Isomorphism Elimination by Zero-Suppressed Binary Decision Diagrams*

Takashi Horiyama† Masahiro Miyasaka† Riku Sasaki‡

Abstract

In this paper, we focus on the isomorphism elimination. More precisely, our problem is as follows: Given a graph G with labeled edges and a family $\mathcal{F}$ of its subgraphs, we extract all automorphisms $\mathrm{Aut}G = \{\pi_1, \pi_2, \ldots, \}$ on the given graph, define the lexicographically largest subgraph for each set of the mutually isomorphic subgraphs on each automorphism π_i, and select the lexicographically largest subgraphs on any of the automorphisms. In this paper, the families of subgraphs are manipulated by ZDDs. We also apply our algorithms to the enumeration of nonisomorphic developments of Platonic and Archimedean solids and d-dimensional hypercubes. Experimental results show that the proposed method is more than 300 times faster and 3,000 times less memory than the conventional method in the best case. Our algorithms are applicable to many other enumeration problems with eliminating isomorphic solutions.

1 Introduction

Suppose that we are given a cube. By cutting along the set of edges $\{e_2, e_3, e_4, e_6, e_{10}, e_{11}, e_{12}\}$ of the cube in Figure 1(a), we can obtain the development in Figure 1(c). When we rotate the positions of cut edges by 90 degrees, i.e., by cutting along the set of edges $\{e_1, e_3, e_4, e_7, e_9, e_{11}, e_{12}\}$ as depicted in Figure 1(b), we can also obtain the development in Figure 1(c). Are these the same? If we focus on the fact that the edges are *labeled*, the positions of cut edges are different, and thus we can say they are different. If we do not care about the labels, i.e., the edges are *unlabeled*, the shape of the developments are the same, and thus we can say they are *isomorphic*.

A cube has 384 labeled developments, and they are classified into 11 nonisomorphic developments (i.e., essentially different unlabeled developments). Here, a development and its mirror shape are regarded as isomorphic. As for the labeled developments, we can count their numbers by combining the following two theorems:

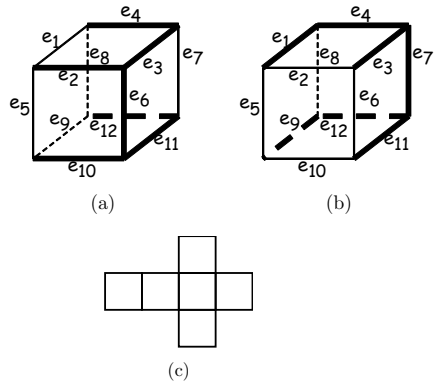

Figure 1: The developments of a cube by different cut edges (a) and (b) are isomorphic.

Theorem 1 *(See, e.g., [[5], Lemma 22.1.1]) The cut edges of a development of a polyhedron form a spanning tree of the 1-skeleton (i.e., the graph formed by the vertices and the edges) of the polyhedron, and vice versa.*

Theorem 2 *Matrix Tree Theorem [12]: The number of spanning trees of a graph is equal to any cofactor of the Laplacian matrix of the graph.*

By applying the theorems, Brown et al. showed that a Buckminsterfullerene (also known as an icosahedral C_{60}, or a truncated icosahedron) has 375,291,866,372,898,816,000 (approximately 3.75×10^{20}) labeled developments [1]. The numbers of labeled developments of Handballene (truncated dodecahedral C_{60}) and Archimedene (truncated icosidodecahedral C_{120}) are given in [2].

As for counting the nonisomorphic developments, the numbers for Platonic solids are obtained in the 1970s [7][10]. Recently, Horiyama and Shoji proposed a technique for counting the number of nonisomorphic developments of any polyhedron (including nonconvex polyhedron) [9]. By applying this method, they also listed the number of nonisomorphic (and also labeled) developments of all regular-faced convex polyhedra (i.e., Platonic solids, Archimedean solids, Johnson-Zalgaller solids, Archimedean prisms, and antiprisms), Catalan solids, bipyramids and trapezohedra. For example, a Buckminsterfullerene (i.e., a truncated icosahedron) has 3,127,432,220,939,473,920 (approximately 3.13×10^{18})

*A preliminary version was presented at AAAC2018.

†Graduate School of Science and Engineering, Saitama University, {horiyama,miyasaka}@al.ics.saitama-u.ac.jp

‡Faculty of Engineering, Saitama University, sasaki@al.ics.saitama-u.ac.jp

nonisomorphic developments. We here note that the technique in [9] counts the number of nonisomorphic developments without enumerating developments.

