Group Decision Diagram (GDD): A Compact Representation for Permutations

Takanori Maehara,[1] **Yuma Inoue**[2]
[1]RIKEN AIP, [2]Google Japan
takanori.maehara@riken.jp, yumai@google.com

Abstract

Permutation is a fundamental combinatorial object appeared in various areas in mathematics, computer science, and artificial intelligence. In some applications, a subset of a permutation group must be maintained efficiently. In this study, we develop a new data structure, called *group decision diagram (GDD)*, to maintain a set of permutations. This data structure combines the zero-suppressed binary decision diagram with the computable subgroup chain of the permutation group. The data structure enables efficient operations, such as membership testing, set operations (e.g., union, intersection, and difference), and Cartesian product. Our experiments demonstrate that the data structure is efficient (i.e., 20–300 times faster) than the existing methods when the permutation group is considerably smaller than the symmetric group, or only subsets constructed by a few operations over generators are maintained.

1 Introduction

Background and Motivation A *permutation* is a bijection on a (finite) set. It is a fundamental object in combinatorics (Knuth 1997), and is applied in several areas, such as sorting (Waksman 1968), scheduling (Pruesse and Ruskey 1994), and puzzles (Egner and Püschel 1998; Mulholland 2013). In certain applications, a subset of permutations must be maintained (i.e., stored and manipulated) efficiently.

For example, let us consider a $3 \times 3 \times 3$ Rubik's cube. We want to count the number of configurations that can be generated by at most k moves from the initial configuration. As the configuration of the cube is represented by a permutation of 48 non-center facets, the problem is equivalent to count the number of permutations on $[48] = \{1, \dots, 48\}$, denoted by $\mathrm{Sym}(48)$, that can be generated by at most k-fold compositions of the basic permutations, i.e., by the rotations of the faces. We can enumerate such configurations by the breadth-first search: Let $S^0 = \{e\}$ be the singleton set of the identity permutation, which is associated with the initial configuration. Then, the configurations obtained by at most i moves are recursively computed by

$$\begin{aligned} S^i &= S^{i-1}\{e, \mathrm{F}, \mathrm{B}, \mathrm{R}, \mathrm{L}, \mathrm{D}, \mathrm{U}\} \\ &= \left\{ sg : s \in S^{i-1}, g \in \{e, \mathrm{F}, \mathrm{B}, \mathrm{R}, \mathrm{L}, \mathrm{D}, \mathrm{U}\} \right\}, \end{aligned} \quad (1)$$

where F, B, R, L, D, and U are the basic permutations that rotate the front, back, right, left, down, and up faces, respectively, and sg denotes the composition of s and g.

If we have a data structure that stores a set of permutations compactly, and permits union and multiplication operations efficiently, we can implement the above breadth-first search procedure using the data structure. Minato (Minato 2011) proposed such a data structure called *permutation decision diagram (πDD)*, which represents a permutation by a sequence of transpositions and stores them using the *zero-suppressed binary decision diagram (ZDD)* (Minato 1993). Inoue and Minato (Inoue and Minato 2014) proposed another data structure called *rotation permutation decision diagram (ρDD)*, which uses left-rotations instead of transpositions; see Section 2. These data structures perform well on several problems including the Rubik's cube (Minato 2011), circuit design (Tague et al. 2013), and topological sorting (Matsumoto, Hatano, and Takimoto 2018).

However, in the Rubik's cube problem, using the πDD or ρDD appears redundant. These data structures can represent *any* subset of $\mathrm{Sym}(48)$; here, $\mathrm{Sym}(48)$ has a cardinality of $48! \approx 10^{61}$. On the other hand, the number of possible configurations of the Rubik's cube is $2^{27}3^{14}5^{3}7^{2}11 \approx 10^{19}$, which is significantly smaller than $48!$. This may imply that πDD and ρDD are "too powerful" to represent a subset of the possible configurations of the Rubik's cube; therefore, there can be more "suitable" data structures to maintain the possible configurations of the Rubik's cube.

The above question is generalized in the notion of *computational group theory* (Seress 2003; Holt, Eick, and O'Brien 2005). Let $\mathrm{Sym}(n)$ be the set of all permutations of $[n] = \{1, \dots, n\}$, called the *symmetric group of degree* n. Then, the πDD and ρDD are data structures that can store any subset of $\mathrm{Sym}(n)$. Now, we are interested in subgroup G of $\mathrm{Sym}(n)$, and want to maintain a subset of G efficiently. Our research question is as follows.

> Is there an efficient data structure for maintaining a subset of G that is a subgroup of $\mathrm{Sym}(n)$? Is it beneficial to considering G instead of $\mathrm{Sym}(n)$?

[1]This work is done at JSPS KAKENHI (S) 15H05711 and not related to Google.

Here, we emphasize that a new data structure is needed to handle *subsets*, instead of *subgroups*; otherwise, one can use classical data structures in computational group theory such as the Schreier–Sims table (see Preliminaries).

Our Contribution We answer the above research question *affirmatively* by proposing a new data structure called *group decision diagram (GDD)*. Our contributions are as follows.

- We propose the GDD, which maintains a subset of a (permutation) group G. The GDD is built by a combination of the ZDD and the *computable subgroup chain* of G, which is a standard tool in computational group theory. It includes the πDD, ρDD, and ZDD as special cases (Section 3).
- We show that, as similar to the πDD and ρDD, the GDD admits efficient operations such as the membership testing, set operations (intersection, union, and difference), and Cartesian product, whose complexities only depend on the size of the GDDs, instead of the cardinalities of the subsets (Section 4).
- We conducted experiments to establish that the GDD is more efficient in representing a set of permutations, compared to the πDD and ρDD (Section 5).

2 Preliminaries

This section presents the basics of the computational group theory and the ZDD with its families for permutations.

Computational Group Theory We follow the standard notations of computational group theory (Cameron 1999; Seress 2003; Holt, Eick, and O'Brien 2005).

A *group* $(G, \circ)$ is a set G with an associative binary operator $\circ$ that has an identity $e \in G$ and an inverse g^{-1} for all $g \in G$. We abbreviate $(G, \circ)$ as G, and $g_1 \circ g_2$ as $g_1 g_2$. The cardinality of G is referred to as the *order*. In this paper, we only consider groups of finite orders.

