𝕋ℂ𝕊-𝕋ℝ-𝔸-**12-56**

𝕋ℂ𝕊 Technical Report

A πDD-Based Method for Generating Conjugacy Classes of Permutation Groups

by

Norihiro Yamada and Shin-ichi Minato

Division of Computer Science

Report Series A

April 22, 2012

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

A πDD-Based Method for Generating Conjugacy Classes of Permutation Groups

Norihiro Yamada
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

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

April 22, 2012

(Abstract) Conjugacy classes are important notions in group theory and computational group theory, since in our method for example, computing conjugacy classes of the symmetric group S_n of degree $n \in N^+$ enables us to partition a set of permutations according to cycle types; also, they are essential in computing the character table of a group. In this paper, we present the method to compute the conjugacy classes of a permutation group by utilizing πDDs, the efficient data structure for manipulating sets of permutations. The advantages of using πDDs are that they achieve the compact representation of sets of permutations, which is suitable for retaining all class elements, while previous methods have had difficulty in storing all class elements of a vast permutation group because of storage limitations; also they provide convenient operations such as membership test and computing unions or intersections. These strong points propose useful applications; for example, πDDs provide an algorithm to partition a set of permutations according to cycle types, as stated above. The experimental results for S_n $(n \leqslant 12)$ demonstrate the efficiency of πDDs. Thus, we have shown that πDDs are excellent for computing conjugacy classes of some permutation groups, and suggested effective applications, which have been difficult for previous methods.

1 Introduction

Conjugacy classes are important notions in group theory and computational group theory, since in our method for example, computing conjugacy classes of the symmetric group S_n of degree $n \in N^+$ enables us to partition a set of permutations according to cycle types. Also, they are essential in computing the character table of a group. So numerous algorithms to compute them have been developed [6].

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

However, although many algorithms for computing conjugacy classes have been invented, listing or retaining all elements of conjugacy classes is time and/or space consuming, particularly when n gets large. For instance, the Basic Orbit Algorithm [6] takes $O(n^2 \cdot n!)$ products of two permutations to compute all class elements of S_n, and a simple data structure such as an array takes about $(n! \cdot n \cdot \log_2 n)$ bits to store them. Also, there is a method to list only class representatives, which is a less space consuming alternative to listing all class elements; however, it is sometimes inconvenient or less informative. For example, the simplest algorithm for computing conjugacy classes, that of Butler and Cannon [6], takes random elements of the input group until we obtain representatives for all classes; however, it has the main drawback that we have only a small chance to find representatives of small classes, and also, retaining only class representatives is not appropriate for membership test, counting the cardinalities of classes, nor computing unions or intersections.

To solve the above problems, we approach them by utilizing πDDs [4], the efficient data structure for manipulating sets of permutations. That is, in this paper, we present the method to compute the conjugacy classes of a permutation group and applications by using πDDs.

The advantages of using πDDs are that it achieves the compact representation of sets of permutations, suitable for retaining all class elements, while previous methods have had difficulty in storing all class elements of a vast permutation group because of storage limitations. In addition, the *Cartesian product operation* [4] can improve the time complexity of computing class elements since it can avoid multiplying every pair of permutations from each operand set and execute multiple multiplications simultaneously. To evaluate their efficiency, we have computed the conjugacy classes of S_n for $n = 5, 6, ..., 12$ and measured time and memory usage. The experimental results demonstrate the potential capacities in πDDs.

Also, they provide convenient operations such as membership test and computing unions or intersections. These strong points propose useful computations; for example, our algorithm offers a way to partition a set P of permutations according to cycle types, by first constructing a library of conjugacy classes of S_n, and then taking the intersections $\tilde{P}(i) \overset{\text{def.}}{=} P \cap S_n(i)$ for $i = 1, 2, ..., m(S_n)$, where $m(S_n)$ is the number of cycle types of permutations in S_n, the index i is assigned according to the order of construction of classes of S_n, and $S_n(i)$ is the i^{th} conjugacy class of S_n so that $\tilde{P}(i)$ is the set of all permutations in P of the i^{th} cycle type (for details, cf. Section 5.3). Furthermore, constructing a library of conjugacy classes with convenient algebraic operations will be useful in many ways since we can avoid constructing them repeatedly, and the operations provide useful algorithms.

Thus in this paper, we have shown that πDDs are excellent for computing conjugacy classes of some permutation groups, and suggested effective applications, which have been difficult for previous methods [6]. In other words, πDDs are our solution to the above problems, i.e., it is the compact expression suitable for storing all class elements, and it provides convenient operations, suggesting useful applications.

In the rest of this paper, we first present the notion of conjugacy classes, which we aim to compute, together with the basics in group theory in Section 2. And, we introduce the notion of cycle types in Section 3 to explain an application πDDs offer. Then, we describe our data structure in Section 4 and our algorithms in Section 5. Also, we evaluate our method by experimental results in Section 6. Finally, we conclude this paper in Section 7 with discussions and plans for future works.

2 Conjugacy Classes

In this section, we introduce the notion of *conjugacy classes*, which are what we aim to compute, together with basics in group theory.

2.1 Groups

A group is an ordered pair $(G, \star)$ where G is a set, and $\star$ is a binary operation on G satisfying the following axioms:

(1) $(a \star b) \star c = a \star (b \star c)$ for all $a, b, c \in G$;
(2) there exists an element $e \in G$, called an *identity* of G, such that we have $a \star e = e \star a = a$ for all $a \in G$; and
(3) for every $a \in G$, there exists an element a^{-1}, called an *inverse* of a, such that $a \star a^{-1} = a^{-1} \star a = e$.

We often abbreviate $(G, \star)$ to G for convenience, provided the operation $\star$ is obvious from the context. Also for $a, b \in G$, we frequently write ab instead of $a \star b$ if it brings no confusion.

2.2 Subgroups

Let $(G, \star)$ be a group. Then, an ordered pair $(H, \star)$, where $\emptyset \subset H \subseteq G$, and $\star$ is the operation on G restricted to H, is called a *subgroup of* $(G, \star)$ if H is closed under the the operation $\star$, and the pair $(H, \star)$ satisfies all the axioms of a group. If H is a subgroup of G, we shall write $H \leqslant G$.

2.3 Permutation Groups

Let $n \in N^+$, and let $\Delta(n) \overset{\text{def.}}{=} \{1, 2, ..., n\}$. We define S_n as the set of all permutations of $\Delta(n)$, and define $\circ$ as a function composition of two permutations in S_n. Then, all the axioms for a group hold for the pair $(S_n, \circ)$, and it is called the *symmetric group of degree n.* A subgroup of S_n is called a *permutation group of degree n.*

2.4 Group Actions

A *group action* of a group $(G, \star)$ on a set A is a map $\cdot$ from $G \times A$ to A satisfying the following axioms:

(1) $g_1 \cdot (g_2 \cdot a) = (g_1 \star g_2) \cdot a$ for all $g_1, g_2 \in G, a \in A$; and
(2) $e \cdot a = a$ for all $a \in A$, where e is an identity of G.

In this case, we call G *a group acting on a set* A.

2.5 Conjugacy Classes

Let $(G, \star)$ be a group acting on a nonempty set A. Then the binary relation $\sim$ on A defined by:

$$\text{for any } a, b \in A,\ a \sim b \text{ if and only if } a = g \cdot b \text{ for some } g \in G$$

where $\cdot$ is the action of G, forms an equivalence relation on A. Thus, the group action of G on A partitions A, and we obtain the corresponding equivalence classes of A, which is called the *orbits*.

Now, we consider the group action $\cdot$ of G on itself by *conjugation*:

$$b \cdot g \overset{\text{def.}}{=} b \star g \star b^{-1} \text{ for all } b, g \in G$$

where $\star$ denotes the group operation of G. We can easily check that this definition satisfies the axioms for a group action. Thus, this action of G partitions itself into the orbits, which have the special name, the *conjugacy classes* of G.

In this paper, we take a permutation group $(G, \circ)$, where $\circ$ denotes the function composition of two permutations, and compute the conjugacy classes of G. In our approach, we construct the πDDs such that each πDD corresponds to a conjugacy class. And for an experiment, we take $G = S_n$ for $n = 5, 6, ..., 12$ and evaluate our method's efficiency.

