πDD: A New Decision Diagram for Efficient Problem Solving in Permutation Space

Shin-ichi Minato

Hokkaido University, Sapporo 060-0814, Japan**
minato@ist.hokudai.ac.jp

Abstract. Permutations and combinations are two 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, "permutatorial" SAT and CSPs also constitute an interesting and practical research topic.
In this paper, we propose a new type of decision diagram named "πDD," for compact and canonical representation of a *set of permutations*. Similarly to an ordinary BDD or ZDD, πDD has efficient algebraic set operations such as union, intersection, etc. In addition, πDDs hava a special Cartesian product operation which generates all possible composite permutations for two given sets of permutations. This is a beautiful and powerful property of πDDs.
We present two examples of πDD applications, namely, designing permutation networks and analysis of Rubik's Cube. The experimental results show that a πDD-based method can explore billions of permutations within feasible time and space limits by using simple algebraic operations.

1 Introduction

Permutations and combinations are two 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, "permutatorial" SAT and CSPs also constitute an interesting research topic.

In this paper, we propose a new type of decision diagram named "πDD," for compact and canonical representation of *sets of permutations*. πDDs are based on BDDs (Binary Decision Diagrams)[1] and ZDDs (Zero-suppressed BDDs)[6]. Ordinary BDDs/ZDDs provide representations of propositional logic functions or sets of combinations, namely, they represent partial sets of combinatorial space. Data structures and algorithms on BDDs/ZDDs have been researched for more than twenty years, and BDD/ZDD-based SAT solving techniques have also been explored [2]. However, most DD-based methods are limited to combinatorial space, and no practical techniques for direct solving of permutational problems are known, even though they have various important applications.

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

πDDs are the first practical idea for efficient manipulation of sets of permutations on the basis of decision diagrams. This data structure can compress a large number of permutations into a compact and canonical representation. Similarly to ordinary BDDs/ZDDs, πDDs have efficient algebraic set operations such as union, intersection, and difference. In addition, πDDs have a special Cartesian product operation which generates all possible composite permutations (cascade of two permutations) for two given sets of permutations. This is a beautiful and powerful property for solving various problems in permutation space. For example, we can represent the primitive moves of Rubik's Cube with a small πDD, and by simply multiplying this πDD by itself k times, we can generate a single canonical πDD representing all possible positions reachable within k moves. The computation time depends on the size of the πDD, which is sometimes much smaller than the number of positions. Once we have generated πDDs for a problem, we can easily apply various analysis or testing techniques, such as counting the exact number of permutations, exploring satisfiable permutations for a given constraint and calculating the minimal or the average cost of all permutations.

The idea of πDDs provide hints about the application of state-of-the-art SAT techniques used for solving combinatorial problems in the "permutatorial world." There is a rich body of studies in group theory led by Galois and many researchers in discrete mathematics [3]. πDDs represent a new computational technique which can be applied in such research fields, and we can expect it to yield numerous exciting results in the future.

In the rest of this paper, Section 2 describes some notations and the basics of BDDs/ZDDs. In Section 3, we propose the general structure of πDDs, and Section 4 gives the algorithms of algebraic operations for πDDs, followed by Section 5, which presents experimental results for two typical problems, namely, designing permutation networks and analyzing Rubik's Cube.

