TCS-TR-A-**11-50**

TCS Technical Report

πDD: A New Decision Diagram for Manipulating Sets of Permutations

by

Shin-ichi Minato

Division of Computer Science

Report Series A

February 21, 2011

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682

Fax: +81-011-706-7682

πDD: A New Decision Diagram for Manipulating Sets of Permutations

Shin-ichi Minato*
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

February 21, 2011

(Abstract) Permutations and combinations are a couple of basic concepts in elementary combinatorics. Permutations appear in various problems such as sorting, ordering, matching, coding, and many other real-life situations. While conventional SAT problems are discussed in combinatorial space, considering "permutatorial" SAT or CSPs is also an interesting and practical research topic.

In this paper, we propose a new decision diagram "πDD," for compact and canonical representation of a *set of permutations*. Similarly to ordinary BDD or ZDD, πDD has efficient algebraic set operations such as union, intersection, etc. In addition, πDD has a special Cartesian product operation to generate all possible composite permutations for given two sets of permutations. This is a beautiful and powerful property of πDDs.

We present two application examples of πDDs: designing permutation networks and analysis of Rubik's Cube. The experimental results show that πDD-based method can explore billions of permutations in a feasible time and space, using simple algebraic operations for solving the problems.

1 Introduction

Permutations and combinations are a couple of basic concepts in elementary combinatorics and discrete mathematics [4]. Permutations appear in various problems such as sorting, ordering, matching coding, and many other real-life situations. Permutations are also important in group theory since they correspond to bijective functions and generate symmetric groups. While conventional SAT problems are defined in combinatorial space, considering "permutatorial" SAT or CSPs is also an interesting research topic.

In this paper, we propose a new decision diagram "πDD," for compact and canonical representation of a *set of permutations.* πDD is based on BDD (Binary Decision Diagram)[1] and ZDD (Zero-suppressed BDD)[6]. Ordinary BDDs/ZDDs are the representations of propositional logic functions or *sets of combinations*, namely, they represent partial sets of combinatorial space. The data structures and algorithms on BDDs/ZDDs have been researched for more than twenty years. BDD/ZDD-based SAT solving techniques has also been studied [2]. However, most of DD-based methods are limited to combinatorial space, and no practical results are known about directly solving permutational problems although there are so many important applications.

πDD is the first practical idea for efficiently manipulating sets of permutations based on decision diagrams. This data structure can compress a large number of permutations into a compact and canonical representation. As well as ordinary BDDs/ZDDs, πDDs have efficient algebraic set operations such as union, intersection, and difference. In addition, πDDs have a special Cartesian product operation to generate all possible composite permutations (cascade of two permutations) for given two sets of permutations. This is a beautiful and powerful property for solving various problems in

*He also works for ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency.

permutation space. For example, we can represent the primitive moves of Rubik's Cube by a small πDD, and just multiplying itself for k times, we can generate one canonical πDD representing all possible positions of up to k moves. The computation time depends on the πDD size, which is sometimes much less than the number of positions. Once we have generated πDDs for a problem, we can easily apply various analysis or testing such as counting the exact number of permutations, exploring satisfiable permutations for a given constraint, and calculating the minimum or average cost of all permutations.

The idea of πDD gives us a hint for applying the state-of-the-art techniques for combinatorial problems into "permutatorial world." There is a rich body of group theory led by Galois and many researchers in discrete mathematics [3]. πDD is a new computation technique in such research fields, and we can expect a lot of exciting future work.

In the rest of this paper, Section 2 describes some notations and the basics of BDDs/ZDDs. In Section 3, we propose the structures of πDDs. Section 4 gives the algorithms of algebraic operations for πDDs, followed by Section 5 to show experimental results for two typical problems: designing permutation networks, and analysis of Rubik's Cube.

2 Preliminaries

2.1 Sets of Permutations

A *permutation* is a bijective function $\pi : S \rightarrow S$, where S is a finite set $\{1, 2, 3, \ldots, n\}$. It is often confusing but in this paper we use the notation for a permutation $\pi = (a_1, a_2, a_3, \ldots, a_n)$ as each item k moves to a_k. For example, $\pi = (4, 2, 1, 3)$ means $1 \rightarrow 4,\ \ 2 \rightarrow 2,\ \ 3 \rightarrow 1$, and $4 \rightarrow 3$. In this case, we may also use multiplicative forms as $1\pi = 4$, $2\pi = 2,\ \ 3\pi = 1$, and $4\pi = 3$. *Composition* of two permutations $\pi_1\pi_2$ means just a composition of the two bijective functions. For example, $\pi_1 = (3, 1, 2)$ and $\pi_2 = (3, 2, 1)$ then $\pi_1\pi_2 = (1, 3, 2)$ because $1\pi_1\pi_2 = 3\pi_2 = 1,\ \ 2\pi_1\pi_2 = 1\pi_2 = 3$, and $3\pi_1\pi_2 = 2\pi_2 = 2$. In general, $\pi_1\pi_2 \neq \pi_2\pi_1$.

In this paper, π_e means an *identical* permutation $(1, 2, 3, \ldots, n)$. Clearly $\pi\pi_e = \pi_e\pi = \pi$ for any π. We define *dimension* of a permutation $dim(\pi)$ as the largest item number moved by π. For example, $dim((3, 1, 2, 4)) = 3$ because the item 4 does not move. We set $dim(\pi_e) = 0$, and otherwise $dim(\pi) \geq 2$. We sometimes omit the items larger than $dim(\pi)$. For example, (3,2,1,4,5) can be written as just (3,2,1).