A *subgroup* H of a group G is a subset of G that forms a group in its own right. If H is a subgroup of G, we write $H \leq G$. For a subset $S \subseteq G$, we define $\langle S \rangle$ as the smallest subgroup that contains S. If $\langle S \rangle = H$, then S is referred to as the *generator* of H. For $g \in G$, the set $Hg = \{hg : h \in H\}$ is called the *(right) coset of H in G with respect to g*. The set of all the cosets is denoted by $G/H = \{Hg : g \in G\}$. We can define an equivalence relation, $g \equiv g'$, by $Hg = Hg'$. A set of representatives of this equivalence relation is referred to as a *transversal of G/H*. Note that $|G/H| = |G|/|H|$ by the Lagrange theorem.

For a group G, we consider a chain of subgroups

$$G = G_0 \geq G_1 \geq \cdots \geq G_r = \{e\}. \tag{2}$$

For each $i = 1, \ldots, r$, we fix a transversal T_i of G_{i-1}/G_i. Then, any $g \in G$ is uniquely factorized by

$$g = g_r \cdots g_1, \tag{3}$$

where $g_i \in T_i$ $(i = 1, \ldots, r)$. The chain of subgroups with transversals is *computable* if, for each $g \in G_{i-1}$, we can (efficiently) compute a representative $g_i \in T_i$ such that $G_i g = G_i g_i$. Many algorithms in computational group theory can be implemented if we have a computable subgroup chain (Cooperman and Murray 2005).

Let Ω be a set. A group G *acts on* Ω if mapping $g \colon \Omega \ni \omega \mapsto \omega^g \in \Omega$ is defined for all $g \in G$, such that it is compatible with the group operation, i.e., $\omega^e = \omega$ and $(\omega^{g_1})^{g_2} = \omega^{g_1 g_2}$ for all $\omega \in \Omega$ and $g_1, g_2 \in G$. An element $g \in G$ *stabilizes* $\beta \in \Omega$ if $\beta^g = \beta$. The set of all elements $g \in G$ that stabilizes $\beta_1, \ldots, \beta_k \in \Omega$ forms a subgroup of G, called the *point-wise stabilizer of* $\beta_1, \ldots, \beta_k$, and is denoted by $\mathrm{Stab}_G(\beta_1, \ldots, \beta_k)$.

The *symmetric group* $\mathrm{Sym}(n)$ *of degree* n is the set of all permutations of $[n]$. Each element $g \in \mathrm{Sym}(n)$ is represented as a list of n distinct integers as

$$g = [1^g, 2^g, \ldots, n^g], \tag{4}$$

where k^g denotes an integer that is mapped from k. The composition of permutations is defined by

$$gh = [(1^g)^h, (2^g)^h, \ldots, (n^g)^h]. \tag{5}$$

Note that it is a reverse order of the function composition. For notational convenience, we denote by

$$g = (i_1, i_2, \ldots, i_k) \tag{6}$$

for the permutation such that $i_j^g = i_{j+1}$, for all $j \in [k]$ where $i_{k+1} = i_1$, and otherwise, $i^g = i$.

Example 1. The composition of permutations are defined by the reverse-composition order. For example, $(1, 2, 3) = (2, 3)(1, 2)$ since $k^{(1,2,3)} = (k^{(2,3)})^{(1,2)}$ for $k = 1, 2, 3$. □

A subgroup G of $\mathrm{Sym}(n)$ is referred to as a *permutation group (of degree n)*. A permutation group of degree n naturally acts on set $[n]$ by the induced action of $\mathrm{Sym}(n)$.

For a permutation group G of degree n, there is a standard method to obtain a computable subgroup chain, called the *base and strong generating set (BSGS)*. Let $\{\beta_1, \ldots, \beta_r\} \subseteq [n]$ be a sequence of integers and G_i be the point-wise stabilizer of $\{\beta_1, \ldots, \beta_i\}$. Then, we have a chain of subgroups

$$G = G_0 \geq G_1 \geq \cdots \geq G_r. \tag{7}$$

If $G_r = \{e\}$ then the sequence is called a *base*. A subset $S \subseteq G$ is a *strong generating set* if $\langle G_i \cap S \rangle = G_i$ holds for $i = 0, \ldots, r-1$. The union of the set of representatives of G_{i-1}/G_i for $i = 1, \ldots, r$ forms a strong generating set. The representatives of G_{i-1}/G_i are denoted by $T_i = \{g_{i,\alpha} \in G : \alpha \in [n]\}$ such that $g_{i,\alpha}$ maps β_i to α. Here, $g_{i,\alpha}$ can be empty. For each $g \in G_{i-1}$, we can identify $G_i g = G_i g_{i,\alpha}$ by checking $\beta_i^g = \alpha$. Thus, the chain (7) forms a computable chain.

Example 2. Let $\mathrm{Sym}(n)$ be the permutation group of degree n, which is identified as the permutations of set $\{1, \ldots, n\}$. We consider the following subgroup chain

$$\mathrm{Sym}(n) \gneq \mathrm{Sym}(n-1) \gneq \cdots \gneq \mathrm{Sym}(2) = \{e\}, \tag{8}$$

where $\mathrm{Sym}(i)$ (i.e., permutation of $[i]$) is identified as the subgroup of $\mathrm{Sym}(n)$. Let $k = n - i - 1$. The coset $T_i = \mathrm{Sym}(k)/\mathrm{Sym}(k-1)$ has cardinality $k1$. We can choose

Algorithm 1 Naive Schreier–Sims algorithm.

procedure SCHREIERSIMS(S)
  $Q \leftarrow$ empty queue; $g_\alpha \leftarrow \perp$ for all $\alpha \in [n]$
  find $\beta \in [n]$ such that S does not fix β
  Q.push(β); $g_\beta \leftarrow e$
  while $Q \neq \emptyset$ **do**
    $\alpha \leftarrow Q$.pop()
    for $s \in S$ **do**
      if $g_{\alpha^s} = \perp$ **then**
        $g_{\alpha^s} \leftarrow g_\alpha s$
        Q.push(α^s)
      end if
    end for
  end while
  return $\{g_\alpha s g_{\alpha^s}^{-1} : s \in S, \alpha \in [n]\}$ and β
end procedure

$T_i = \{e, (1,k), (2,k), \ldots, (k-1,k)\}$ as the transversal for each i. Here, each (j,k) maps $\beta_i = k$ to j. This means that any permutation on $[k]$ is uniquely factored into the product of (j,k) for some j and the permutation on $[k-1]$. □