If we turn to the developments of polytopes in 4 (or more) dimensions. We can apply the matrix tree theorem to any polytope, and thus we can count the number of the labeled developments. As for the number of the nonisomorphic developments, Gardner asked to enumerate all of the nonisomorphic developments of a 4-dimensional hypercube [6], and Turney enumerated 261 nonisomorphic developments by hands [19]. He also says "As far as I know, the only way is to exhaustively examine the possibilities" in [19]. Later, a technique for counting the number of the nonisomorphic developments of 4-dimensional regular convex polytopes [4] is proposed. The technique is an extension of those for the Platonic solids [7][10], and is further extended to that for any 3-dimensional polyhedron [9]. These techniques avoid explicitly enumerating the developments, but count their numbers by exploiting Polya's counting theorem [17].

As for the enumeration of nonisomorphic developments, an efficient exhaustive search technique using BDDs (Binary Decision Diagrams) is proposed in [8], where a BDD [3] is a succinct data structure that represents a family of sets by a graph. In [8], a method to construct a BDD corresponding to a family of labeled developments is proposed, where each development are represented as a set of labeled edges that form a spanning tree. Then, by omitting mutually isomorphic developments, the nonisomorphic developments are obtained.

Later, a sophisticated technique called a "frontier-based search" [11] is proposed for constructing BDDs/ZDDs representing all constrained subgraphs, and we can adopt this technique to the first step of the method in [8]. A ZDD (Zero-suppressed Binary Decision Diagram) [16] is a variant of BDDs, and also represents a family of sets. The frontier-based search is an extension of Simpath algorithm [13] by Knuth for enumerating all st-paths (i.e., simple paths from vertex s to t) in a given graph. The method can be considered as one of DP-like algorithms, and it constructs the resulting BDDs/ZDDs in a top-down manner. By applying this method to the first step in [8], we can speed-up the construction of the BDD/ZDD representing a family of spanning trees.

Our contribution. In this paper, we focus on the second step of the method in [8], i.e., the isomorphism elimination. More precisely, our problem is as follows: Given a graph G with labeled edges and a family $\mathcal{F}$ of its subgraphs, we extract all automorphisms $\mathrm{Aut}G = \{\pi_1, \pi_2, \dots, \}$ on the given graph, define the lexicographically largest subgraph for each set of the mutually isomorphic subgraphs on each automorphism π_i, and select the lexicographically largest subgraphs on any of the automorphisms. In this paper, both of the given and resulting families of subgraphs are in the form of ZDDs, and the computation are performed on ZDDs. This is because (1) ZDDs can represent a family of sets compactly, (2) the manipulation of ZDDs are faster than the other representations in many cases.

In general, the first step for extracting all automorphisms on a given graph is not tractable: It is still open whether the graph automorphism problem (i.e., the problem deciding whether a given graph has a nontrivial automorphism or not) is in P or in NP-complete [15]. Fortunately, however, we can solve the problem in polynomial time if the degrees of vertices in a graph graph are bounded by a constant [14].

Our main issue is to select the lexicographically largest subgraphs on any of the automorphisms. In [8], BDDs $G_1, G_2, \dots$ are constructed so that G_i represents a family of the lexicographically largest subgraphs on automorphism π_i, and their intersection is taken for selecting a family of subgraphs that appear in all of the families of $G_1, G_2, \dots$ Unfortunately, the method was proposed before the era of the frontier-based search algorithms. Thus, similarly to the BDD/ZDD algorithms in those days, it obtains the resulting BDD by an old-fashioned manner, i.e., by the repetition of so-called "apply operations." In this paper, we renovate this step by introducing the framework of the frontier-based search: We propose algorithms for the top-down construction of the ZDD representing a family of the lexicographically largest subgraphs on π_i.

2 Enumeration by Zero-Suppressed Binary Decision Diagrams

A *zero-suppressed binary decision diagram (ZDD)* [16] is directed acyclic graph that represents a family of sets. As illustrated in Figure 2, it has the unique source node[1], called *the root node*, and has two sink nodes 0 and 1, called *the 0-node* and *the 1-node*, respectively (which are together called the constant nodes). Each of the other nodes is labeled by one of the variables $x_1, x_2, \dots, x_n$, and has exactly two outgoing edges, called *0-edge* and *1-edge*, respectively. On every path from the root node to a constant node in a ZDD, each variable appears at most once in the same order. The size of a ZDD is the number of nodes in it.