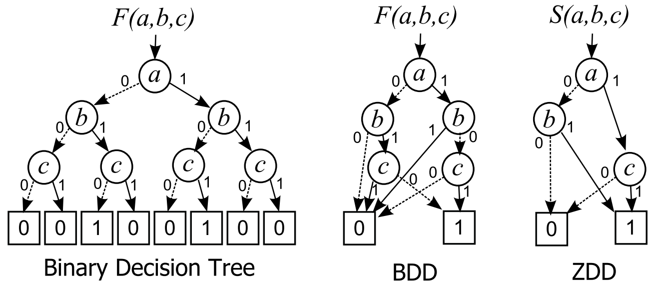

Figure 1: Binary Decision Tree, BDD and ZDD

The main subject of this paper is representing *sets of permutations*. We describe them as $P = \{\pi_e, (2, 1), (2, 3, 1)\}$. Empty set is denoted as $\emptyset$. We also define *dimension for a set of permutations*, such that $dim(P) = max(\{dim(\pi) |\ \pi \in P\})$. We set $dim(P) = 0$ iff $P = \emptyset$ or $P = \{\pi_e\}$, otherwise $dim(P) \geq 2$.

We may use a multiplicative notation between a set of permutation P and a permutation π, as $P \cdot \pi = \{\pi'\pi \mid \pi' \in P\}$

2.2 BDDs and ZDDs

A Binary Decision Diagram (BDD) [1] is a graph representation for a Boolean function. As illustrated in Fig. 1, It is derived by reducing a binary decision tree graph, which represents a decision making process by the input variables. If we fix the order of input variables, and apply the following two reduction rules, then we have a compact canonical form for a given Boolean function:

(1) Delete all redundant nodes whose two edges have the same destination, and

(2) Share all equivalent nodes having the same child nodes and the same variable.

The compression ratio by using a BDD depends on the property of Boolean function to be represented,

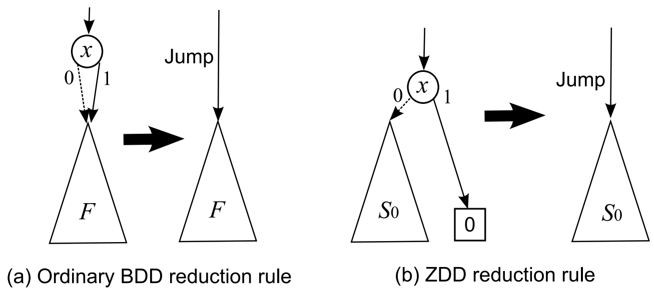

Figure 2: ZDD reduction rule.

but it can be 10 to 100 times in some practical cases. In addition, we can systematically construct a BDD as the result of a binary logic operation (i.e. AND, OR) for a given pair of operand BDDs. This algorithm is based on hash table techniques, and computation time is almost linear to the BDD size.

Zero-suppressed BDD (ZDD) [6] is a variant of BDD, customized for manipulating *sets of combinations*. ZDDs are based on the special reduction rules different from ordinary ones. As shown in Fig. 2, we delete all nodes whose 1-edge directly points to the 0-terminal node, but do not delete the nodes which were deleted in ordinary BDDs. As well as ordinary BDDs, ZDDs give compact canonical representations for sets of combinations. We can construct ZDDs by applying algebraic set operations such as union, intersection and difference, which correspond to logic operations in BDDs.

The zero-suppressing reduction rule is extremely effective if we handle a set of sparse combinations. If the average appearance ratio of each item is 1%, ZDDs are possibly more compact than ordinary BDDs, up to 100 times. Such situations often appear in real-life problems, for example, in a supermarket, the number of items in a customer's basket is usually much less than all the items displayed there. ZDD is now widely recognized as the most important variant of BDD. (for details, see Knuth's book fascicle [5].)

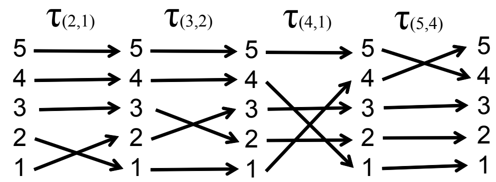

Figure 3: Decomposition for a permutation (3,5,2,1,4).

3 Data Structures

3.1 Desired Properties for πDDs

Before discussing the structure of πDDs, we list the basic properties desired for πDDs to represent sets of permutations.

- Empty set $\emptyset$ corresponds to a 0-terminal node in πDD since this is a zero element for union operation.
- Singleton set $\{\pi_e\}$ may correspond to a 1-terminal node since this is an identity element for composite operation.
- The form of πDD for P does not depend on the items larger than $dim(P)$. For example, $\{(3,2,1),(2,1)\}$ and $\{(3,2,1,4,5),(2,1,3,4,5)\}$ should be the same πDD.
- A πDD should provide a canonical (unique) representation for a set of permutations. This enables efficient equivalence checking and satisfiability testing.
- Each path from the root node to a 1-terminal node should correspond to a permutation included in the set, namely, the number of paths corresponds to the cardinality of the set.