Further, the construction of BSGS needs to be addressed. Suppose that G is specified by generators $S \subseteq \mathrm{Sym}(n)$ as $G = \langle S \rangle$. The *Schreier–Sims algorithm* (Holt, Eick, and O'Brien 2005) constructs a BSGS of G as follows. Let $S_0 = S$ and $G_0 = G$. For $i = 1, 2, \ldots$, we select $\beta_i \in [n]$ such that S_{i-1} does not stabilize β_i; we often select β_i that has the longest orbit. We define $G_i = \mathrm{Stab}_{G_{i-1}}(\beta_i) = \mathrm{Stab}_G(\beta_1, \ldots, \beta_i)$. By performing a breadth-first search (or a depth-first search) on $[n]$, we obtain $g_{i,\alpha} \in G_{i-1}$ such that $\beta_i^{g_{i,\alpha}} = \alpha$ for each $\alpha \in [n]$ (if exists); these forms the representatives of G_{i-1}/G_i. The generators of G_i are obtained by the following lemma.

Theorem 3 (Schreier's Lemma)**.** The set $\{g_\alpha s g_{\alpha^s}^{-1} : s \in S_{i-1}, \alpha \in [n]\}$ generates G_i. □

Thus, by setting $S_i = \{g_\alpha s g_{\alpha^s}^{-1} : s \in S_{i-1}, \alpha \in [n]\}$, we proceed to the next iteration. Here, if $S_i = \{e\}$, we obtained the BSGS. Each iteration of the algorithm is shown in Algorithm 1.

The naive algorithm based on the Schreier's lemma creates exponentially many generators during the process. To reduce the number of generators, several methods, such as Sims's filter, Jerrum's filter, and the incremental Schreier–Sims method have been proposed; see (Cameron 1999; Seress 2003; Holt, Eick, and O'Brien 2005). The elaborate implementations of these procedures can be found in the computational group theory softwares (Bosma, Cannon, and Playoust 1997; GAP 2018; Meurer et al. 2017).

Zero-Suppressed Binary Decision Diagram and Permutation Decision Diagrams The ZDD is a data structure to maintain a family of a finite set $[n]$ (Minato 2011). It is a variant of the *binary decision diagram (BDD)* (Bryant 1992), tailored to represent a combination of a few items.

The ZDD is a directed acyclic graph (DAG) with a single root and two terminals $\top$ and $\perp$. Each non-terminal node is associated with an element $i \in [n]$ and has two outgoing edges, 1-edge and 0-edge. Each outgoing edge points to a terminal or node associated with a larger integer than the source node. Each path from the root to the $\top$ corresponds to a set in the family. Here, a path through the 1-edge of the node associated with i corresponds to a set containing i. To make the diagram compact, we apply the following two reduction rules:

$$\begin{cases} 1. & \text{There is no node that has a 1-edge pointing to } \perp \\ 2. & \text{There are no two nodes that have outgoing edges pointing to the same destination.} \end{cases} \tag{9}$$

As per the reduction rules, any family is uniquely represented by a ZDD.

The πDD (Minato 2011) stores a set of permutations using a ZDD as follows. A *transposition* $(u,v) \in \mathrm{Sym}(n)$ is a permutation that swaps $u \in [n]$ and $v \in [n]$. Since $(u,v) = (v,u)$, we assume that $u < v$ in this notation. Any permutation g is factorized by a sequence of transpositions $g = (u_k, v_k) \cdots (u_1, v_1)$ where $v_i > v_j$ if $i < j$. For example, permutation $g = [3,1,4,2]$ is factorized as $g = (1,2)(2,3)(2,4)$. We define the ordering on the pairs of integers as mentioned above . Then, a permutation is represented by a set of integer pairs; hence, a set of permutations is represented by a family of sets of pairs, which can be efficiently stored by a ZDD. This data structure is called the πDD.

The ρDD (Inoue and Minato 2014) is defined using left-rotations instead of transpositions as follows. A *left-rotation* is a permutation of the form $\rho_{u,v} = (u, u+1, \ldots, v) \in \mathrm{Sym}(n)$, where $u < v$. Any permutation g is factorized by a sequence of consecutive rotations $g = \rho_{u_k,v_k} \cdots \rho_{u_1,v_1}$ where $v_i > v_j$ if $i < j$. For example, permutation $g = [3,1,4,2]$ is factorized as $g = (1,2)(2,3,4) = \rho_{1,2}\rho_{2,4}$. Using this factorization, as same as πDD, we can represent a set of permutations by a family of sets of pairs. This data structure is called the ρDD.

ZDD-type structures permit efficient set operations, such as the intersection, union, difference, and Cartesian product, implemented by recursion with memoization in a unified manner. Such implementations are referred to as *apply operations* (Bryant 1992).

3 Group Decision Diagram

In this section, we introduce a new data structure, the *group decision diagram (GDD)*, for maintaining a subset of a group G. The GDD is based on the following: If we fix a subgroup chain (2) with transversals, any subset $S_{i-1} \subseteq G_{i-1}$ is *uniquely* factorized by

$$S_{i-1} = \bigcup_\alpha S_{i,\alpha} g_{i,\alpha} \tag{10}$$

where $S_{i,\alpha} \subseteq G_i$, and $S_{i,\alpha} g_{i,\alpha} = \{s g_{i,\alpha} : s \in S_{i,\alpha}\}$. This immediately follows from the factorization of an element (3). The union over α in (10) is further factorized as

$$S_{i-1} = S'_{i-1} \cup S''_i g_{i,\alpha}. \tag{11}$$

Here, S'_{i-1} is a subset of G_{i-1} that contains the transversal element $g_{i,\alpha'}$ for some α', and S''_i is a subset of G_i. The GDD applies factorization (11) recursively, and represents the structure by a DAG as in the ZDD-type structures.