2 Preliminaries

2.1 Sets of Permutations

A *permutation* is a bijective function $\pi : S \to S$, where S is a finite set $\{1, 2, 3, \ldots, n\}$. Although it is often confusing, in this paper we use the notation for permutation $\pi = (a_1, a_2, a_3, \ldots, a_n)$, in which each item k moves to a_k. For example, $\pi = (4, 2, 1, 3)$ implies $1 \to 4,\ 2 \to 2,\ 3 \to 1$, and $4 \to 3$. In this case, we may also use multiplicative forms, such as $1\pi = 4,\ 2\pi = 2,\ 3\pi = 1$, and $4\pi = 3$. *A composition* of two permutations $\pi_1\pi_2$ simply indicates a composition of two bijective functions. For example, if $\pi_1 = (3, 1, 2)$ 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 denotes an *identical* permutation $(1, 2, 3, \ldots, n)$. Clearly $\pi\pi_e = \pi_e\pi = \pi$ for any π. We define the *dimension* of a permutation $dim(\pi)$ as the highest item number moved by π. For example, $dim((3, 1, 2, 4)) = 3$ as item 4 does not move. We set $dim(\pi_e) = 0$, and otherwise $dim(\pi) \geq 2$. Also, we sometimes omit items larger than $dim(\pi)$. For example, (3,2,1,4,5) can be written simply as (3,2,1).

The main objective of this paper is the representation of *sets of permutations*. We describe such set as $P = \{\pi_e, (2, 1), (2, 3, 1)\}$. The empty set is denoted as $\emptyset$. We also

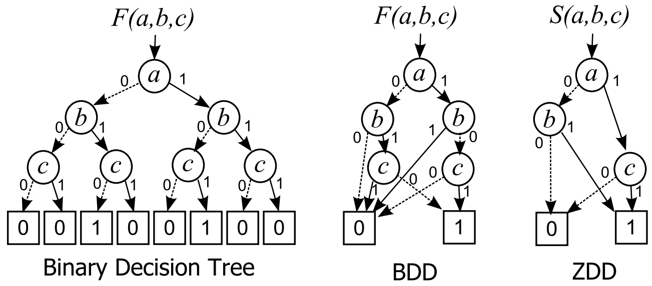

Fig. 1. Binary Decision Tree, BDD and ZDD

define the *dimension of a set of permutations* such that $dim(P) = max(\{dim(\pi) | \ \pi \in P\})$. Finaly, 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, which is defined as follows: $P \cdot \pi = \{\pi'\pi \mid \pi' \in P\}$.

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 through the input variables. If we fix the order of the input variables and apply the following two reduction rules, then we obtain a compact canonical form for a given Boolean function:

(1) Delete all redundant nodes whose both edges have the same destination, and
(2) Share all equivalent nodes having the same child nodes and the same variables.

Although the compression ratio achieved by using a BDD depends on the properties of the Boolean function to be represented, it can be between 10 and 100 times in some practical cases. In addition, we can systematically construct a BDD as a result of a binary logic operation (i.e., AND or OR) for a given pair of operand BDDs. This algorithm is based on hash table techniques, and the computation time is almost linear with respect to the size of the BDD.

A zero-suppressed BDD (ZDD) [6] is a variant of BDD customized for manipulating *sets of combinations*. ZDDs are based on special reduction rules which differ from ordinary ones. As shown in Fig. 2, we delete all nodes whose 1-edge points directly to the 0-terminal node and do not delete the nodes that would be deleted in ordinary BDDs. Similarly to 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 for sets of sparse combinations. If the average appearance rate of each item is 1%, ZDDs are possibly up to 100 times more compact than ordinary BDDs. Such situations often appear in real-life problems, for example, in a supermarket, the number of items in a customer's basket

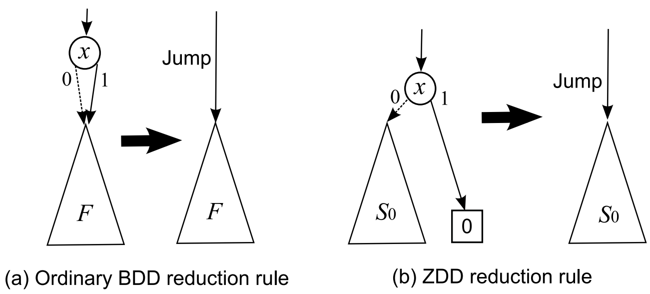

Fig. 2. ZDD reduction rule.

is usually much smaller than the number of all items displayed at the supermarket. ZDDs are now widely recognized as the most important variant of BDDs (for details, see Knuth's book fascicle [5].)