Every node v of a ZDD represents a family of sets $\mathcal{F}_v$, defined by the subgraph consisting of those edges and nodes reachable from v. If node v is the 1-node (respectively, 0-node), $\mathcal{F}_v$ equals to $\{\{\}\}$ (respectively, $\{\}$). Otherwise, $\mathcal{F}_v$ is defined as $\mathcal{F}_{0\text{-}succ(v)} \cup \{S \mid S = \{var(v)\} \cup S', S' \in \mathcal{F}_{1\text{-}succ(v)}\}$, where 0-$succ(v)$ and

[1] We distinguish *nodes* of a ZDD from *vertices* of a graph (or a 1-skeleton).

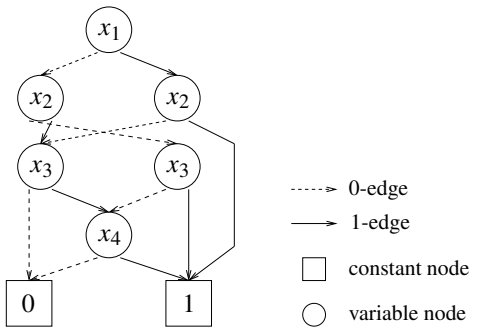

Figure 2: A ZDD representing $\{\{1,2\},\{1,3,4\},\{2,3,4\},\{3\},\{4\}\}$.

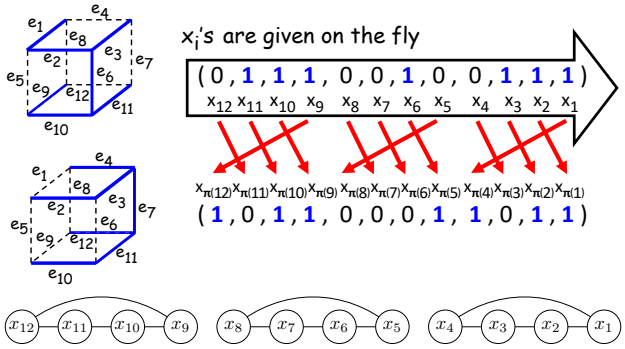

Figure 3: Comparison of x and $\pi(x)$, and propagation graph G_π.

1-$succ(v)$, respectively, denote the nodes pointed by the 0-edge and the 1-edge from node v, and $var(v)$ denotes the label of node v. The family $\mathcal{F}$ of sets represented by a ZDD is the one represented by the root node. Figure 2 is a ZDD representing $\mathcal{F} = \{\{1,2\},\{1,3,4\},\{2,3,4\},\{3\},\{4\}\}$. Each path from the root node to the 1-node, called *1-path*, corresponds to one of the sets in $\mathcal{F}$.

The frontier-based search [11] constructs ZDDs in a top-down manner, and it can be considered as one of DP-like algorithms. We can modify DP algorithms for recognition (i.e., testing whether a given instance satisfies some property) to the frontier-based search algorithm that construct a ZDD representing the family of the yes-instances of the property. Thus, in Section 3, we mainly focus on the method in the form of DP algorithms. The key of the frontier-based search is to share ZDD-nodes by simple "knowledge" of partially given input, and not to traverse the same subproblems more than once. In the context of DP, this means that "internal state" for partially given input should be small. For more details, see [11].

3 Isomorphism Elimination

Let π be a permutation on $\{1,2,\ldots,n\}$, and $\preceq$ be a lexicographical order on $x = (x_n, x_{n-1}, \ldots, x_1) \in \{0,1\}^n$. For any x, we can obtain $\pi(x) = (x_{\pi(n)}, x_{\pi(n-1)}, \ldots, x_{\pi(1)})$, and thus we can define a family $\mathcal{F}_\pi$ of lexicographically larger x's as $\mathcal{F}_\pi = \{x \mid x \succeq \pi(x)\}$. Here, we regard a vector x as a set $\{x_i \mid x_i = 1\}$, which implies that $\mathcal{F}_\pi$ can be regarded as a family of sets $\{x_{i_1}, x_{i_2}, \ldots\}$ $(\subseteq \{x_n, x_{n-1}, \ldots, x_1\})$ that are lexicographically larger than their π-mapped set $\{x_{\pi(i_1)}, x_{\pi(i_2)}, \ldots\}$. Given a set of permutations $\mathrm{Aut}G = \{\pi_1, \pi_2, \ldots\}$, by taking the intersection of $\mathcal{F}_{\pi_1}, \mathcal{F}_{\pi_2}, \ldots$, we can obtain a family of sets each of which is the lexicographically largest on $\mathrm{Aut}G$. By our algorithms described below, we can construct ZDDs of $\mathcal{F}_{\pi_1}, \mathcal{F}_{\pi_2}, \ldots$ in the top-down manner. By combining the top-down construction of the ZDD for spanning trees $\mathcal{F}_s$, we can directly construct the ZDD of their intersection $\mathcal{F}_s \cap \mathcal{F}_{\pi_1} \cap \mathcal{F}_{\pi_2} \cap \cdots$ in the top-down manner [18].