Formally, the GDD is defined as follows. We fix a computable subgroup chain with transversals, and total orders on the transversals (we assume that these are provided or precomputed from the input). The GDD is a DAG $\mathcal{D} = (V, E)$ that has a unique root and two special terminals $\top$ and $\bot$. Each non-terminal node $u \in V \setminus \{\top, \bot\}$ is associated with an element $g_{i,\alpha} \in T_i$ in the transversal of G_{i-1}/G_i, and has two outgoing edges, 1-edge and 0-edge. Each 1-edge points a terminal or node associated with $g_{j,\alpha} \in T_j$ for $j > i$ (corresponds to the second term in the right-hand side of (11)), and each 0-edge points a terminal or node associated with $g_{j,\alpha'} \in T_j$ for $j > i$ or $g_{i,\alpha'} \in T_i$ for $g_{i,\alpha'} > g_{i,\alpha}$ (corresponds to the first term on the right-hand side of (11)). Then, each path from the root to the $\top$ corresponds to a single group element, where $g_{i,\alpha}$ appears in the formula if the path through the 1-edge of the node associated with $g_{i,\alpha}$. By definition, $\top$ corresponds to the singleton of the identity element $\{e\}$ and $\bot$ corresponds to the empty set $\{\ \}$.

To make the diagram compact, we apply the same reduction rules (9) as that of the ZDD. As per the reduction rules and the uniqueness of the representation (11) under the fixed orderings on the transversals, any subset S is uniquely represented by a GDD $\mathcal{D}(S)$. This means that, GDD is a *canonical data structure* for subsets of the group.

For each node $u \in V$, its descendants form a GDD. Thus, we identify the node in a GDD and the GDD. In particular, the GDD $\mathcal{D}(S)$ is identified as the root node of $\mathcal{D}(S)$.

Remark 4. It may be more natural to construct a DAG with arbitrary many out-going edges from (10). The GDD is isomorphic to this DAG as each "fat" node in the DAG is expanded to binary nodes (the 0-edge is for a sibling and the 1-edge is for a child). However, in practice, the GDD offers a better run time than the DAG because it uses less memory, and has higher hash efficiency because it can store the "intermediate" states of the fat nodes in the DAG.

Here, we provide several examples of the GDD.

Example 5. Let us consider the symmetric group $\mathrm{Sym}(5)$ of degree five. It has a chain of subgroups whose transversals are

$$T_1 = \{e, (1,2), (1,3), (1,4), (1,5)\}, \tag{12}$$
$$T_2 = \{e, (2,3), (2,4), (2,5)\}, \tag{13}$$
$$T_3 = \{e, (3,4), (3,5)\}, \tag{14}$$
$$T_4 = \{e, (4,5)\}. \tag{15}$$

Note that this is the reverse numbering as Example 2.

We try to represent subset $S = \{e, (2,3,4), (2,4,3)\}$. Since $(2,3,4) = (3,4)(2,3)$ where $(3,4) \in T_3$ and $(2,3) \in T_2$ and $(2,4,3) = (3,4)(2,4)$ where $(3,4) \in T_3$ and $(2,4) \in T_2$, this set is factorized by

$$S = \{e\} \cup \{(3,4)(2,3)\} \cup \{(3,4)(2,4)\} \tag{16}$$
$$= (\{\{e\}) \cup \{(3,4)\}(2,4)) \cup \{(3,4)\}(2,3) \tag{17}$$

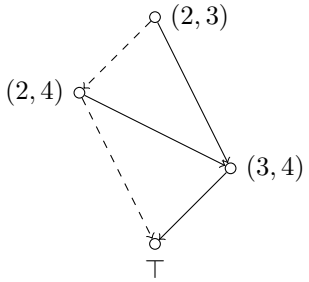

Figure 1: GDD for representing $\langle (2,3,4) \rangle$ in $\mathrm{Sym}(5)$; see Example 5

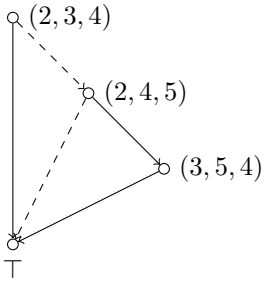

Figure 2: GDD for representing $\langle (2,3,4) \rangle$ in $\mathrm{Alt}(5)$; see Example 6.

Thus, $\mathcal{D}(S)$ is represented by Figure 1. Note that the substructure for $\{(3,4)\}$, which appeared twice in (16) is shared. □

Example 6. Let us consider the alternate group $\mathrm{Alt}(5)$ of degree five, which is the set of all even permutations on $[5]$ (i.e., obtained by the even number of transpositions). Then, it has a chain of subgroups

$$T_1 = \{e, (1,2,3), (1,3,4), (1,4,5), (1,5,2)\}, \tag{18}$$
$$T_2 = \{e, (2,3,4), (2,4,5), (2,5,3)\}, \tag{19}$$
$$T_3 = \{e, (3,4,5), (3,5,4)\}. \tag{20}$$

We try to represent subset $S = \{e, (2,3,4), (2,4,3)\}$. This set is factorized by

$$S = ((\{e\}) \cup \{(3,5,4)\}(2,4,5)) \cup \{(2,3,4)\}. \tag{21}$$

Thus, it is represented by Figure 2. □

We show that the GDD contains the ZDD, πDD, and ρDD as special cases.

Example 7 (GDD $\supset$ ZDD)**.** Let $C_2^n = \langle \{(1,2), (3,4), \ldots, (2n-1, 2n)\} \rangle$ be the direct product of n cyclic groups of order two. We consider the subgroup chain

$$C_2^n \gneq C_2^{n-1} \gneq \cdots \gneq C_2 \gneq \{e\}, \tag{22}$$

where C_2^k is the subgroup of C_2^n generated by $(1,2), \ldots, (2k-1, 2k)$. The coset C_2^k / C_2^{k-1} has a cardinality of two, with representatives that correspond to the flipping or non-flipping of $2k-1$ and $2k$. If we select $T_k = \{e, (2k-1, 2k)\}$ as the transversal for each k, the GDD coincides with the ZDD. □

Example 8 (GDD $\supset$ πDD)**.** Let us consider the symmetric group $\mathrm{Sym}(n)$ with the subgroup chain (8) as in Example 2. If we choose $T_i = \{e, (1,k), (2,k), \ldots, (k-1,k)\}$ as the transversal for each k, the GDD coincides with the πDD. □

Example 9 (GDD $\supset$ ρDD)**.** Let us consider the same group $\mathrm{Sym}(n)$ with the same subgroup chain (8) as in Example 2. If we select $T_i = \{\rho_{1,k}, \ldots, \rho_{k-1,k}\}$ as the transversal for each k, the GDD coincides with the ρDD. □

4 Implementation of Basic Operations

Here, we present algorithms to manipulate GDDs. We assume the RAM model with sufficient memory (i.e., we can store/retrieve elements in a hash table in $O(1)$ time).