3.2 Decomposition of Permutations

A *transposition* is a basic permutation of just exchanging two items. In this paper, $\tau_{(x,y)}$ denotes the transposition of the item x and y. Clearly, $\tau_{(x,y)} = \tau_{(y,x)}$ and $(\tau_{(x,y)})^2 = \pi_e$ for any x and y. We set $\tau_{(x,x)} = \pi_e$.

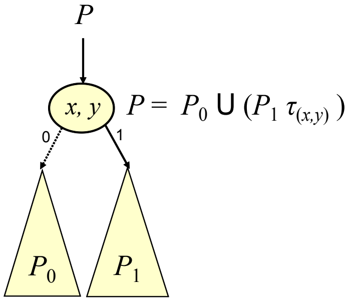

Figure 4: Basic structure of πDD.

The key idea of πDD is based on the observation that any permutation π can be decomposed into a sequence of up to $(dim(\pi) - 1)$ of transpositions. For example, a permutation $(3, 5, 2, 1, 4)$ can be decomposed into $\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)}\tau_{(5,4)}$, as illustrated in Fig. 3.

Theorem 1 *Any non-identical permutation π has a decomposition form which consists of up to $(dim(\pi) - 1)$ transpositions, and there is a way to give a unique decomposition form for any given permutation.*

(Proof) If $dim(\pi) = 2$ then π should be one transposition $\tau_{(2,1)}$. Now we assume $dim(\pi) > 2$. Let $x = dim(\pi)$ and let $\pi_1 = \pi \cdot \tau_{(x,x\pi)}$, then $x\pi_1 = x$ holds. Since x is not moved by π_1, $dim(\pi_1) < dim(\pi)$. The equation $\pi_1 = \pi \cdot \tau_{(x,x\pi)}$ can be transformed into $\pi = \pi_1 \cdot \tau_{(x,x\pi)}$, thus, π can be decomposed into a permutation π_1 followed by one transposition. Applying this procedure to π_1 recursively, the dimension is monotonically decreasing, and eventually we can obtain a unique decomposition form which consists of up to $(dim(\pi) - 1)$ transpositions. ∎

For the example shown in Fig. 3, the dimension is 5 and the item 5 is moved to 4, so we obtain $(3, 5, 2, 1, 4) = (3, 4, 2, 1) \cdot \tau_{(5,4)}$. Next, the dimension is 4 and the item 4 is moved to 1, so we get $(3, 4, 2, 1) = (3, 1, 2) \cdot \tau_{(4,1)}$. Similarly we next get $(3, 1, 2) = (2, 1) \cdot \tau_{(3,2)}$, and finally $(2, 1) = \tau_{(2,1)}$. In total, we obtain a sequence of 4 transpositions. This procedure is deterministic and the result is unique for a given permutation.

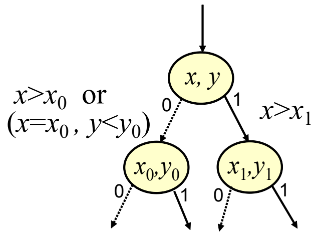

Figure 5: Variable ordering rules in πDD.

3.3 Structures of πDD

From the above observation, we can uniquely represent a permutation by using a combination of transpositions. Since ZDDs are efficient representations for sets of combinations, we may have somehow ZDD-like data structure for representing sets of permutations.

Figure 4 shows the main idea of πDDs. We assign a pair of item IDs (x, y) for each decision node, where $x = dim(P)$ and $x > y \geq 1$. Each decision node has semantics that:

$$P = P_0 \cup (P_1 \cdot \tau_{(x,y)}),$$

where P_0 and P_1 represent a partition of P decided by the existence of $\tau_{(x,y)}$ in their decomposition forms. More formally, they are described as:

$$\begin{aligned} P_0 &= \{\pi \mid \pi \in P,\ x\pi \neq y\}, \\ P_1 &= \{\pi\tau_{(x,y)} \mid \pi \in P,\ x\pi = y\}. \end{aligned}$$

Notice that $dim(P_1) < dim(P)$ holds since x never moved by any permutation in P_1. Applying this expansion recursively, we eventually obtain one of the two trivial set of permutations, empty set $\emptyset$ (0-terminal node) or singleton set $\{\pi_e\}$ (1-terminal node).

Similarly to ordinary ZDDs, we need a fixed variable ordering for all $\tau_{(x,y)}$ to preserve unique representations of πDDs. We use the following ordering from the bottom to top:

$$(2, 1)(3, 2)(3, 1)(4, 3)(4, 2)(4, 1)(5, 4)(5, 3)(5, 2)(5, 1)\ldots$$

Figure 5 shows the rules of variable ordering between the two adjacent decision nodes in our πDDs.

In a πDD, any combination of transpositions can be represented by a unique path from the root node to a 1-terminal node.

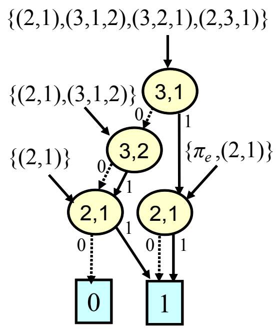

Figure 6: Multi-rooted shared πDD.

Finally we confirm the node reduction rules in πDDs. As well as ordinary ZDDs, equivalent node sharing is effective to πDDs. (Notice that we need to check a pair of items (x, y) instead of only one decision variable in ZDDs.) For redundant node deletion, the zero-suppressing rule works very well for πDDs since unnecessary transpositions are automatically deleted, and thus the nodes for unmoved items never appear in πDDs.