3 Data Structures

3.1 Desired Properties for πDDs

Before discussing the general structure of πDDs, we list the basic properties desired for πDDs which are necessary for representing sets of permutations.

- The empty set $\emptyset$ corresponds to a 0-terminal node in a πDD since this is a zero element for union operation.
- The singleton set $\{\pi_e\}$ corresponds to a 1-terminal node since this is an identity element for composite operations.
- The form of a πDD for P does not depend on items larger than $dim(P)$. For example, $\{(3,2,1),(2,1)\}$ and $\{(3,2,1,4,5),(2,1,3,4,5)\}$ should yield the same πDD.
- A πDD should provide a canonical (unique) representation for a set of permutations. This allows for 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

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

The key idea behind πDDs is based on the observation that any permutation π can be decomposed into a sequence of up to $(dim(\pi) - 1)$ 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.

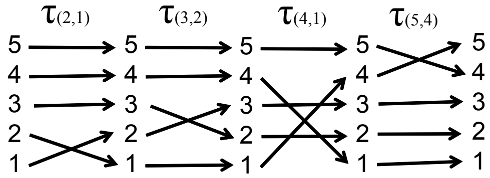

Fig. 3. Decomposition for a permutation (3,5,2,1,4).

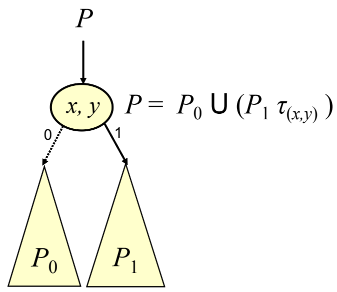

Fig. 4. Basic structure of πDD.

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 obtain a unique decomposition form for any given permutation.*

(Proof) If $dim(\pi) = 2$ then π should be a single transposition $\tau_{(2,1)}$. Next, we assume $dim(\pi) > 2$. If we let $x = dim(\pi)$ and $\pi_1 = \pi \cdot \tau_{(x,x\pi)}$, then $x\pi_1 = x$ holds. Since x is not moved by π_1, then $dim(\pi_1) < dim(\pi)$. The equation $\pi_1 = \pi \cdot \tau_{(x,x\pi)}$ can be transformed into $\pi = \pi_1 \cdot \tau_{(x,x\pi)}$, and thus π can be decomposed into a permutation π_1 followed by one transposition. In applying this procedure to π_1 recursively, the dimension decreses monotonically, 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, item 5 is moved to 4, and we obtain $(3, 5, 2, 1, 4) = (3, 4, 2, 1) \cdot \tau_{(5,4)}$. Next, the dimension is 4, item 4 is moved to 1, and we obtain $(3, 4, 2, 1) = (3, 1, 2) \cdot \tau_{(4,1)}$. Similarly, we subsequently obtain $(3, 1, 2) = (2, 1) \cdot \tau_{(3,2)}$, 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 any given permutation.

3.3 General structure of πDDs

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 might arrive at a ZDD-like data structure for representing sets of permutations.

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

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

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

$$P_0 = \{\pi \mid \pi \in P,\ x\pi \neq y\}, \text{ and } P_1 = \{\pi\tau_{(x,y)} \mid \pi \in P,\ x\pi = y\}.$$

Note that $dim(P_1) < dim(P)$ holds since x has not been moved by any of the permutations in P_1. Applying this expansion recursively, we eventually obtain one of the

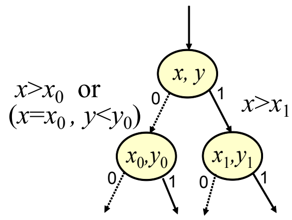

Fig. 5. Variable ordering rules in πDD.