3 Cycle Types

There is a notion the *cycle type* of a permutation, which reveals a structual information of a permutation. Also, this notion is closely relevant to combinatorial problems, e.g., for every $n \in N^+$, the number of cycle types of permutations in S_n equals the number of partitions of a positive integer n. In particular for our method, we have an application relevant to cycle types: πDDs provide an algorithm to partition a set of permutations according to cycle types. So in this section, we introduce the definition of cycle types and how we execute the computation (for the formal description of the algorithm, see *Algorithm 2* in Section 5.3). Here, we fix an $n \in N^+$ and consider the permutations in S_n; so the integers appearing in cycles are always the ones in $\Delta(n) = \{1, 2, ..., n\}$.

3.1 Cycles and Cycle Decomposition

In order to define cycle types, we need to first introduce a *cycle*, the notation of the specific permutations, and the *cycle decomposition* of a permutation, the important notation of any permutation. A *cycle* is a string $(a_1\ a_2\ ...\ a_m)$ of integers in $\Delta(n)$ representing a permutation, which cyclically permutes the integers $a_1, a_2, ..., a_m$ with $m \in N^+$. More specifically, it denotes the permutation which maps a_i to a_{i+1} for $i = 1, 2, ..., m-1$, maps a_m to a_1, and fixes other integers in $\Delta(n)$. For example, let $n = 5$, and then, (2 1 3) is the permutation mapping 2 to 1, 1 to 3, 3 to 2, 4 to 4, and 5 to 5. The *length* of a cycle is the number of integers appearing in the cycle; and a cycle of length $t \in N^+$ is called a *t-cycle*. In the above case for example, the length of the cycle $(a_1\ a_2\ ...\ a_m)$ is m, or it is an m-cycle. Two cycles are called *disjoint* if they have no numbers in common. It is clear that disjoint cycles commute. Note that a t-cycle can be written in t-many ways since we have t-many choices for the leftmost number of the cycle; however by convention, the smallest integer in the cycle is usually written at the leftmost position to have the unique notation of cycles.

Next, we introduce the definition of the *cycle decomposition* of a permutation and an important theorem. It is known that for each $g \in S_n$, there exists some $k \in \Delta(n)$, and all the integers in $\Delta(n)$ can be rearranged and grouped into k-many cycles to represent g in the following form:

$$(a_1\ a_2\ ...\ a_{m_1}) \circ (a_{(m_1+1)}\ a_{(m_1+2)}\ ...\ a_{m_2}) \circ ... \circ (a_{(m_{k-1}+1)}\ a_{(m_{k-1}+2)}\ ...\ a_{m_k}) \quad (1)$$

where $\circ$ denotes the function composition of two permutations (cycles), and m_j with $1 \leqslant j \leqslant k$ denotes the number of integers appearing from the 1^{st} to the j^{th} cycles [5]. In other words, every permutation $g \in S_n$ can be written in the form (1), called the *cycle decomposition* of a permutation g. Note that in general, $m_k \leqslant n$ since we omit the integers fixed by g in the above form for convenience; however in this paper, we include all the integers in $\Delta(n)$, and so we always have $m_k = n$. We read the notation (1) of g as follows: for each $x \in \Delta(n)$, we first locate x in the above expression; if x is at the right most position of a cycle (say $x = a_{m_j}$), then it means that g maps x to the integer $a_{(m_{j-1}+1)}$ ($m_0 \overset{\text{def.}}{=} 0$ for $j = 1$) at the leftmost position of the cycle; if x is not at the rightmost position of a cycle, then it means that g maps x to the integer appearing to just the right of x. Note that the cycles in the form (1) are pairwise disjoint, so they commute each other. Thus, the form is independent of the order of constituent cycles; and also, it is independent of the leftmost number of each cycle. In this context, we have the following theorem [5]:

Theorem 1 (Uniquness and Existence of Cycle Decompositions [5]). *For every $g \in S_n$, there exists the unique cycle decomposition of g (up to rearranging its cycles and cyclically permuting the numbers within each cycle).*

Proof. See [5]. ▮

3.2 Cycle Types

Now, we are ready to introduce the notion of *cycle types.* Let $g \in S_n$ such that the cycle decomposition of g consists of e_1-many 1-cycles, e_2-many 2-cycles, ..., and e_n-many n-cycles, where $e_j \in N$ for $j = 1, 2, ..., n$ (note that in particular, e_n must be 0 or 1). Then, we define the *cycle type* of a permutation g, denoted by $type(g)$, as follows:

$$type(g) \overset{\text{def.}}{=} (e_1, e_2, ..., e_n)$$

We also define

$$\sharp(g) \overset{\text{def.}}{=} \sum_{j=1}^{n} e_j$$

Note that $\sum_{j=1}^{n} j \cdot e_j = n$.

Then, we present an important connection between conjugacy classes and cycle types. Let G be a permutation group of degree n. Then for each conjugacy class of G, all elements in the class have the same cycle type. However, the converse does not hold in general, i.e., a set of all elements in G of a fixed cycle type may consist of the union of multiple conjugacy classes of G. It is also known that in the special case of $G = S_n$, the set of conjugacy classes of G exactly corresponds to the partition of G according to cycle types (for the proof, cf. Proof of Theorem 3 in Appendix). Thus, for a permutation group G in general, just computing conjugacy classes is not sufficient for obtaining the partition of G according to cycle types. Then our approach to compute the partition $\{\tilde{G}(i) | \ i = 1, 2, ..., \tilde{m}(G)\}$ of G, where $\tilde{m}(G)$ is the number of cycle types in G, is as follows: (1) compute the conjugacy classes $S_n(i)$ of S_n for $i = 1, 2, ..., m(S_n)$, where $m(S_n)$ is the number of cycle types in S_n; and (2) take the intersections: $\tilde{G}(i) = G \cap S_n(i)$ for $i = 1, 2, ..., m(S_n)$. Note that $\tilde{m}(G) \leqslant m(S_n)$ in general since $G \subseteq S_n$, which means that we may have $G \cap S_n(i) = \emptyset$ for some i. Additionally, our approach actually can partition any subset P of S_n, not only subgroups of S_n (for the detail, cf. Algorithm 2 in Section 5.3).

4 BDD, ZDD, and πDD

In this section, we briefly introduce our data structure πDDs [4], the fundamental component of our method. In particular, we describe what characteristics of πDDs are the advantages in computing conjugacy classes and their applications. In order to explain them, we need to first introduce a Binary Decision Diagram (BDD) [1] and a zero-suppressed BDD (ZDD) [3] since a πDD is based on them.

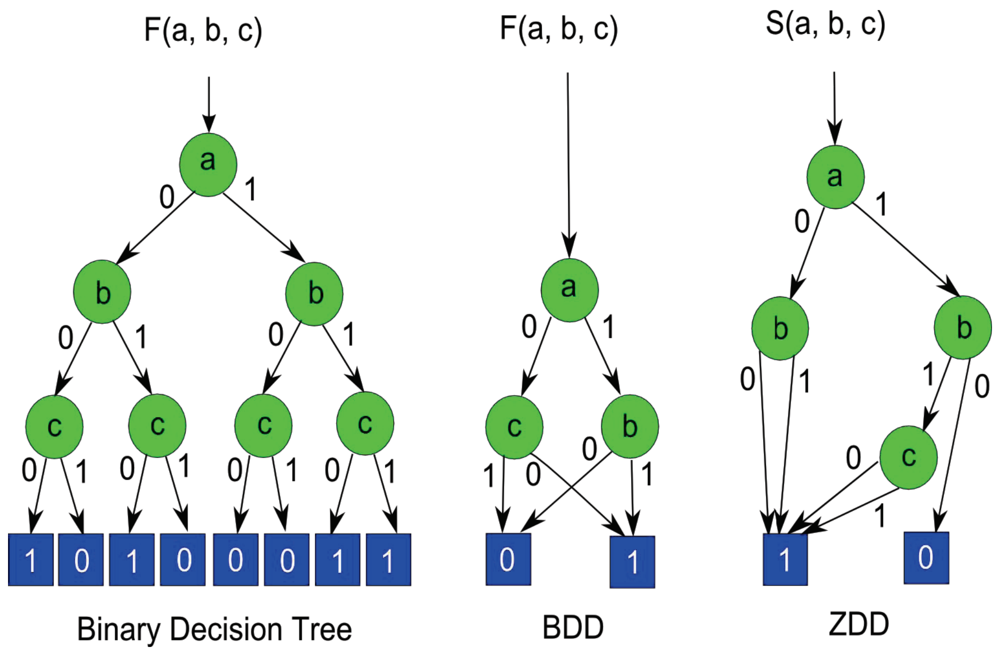

Figure 1: BDD and ZDD

4.1 BDD

A Binary Decision Diagram (BDD) [1] is a compact and canonical graph representation for a Boolean function. As illustrated in Figure 1, it is derived from a binary decision tree for a Boolean function $F(a, b, c)$, which represents a decision making precess through the input variables, by first fixing the order of the input variables and then applying the following reduction rules:

> (1) delete all redundant nodes whose both edges have the same destination (cf. Figure 2); and
> (2) share all equivalent nodes having the same child nodes and the same variables (cf. Figure 2).