As well as ordinary BDDs/ZDDs, multiple πDDs can share their subgraphs to each other in a multi-rooted πDD, as shown in Fig. 6.

4 Algorithms for Algebraic Operations

In the previous section, we presented the basic structures of πDDs. However, we should consider not only the compact representation but also efficient manipulation algorithms. Similarly to ordinary BDDs/ZDDs, πDDs can be constructed by applying algebraic operations, as illustrated in Fig. 7. Table 1 summarizes the primitive operations of πDDs for manipulating sets of permutations. Here we present how to compute these operations efficiently. We want to develop a good algorithm in linear or a small order of polynomial time for the size of πDD, which is sometimes much less than the total number of permutations.

4.1 Binary Set Operations

First we consider the three binary set operations: union, intersection, and difference. As written above, πDD is based on the expansion: $P = P_0 \cup (P_1 \cdot \tau_{(x,y)})$ on each decision node. Since the two parts P_0 and $(P_1 \cdot \tau_{(x,y)})$ are disjoint, and since τ operation is independent of union, intersection, and difference operations, we can execute those set operations as well as ordinary BDDs/ZDDs. For example, intersection operation can be written as follows:

$$
\begin{aligned}
P \cap Q &= (P_0 \cup (P_1 \cdot \tau_{(x,y)})) \cap (Q_0 \cup (Q_1 \cdot \tau_{(x,y)})) \\
&= (P_0 \cap Q_0) \cup ((P_1 \cap Q_1) \cdot \tau_{(x,y)}).
\end{aligned}
$$

Then, $(P_0 \cap Q_0)$ and $(P_1 \cap Q_1)$ are called recursively. Similarly to ordinary BDDs/ZDDs, we can avoid duplicated recursive calls by using cache to store the previous operations and their results.

4.2 Transposition

Next we consider the transposition operation with any pair of items for a given set of permutations. Let P is a given πDD and $P.top = (x, y)$. Now we want to compute $P \cdot \tau_{(u,v)}$. If $u > x$, we may just return a decision node with items (u, v), whose 0-edge points to $\emptyset$ and whose 1-edge points to P. On the other hand, if $u \leq x$, we need more complicated work to traverse the internal nodes of P.

To consider the algorithm, we recall an example of permutation $(3, 5, 2, 1, 4)$ shown in Fig. 3, and let us compute $(3, 5, 2, 1, 4)\ \tau_{(3,1)}$. In πDDs, $(3, 5, 2, 1, 4)$ is represented by a sequence of transpositions $\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)}\tau_{(5,4)}$, thus we should compute $(\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)}\tau_{(5,4)})\ \tau_{(3,1)}$. Then, we can observe the following transformation:

$$
\begin{aligned}
& (\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)}\tau_{(5,4)})\ \tau_{(3,1)} \\
=\ & (\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)})\ (\tau_{(5,4)}\tau_{(3,1)}) \\
=\ & (\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)})\ (\tau_{(3,1)}\tau_{(5,4)}) \\
=\ & (\tau_{(2,1)}\tau_{(3,2)})\ (\tau_{(4,1)}\tau_{(3,1)})\ \tau_{(5,4)} \\
=\ & (\tau_{(2,1)}\tau_{(3,2)})\ (\tau_{(3,1)}\tau_{(4,3)})\ \tau_{(5,4)} \\
=\ & \tau_{(2,1)}\ (\tau_{(3,2)}\tau_{(3,1)})\ \tau_{(4,3)}\tau_{(5,4)} \\
=\ & \tau_{(2,1)}\ (\tau_{(2,1)}\tau_{(3,2)})\ \tau_{(4,3)}\tau_{(5,4)} \\
=\ & (\tau_{(2,1)}\tau_{(2,1)})\ \tau_{(3,2)}\tau_{(4,3)}\tau_{(5,4)} \\
=\ & \tau_{(3,2)}\tau_{(4,3)}\tau_{(5,4)}.
\end{aligned}
$$

In this transformation, adjacent two transpositions are compared, and if the order violates the πDD's manner, then the two transpositions

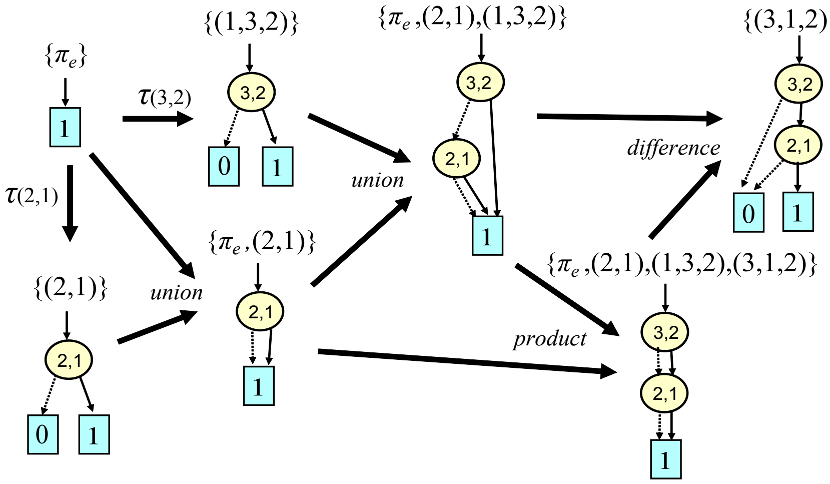

Figure 7: Construction of πDDs by algebraic operations.