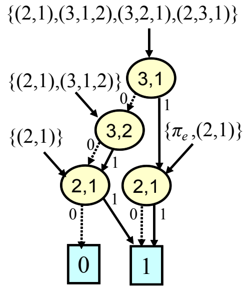

Fig. 6. Multi-rooted shared πDD.

Table 1. Primitive πDD operations.

$\emptyset$	Returns the empty set. (0-termial node)
$\{\pi_e\}$	Returns the singleton set. (1-terminal node)
$P.top$	Returns the IDs (x, y) at the root node of P.
$P \cup Q$	Returns $\{\pi \mid \pi \in P \text{ or } \pi \in Q\}$.
$P \cap Q$	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.

two trivial sets of permutations, namely, the empty set $\emptyset$ (0-terminal node) or the singleton set $\{\pi_e\}$ (1-terminal node).

Similarly to ordinary ZDDs, a fixed order of variables is necessary for all $\tau_{(x,y)}$ in order to preserve the unique representation of the πDD. We use the following order from bottom to top:

$$(2, 1)(3, 2)(3, 1)(4, 3)(4, 2)(4, 1)(5, 4)(5, 3)(5, 2)(5, 1)(6, 5)(6, 4) \ldots$$

Figure 5 shows the rules for variable ordering between 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.

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

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

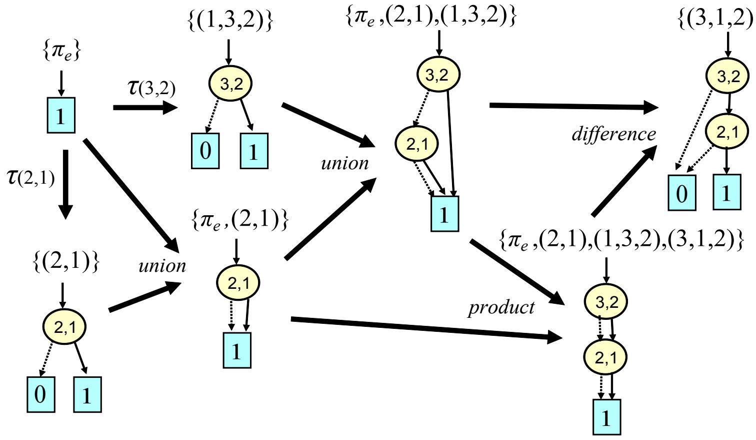

Fig. 7. Construction of πDDs by algebraic operations.

4 Algorithms for Algebraic Operations

In the previous section, we presented the basic structure of πDDs. However, we should consider not only 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 used in πDDs for manipulating sets of permutations. Here, we present a method for computing these operations efficiently. We are aiming at developing an efficient algorithm which computes in linear or small-order polynomial time with respect to the size of the relevant πDD, which is sometimes much smaller than the total number of permutations.

4.1 Binary Set Operations

First we consider the following three binary set operations: union, intersection and difference. As mentioned 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 the τ operation is independent of the union, intersection and difference operations, we can execute those set operations in the same manner as for ordinary BDDs/ZDDs. For example, the 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 duplicate recursive calls by using cache to store previous operations and their results.

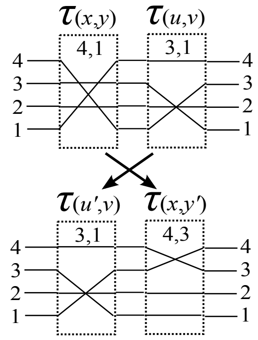

Fig. 8. Swapping of adjacent transpositions.

4.2 Transposition

Next, we consider the transposition operation with any pair of items for a given set of permutations. Let P be a given πDD and $P.top = (x, y)$, after which we compute $P \cdot \tau_{(u,v)}$. If $u > x$, we can simply 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 $u \leq x$, more complex work is needed in order to traverse the internal nodes of P.