It is known that for any Boolean function, the above reduction rules provide the compact and canonical BDD representation. Although the compression ratio achieved by a BDD depends on the Boolean function being represented, it can be between 10 and 100 times in some practical cases. Additionally, 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 (i.e., the number of nodes) of the BDD.

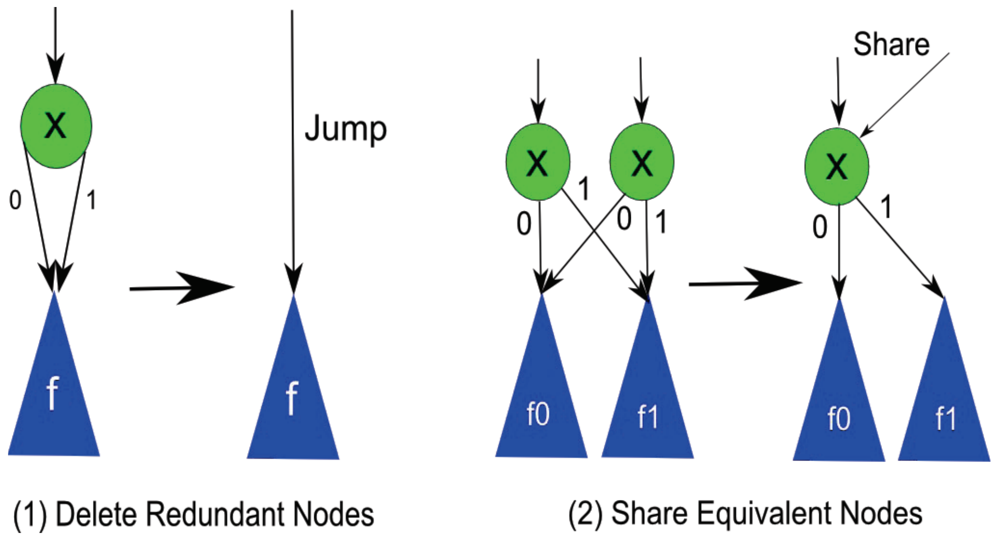

Figure 2: Reduction Rules (1) and (2)

4.2 ZDD

A zero-suppressed BDD (ZDD) [3] is a variant of a BDD customized for manipulating *sets of combinations*. ZDDs are based on special reduction rules suitable for sets of combinations, which differ from ordinay ones for BDDs. Broadly speaking, the semantics of a ZDD representation is that a node variable expresses an item of combinations, and a path from the top to the 1-terminal node corresponds to the combination containing the node items having the outgoing 1-edges in the path; then, the set of all such combinations is the semantics of the ZDD representation (e.g., in Figure 1, $S(a,b,c) = \{\emptyset, \{b\}, \{a,b\}, \{a,b,c\}\}$; also, note that the correspondence between the BDD and ZDD derived from the same binary decision tree, i.e., $F(a,b,c) = 1$ iff $\{x \in \{a,b,c\} | \ x$ has an outgoing 1-edge$\} \in S(a,b,c)$ for each path). To represent a set of combinations efficiently in the similar way to a BDD, we first fix the order of items in combinations (node variables) in a ZDD representation, and then employ the following special reduction rules:

($\acute{1}$) delete all nodes whose 1-edge points directly to the 0-terminal node (cf. Figure 3); and
(2) share all equivalent nodes having the same child nodes and the same variables (cf. Figure 2).

Note that just the rule ($\acute{1}$) differs from the rule (1) for a BDD, while the rule (2) is common for a ZDD and a BDD. Similar to ordinary BDDs, ZDDs give compact and canonical representations for sets of combinations. Also, we can construct ZDDs by applying algebraic set operations such as union, intersection, and difference of two sets, which correspond to logic operations in BDDs.

The zero-suppressing reduction rule ($\acute{1}$) is extremely effective for sets of sparse combinations. If the average apperance rate of each item is 1%, ZDDs are possibly

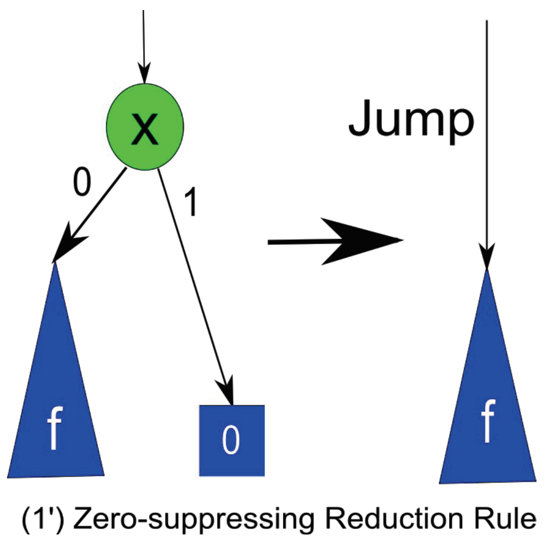

Figure 3: (1') Zero-suppressing Reduction Rule

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 is usually much smaller than the number of all items displayed at the super market. ZDDs are now widely recognized as the most important variant of BDDs (for details, see Knuth's book fascicle [2]).

4.3 πDD

We first fix the degree $n \in N^+$ of permutation groups. A πDD is a variant of a BDD for representing *a set of permutations* compactly and canonically. It has been developed in the similar way to a ZDD. As illustrated in Figure 4, each node of a πDD corresponds to an ordered pair (x, y) with $x, y \in \Delta(n) = \{1, 2, ..., n\}$, which represents a transposition $\tau_{(x,y)} = (x\ y)$ mapping x to y and y to x. Roughly speaking, the semantics of a graph representation of a πDD is the following: each path from the top node to the 1-terminal node represents a permutation in the set, which is obtained by multiplying the transpositions of the nodes in the path having the outgoing 1-edges in the path, from the bottom to the top (e.g., the πDD in Figure 4 represents the set $\{(1\ 2)(3), (1\ 3\ 2), (1\ 3)(2), (1\ 2\ 3)\}$). To represent a set of permutations efficiently, in the similar way to a ZDD, we first fix the order of transpositions (node variables) in a πDD representation, and then employ the ZDD's reduction rules. In particular for a πDD, the rule $(\acute{1})$ means that the numbers in $\Delta(n)$ which are fixed by every permutation in the set being represented will never appear in the πDD expression, which contributes to the compactness of a πDD. In fact, it is about 1000 times more compact than explicit representations for some cases [4].

Additionally, we can systematically construct πDDs as the result of algebraic

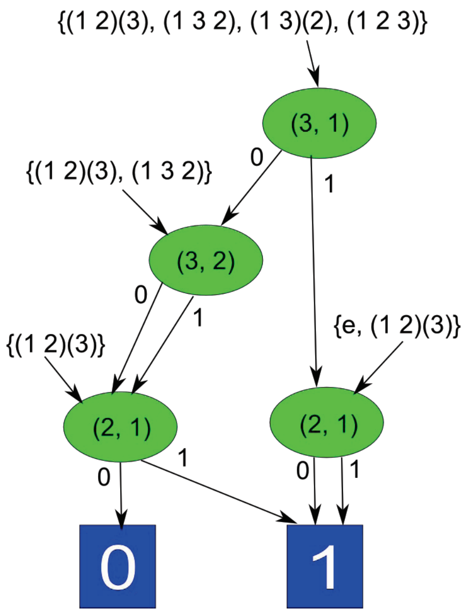

Figure 4: An Example of a πDD

set operations such as computing a union, intersection, and difference in the same way as ZDDs. Moreover, a πDD has a powerful Cartesian product operation which generates all possible composite permutations for two sets of permutations (i.e., for any $P_1, P_2 \subseteq S_n$, $P_1 \times P_2 \overset{\text{def.}}{=} \{p_1 \circ p_2 |\ p_1 \in P_1,\ p_2 \in P_2\}$, where $\times$ denotes the Cartesian product; from here, we abbreviate $P_1 \times P_2$ to $P_1 P_2$). Since the time complexity of these operations depend on the size (i.e., the number of nodes) of πDDs, not the number of permutations in the sets, πDDs sometimes achieve very fast computation time. Once we have generated πDDs, we can apply various analysis or testing techniques, such as counting the exact number of permutations in the set, exploring the satisfiable permutations for a given constraint, and calculating the minimal or the average cost of all permutations.