Table 1: Primitive πDD operations.

$\emptyset$	Returns empty set. (0-termial node)
$\{\pi_e\}$	Returns singleton set. (1-terminal node)
$P.top$	Returns IDs (x, y) at the root node of P.
$P \cup Q$	Returns $\{\pi \mid \pi \in P$ or $\pi \in Q\}$.
$P \cap Q$	Returns $\{\pi \mid \pi \in P,\ \pi \in Q\}$.
$P \setminus Q$	Returns $\{\pi \mid \pi \in P,\ \pi \notin Q\}$.
$P.\tau(x, y)$	Returns $P \cdot \tau_{(x,y)}$.
$P * Q$	Returns $\{\alpha\beta \mid \alpha \in P,\ \beta \in Q\}$.
$P.cofact(x, y)$	Returns $\{\pi\tau_{(x,y)} \mid \pi \in P,\ x\pi = y\}$.
$P.count$	Returns the number of permutations.

are exchanged. For example, $(\tau_{(5,4)}\tau_{(3,1)})$ is exchanged into $(\tau_{(3,1)}\tau_{(5,4)})$, and $(\tau_{(4,1)}\tau_{(3,1)})$ becomes $(\tau_{(3,1)}\tau_{(4,3)})$. In this way, eventually we can obtain a normalized decomposition form of πDDs. We should be careful that some item numbers are slightly changed in this process.

Figure 8 illustrates an example of exchange from $\tau_{(x,y)}\tau_{(u,v)}$ to $\tau_{(u',v)}\tau_{(x,y')}$. In this example, u, v, and x are kept but y is changed. Here we determine that this exchange is always possible for any pair of transpositions, and in which cases the items need to be changed.

Theorem 2 *For given positive integers x, y, u, v with $x > y > 0$ and $x \geq u > v$, a pair of cascaded transpositions $\tau_{(x,y)}\tau_{(u,v)}$ can be transformed into π_e or $\tau_{(u',v)}\tau_{(x,y')}$, where u' and y' are some positive integers satisfying $u' < x$ and $x > y' > 0$.*

(Proof) If $\tau_{(x,y)}$ and $\tau_{(u,v)}$ have no collisions of items, they can be exchanged transparently. Now we check all the collision cases. If $y = u$ then $u' = u$ and $y' = v$. If $y = v$ then $u' = y' = u$. If $x = u$ then $u' = y' = y$. If $x = u$ and $y = v$ then $\tau_{(x,y)}\tau_{(u,v)} = \pi_e$. Otherwise, just $u' = u$ and $y' = y$. ∎

Based on this theorem, now we can implement a recursive algorithm for transposition operation. If $P.top = (x, y)$ and $u \leq x$, then $P \cdot \tau_{(u,v)}$ can be written as follows:

$$
\begin{aligned}
P \cdot \tau_{(u,v)} &= (P_0 \cup (P_1 \cdot \tau_{(x,y)})) \cdot \tau_{(u,v)} \\
&= P_0 \cdot \tau_{(u,v)} \cup (P_1 \cdot (\tau_{(x,y)}\tau_{(u,v)})) \\
&= (P_0 \cdot \tau_{(u,v)}) \cup ((P_1 \cdot \tau_{(u',v)}) \cdot \tau_{(x,y')})
\end{aligned}
$$

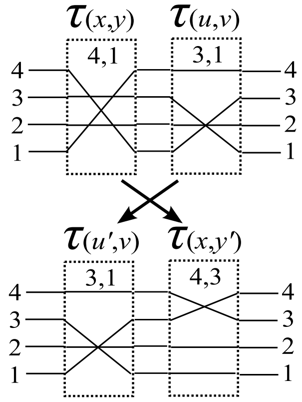

Figure 8: Exchange of adjacent transpositions.

This formula shows that we may return a decision node with IDs (x, y'), whose 0-edge points to the result of $P_0 \cdot \tau_{(u,v)}$ and whose 1-edge points to the result of $P_1 \cdot \tau_{(u',v)}$. (Here we should notice that $dim(P_1 \cdot \tau_{(u',v)})$ must be less than x.) Each sub-operation can be computed by a recursive call, and eventually we have a trivial case. As well as other operations, we can avoid duplicated recursions by using the operation cache.

4.3 Cartesian Product

Cartesian product, $P * Q = \{\alpha\beta \mid \alpha \in P,\ \beta \in Q\}$, computes a sets of all possible composite permutations chosen from P and Q. This is the most important and useful operation in manipulating permutations.

Using transposition operations, the product $P * Q$ can be written as follows. Here we assume $Q.top = (x, y)$.

$$\begin{aligned} P * Q &= P * (Q_0 \cup (Q_1 \cdot \tau_{(x,y)})) \\ &= (P * Q_0) \cup ((P * Q_1) \cdot \tau_{(x,y)}) \end{aligned}$$

This formula indicates that we may recursively call sub-operations $(P * Q_0)$ and $(P * Q_1)$, and eventually we have trivial operation $P * \emptyset$ or $P * \{\pi_e\}$. As well as the other operations, we can avoid duplicated recursions by using the operation cache. However, one different point here is that we cannot assure $dim(P * Q_1) < x$, so we need to apply general transposition operation for $(P * Q_1) \cdot \tau_{(x,y)}$.