To illustrate the algorithm, we recall the example permutation $(3, 5, 2, 1, 4)$ shown in Fig. 3, and we compute $(3, 5, 2, 1, 4)\ \tau_{(3,1)}$. In a πDD, $(3, 5, 2, 1, 4)$ is represented by a sequence of transpositions $\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)}\tau_{(5,4)}$, and 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, two adjacent transpositions are compared, and if the order violates the fixed order of the πDD, then the two transpositions are swapped. For example, $(\tau_{(5,4)}\tau_{(3,1)})$ is transformed 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 the πDD. Care should be taken since some item numbers are slightly altered in this process.

Figure 8 illustrates an example of swapping $\tau_{(x,y)}\tau_{(u,v)}$ with $\tau_{(u',v)}\tau_{(x,y')}$. In this example, u, v, and x are kept while y is changed. Here, we determine that such swapping is always possible for any pair of transpositions, and we also determine the cases in which the items should be changed.

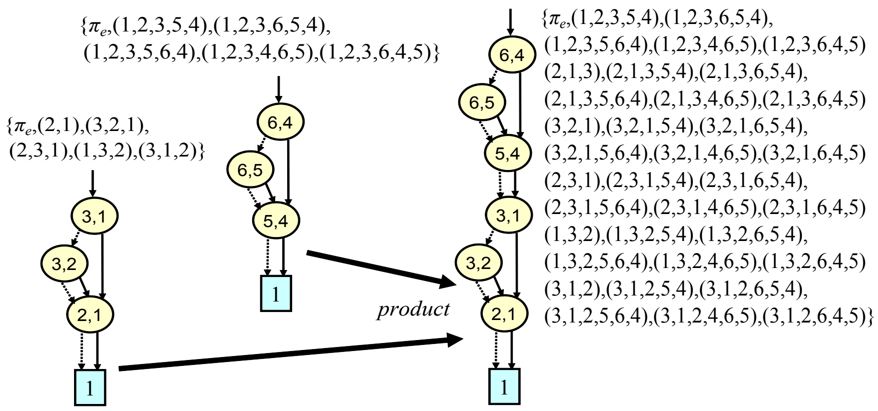

Fig. 9. Example of Cartesian product.

Theorem 2 *For given positive integers x, y, u, v where $x > y > 0$ and $x \geq u > v$, a pair of cascading 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 there are no colliding items for $\tau_{(x,y)}$ and $\tau_{(u,v)}$, they can be swapped transparently. Next, we check all 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, simply $u' = u$ and $y' = y$. □

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

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

This formula shows that we can obtain 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, it should be noted that $dim(P_1 \cdot \tau_{(u',v)})$ must be lower than x. Each sub-operation can be computed by a recursive call, and eventually we arrive at a trivial case. Similarly to other operations, we can avoid duplicate recursions by using operation cache.

4.3 Cartesian Product

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

By 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 we eventually arrive at a trivial operation $P * \emptyset$ or $P * \{\pi_e\}$. As in the case of other operations, we can avoid duplicate recursions by using operation cache. However, one different point here is that we cannot ensure $dim(P * Q_1) < x$, and therefore it is necessary to apply a 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. In this case, even though the number of permutations increases multiplicatively, the size of the πDD increases only additively. Since the computation time also depends on the size of the πDD, in such cases the effectiveness of the πDD-based method increses exponentially as compared to using an explicit data structure.

4.4 Cofactor

After generating a πDD for a set of permutations, it is necessary to extract a subset of permutations in order to check whether a certain property is satisfied. *A cofactor* operation $P.cofact(u, v) = \{\pi\tau_{(u,v)} | \ \pi \in P, \ u\pi = v\}$ generates a subset of permutations such that the item u is moved to v. For example,

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

Note 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 method for executing the cofactor operation. If (u, v) corresponds to $P.top$, we may simply return the 1-edge of the root node. Otherwise, it is necessary 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 executed by using a transposition operation. Due to space limitations, we omit the details regarding the implementation of this operation.