Our basic approach for computing conjugacy classes efficiently is to utilize the Cartesian product operation to compute multiple products simultaneously, and compactly retain all class elements by πDD representations. Also, the algebraic set operations provide the effective applications of the constructed conjugacy classes, which is another advantage in πDDs.

5 Our Method

So far, we have described what to compute together with relevant definitions and notions. In this section, we explain how to execute the computations.

5.1 The Subgroup Generated by a Subset

Before describing our algorithms, we need to introduce the notion of the *subgroup generated by a subset* since it is frequently used for the inputs of the algorithms. Let $(G, \star)$ be a group, and let A be any subset of G. Then, we difine:

$$< A > \overset{\text{def.}}{=} \{{a_1}^{e_1}{a_2}^{e_2}...\ {a_n}^{e_n} \mid n \in N, a_i \in A, e_i \in \{+1, -1\} \text{ for each } i\}$$

where an identity $e \in G$ is included in $< A >$ as the element corresponding to $n = 0$, and $< A > \overset{\text{def.}}{=} e$ if $A = \emptyset$. We can easily check that $< A >$ is closed under $\star$, and it satisfies all the axioms of a group. So it is a subgroup of G. We call $< A >$ the *subgroup of* G *generated by* A.

Roughly speaking, for a group G, the subset A of G such that $< A > = G$ is considered to be a compressed set of G or a set of base components of G since every element of G can be restored as a product of elements of A and their inverses. Moreover, this *compression* of G is suitable for our algorithm (in particular for the Cartesian product operation) since it suffices to compute the action of conjugation by each element in A, instead of each element in G (i.e., for the conjugacy class $CC(g)$ of G containing g, we iterate $CC(g) \leftarrow (CC(g) \cup (\bigcup_{a \in A} \{a\}CC(g)\{a^{-1}\}))$ with the initial value $CC(g) \leftarrow \{g\}$ until $CC(g)$ does not increase, instead of $CC(g) \leftarrow \bigcup_{\hat{g} \in G} \{\hat{g}g\hat{g}^{-1}\}$).

Note that for a group G and the subset $A \subseteq G$ such that $< A > = G$, $|A|$ tends to be much smaller than $|G|$ (many groups can be generated by two elements, and in most cases, we have $|A| \leqslant 10$; moreover, it is theoretically proved that for any permutation group G of degree n, we have $|A| \leqslant n/2$ [6]). Thus, dealing with A is a fundamental point for our computation since $|A|$ tends to be much smaller than $|G|$, and if we have to compute the action by each element of G (i.e., $CC(g) \leftarrow (CC(g) \cup (\bigcup_{\hat{g} \in G} \{\hat{g}\}CC(g)\{\hat{g}^{-1}\})))$, the number of the Cartesian product operation will equal the number of simple multiplication of permutations, where our method has no advantage. In this paper, we always assume that every permutation group G is given together with its subset A such that $< A > = G$.

5.2 Algorithm to Compute Conjugacy Classes

Now, we present our first algorithm to compute conjugacy classes of a permutation group. Let G be a permutation group such that it has $m(G)$-many conjugacy classes $G(1), G(2), ..., G(m(G))$. Then the following *Algorithm 1* computes $G(i)$ for $i = 1, 2, ..., m(G)$, and obtains them as the corresponding πDDs:

Theorem 2 (Correctness of Algorithm 1). *Let a finite permutation group* G *and a subset* $A \subseteq G$ *such that* $< A > = G$ *be inputs for Algorithm 1. Then, the algorithm stops after finitely many steps. Moreover, the corresponding outputs are exactly the conjugacy classes of* G.

Algorithm 1 (Compute Conjugacy Classes)

```
Input: $G$, $A \subseteq G$ such that $< A > = G$
Output: $G(1), G(2), ..., G(m(G))$
Method:
$i \leftarrow 0$
while $G \neq \emptyset$ do
  $i \leftarrow (i+1)$
  Take an arbitrary $\delta_i \in G$
  $G(i) \leftarrow \{\delta_i\}$; $Add \leftarrow \{\delta_i\}$
  while $Add \neq \emptyset$ do
    $Add \leftarrow \{(\bigcup_{a \in A} \{a\}G(i)\{a^{-1}\}) \setminus G(i)\}$
    $G(i) \leftarrow (G(i) \cup Add)$
  end while
  $G \leftarrow (G \setminus G(i))$
end while
```

Proof. See Appendix. ∎

5.3 Algorithm to Partition a Set of Permutations According to Cycle Types

Next, we provide our second algorithm to partition a set of permutations according to cycle types, which is based on Algorithm 1 described above. Let P be a set of permutations of degree $n \in N^+$ (i.e., $P \subseteq S_n$) such that it has $\tilde{m}(P)$ cycle types among its elements. Then, the following *Algorithm 2* computes all the subsets $\tilde{P}(i)$ for $i = 1, 2, ..., \tilde{m}(P)$ of P, each of which consists of permutations in P of a fixed cycle type, and obtains them as the corresponding πDDs:

Algorithm 2 (Partition a Set of Permutations According to Cycle Types)

```
Input: $P$, $S_n$, $B \subseteq S_n$ such that $< B > = S_n$
Output: $\tilde{P}(1), \tilde{P}(2), ..., \tilde{P}(\tilde{m}(P))$
Method:
Execute Algorithm 1 for the input $S_n$ and $B$,
and obtain the output $S_n(i)$ for $i = 1, 2, ..., m(S_n)$
$j \leftarrow 1$
for $i = 1, 2, ..., m(S_n)$ do
  $\tilde{P}(j) \leftarrow P \cap Sn(i)$
  if $\tilde{P}(j) \neq \emptyset$ then
    $j \leftarrow (j+1)$
  end if
end for
```

Theorem 3 (Correctness of Algorithm 2). *Let P, S_n with $n \in N^+$, and $B \subseteq S_n$ such that $< B > = S_n$ be inputs for Algorithm 2. Then, the algorithm stops* *after finitely many steps. Moreover, if we denote the corresponding outputs by $\tilde{P}(1), \tilde{P}(2), ..., \tilde{P}(\tilde{m}(P))$, then, the set $\{\tilde{P}(1), \tilde{P}(2), ..., \tilde{P}(\tilde{m}(P))\}$ is exactly the partition of P according to cycle types, i.e., each $\tilde{P}(j)$, $j \in \{1, 2, ..., \tilde{m}(P)\}$ is the set of all elements in P of a fixed cycle type.*

Proof. See Appendix. ∎

6 Experimental Results

6.1 Setting Up

In order to evaluate our method's efficiency, we have measured time and memory usage in generating S_n, computing the conjugacy classes of S_n, and retaining them as the corresponding πDDs for $n = 5, 6, ..., 12$. The machine we used for the experiment was VT64 Server4600, Opteron6176 2.3GHz, 256MB memory, SuSE 64 Linux.

We have recorded the results in Table 1, where *Final* (kB) is the final memory usage after the computation, and *Peak* (kB) is the maximum memory usage during the computation. For comparison with πDDs, we used an array as a standard data structure for sets of permutations, and recorded its virtual memory usage as *Array* (kB). We assume that for a set P of permutations in S_n, a πDD takes $(30 \cdot$ (the number of its nodes)) bits, while an array takes $(n \cdot \log_2 n \cdot |P|)$ bits.

And, *C-Ratio* (*Compression Ratio*) is defined as the value Array/Final, and *P-Ratio* (*Peak Ratio*) is defined as the value Peak/Final. Finally, we have recorded the computation time as *Time* (s).

6.2 Results and Evaluations

Since simple data structures such as an array are likely to take more than the exponential function of n for both time and memory usage for dealing with S_n, the point is how much πDDs improve such costs. Also, since πDDs tend to expand during computations in some cases [4], we should focus on how much it swells out to see whether or not πDDs are suitable for computing conjugacy classes.