Similar to the existing decision diagrams (Bryant 1992; Minato 1993; 2011; Inoue and Minato 2014), our algorithms are implemented using recursion with memoization. Their run time depends on the size of the GDDs, rather the cardinality of the subsets. As the size of GDD is often exponentially smaller than the cardinality of the subset, it provides exponential speedup.

Preprocessing: Subgroup Chain

In several applications, we only have generators S of the permutation group G. Thus, we have to compute the subgroup chain (2) with transversals from S. It is desired that the transversals are "close" to the generators, as this may yield a smaller GDD if the sets are generated by a few operations from the generators. For this purpose, we employ the *incremental Schreier–Sims* method (Holt, Eick, and O'Brien 2005), which finds the subgroup chain in polynomial time.[1]

The algorithm works as follows (see (Holt, Eick, and O'Brien 2005) for more detail). First, we find $\beta_1 \in [n]$ that is not stabilized by the generators, and we define $S_1 = \{s \in S : \beta_1^s = \beta_1\}$ the set of generators that stabilizes β_1. We compute the subgroup chain of subgroup $H = \langle S_1 \rangle$ recursively. We also compute the generators for $G_1 = \mathrm{Stab}_G(\{\beta_1\})$ using Schreier's lemma (Theorem 3). Then, for each generator h of G_1, we check $h \in H$. If $h \notin H$, we add h to S_1 and repeat the procedure; else, we have $G_1 = H$; therefore, we obtain the subgroup chain. Here, we can check $h \in H$ efficiently by the following procedure, called the *sifting*. By the definition of the incremental Schreier–Sims algorithm, we have a subgroup chain $H = H_0 \gneq H_1 \gneq \cdots \gneq H_{r'} = \{e\}$. Let $h_{1,\alpha}$ be a representative of h in H/H_1; if it does not exist, we have $h \notin H$. Then, $h \in H$ if and only if $hh_{1,\alpha}^{-1} \in H_1$, which is computed recursively. According to the construction, all the generators in the input are used in the transversals (if it is not redundant).

[1] We compared the incremental Schreier–Sims method and the Schreier–Sims method with Jerrum's filter in a preliminary experiment, and found that although the former is slower in computing the transversals, it produces better transversals, which provide better performance in GDD applications shown in Section 5.

Algorithm 2 Create a node for (i, α, lo, hi)

```
1: procedure GetNode(i, α, lo, hi)
2:     if hi = ⊥ then return lo
3:     if (i, α, lo, hi) ∉ Cache then
4:         Cache[(i, α, lo, hi)] ← new Node(i, α, lo, hi)
5:     end if
6:     return Cache[(i, α, lo, hi)]
7: end procedure
```

Algorithm 3 Create a GDD for a singleton.

```
1: procedure Singleton(h, i = 0)
2:     if h = e then return ⊤
3:     if β_i^h = β_i then return Singleton(h, i + 1)
4:     return GetNode(i, β_i^h, ⊥, Singleton(h g_{i,β_i^h}^{-1}, i + 1))
5: end procedure
```

Node Representation

Let r be the length of the subgroup chain. A node of the GDD is implemented by a tuple $x = (i, \alpha, \mathrm{lo}, \mathrm{hi})$ of $i \in [r]$, $\alpha \in [n]$, and two pointers lo and hi. This node is associated with an element $g_{i,\alpha} \in T_i$ in the transversal of G_{-1}/G_i, and the 1 and 0-edges are implemented by pointers hi and lo, respectively. We denote by $x.i$, $x.\alpha$, $x.\mathrm{lo}$, and $x.\mathrm{hi}$ to denote the entries of the tuple.

To maintain the reduction rule (2) efficiently, we store the nodes in a hash table and retrieve the stored node when we refer to the node as in Algorithm 2.

Owing to the uniqueness of the representation, and the usage of the hash table, the equality of the two subsets represented by GDDs are checked in $O(1)$ time by checking the equality of the addresses.

Singleton

For $h \in G$, we can construct the GDD $\mathcal{D}(\{h\})$ of singleton $\{h\}$ in $O(nr)$ time as follows. First, we compute $\alpha = \beta_1^h$. Then, the corresponding element in the first transversal is given by $g_{1,\alpha}$. We construct GDD $\mathcal{D}(\{hg_{1,\alpha}^{-1}\})$ recursively, and let x be the pointer of the root node of this GDD. Then, it returns GDD node $(1, \alpha, \bot, x)$ as the solution. The implementation is shown in Algorithm 3.

Membership

For $h \in G$ (or $h \in \mathrm{Sym}(n)$), we can test whether $h \in S$ in $O(n\mathrm{depth}(\mathcal{D}(S)))$ time, where $\mathrm{depth}(\mathcal{D}(S))$ is the length of the longest path in $\mathcal{D}(S)$, which is bounded by $\sum_i |T_i|$, as the following procedure, which is almost the same as the sifting.

Let $S = S' \cup S''g_{i,\alpha}$. Then, if $\beta_i^h = \alpha$ then $h \in S$ if and only if $hg_{i,\alpha}^{-1} \in S''$; else, $h \in S$ if and only if $h \in S'$, if S and S' correspond to the same coset. Each step of this procedure requires $O(n)$ time, and the number of recursion is at most the depth of $\mathcal{D}(S)$. Therefore, the complexity is obtained. The implementation is shown in Algorithm 4.

Algorithm 4 Membership predicate.

```
1: procedure IsMember(h, x)
2:     if x = ⊥ then return False
3:     if h = e and x = ⊤ then return True
4:     if β_i^h = α then return IsMember(h g_{i,α}^{-1}, x.hi)
5:     if x.i = x.lo.i then return IsMember(h, x.lo)
6:     return False
7: end procedure
```

Algorithm 5 Union of two sets.

```
1: procedure Union(x, y)
2:     if x = ⊥ then return y
3:     if y = ⊥ then return x
4:     if x = y then return x
5:     if {x, y} ∉ Cache then
6:         if (x.i, x.α) = (y.i, y.α) then
7:             z ← Union(x.lo, y.lo)
8:             w ← Union(x.hi, y.hi)
9:             Cache[{x, y}] ← GetNode(x.i, x.α, z, w)
10:        else if (x.i, x.α) < (y.i, y.α) then
11:            z ← Union(x.lo, y)
12:            Cache[{x, y}] ← GetNode(x.i, x.α, z, x.hi)
13:        else
14:            z ← Union(y.lo, x)
15:            Cache[{x, y}] ← GetNode(y.i, y.α, z, y.hi)
16:        end if
17:    end if
18:    return Cache[{x, y}]
19: end procedure
```