5 Application Examples

Here, we present two application examples and the respective experimental results. We implemented a prototype version of a πDD manipulator based on our own BDD/ZDD package. The program consists of 330 lines of C++ code, newly added to the basic libraries including 6,000 lines of C/C++ code. The following experiments were performed by using a 2.4 GHz Core2Duo PC with 2 GB of RAM, SuSE 10 OS and GNU C++ compiler.

5.1 Design of Permutation Networks

A *permutation network* is an n-input and n-output network which can generate any permutation of the input items. Such circuits are often used in customized hardware

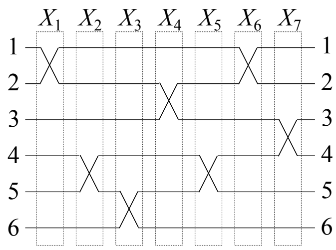

Fig. 10. A permutation network for (4,2,1,6,5,3).

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

P_k	πDD size	# of perm.	total $\#\tau$	P_k	πDD size	# of perm.	total $\#\tau$	P_k	πDD size	# of perm.	total $\#\tau$
P_0	0	1	0	P_{16}	3956	528441	3412177	P_{32}	8655	3497165	24691907
P_1	9	10	9	P_{17}	4685	690778	4522462	P_{33}	7669	3544208	25039740
P_2	31	54	97	P_{18}	5455	878737	5821218	P_{34}	6590	3576891	25279788
P_3	63	209	546	P_{19}	6249	1089826	7296041	P_{35}	5470	3598561	25439624
P_4	109	649	2152	P_{20}	7047	1319957	8915085	P_{36}	4374	3612201	25539440
P_5	172	1717	6704	P_{21}	7834	1563651	10645703	P_{37}	3353	3620296	25598543
P_6	261	4015	17632	P_{22}	8591	1814400	12433871	P_{38}	2444	3624785	25630975
P_7	390	8504	40751	P_{23}	9293	2065149	14239194	P_{39}	1671	3627083	25647411
P_8	558	16599	84985	P_{24}	9905	2308843	15996836	P_{40}	1055	3628151	25654943
P_9	773	30239	162995	P_{25}	10397	2538974	17671711	P_{41}	602	3628591	25657983
P_{10}	1034	51909	291537	P_{26}	10735	2750063	19206325	P_{42}	305	3628746	25659023
P_{11}	1353	84592	491272	P_{27}	**10894**	2938022	20584666	P_{43}	136	3628790	25659303
P_{12}	1727	131635	786100	P_{28}	10857	3100359	21772380	P_{44}	59	**3628799**	25659355
P_{13}	2169	196524	1201963	P_{29}	10614	3236212	22773147	P_{45}	**45**	**3628800**	25659360
P_{14}	2688	282578	1764353	P_{30}	10157	3346222	23579581	P_{46}	**45**	**3628800**	25659360
P_{15}	3286	392588	2495497	P_{31}	9497	3432276	24214975				

of cryptographic systems and signal processing systems. Here, we consider a type of permutation networks using a set of n-bit parallel lines with a number of swapping switches X_k between any two adjacent lines, as shown in Fig. 10. We then consider 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.

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

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

Table 2 shows the experimental results for a 10-bit permutation network. In this table, "πDD size" shows the number of decision nodes in the πDD, "# of perm." indicates the number of permutations included in P_k, and "total $\#\tau$" is the total number of transpositions included in all permutations in P_k. Note 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}, and thus we can see that $m = 45$. In other words, 45 switches are sufficient to cover all 362,880 (=10!) permutations.

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

n	m	πDD size		# of	total	time
		(peak)	(final)	perm.	$\#\tau$	(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

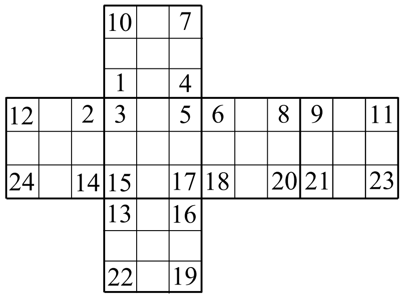

Fig. 11. Assignment of items for the corner cubes of Rubik's Cube.