First, see the C-Ratio in Table 1. Except for $n = 5$, πDDs take much less memory usage than an array; in particular, when n gets large ($n \geqslant 9$). For example, it is 97.70 when $n = 10$ (i.e., πDDs are about 100 times more compact than an array). And focusing on *how much the C-Ratio increases as n becomes larger*, we can observe that it gets about $2 \sim 4$ times every time n increases by 1; more detailed observation tells us that it is more than the exponential function of n. To sum up, πDDs are compact representations of conjugacy classes; in particular, they are very efficient when n becomes large.

Table 1: Computing Conjugacy Classes of S_n

S_n	Final	Peak	Array	C-Ratio	P-Ratio	Time
S_5	0.33	0.39	0.17	0.53	1.19	0.03
S_6	1.17	1.33	1.40	1.20	1.14	0.02
S_7	4.07	4.66	12.38	3.04	1.15	0.11
S_8	14.07	15.50	121.00	8.60	1.10	0.64
S_9	46.10	52.37	1294.00	28.08	1.14	4.14
S_{10}	154.20	215.10	15068.00	97.70	1.40	32.84
S_{11}	491.90	1470.00	189873.00	386.00	2.99	616.40
S_{12}	1534.00	11274.00	2575806.00	1679.00	7.35	3082.00

Next, see the P-Ratio. For $n = 5, 6, ..., 10$, it is between 1.10 and 1.40, which is very small since it can be thousands in some cases (cf. [4]). However, it is 2.99 and 7.35 for $n = 11$ and $n = 12$ respectively, which indicates that it will relatively increase as n gets larger ($n \geqslant 10$). Thus, it seems that πDDs do not swell out in computing the conjugacy classes of S_n when n is small, but they tend to expand when n is large.

Finally, we consider the Time. When we see how the values change as n becomes larger, the increasing rate is more than the exponential function of n (say, more than $1.28 \cdot 10^{-6} \cdot 5^n$). So it seems that the computation time will not be feasible when n is very large; for example, it took more than 14 hours when $n = 13$.

6.3 Consideration to the P-Ratio

Here, we consider a particularly interesting problem: why the P-Ratio tends to be very small (between 1.10 and 1.40) for $n = 5, 6, ..., 10$. In other words, why a πDD does not expand in computing the conjugacy classes of S_n at least for small n. Then, more detailed experimental results (cf. Table 2 which records the detailed process of the computation for S_9, which has 30-many conjugacy classes $S_9(i)$ for $i = 1, 2, ..., 30$; *Card* $\overset{\text{def.}}{=} |S_9(i)|$, *Peak* $\overset{\text{def.}}{=}$ the maximum memory usage, and *Final* $\overset{\text{def.}}{=}$ the resulting memory usage in computing $S_9(i)$ for $i = 1, 2, ..., 30$) indicate an answer: seeing the memory usage in computations of each class , the conjugacy classes constructed at the beginning of the computation (say, $S_9(1), S_9(2), ...,$ and $S_9(7)$) tend to expand (i.e., $P/F \overset{\text{def.}}{=}$ Peak/Final is relatively large) in the construction process, while the classes produced in the latter half of the computation (say, $S_9(16), S_9(17), ...,$ and $S_9(30)$) do not expand (i.e., P/F is relatively small); however in the first half of the process, the total memory usage tends to be relatively small since there remain many unprocessed permutations kept as a single πDD, which is compact since the nodes sharings are likely to occcur, while the memory usage in the latter half is relatively large since many πDDs corresponding to the conjugacy classes have been constructed, and so the nodes sharings are less frequent. As a result, the peak time (i.e., the moment when the entire memory usage gets the

Table 2: Detailed Record of Computing S_9

Classes	Card	Peak	Final	P/F
$S_9(1)$	40320	32.741	0.570	57.440
$S_9(2)$	45360	34.121	4.654	7.332
$S_9(3)$	25920	30.750	8.220	3.741
$S_9(4)$	25920	29.550	11.085	2.666
$S_9(5)$	20160	29.453	13.316	2.212
$S_9(6)$	30240	40.770	16.917	2.410
$S_9(7)$	10080	25.868	17.981	1.439
$S_9(8)$	18144	31.868	20.531	1.552
$S_9(9)$	24192	39.728	23.704	1.676
$S_9(10)$	9072	33.773	26.145	1.292
$S_9(11)$	18144	41.104	28.241	1.455
$S_9(12)$	3024	30.956	28.350	1.092
$S_9(13)$	11340	39.041	31.328	1.246
$S_9(14)$	15120	41.583	34.223	1.215
$S_9(15)$	15120	43.613	36.255	1.203
$S_9(16)$	11340	44.674	39.251	1.138
$S_9(17)$	7560	44.828	40.170	1.116
$S_9(18)$	756	40.924	39.870	1.026
$S_9(19)$	2240	43.005	41.228	1.043
$S_9(20)$	10080	50.543	43.181	1.170
$S_9(21)$	3360	46.223	43.298	1.068
$S_9(22)$	2520	47.021	44.921	1.047
$S_9(23)$	7560	52.373	45.885	1.141
$S_9(24)$	2520	49.241	46.061	1.069
$S_9(25)$	168	46.328	45.821	1.011
$S_9(26)$	945	46.804	46.204	1.013
$S_9(27)$	1260	47.460	46.163	1.028
$S_9(28)$	378	46.774	46.095	1.015
$S_9(29)$	36	46.200	46.091	1.002
$S_9(30)$	1	46.091	46.088	1.000

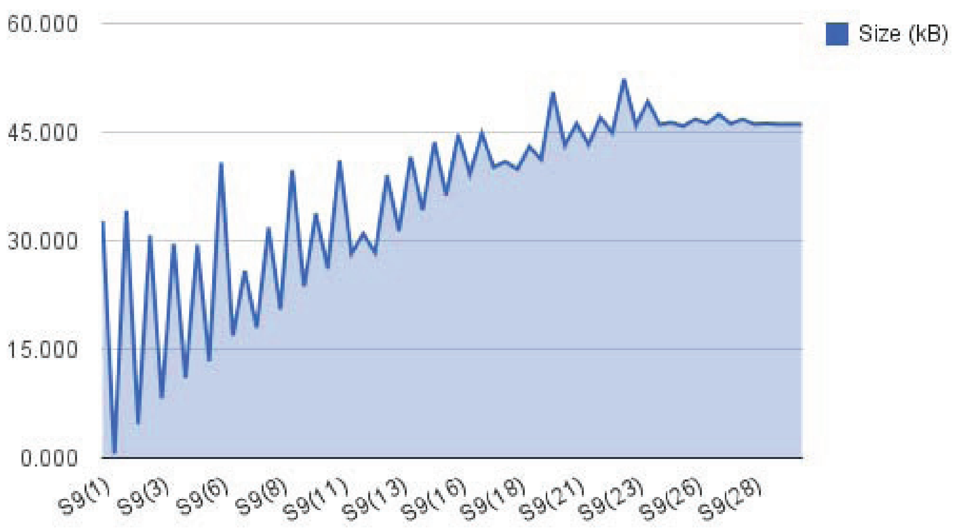

Figure 5: An Illustration of Transition of a πDD Size

largest) will be the one in the latter half of the process, where Peak is not very far away from the resulting value, i.e., the memory usage after computing $S_9(30)$. This seems to be a reasonable answer for the question why πDDs do not expand in computing conjugacy classes. Also, see Figure 5, a graph representation of Table 2, describing the transition of the memory usage throughout the computation.

7 Future Works

In this paper, we have evaluated the efficiency of πDDs for computing the conjugacy classes of S_n for $n = 5, 6, ..., 12$ and suggested some applications. Althoguh our results demonstrate the potential capabilities in πDDs, there remain many things to be examined and achieved. First, we need to evaluate the efficiency of πDDs for computing the conjugacy classes of the permutation groups other than S_n to investigate they are efficient equally for most permutation groups, or only for specific ones; and if they are depending on some properties of permutation groups, we shall clarify such properties, and how they affect πDDs.

Also, it is an interesting problem to clarify the reason why the conjugacy classes constructed at the beginning of the process tend to expand, while the classes produced in the latter half do not swell out; in particular, we would like to know why the conjugacy class $S_n(1)$ constructed first has an extremely large value for P/F, in comparison with other classes. Moreover, we would like to know why the

P-Ratio suddenly starts to increase and gets larger dramatically for $n \geqslant 10$, while it has been static and small for $n \leqslant 9$.