Set Operations

Set operations, e.g., union, intersection, and difference, can be implemented as in the ZDD. For example, we consider union operation. Suppose that $S = S' \cup S'' g_{i,\alpha}$ and $T = T' \cup T'' g_{j,\beta}$. If $(i, \alpha) \neq (j, \beta)$, we call the procedure recursively to the suitable child; else, we have

$$S \cup T = (S' \cup T') \cup (S'' \cup T'') g_{i,\alpha}. \tag{23}$$

Therefore, we call the procedure recursively to the children. To reduce the computational cost, we memoize the computation using a hash table. Algorithm 5 shows the implementation of the union operation. The intersection and difference can be implemented similarly.

The complexity of the union, intersection, and difference operations are analyzed as follows:

Proposition 10. Let $\mathcal{D}(S)$ and $\mathcal{D}(T)$ be GDD nodes representing subsets S and T, respectively. Then, the union, intersection, and difference run in $O(|\mathcal{D}(S)||\mathcal{D}(T)|)$ time.

Proof. Due to memoization, for each pair of nodes x, y of $\mathcal{D}(S)$, $\mathcal{D}(T)$, respectively, the procedure is called at most $O(1)$ times. □

Remark 11. The above estimation appears pessimistic. For the ZDD, in the worst case, there exists example that requires quadratic time; however, in many applications, it runs in almost linear time (Yoshinaka et al. 2012).

Algorithm 6 Right multiplication.

```
1: procedure RMul(x, h)
2:     if x = ⊥ then return ⊥
3:     if x = ⊤ then return Singleton(h)
4:     if (x, h) ∉ Cache then
5:         h_α = g_{x.i,x.α} h g_{x.i,x.α^h}^{-1}
6:         z ← RMul(x.lo, h)
7:         w ← GetNode(x.i, x.α^h, ⊥, RMul(x.hi, h_α))
8:         Cache[(x, h)] ← Union(z, w)
9:     end if
10:    return Cache[(x, y)]
11: end procedure
```

Cartesian Product

The *Cartesian product* of two sets $S, T \subseteq G$ is defined by

$$SS = \{st : s \in S, t \in T\} \tag{24}$$

This operation is highly beneficial in several applications, including the Rubik's cube problem, as shown in Section 1.

To implement the Cartesian product, we first implement the *right multiplication*, Sh, for the subroutine. Note that it is a special case (T is a singleton) of the Cartesian product. Let $S = S' \cup (S'' g_{i,\alpha})$. We define $h_\alpha = g_{1,\alpha} h g_{1,\alpha^h}^{-1}$. Then, we have

$$\begin{aligned} Sh &= (S' \cup (S'' g_{i,\alpha}))h \\ &= (S'h) \cup \left((S'' h_\alpha) g_{i,\alpha^h}\right). \end{aligned} \tag{25}$$

Here, $S'' h_\alpha$ is a multiplication in G_i. Thus, no additional computation is needed to multiply g_{i,α^h} from the right. Therefore, we can implement right multiplication recursively. The implementation is shown in Algorithm 6.

Using the right multiplication as a subroutine, we can implement the Cartesian product as follows. For the base case, we have

$$S\{e\} = S, \quad S\{\} = \{\}. \tag{26}$$

For general case, let $T = T' \cup (T'' g_{i,\alpha})$. Then, we have

$$\begin{aligned} ST &= S\left(T' \cup (T'' g_{i,\alpha})\right) \\ &= (ST') \cup \left((ST'') g_{i,\alpha}\right). \end{aligned} \tag{27}$$

Therefore, by calling the procedure recursively, we compute the Cartesian product. The worst case complexity of the Cartesian product may be exponential.

Remark 12. Here, we mention a disadvantage of the GDD compared to the πDD and ρDD. In the implementations of the Cartesian product in the πDD and ρDD, the following relation can be used: for any α and β, there exists γ such that

$$g_{i,\alpha} g_{i,\beta} = g_{i-1,\gamma} g_{i,\alpha^{g_{i,\beta}}}, \tag{28}$$

Using this identity, we can assume that the group elements appeared in the right multiplication for the Cartesian product are transversal elements, which reduces the size of the hash table. However, in the GDD, we cannot assume such a relation. Thus, it may increase the constant factor of the Cartesian product operation.

This phenomenon will be particularly observed when $G = \mathrm{Sym}(n)$ and the GDD reduced to the πDD or ρDD; see Pancake Sorting problem in our Experiments.

Algorithm 7 Cartesian product.

```
1: procedure CartesianProd(x, y)
2:   if y = ⊥ then return ⊥
3:   if y = ⊤ then return x
4:   if {x, y} ∉ Cache then
5:     z = CartesianProd(x, y.lo)
6:     w = RMul(CartesianProd(x, y.hi), g_{y.i,y.α})
7:     Cache[{x, y}] ← Union(x, w)
8:   end if
9:   return Cache[{x, y}]
10: end procedure
```

5 Experiment

We conducted experiments to evaluate the performance of the GDD. The codes was implemented in C++. For comparison, we obtained the original source codes of πDD and ρDD from the authors of the papers (Minato 2011; Inoue and Minato 2014), which are also implemented in C++. These were compiled by g++ (GCC) 4.8.5 with the -O3 option. All the experiments were performed on a machine with Intel(R) Xeon(R) CPU E5-2690 v4, 2.60GHz with a single core, with 50GB of RAM.

Our code is available at the project page[2]. Also, some additional experimental results are available in the project page, which are omitted from this manuscript due to the space limitation.

Rubik's Cube (Pocket Cube) Puzzle

The Rubik's cube is one of the famous permutation puzzles. Here, we consider the *Pocket Cube*, which is a $2 \times 2 \times 2$-variant of the Rubik's cube[3]. The possible configurations of the puzzle forms a group $G \leq \mathrm{Sym}(24)$ generated by the following three elements:

$$R = (13, 14, 16, 15)(10, 2, 19, 22)(12, 4, 17, 24), \quad (29)$$

$$U = (1, 2, 4, 3)(9, 5, 17, 13)(10, 6, 18, 14), \quad (30)$$

$$F = (9, 10, 12, 11)(3, 13, 22, 8)(4, 15, 21, 6). \quad (31)$$

The order of G is $3,674,160$, which is significantly smaller than that of $|\mathrm{Sym}(24)| = 24! \approx 6 \times 10^{23}$. Therefore, we expect the GDD to perform better than the πDD and ρDD.