Figure 9 shows an example of product operation for two πDDs whose items are disjoint to each other. In this case, the number of permutations increases multiplicatively but the πDD size increases only additively. Computation time also depends on the πDD size, and in such cases πDD-based method is exponentially effective than using an explicit data structure.

4.4 Cofactor

After generating a πDD for a set of permutations, we want to extract a subset of permutations to check a certain property is satisfied or not. *Cofactor* operation,

$$P.cofact(u, v) = \{\pi\tau_{(u,v)} \mid \pi \in P,\ u\pi = v\},$$

generates a subset of permutations such that the item u is moved to v. For example,

$$\begin{aligned} & \{(3,2,1),(2,3,1),(1,3,2),(2,1)\}.cofact(3,1) \\ = \;& \{(3,2,1)\tau_{(3,1)},(2,3,1)\tau_{(3,1)}\} \\ = \;& \{\pi_e,(2,1)\}. \end{aligned}$$

Notice that $P.cofact(u, u)$ can extract the permutations where u is not moved. Using cofactor and other set operations, various constraints can be specified and applied to πDDs.

Here we discuss the way to compute cofactor operation. If (u, v) corresponds to $P.top$, we may just return the 1-edge of the root node. Otherwise, we need to traverse the internal nodes in P. We can observe that the following equation holds.

$$P.cofact(u, v) = (P \cdot \tau_{(u,v)}).cofact(u, u),$$

Thus, the cofactor operation can be computed by using transposition operation. Due to the space limitation, we omit the detailed algorithm for implementation of cofactor operation.

5 Application Examples

Here we present two application examples and experimental results. We implemented a prototype version of πDD manipulator based on our own

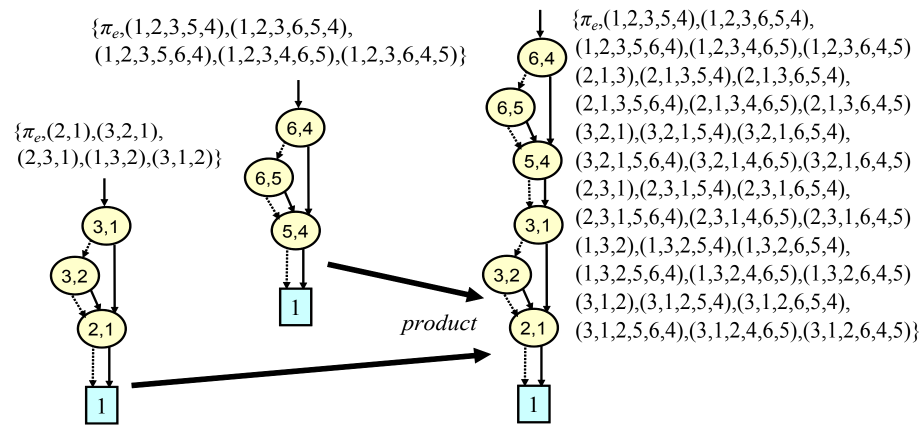

Figure 9: Example of Cartesian product.

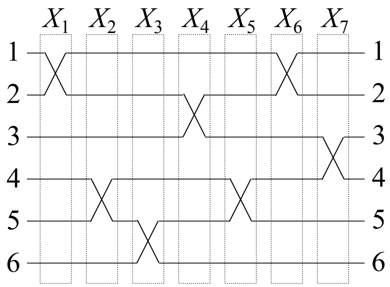

Figure 10: A permutation network for (4,2,1,6,5,3).

BDD/ZDD package. The program is written in 330 lines of C++ codes, newly added to the basic libraries including 6,000 lines of C/C++ codes. The following experiments are performed by a 2.4GHz Core2Duo PC with 2GB memory, SuSE 10, and GNU C++.

5.1 Design of Permutation Networks

A *permutation network* is an n-input and n-output network to generate any one permutation. Such circuits are often used in customized hardware of cryptographic systems and signal processing systems. Here we consider a style of permutation networks using a set of n-bit parallel lines with a number of exchange switches X_k between any pair of adjacent lines, as shown in Fig. 10. Now we want to design an optimal layout of switches for a given permutation.

A set of permutations given by one switch can be written as $\bigcup_{i=1}^{n-1} \tau_{(i,i+1)}$. Thus, all possible permutations generated by up to k switches are described as follows.

$$
\begin{aligned}
P_0 &= \pi_e \\
P_1 &= P_0 \cup \left(\bigcup_{i=1}^{n-1} \tau_{(i,i+1)}\right) \\
P_k &= P_{k-1} * P_1 \qquad (\text{for } k \geq 2)
\end{aligned}
$$

According to this iterative formula, we can generate πDDs for $P_0, P_1, P_2, \ldots$ by increasing k, and eventually we must have $P_{k+1} = P_k$ for any $k \geq m$. Then, m shows the minimum number of switches to cover all permutations.

Table 2 shows the experimental result for 10-bit permutation network. In this table, "πDD size" shows the number of decision nodes in the πDD, "# of perm." means the number of permutations included in P_k, "total $\#\tau$" is the total number of transpositions included in all permutations in P_k. Notice that the total $\#\tau$ corresponds to the data size when using an explicit representation for P_k.