Finally, it is also important to consider the ways to utilize πDDs' advantages, i.e., retaining all class elements by the compact representation and providing convenient algebraic operations. We believe that our approach will offer useful applications; for example, as we have described, it gives an algorithm to partition a set of permutations according to cycle types. In this way, it seems a promising direction to consider how to utilize a *library of conjugacy classes with algebraic operations*.

Acknowledgements

In writing this paper, Dr. Saitoh at JST ERATO Minato Discrete Structure Manipulation System Project precisely advised the author on how to implement the algorithms, how to account for experimental results, and how to organize the paper; also, Professor Zeugmann at Hokkaido University helped to determine the research direction of this paper. Moreover, Professor Hulpke at Colorado State University politely taught me the current situation in the field of computational group theory and directed the author. The author deeply appreciates their kind help.

References

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

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

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

[4] Shin-ichi Minato. *PiDD: A New Decision Diagram for Efficient Problem Solving in Permutation Space.* Proc. of 14th International Conference on Theory and Applications of Satisfiability Testing, pp. 90-104, Jun. 2011.

[5] David S. Dummit, and Richard M. Foote. 2003. *Abstract Algebra Third Edition.* Wiley.

[6] Ákos Seress. 2003. *Permutation Group Algorithms.* Cambridge University Press.

Appendix

Here, we present the proofs of the theorems described in Section 5.

Proof of Theorem 2

Theorem 2 (Correctness of Algorithm 1)**.** *Let a finite permutation group G and a subset $A \subseteq G$ such that $< A > = G$ be inputs for Algorithm 1. Then, the algorithm stops after finitely many steps. Moreover, the corresponding outputs are exactly the conjugacy classes of G.*

Proof. First of all, the algorithm will stop at most after $|G|$ $(< \infty)$ many While loops (at the fifth line), since for each i, we have $G(i) \neq \emptyset$ and execute $G \leftarrow (G \setminus G(i))$ so that we will obtain $G = \emptyset$, and the algorithm will stop after a finitely many steps. And, since we will iterate the While loop until $G = \emptyset$, we have $\bigcup_{i=1}^{m(G)} G(i) = G$, where $m(G)$ is the resulting value of i. Also, since we execute $G \leftarrow (G \setminus G(i))$ for each i, we have $G(i) \cap G(j) = \emptyset$ if $i \neq j$ for any $i, j \in \{1, 2, ..., m(G)\}$. Thus, we have confirmed that the set $\{G(i)|\ i \in \{1, 2, ..., m(G)\}\}$ is a partition of G. Now, we denote the conjugacy class of G containing the initial element $\delta_i \in G(i)$ by $CC(i)$; and so it suffices to show that $G(i) = CC(i)$ for $i = 1, 2, ..., m(G)$.

Note that every $g \in G$ can be written as $a_1 a_2 ... a_m$ for some $m \in N^+$, $a_1, a_2, ..., a_m \in A$ since G is a finite group. Moreover, there exists the minimum $m(g) \in N^+$ to write g in the form of $g = a_1 a_2 ... a_{m(g)}$. Furthermore, for any $g_1, g_2 \in G$, there exists some $\hat{g} \in G$ such that $g_1 = \hat{g} g_2 \hat{g}^{-1}$ with $m(\hat{g}) \leqslant m(g)$ for all $g \in G$ such that $g_1 = g g_2 g^{-1}$. We define the number $N(g_1, g_2, A) \overset{\text{def.}}{=} m(\hat{g})$, which is called the *least conjugation number of g_2 for g_1 in the presence of A.*

$(i = 1)$ We first show the theorem for $i = 1$. We start with showing $G(1) \subseteq CC(1)$. In Algorithm 1, we first take an arbitrary $\delta_1 \in G$ as the initial element of $G(1)$. Then, it is clear that every $x \in G$ newly added to *Add* must be in the form of $x = (a_m ... (a_2(a_1 \delta_1 a_1^{-1}) a_2^{-1}) ... a_m^{-1}) = (a_m ... a_2 a_1) \delta_1 (a_1^{-1} a_2^{-1} ... a_m^{-1})$ with $m \in N$, $a_1, a_2, ..., a_m \in A \subseteq G$ (note that $m = 0$ iff $x = \delta_1$). And since $a_1^{-1} a_2^{-1} ... a_m^{-1} = (a_m ... a_2 a_1)^{-1}$, we consequently obtain $x = g \delta_1 g^{-1}$, where $g = a_m ... a_2 a_1 \in G$. Thus, we conclude that $x \in CC(1)$ since $CC(1) = \{\tilde{g} \delta_1 \tilde{g}^{-1}|\ \tilde{g} \in G\}$, showing $G(1) \subseteq CC(1)$.

It remains to show $CC(1) \subseteq G(1)$. Let $x \in CC(1)$. Then, x has the form of $x = g \delta_1 g^{-1}$ for some $g \in G$. Moreover, g can be written as $g = a_1 a_2 ... a_k$ for $a_1, a_2, ..., a_k \in A$ with $k = N(x, \delta_1, A)$. Thus, $x = (a_1 a_2 ... a_k) \delta_1 (a_1 a_2 ... a_k)^{-1} = (a_1 a_2 ... a_k) \delta_1 (a_k^{-1} ... a_2^{-1} a_1^{-1}) = (a_1 (a_2 ... (a_k \delta_1 a_k^{-1}) ... a_2^{-1}) a_1^{-1})$. Now, we focus on the iteration of the following computations in the algorithm:

$$Add \leftarrow \{(\bigcup_{a \in A} \{a\}G(i)\{a^{-1}\}) \setminus G(i)\} \tag{2}$$

$$G(i) \leftarrow (G(i) \cup Add) \tag{3}$$

Note that after k-many iterations of the computations (2) and (3), $G(1)$ contains all $\tilde{x} \in CC(1)$ such that $N(\tilde{x}, \delta_1, A) \leqslant k$, including x. Thus, it suffices to show that the algorithm must execute the above computations at least k-many times. Then suppose the converse, i.e., suppose that the algorithm stops after a l-many iterations of the above computations with $l < k$. By the iteration condition *While* $Add \neq \emptyset$ *do* (at the ninth line), this means that at the l^{th} iteration, conjugating any already obtained element of $G(1)$ by any element of A must be some already obtained element; in particular, we must have $(a_l...(a_2(a_1\delta a_1^{-1})a_2^{-1})...a_l^{-1}) = (\hat{a}_{\hat{m}}...(\hat{a}_2(\hat{a}_1\delta_1\hat{a}_1^{-1})\hat{a}_2^{-1})...\hat{a}_{\hat{m}}^{-1})$ for some $\hat{a}_1, \hat{a}_2, ..., \hat{a}_{\hat{m}} \in A$ with $\hat{m} < l$. Consequently, x can be written as:

$$\begin{aligned} x &= a_k...a_{l+1}a_l...a_2a_1\delta_1a_1^{-1}a_2^{-1}...a_l^{-1}a_{l+1}^{-1}...a_k^{-1} \\ &= a_k...a_{l+1}\hat{a}_{\hat{m}}...\hat{a}_2\hat{a}_1\delta_1\hat{a}_1^{-1}\hat{a}_2^{-1}...\hat{a}_{\hat{m}}^{-1}a_{l+1}^{-1}...a_k^{-1} \end{aligned}$$

This implies that $N(x, \delta_1, A) = k - l + \hat{m} = k - (l - \hat{m}) < k = N(x, \delta_1, A)$, a contradiction. Thus, we have completed the proof for $i = 1$.

($i > 1$) Next, we prove the theorem for $i > 1$. Let $i > 1$, and suppose that $G(1), G(2), ..., G(i-1)$ have been already constructed by the algorithm, and each of them corresponds to a distinct conjugacy class of G. Then, the algorithm takes an arbitrary δ_i from the remaining (unpartitioned) elements of G, i.e., $\delta_i \in G \setminus (\bigcup_{j=1}^{i-1} G(j))$, and so it must be a representative of one of the conjugacy classes which have not been constructed yet, which we denote by $CC(i)$. By induction on i, it suffices to prove that $G(i) = CC(i)$, which can be shown in the same way as the proof for $i = 1$. Therefore, we have completed the entire proof. ∎

Proof of Theorem 3

Theorem 3 (Correctness of Algorithm 2). *Let P, S_n with $n \in N^+$, and $B \subseteq S_n$ such that $< B > = S_n$ be inputs for Algorithm 2. Then, the algorithm stops after finitely many steps. Moreover, if we denote the corresponding outputs by $\tilde{P}(1), \tilde{P}(2), ..., \tilde{P}(\tilde{m}(P))$, then, the set $\{\tilde{P}(1), \tilde{P}(2), ..., \tilde{P}(\tilde{m}(P))\}$ is exactly the partition of P according to cycle types, i.e., each $\tilde{P}(j)$, $j \in \{1, 2, ..., \tilde{m}(P)\}$ is the set of all elements in P of a fixed cycle type.*

Proof. In this proof, we explicitly write the binary operation $\circ$, which is the composite function operation of two permutations. First, it is clear that the algorithm must stop after $m(S_n)$, a finitely many ($m(S_n) \leqslant |S_n| = n! < \infty$) loops.