We count the number of configurations that can be generated by k moves for $k = 1, 2, \ldots$. These numbers are obtained by computing the k-fold Cartesian products $S \cdots S$ for $S = \{e, R, R^{-1}, U, U^{-1}, F, F^{-1}\}$. The result is shown in Table 1. As expected, in the most expensive case, i.e., $k = 11$, the size of GDD is 6 and 14 times smaller than those of the πDD and ρDD, respectively, making the computation time 12 and 28 times faster than those of the πDD and ρDD, respectively. It should be emphasized that the number of nodes of the GDD is smaller than the number of configurations, implying that multiple configurations are represented compactly in the GDD. This demonstrates the effectiveness of the GDD.

[2]https://github.com/spaghetti-source/gdd

[3]The results of the original $3 \times 3 \times 3$ Rubik's cube is presented in the project page — all the methods can enumerate at most six steps; thus, not suited for comparing algorithms.

Table 1: Pocket Cube. #Conf. shows the number of configurations. The first rows of GDD, πDD, and ρDD show the number of nodes of the data structures, and the second rows show the computation time (in [sec]). The time to construct the subgroup chain for GDD is not included in the below table, but it is less than 0.01s.

k	2	5	8	11	14
#Conf.	27	2,256	114,149	1,350,852	276
GDD	53	3,047	95,152	445,928	76
	0.00s	0.01s	0.88s	19.50s	29.18
πDD	354	24,917	647,999	2,455,878	1,400,951
	0.00s	0.55s	36.83s	374.46s	481.07s
ρDD	495	33,427	1,090,124	6,369,842	6,149,175
	0.01s	0.97s	54.17s	745.75s	1069.31s

Pancake Sorting Problem

The *pancake sorting problem* (Dweighter 1975) is as follows. Let τ_k be a prefix-reversal permutation, i.e., $\tau_k = [k, k-1, \ldots, 1, k+1, k+2, \ldots, n]$. Then, $\tau_2, \ldots, \tau_n$ generates the symmetric group $\mathrm{Sym}(n)$, i.e., any $\pi \in \mathrm{Sym}(n)$ is generated by the product of $\tau_2, \ldots, \tau_k$. The problem seeks the smallest integer k such that any permutation $g \in \mathrm{Sym}(n)$ is generated by at most k products of $\tau_2, \ldots, \tau_n$ (i.e., it is the diameter of the Cayley graph of $\mathrm{Sym}(n)$). This value is useful in a network design problem for parallel computation (Akl, Qiu, and Stojmenović 1993). Currently, the values of $n \leq 17$ are obtained using a sophisticated algorithm with massive parallel computation (Asai et al. 2006).

The result is shown in Table 2. Here, the GDD does not outperform πDD and ρDD — it is slightly slower than πDD and three times slower than ρDD. The reason is that, in this example, the size of GDD is not so smaller than that of πDD and ρDD since the Pancake group is $\mathrm{Sym}(n)$, and the generators of πDD and ρDD are not so redundant — The generators of this problem is the prefix reversals, e.g., $[2, 1, 3, 4]$, $[3, 2, 1, 4$, and $[4, 3, 2, 1]$ for $n = 4$. These are efficiently represented by the successive applications of the left rotations as $[2, 1, 3, 4] = (1, 2)$, $[3, 2, 1, 4] = (1, 2)(1, 2, 3)$, and $[4, 3, 2, 1] = (1, 2)(1, 2, 3)(1, 2, 3, 4)$. Thus, ρDD is suitable for this problem.

Shortest Zero-Walk in Group-Labeled Graphs

A *group-labeled graph* (Huynh 2009) (with group G) is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a group-valued edge label $g \colon \mathcal{E} \to G$. A group-labeled graph is also known as a *voltage graph* (Gross and Tucker 1977) or *biased graph* (Zaslavsky 1989), and has been studied for graph embedding.

Here, we consider the *shortest zero-walk problem*. A *walk* $W = [u_1, \ldots, u_l]$ is a sequence of vertices that traverses edges, i.e., $(u_i, u_{i+1}) \in \mathcal{E}$ for all $i = 1, \ldots, l-1$. We define $g(W) = g((u_1, u_2)) \cdots g((u_{l-1}, u_l))$. Then, the problem seeks the shortest walk W from s to t such that $g(W) = e$. As mentioned in (Kobayashi and Toyooka 2017), this problem is solved in $O(|G||\mathcal{V}||\mathcal{E}|)$ time using the dynamic pro-

Table 2: Pancake Sorting Problem. Ans. shows the solution for each n. The first rows of GDD, πDD, and ρDD show the number of nodes appeared during the computation, and the second rows shows the computation time (in [sec]). The time to construct the subgroup chain for GDD is not included in the below table, but it is less than 0.01s.

n	7	8	9	10	11
Ans.	8	9	10	11	13
GDD	611	3,224	19,751	140,360	1,165,544
	0.02s	0.09s	0.82s	11.00s	126.81s
πDD	590	3,309	20,343	144,630	1,195,767
	0.01s	0.06s	0.85s	10.38s	117.69s
ρDD	517	2,801	17,429	122,781	1,021,055
	0.00s	0.03s	0.20s	3.25s	39.39s

Table 3: Shortest zero walk problem. The row of Ans. shows the answer of the solution for each n. The first rows of GDD, πDD, and ρDD show the maximum size of xDDs appeared during the computation, and the second rows show the computation time of the data structures (in [sec]). The time to construct the subgroup chain for GDD is not included in the below table, but it is less than 0.01s.

n	10	30	50	70	90
Ans.	18	34	54	74	94
GDD	1,016	13,915	51,414	128,159	255,004
	0.21s	1.25s	5.20s	15.40s	32.22s
πDD	1,258	298,555	—	—	—
	0.20s	290.74s	—	—	—
ρDD	1,220	57,792	363,028	1,397,459	3,813,833
	0.16s	9.16s	83.25s	310.00s	926.80s

gramming algorithm:

$$S_{v,k+1} \leftarrow S_{v,k} \cup \bigcup_{(u,v)\in\mathcal{E}} S_{u,k}g((u,v)). \qquad (32)$$

We implemented the above algorithm using the GDD, πDD, and ρDD, and compared their performances.