The result shows that P_{46} is equivalent to P_{45}, thus we can see $m = 45$. In other words, 45 switches are enough to cover all 362,880 (=10!) permutations. The numbers of permutations and total transpositions increase monotonically in this iteration process, however, πDD sizes have a peak of 10,894 at P_{27}, and finally we need only 45 decision nodes of πDD for representing all the 10! permutations. The latter P_k's may have more beautiful structures and the πDD nodes are well shared, even they include very large number of permutations.

Table 2: Experimental results for 10-bit permutation network.

P_k	πDD size	# of perm.	total $\#\tau$
P_0	0	1	0
P_1	9	10	9
P_2	31	54	97
P_3	63	209	546
P_4	109	649	2152
P_5	172	1717	6704
P_6	261	4015	17632
P_7	390	8504	40751
P_8	558	16599	84985
P_9	773	30239	162995
P_{10}	1034	51909	291537
P_{11}	1353	84592	491272
P_{12}	1727	131635	786100
P_{13}	2169	196524	1201963
P_{14}	2688	282578	1764353
P_{15}	3286	392588	2495497

P_k	πDD size	# of perm.	total $\#\tau$
P_{16}	3956	528441	3412177
P_{17}	4685	690778	4522462
P_{18}	5455	878737	5821218
P_{19}	6249	1089826	7296041
P_{20}	7047	1319957	8915085
P_{21}	7834	1563651	10645703
P_{22}	8591	1814400	12433871
P_{23}	9293	2065149	14239194
P_{24}	9905	2308843	15996836
P_{25}	10397	2538974	17671711
P_{26}	10735	2750063	19206325
P_{27}	**10894**	2938022	20584666
P_{28}	10857	3100359	21772380
P_{29}	10614	3236212	22773147
P_{30}	10157	3346222	23579581
P_{31}	9497	3432276	24214975

P_k	πDD size	# of perm.	total $\#\tau$
P_{32}	8655	3497165	24691907
P_{33}	7669	3544208	25039740
P_{34}	6590	3576891	25279788
P_{35}	5470	3598561	25439624
P_{36}	4374	3612201	25539440
P_{37}	3353	3620296	25598543
P_{38}	2444	3624785	25630975
P_{39}	1671	3627083	25647411
P_{40}	1055	3628151	25654943
P_{41}	602	3628591	25657983
P_{42}	305	3628746	25659023
P_{43}	136	3628790	25659303
P_{44}	59	**3628799**	25659355
P_{45}	**45**	**3628800**	25659360
P_{46}	**45**	**3628800**	25659360

Table 3: Experimental results for n-bit permutation networks.

n	m	πDD size (peak)	πDD size (final)	# of perm.	total $\#\tau$	time (sec)
1	0	0	0	1	0	0.00
2	1	1	1	2	1	0.00
3	3	3	3	6	7	0.00
4	6	9	6	24	46	0.00
5	10	27	10	120	326	0.00
6	15	89	15	720	2556	0.01
7	21	292	21	5040	22212	0.02
8	28	972	28	40320	212976	0.06
9	36	3241	36	362880	2239344	0.26
10	45	10894	45	3628800	25659360	1.19
11	55	36906	55	39916800	318540960	5.77
12	66	125904	66	479001600	4261576320	27.06
13	78	435221	78	6227020800	61148511360	126.80
14	91	1520439	91	87178291200	937030429440	666.29

We can also observe that P_{45} and P_{44} have only one difference in the number of permutations, and just by applying the difference set operation ($P_{45} \setminus P_{44}$), we can confirm that the last permutation is (10,9,8,7,6,5,4,3,2,1). By applying algebraic operations of πDDs for P_k's, we can determine the minimum number of switches for any given permutation, and we can find a layout of the switches to make it.

Table 3 presents the results for n-bit permutation networks, up to $n = 14$. We show the peak and the final size of πDDs and their computation time. The number of all permutations is clearly $n!$, however, the final πDD size is only $n(n-1)/2$. The peak πDD size grows exponentially, but seems slower than $n!$. Here we can observe that πDDs are 1000 times or more compact than explicit representations.

5.2 Analysis of Rubik's Cube

Rubik's Cube$^{\mathrm{TM}}$would be one of the most popular puzzles related to permutation group theory. πDD is also useful for analyzing Rubik's Cube. Here we focus only the moves of the eight corner cubes. Figure 11 illustrates our assignment of the items to all the 24 faces of the corner cubes. Then we can describe 90° moves for X-, Y-, and Z-axis, as

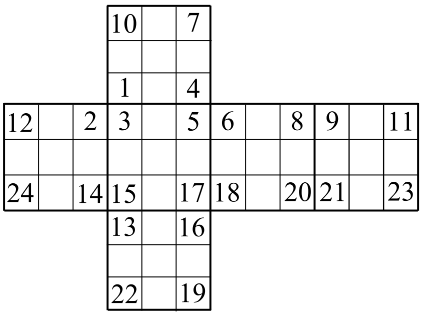

Figure 11: Item assignments for the corner cubes of Rubik's Cube.

Table 4: Experimental results for Rubik's Cube.