Next, we shall show that the set $\{S_n(1), S_n(2), ..., S_n(m(S_n))\}$ is the partition of S_n according to cycle types since it directly proves the theorem as the following claim states:

Claim: Let the set $\{S_n(1), S_n(2), ..., S_n(m(S_n))\}$ be the partition of S_n according to cycle types. Moreover, let the cycle type corresponding to $S_n(i)$ be called the i^{th} *cycle type* for $i = 1, 2, ..., m(S_n)$. Then, for $i = 1, 2, ..., m(S_n)$, we have:

$$P \cap S_n(i) = \{p \in P | \text{ p is a permutation of the } i^{th} \text{ cycle type}\}$$

Also, we have the following *if and only if* condition:

> P contains a permutation of the i^{th} cycle type (i.e., $P \cap S_n(i) \neq \emptyset$) if and only if the algorithm constructs the set $\tilde{P}(l) \overset{\text{def.}}{=} P \cap S_n(i)$, where l is the least positive integer such that $\tilde{P}(l)$ has not been constructed yet.

Therefore, we conclude that:

$$\{\tilde{P}(1), \tilde{P}(2), ..., \tilde{P}(\tilde{m}(P))\} = \{P \cap S_n(i) | \, i \in \{1, 2, ..., m(S_n)\}, P \cap S_n(i) \neq \emptyset\} \quad (4)$$

where $\tilde{m}(P)$ denotes the resulting value of j, which also equals the number of cycle types in P.

Proof of the Claim. First of all, by the definition of $S_n(i)$, we directly obtain $P \cap S_n(i) = \{p \in P | \text{ p is a permutation of the } i^{th} \text{ cycle type}\}$ for $i = 1, 2, ..., m(S_n)$.

Next, we show that the *if and only if* condition holds. We first consider the loop for $i = 1$. In the loop, if $P \cap S_n(1) = \emptyset$, then, we will just go to $i = 2$ and keep $j = 1$; if $P \cap S_n(1) \neq \emptyset$, then, we construct $P(1) \leftarrow (P \cap S_n(1))$, and we will go to $i = j = 2$. Therefore the condition holds for $i = 1$.

Next, we consider the *if and only if* condition for $i > 1$. Suppose that we have executed the algorithm for $i = 1, 2, ..., (k-1)$ with $k \in N^+$ such that $k > 1$, and $j = l$, where l is the least positive integer such that $\tilde{P}(l)$ has not been constructed yet (i.e., the algorithm has constructed the sets $\tilde{P}(1), \tilde{P}(2), ..., \tilde{P}(l-1)$). Then for $i = k$, if $P \cap S_n(k) = \emptyset$, then, we will just go to $i = k + 1$ and keep $j = l$; if $P \cap S_n(k) \neq \emptyset$, then, we will construct $P(l) \leftarrow (P \cap S_n(k))$, and we will go to $i = (k+1)$, $j = (l+1)$. Thus, the condition holds for k; and by induction, it holds for $i = 1, 2, ..., m(S_n)$.

Finally by what we have shown above (i.e., the *if and only if* condition), we directly obtain the equation (4). And, the condition also implies that the number $\tilde{m}(P)$ of the constructed subsets of P equals the number of cycle types in P. ▮

Therefore, by this claim and Theorem 2, it suffices to prove that every conjugacy class of S_n exactly corresponds to a set of elements in S_n of a fixed cycle type, i.e., for any $x, y \in S_n$,

$y = \tau \circ x \circ \tau^{-1}$ for some $\tau \in S_n$ if and only if x and y share the same cycle type.

Necessity. We first show the necessity. Let $x, y \in S_n$ such that $y = \tau \circ x \circ \tau^{-1}$ for some $\tau \in S_n$. Then by Theorem 1, we can assume that x has the unique cycle decomposition:

$$x = \sigma_1 \circ \sigma_2 \circ ... \circ \sigma_k$$

with some $k \in N^+$, where σ_i is an m_i-cycle with $m_i \in N^+$ for $i = 1, 2, ..., k$. Now, we need the following lemma:

Lemma 1 (Conjugation in S_n). *Let $x \in S_n$. Then by Theorem 1, x has the unique cycle decomposition:*

$$x = \sigma_1 \circ \sigma_2 \circ ... \circ \sigma_k \tag{5}$$

with some $k \in N^+$, where σ_i is an m_i-cycle with $m_i \in N^+$ for $i = 1, 2, ..., k$. Then for any $\tau \in S_n$, the conjugate $\tau \circ x \circ \tau^{-1}$ of x has the following cycle decomposition:

$$\tau \circ x \circ \tau^{-1} = (\tau \circ \sigma_1 \circ \tau^{-1}) \circ (\tau \circ \sigma_2 \circ \tau^{-1}) \circ ... \circ (\tau \circ \sigma_k \circ \tau^{-1})$$

where $\tau \circ \sigma_i \circ \tau^{-1}$ is an m_i-cycle for $i = 1, 2, ..., k$. Moreover, if we write σ_i for $i = 1, 2, ..., k$ as: $\sigma_i = (a_1^i \ a_2^i \ ... \ a_{m_i}^i)$, where $a_j^i \in \Delta(n) = \{1, 2, ..., n\}$ for $j = 1, 2, ..., m_i$, then, we have:

$$\tau \circ \sigma_i \circ \tau^{-1} = (\tau(a_1^i) \ \tau(a_2^i) \ ... \ \tau(a_{m_i}^i))$$

Also, note that the equation (5) is now written as:

$$x = (a_1^1 \ a_2^1 \ ... \ a_{m_1}^1) \circ (a_1^2 \ a_2^2 \ ... \ a_{m_2}^2) \circ ... \circ (a_1^k \ a_2^k \ ... \ a_{m_k}^k) \tag{6}$$

Consequently, we obtain the following equation:

$$\tau \circ \sigma \circ \tau^{-1} = (\tau(a_1^1) \ \tau(a_2^1) \ ... \ \tau(a_{m_1}^1)) \circ ... \circ (\tau(a_1^k) \ \tau(a_2^k) \ ... \ \tau(a_{m_k}^k))$$

i.e., the cycle decomposition of $\tau \circ \sigma \circ \tau^{-1}$ is obtained by replacing each entry $a \in \Delta(n)$ of the cycle decomposition (6) of x by $\tau(a)$.

Proof of the Lemma. First of all, for each σ_i, observe that $\sigma_i(a_{m_i}^i) = a_1^i$ and $\sigma_i(a_j^i) = a_{j+1}^i$ for $j = 1, 2, ..., m_i - 1$. Also, observe the following:

$$\begin{aligned}(\tau \circ \sigma_i \circ \tau^{-1})(\tau(a_j^i)) &= \tau(\sigma_i(\tau^{-1}(\tau(a_j^i)))) \\ &= \tau(\sigma_i((\tau^{-1} \circ \tau)(a_j^i))) \\ &= \tau(\sigma_i(a_j^i))\end{aligned}$$

Consequently, $(\tau\circ\sigma_i\circ\tau^{-1})(\tau(a^i_j)) = \tau(a^i_1)$ if $j = m_i$; $(\tau\circ\sigma_i\circ\tau^{-1})(\tau(a^i_j)) = \tau(a^i_{j+1})$ otherwise. And for all $a \in \Delta(n)$ such that $\sigma_i(a) = a$ (i.e., a does not exist in the cycle σ_i), we have:

$$\begin{aligned}(\tau \circ \sigma_i \circ \tau^{-1})(\tau(a)) &= \tau(\sigma_i(\tau^{-1}(\tau(a)))) \\ &= \tau(\sigma_i((\tau^{-1} \circ \tau)(a))) \\ &= \tau(\sigma_i(a)) \\ &= \tau(a)\end{aligned}$$