We defined $G = \langle\{(1,2),(1,\ldots,n)\}\rangle = \mathrm{Sym}(n)$. Then, we generated a graph of 10×10 grid, which consists of 100 vertices; each edge is bidirected and assigned $(1,2)$ or $(1,\ldots,n)$ randomly. Then, the shortest walk was determined using (32). The result is shown in Table 3. The GDD found the shortest walk for $n = 90$ in 30[sec]. On the other hand, the πDD only solved $n = 30$ within time limit (< 3000 [sec]). The ρDD solved $n = 90$ but requires approximately 15 [min].

This result implies that, even if the group is the symmetric group, the GDD has an advantage because it can find a better representation suited to the input.

6 Conclusion

We have proposed a new data structure, called the GDD, to maintain a set of permutations. The data structure combines the ZDD with the computable subgroup chain. The data structure admits efficient operations such as the membership testing, set operations (union, intersection, difference), and Cartesian product. Our experiments demonstrated that the GDD is highly efficient than the existing data structures if the group G is considerably smaller than the symmetric group.

The most important future direction is to characterize when the data structure works well. For the BDD and ZDD, we know that any subsets of a bounded treewidth graph specified by a monadic second-order logic are maintained by polynomial size structure (Amarilli et al. 2017). However, there are no corresponding result for permutation diagrams. We believe that establishing such result increases applicability of the data structures in practice.

Related with this direction, we need a method for constructing a "nice" subgroup chain for given problems. As we see in experiments, it is very important to reduce the size of GDD, which is determined by the subgroup chain. Therefore, it will be effective for the running time.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant number 15H05711. The authors thank Shin'ichi Minato for providing source code for BDD.

References

Akl, S. G.; Qiu, K.; and Stojmenović, I. 1993. Fundamental algorithms for the star and pancake interconnection networks with applications to computational geometry. *Networks* 23(4):215–225.

Amarilli, A.; Bourhis, P.; Jachiet, L.; and Mengel, S. 2017. A circuit-based approach to efficient enumeration. In *44th International Colloquium on Automata, Languages, and Programming (ICALP'17)*, 1–15.

Asai, S.; Kounoike, Y.; Shinano, Y.; and Kaneko, K. 2006. Computing the diameter of 17-pancake graph using a pc cluster. In *European Conference on Parallel Processing*, 1114–1124.

Bosma, W.; Cannon, J.; and Playoust, C. 1997. The Magma algebra system. I. The user language. *Journal of Symbolic Computation* 24(3-4):235–265.

Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Cameron, P. J. 1999. Permutation groups, london math. soc. student texts 45.

Cooperman, G., and Murray, S. H. 2005. Computable subgroup chains and shadowing. Technical report, http://www.maths.usyd.edu.au/u/pubs/publist/preprints/2008/cooperman-16.pdf.

Dweighter, H. 1975. Problem e2569. *American Mathematical Monthly* 82(10).

Egner, S., and Püschel, M. 1998. Solving puzzles related to permutation groups. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, 186–193. ACM.

The GAP Group. 2018. *GAP – Groups, Algorithms, and Programming, Version 4.9.2*.

Gross, J. L., and Tucker, T. W. 1977. Generating all graph coverings by permutation voltage assignments. *Discrete Mathematics* 18(3):273–283.

Holt, D. F.; Eick, B.; and O'Brien, E. A. 2005. *Handbook of Computational Group Theory*. Chapman and Hall/CRC.

Huynh, T. 2009. The linkage problem for group-labelled graphs. Technical report, University of Waterloo.

Inoue, Y., and Minato, S. 2014. An efficient method for indexing all topological orders of a directed graph. In *International Symposium on Algorithms and Computation*, 103–114. Springer.

Knuth, D. E. 1997. *The Art of Computer Programming*, volume 3. Pearson Education.

Kobayashi, Y., and Toyooka, S. 2017. Finding a shortest non-zero path in group-labeled graphs via permanent computation. *Algorithmica* 77(4):1128–1142.

Matsumoto, K.; Hatano, K.; and Takimoto, E. 2018. Decision diagrams for solving a job scheduling problem under precedence constraints. In *Proceeding of the 17th International Symposium on Experimental Algorithms*.

Meurer, A.; Smith, C. P.; Paprocki, M.; Čertík, O.; Kirpichev, S. B.; Rocklin, M.; Kumar, A.; Ivanov, S.; Moore, J. K.; Singh, S.; Rathnayake, T.; Vig, S.; Granger, B. E.; Muller, R. P.; Bonazzi, F.; Gupta, H.; Vats, S.; Johansson, F.; Pedregosa, F.; Curry, M. J.; Terrel, A. R.; Roučka, v.; Saboo, A.; Fernando, I.; Kulal, S.; Cimrman, R.; and Scopatz, A. 2017. Sympy: symbolic computing in python. *PeerJ Computer Science* 3:e103.

Minato, S. 1993. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th international Design Automation Conference*, 272–277. ACM.

Minato, S. 2011. πdd: A new decision diagram for efficient problem solving in permutation space. In *International Conference on Theory and Applications of Satisfiability Testing*, 90–104. Springer.

Mulholland, J. 2013. Permutation puzzles: a mathematical perspective.

Pruesse, G., and Ruskey, F. 1994. Generating linear extensions fast. *SIAM Journal on Computing* 23(2):373–386.

Seress, Á. 2003. *Permutation Group Algorithms*, volume 152. Cambridge University Press.

Tague, L.; Soeken, M.; Minato, S.; and Drechsler, R. 2013. Debugging of reversible circuits using pdds. In *Proceedings of the IEEE 43rd International Symposium on Multiple-Valued Logic (ISMVL'13*, 316–321.

Waksman, A. 1968. A permutation network. *Journal of the ACM* 15(1):159–163.

Yoshinaka, R.; Kawahara, J.; Denzumi, S.; Arimura, H.; and Minato, S. 2012. Counterexamples to the long-standing conjecture on the complexity of bdd binary operations. *Information Processing Letters* 112(16):636–640.

Zaslavsky, T. 1989. Biased graphs. i. bias, balance, and gains. *Journal of Combinatorial Theory, Series B* 47(1):32–52.