P_k	πDD size	# of perm.	total $\#\tau$
P_0	0	1	0
P_1	63	10	72
P_2	392	64	888
P_3	1789	385	5634
P_4	6860	2232	34446
P_5	23797	12224	194406
P_6	84704	62360	1012170
P_7	290018	289896	4752582
P_8	**608666**	1159968	19087266
P_9	580574	3047716	50272542
P_{10}	18783	3671516	60540732
P_{11}	**511**	**3674160**	60579900
P_{12}	**511**	**3674160**	60579900

follows.

$$
\begin{aligned}
\pi_x &= \tau_{(3,5)}\tau_{(3,17)}\tau_{(3,15)}\tau_{(1,6)}\tau_{(1,16)}\tau_{(1,14)} \\
&\quad \tau_{(2,4)}\tau_{(2,18)}\tau_{(2,13)} \\
\pi_y &= \tau_{(2,14)}\tau_{(2,24)}\tau_{(2,12)}\tau_{(3,13)}\tau_{(3,23)}\tau_{(3,10)} \\
&\quad \tau_{(1,15)}\tau_{(1,22)}\tau_{(1,11)} \\
\pi_z &= \tau_{(1,10)}\tau_{(1,7)}\tau_{(1,4)}\tau_{(3,12)}\tau_{(3,9)}\tau_{(3,6)} \\
&\quad \tau_{(2,11)}\tau_{(2,8)}\tau_{(2,5)}
\end{aligned}
$$

and then all possible permutations of at most one of the primitive moves ($+90°$, $-90°$, and $180°$ for each axis) are described as follows.

$$
\begin{aligned}
P_1 &= \pi_e + \pi_x + {\pi_x}^2 + {\pi_x}^3 + \pi_y + {\pi_y}^2 + {\pi_y}^3 \\
&+ \pi_z + {\pi_z}^2 + {\pi_z}^3
\end{aligned}
$$

Now we can generate the set of permutations by up to k moves by the following simple iterative formula.

$$P_k = P_{k-1} * P_1 \qquad (\text{for } k \geq 2)$$

Similarly to the case of permutation networks, we must have a fixed point m such that $P_{k+1} = P_k$ for any $k \geq m$. If we ignore the edge cubes and center cubes, P_m contains all meaningful patterns of the eight corner cubes. (Notice that the cube of $\{19, 20, 21\}$ is fixed to the original position to eliminate symmetric patterns.)

Table 4 shows the result of generating πDDs for the P_k's. We can see that the number of all possible patterns of corner cubes is 3,674,160. We confirmed that 11 moves are enough to generate all the possible patterns, in other words, any patterns of the corner cubes can be returned to the original positions in 11 or less moves. We need only 511 decision nodes of πDDs for representing all patterns, and P_8 has a peak of πDD size as 608,666. The computation time was 207 seconds for generating all πDDs.

After generating πDDs for the P_k's, we can analyze various properties of Rubik's Cube. For example, we may explore the patterns such that only two corner cubes are moving and the other six cubes stay at the original positions. Such patterns can be detected by cofactor operations as follows.

$$
\begin{aligned}
S_k &= P_k.\mathit{cofact}(9,9).\mathit{cofact}(11,11).\mathit{cofact}(15,15) \\
&\quad .\mathit{cofact}(17,17).\mathit{cofact}(21,21).\mathit{cofact}(23,23)
\end{aligned}
$$

Our experiment shows that, for $k \leq 9$, S_k only includes π_e. In $k = 10$, we newly found (2,3,1,6,4,5), (3,1,2,5,6,4), (4,5,6,1,2,3), and (6,4,5,2,3,1). Using the maximum moves ($k = 11$), we can make (6,4,5,2,3,1). After such a pattern is detected, it is not hard to find a sequence of moves to generate it. We may apply one of the primitive moves into the final pattern to make a candidate of one step previous pattern, and check the existence in P_{k-1}. At least one of the candidates must be in P_{k-1}, and then we can repeat the process until P_1.

Here we considered only the corner cubes, but recently, Rokicki et al. [7] confirmed that all cube's patterns can be solved in 20 moves and this is the exact minimum. They applied some mathematical pruning and then used total 35 CPU years of massive parallel PCs. A straight forward application of πDDs to this problem may cause memory overflow, but in somehow our method will be useful to accelerate such kind of problem solving.

6 Conclusion

In this paper, we proposed a new idea of decision diagrams for manipulating sets of permutations. The method of πDDs provides a hint for applying the state-of-the-art techniques for combinatorial problems into "permutatorial world." There is a rich body of group theory led by Gallois and many researchers in discrete mathematics [3]. We can expect a lot of exciting future work, for example, providing software tools for studying group theory, considering many other practical applications, implementing more various operations for sets of permutations, and considering extended models such as sets of k-out-of-n permutations or multisets of permutations.

Acknowledgement

The author would like to thank all the colleagues in Algorithm Laboratory of Hokkaido University and ERATO MINATO Discrete Structure Manipulation System Project for their fruitful discussions.

References

[1] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[2] P. Chatalic and L. Simon. Zres: The old davis-putnam procedure meets ZBDDs. In *7th International Conference on Automated Deduction (CADE'17), LNAI 1831*, pages 449–454, 2000.

[3] GAP Forum. *GAP – Groups, Algorithms, Programming – a System for Computational Discrete Algebra*, 2008. `http://www.gap-system.org/`.

[4] D. E. Knuth. Combinatorial properties of permutations. In *The Art of Computer Programming*, volume 3, chapter 5.1, pages 11–72. Addison-Wesley, 1998.

[5] D. E. Knuth. *The Art of Computer Programming: Bitwise Tricks & Techniques; Binary Decision Diagrams*, volume 4, fascicle 1. Addison-Wesley, 2009.

[6] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of 30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[7] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. God's number is 20. `http://www.cube20.org/`, 2010.