Since $\{\tau(\hat{a})|\ \hat{a} \in \Delta(n)\} = \Delta(n)$, we have considered every element in the domain $\Delta(n)$ of $\tau \circ \sigma_i \circ \tau^{-1}$; and so, we conclude that $\tau \circ \sigma_i \circ \tau^{-1}$ can be written as the following m_i-cycle:

$$\tau \circ \sigma_i \circ \tau^{-1} = (\tau(a^i_1)\ \tau(a^i_2)\ \ldots\ \tau(a^i_{m_i})) \tag{7}$$

Note that $\tau\circ\sigma_1\circ\tau^{-1}, \tau\circ\sigma_2\circ\tau^{-1}, \ldots$, and $\tau\circ\sigma_k\circ\tau^{-1}$ are pairwise disjoint cycles since $\sigma_1, \sigma_2, \ldots$, and σ_k are pairwise disjoint cycles and τ is a permutation (injective). (That is, for any $i_1, i_2 \in \{1, 2, \ldots, k\}$, $a^{i_1}_{j_1} \in \sigma_{i_1}$, $a^{i_2}_{j_2} \in \sigma_{i_2}$, we have $\tau(a^{i_1}_{j_1}) \neq \tau(a^{i_2}_{j_2})$ if $i_1 \neq i_2$, since $i_1 \neq i_2 \Rightarrow a^{i_1}_{j_1} \neq a^{i_2}_{j_2} \Rightarrow \tau(a^{i_1}_{j_1}) \neq \tau(a^{i_2}_{j_2})$.)

Finally, we consider the following product:

$$(\tau \circ \sigma_1 \circ \tau^{-1}) \circ (\tau \circ \sigma_2 \circ \tau^{-1}) \circ \ldots \circ (\tau \circ \sigma_k \circ \tau^{-1}) \tag{8}$$

Seeing the form (7), the product (8) can be constructed by first arranging every integer in $\Delta(n)$ as:

$$\tau(a^1_1), \tau(a^1_2), \ldots, \tau(a^1_{m_1}), \tau(a^2_1), \tau(a^2_2), \ldots, \tau(a^2_{m_2}), \ldots, \tau(a^k_1), \tau(a^k_2), \ldots, \tau(a^k_{m_k})$$

which forms an arrangement (a permutation) of all integers in $\Delta(n)$; and then from the left, grouping m_1-many integers corresponding to $\tau \circ \sigma_1 \circ \tau^{-1}$, grouping next m_2-many integers corresponding to $\tau \circ \sigma_2 \circ \tau^{-1}$, ..., and finally grouping m_k-many integers corresponding to $\tau\circ\sigma_k\circ\tau^{-1}$. In other words, the product (8) is generated by rearranging all integers in $\Delta(n)$ and grouping it into k-many cycles, so that it is a legal syntax of a cycle decomposition of a permutation in S_n. Next, we see the semantics of (8) as follows:

$$\begin{aligned}&(\tau \circ \sigma_1 \circ \tau^{-1}) \circ (\tau \circ \sigma_2 \circ \tau^{-1}) \circ \ldots \circ (\tau \circ \sigma_k \circ \tau^{-1}) \\ &= \tau \circ \sigma_1 \circ (\tau^{-1} \circ \tau) \circ \sigma_2 \circ (\tau^{-1} \circ \tau) \circ \ldots \circ (\tau^{-1} \circ \tau) \circ \sigma_k \circ \tau^{-1} \\ &= \tau \circ (\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_k) \circ \tau^{-1} \\ &= \tau \circ x \circ \tau^{-1}\end{aligned}$$

Thus, the form (8) is a cycle decomposition of $\tau \circ x \circ \tau^{-1}$, and by Theorem 1, it is unique. Finally, together with the equation (7), we have shown that

$$\tau \circ x \circ \tau^{-1} = \prod_{i=1}^{k} (\tau(a^i_1)\ \tau(a^i_2)\ \ldots\ \tau(a^i_{m_i})).$$

∎

By this lemma, we immediately obtain the cycle decomposition of y:

$$\begin{aligned} y &= \tau \circ x \circ \tau^{-1} \\ &= (\tau \circ \sigma_1 \circ \tau^{-1}) \circ (\tau \circ \sigma_2 \circ \tau^{-1}) \circ ... \circ (\tau \circ \sigma_k \circ \tau^{-1}) \end{aligned} \tag{9}$$

where $\tau \circ \sigma_i \circ \tau^{-1}$ is an m_i-cycle for $i = 1, 2, ..., k$. Thus, the cycle decomposition (9) of y has the same number of 1-cycles, 2-cycles, ..., n-cycles as the cycle decomposition (5) of x. Therefore, x and y have the identical cycle type, completing the necessity.

Sufficiency. Next, we show the sufficiency. Let $a, b \in S_n$ such that both have the same cycle type $(e_1, e_2, ..., e_n)$. Then without loss of generality, we write the two cycle decompositions of a and b as follows: from the left, writing e_1-many 1-cycles, e_2-many 2-cycles, ..., e_n-many n-cycles. Now, we obtain:

$$a = \sigma_1 \circ \sigma_2 \circ ... \circ \sigma_r; \tag{10}$$

$$b = \hat{\sigma}_1 \circ \hat{\sigma}_2 \circ ... \circ \hat{\sigma}_r \tag{11}$$

where $r \in N^+$, and both σ_i and $\hat{\sigma}_i$ are m_i-cycles with $m_i \in N^+$ for $i = 1, 2, ..., r$ such that $m_1 \leqslant m_2 \leqslant ... \leqslant m_r$. Moreover, for $i = 1, 2, ..., r$, we write:

$$\sigma_i = (a^i_1 \ a^i_2 \ ... \ a^i_{m_i}); \tag{12}$$

$$\hat{\sigma}_i = (\hat{a}^i_1 \ \hat{a}^i_2 \ ... \ \hat{a}^i_{m_i}) \tag{13}$$

where $a^i_j, \hat{a}^i_j \in \Delta(n)$ for $j = 1, 2, ..., m_i$. In the forms (10) and (11), we have included every integer in $\Delta(n)$ by convension; so together with the equations (12) and (13), the integers in the decompositions are:

$$a^1_1, a^1_2, ..., a^1_{m_1}, a^2_1, a^2_2, ..., a^2_{m_2}, ..., a^r_1, a^r_2, ..., a^r_{m_r};$$
$$\hat{a}^1_1, \hat{a}^1_2, ..., \hat{a}^1_{m_1}, \hat{a}^2_1, \hat{a}^2_2, ..., \hat{a}^2_{m_2}, ..., \hat{a}^r_1, \hat{a}^r_2, ..., \hat{a}^r_{m_r}$$

Note that they are two arrangements (permutations) of all integers in $\Delta(n)$. Then, we define a permutation $\tau \in S_n$ by:

$$\tau(a^i_j) \overset{\text{def.}}{=} \hat{a}^i_j$$

for all $i \in \{1, 2, ..., r\}$, $j \in \{1, 2, ..., m_i\}$. Thus for $i = 1, 2, ..., r$, we obtain:

$$\begin{aligned} \hat{\sigma}_i &= (\hat{a}^i_1 \ \hat{a}^i_2 \ ... \ \hat{a}^i_{m_i}) \\ &= (\tau(a^i_1) \ \tau(a^i_2) \ ... \ \tau(a^i_{m_i})) \\ &= \tau \circ \sigma_i \circ \tau^{-1} \text{ (by Lemma 1)} \end{aligned}$$

Consequenlty, we have:

$$\begin{aligned} b &= \hat{\sigma}_1 \circ \hat{\sigma}_2 \circ ... \circ \hat{\sigma}_r \\ &= (\tau \circ \sigma_1 \circ \tau^{-1}) \circ (\tau \circ \sigma_2 \circ \tau^{-1}) \circ ... \circ (\tau \circ \sigma_r \circ \tau^{-1}) \\ &= \tau \circ \sigma_1 \circ (\tau^{-1} \circ \tau) \circ \sigma_2 \circ (\tau^{-1} \circ \tau) \circ ... \circ (\tau^{-1} \circ \tau) \circ \sigma_r \circ \tau^{-1} \\ &= \tau \circ (\sigma_1 \circ \sigma_2 \circ ... \circ \sigma_r) \circ \tau^{-1} \\ &= \tau \circ a \circ \tau^{-1} \end{aligned}$$

Thus, we have shown that a and b are conjugates, completing the proof. ∎