The number of permutations and the total number of transpositions increase monotonically in this iteration process, however, the size of the πDD reaches a peak of 10,894 at P_{27}, and consequently we require a πDD of only 45 decision nodes to represent all 10! permutations. The latter P_ks might yield more beautiful structures, and the πDD nodes are well shared, even though they include a rather large number of permutations.

We can also observe that P_{45} and P_{44} differ by only a single number of permutations by simply applying the difference set operation ($P_{45} \setminus P_{44}$), and we can confirm that the last permutation is (10,9,8,7,6,5,4,3,2,1). By applying algebraic operations for πDDs to P_ks, we can determine the minimal number of switches for any given permutation, and we can find the layout of the switches which is necessary in order to obtain this permutation.

Table 3 presents the results for n-bit permutation networks for n up to 14. We show the peak and the final size of the πDDs and their respective computation times. The number of all permutations is clearly $n!$, however, the final size of the πDD is only $n(n-1)/2$. Even though the peak size of the πDD grows exponentially, its growth rate appears to be slower than that of $n!$. Here, we can observe that the πDDs are at least 1000 times more compact than explicit representations.

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

5.2 Analysis of Rubik's Cube

Rubik's Cube™is one of the most popular puzzles related to permutation group theory, and πDD can be useful for analyzing it. Here, we focus only on 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 along the X-, Y- and Z-axis as follows.

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

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

$$P_1 = \pi_e + \pi_x + {\pi_x}^2 + {\pi_x}^3 + \pi_y + {\pi_y}^2 + {\pi_y}^3 + \pi_z + {\pi_z}^2 + {\pi_z}^3$$

Now we can generate the set of permutations for up to k moves by using 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 can find a fixed point m such that $P_{k+1} = P_k$ for any $k \geq m$. If we ignore all edge and center cubes, P_m contains all meaningful patterns for the eight corner cubes. Note that the cube $\{19, 20, 21\}$ is fixed to the original position in order 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 the corner cubes is 3,674,160. We confirmed that 11 moves are sufficient to generate all possible patterns, in other words, any pattern of the corner cubes can be returned to the original positions in 11 or fewer moves. As a result, this requires only 511 decision nodes of πDDs for representing all patterns, and P_8 reaches a peak at a πDD size of 608,666. The computation time for generating all πDDs was 207 seconds.

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

$$S_k = P_k.cofact(9,9).cofact(11,11).cofact(15,15)$$
$$.cofact(17,17).cofact(21,21).cofact(23,23)$$

Our experiment shows that, for $k \leq 9$, S_k only includes π_e. For $k = 10$, we discover (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), and by using the maximal number of moves ($k = 11$), we arrive at (6,4,5,2,3,1). After such a pattern is detected, it is not difficult to find a sequence of moves which generates it. We can apply one of the primitive moves to the final pattern in order to obtain a candidate for a preceding pattern, and we check for its 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 we reach P_1.

Although we have considered only the corner cubes, Rokicki et al. [7] recently confirmed that all patterns of the Rubik's cube can be solved as few as 20 moves, and this is the exact minimum. They applied some mathematical pruning and used a network of PCs for massive parallel computation amounting to a total of 35 CPU years. Although the straight-forward application of πDDs to this problem might cause memory overflow, we nevertheless believe that it will be useful for accelerating 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 hints about the application of state-of-the-art SAT techniques used for solving combinatorial problems to permutational problems. There is a rich body of research in group theory led by Galois and many researchers in discrete mathematics [3]. We can expect much future work in this area, for example, developing software tools for studying group theory, considering many other practical applications, implementing various other operations for sets of permutations and considering extended models, such as sets of k-out-of-n permutations or multisets of permutations.

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.