Now, we discuss a DP algorithm for recognizing $\mathcal{F}_\pi$. As illustrated in Figure 3, $x_n, x_{n-1}, \ldots, x_1$ are given on-the-fly. In other words, x_i is given in time slot i $(i = n, n-1, \ldots, 1)$. We compare x and $\pi(x)$, and output 1 if and only if $x \succeq \pi(x)$ holds.

The outline of our algorithm is as follows. The algorithm consists of two phases. In Phase I, $x_n, x_{n-1}, \ldots, x_1$ are given on-the-fly. In the comparison of x and $\pi(x)$, x_i is compared with $x_{\pi(i)}$. In case $i > \pi(i)$, since $x_{\pi(i)}$ will be given in the future, we store x_i in the memory until $x_{\pi(i)}$ is given. On the other hand, in case $i < \pi(i)$, $x_{\pi(i)}$ is already stored in the memory, and thus we can compare x_i and $x_{\pi(i)}$. We transfer the result (denoted as c_i) of the comparison to Phase II. In case $i = \pi(i)$, we compare x_i and $x_{\pi(i)}$, and transfer $c_i :=$ '=' (i.e., equivalent) to Phase II.

In Phase II, the results of the comparisons $C = \{c_n, c_{n-1}, \ldots, c_1\}$ are given from Phase I. Note that the given order of c_i is not $c_n, c_{n-1}, \ldots, c_1$. The order is defined by π. Let π' denote the order of c_i's given to Phase II: c_i's are given in the order of $c_{\pi'(n)}, c_{\pi'(n-1)}, \ldots, c_{\pi'(1)}$. We also note that no c_i may be given in some time slot, and that two c_i and $c_{i'}$ may be given in the same time slot. In Phase II, by checking such c_i's, we conclude whether $x \succeq \pi(x)$ holds or not.

Now, we move to the details of the algorithm. In Phase I, x_i is stored until $x_{\pi(i)}$ appears. At the same time, x_i is required to compare with $x_{\pi^{-1}(i)}$. Thus, precisely speaking, x_i is stored into the memory if $i > \min\{\pi(i), \pi^{-1}(i)\}$ holds, and it is stored until $x_{\min\{\pi(i),\pi^{-1}(i)\}}$ is given. We define *the propagation graph* G_π as the graph $G_\pi = (V, E)$ with $V = \{x_n, x_{n-1}, \ldots, x_1\}$ and $(x_i, x_{\pi(i)}) \in E$. From G_π, we can estimate the memory consumption. (Note that the ordering of the variables is fixed.)

Proposition 3 *To store x_i's in Phase I, w bit is enough, where w is the cut width of the propaga-*

Algorithm 1: Preparation of Phases I and II

Input : n, π
Output: UpdateMemory[ ], cutwidth, Compare[ ]

1 Prepare an empty array until[ ]
2 **for** $i := n, n-1, \ldots, 1$ **do**
3 **if** $i > \min\{\pi(i),\ \pi^{-1}(i)\}$ **then** // It is necessary to store x_i in the memory
4 $k := \begin{cases} \min\{j \mid i \le \text{until}[j]\} & \text{if } \exists j \text{ s.t. } i \le \text{until}[j] \\ (\text{cardinality of until}[\,]) + 1 & \text{otherwise} \end{cases}$
5 until$[k] := \min\{\pi(i),\ \pi^{-1}(i)\}$ // $M[k]$ should be kept until the level of $x_{\pi(i)}$ or $x_{\pi^{-1}(i)}$
6 position$[i] := k$ // x_i is stored in $M[k]$
7 UpdateMemory$[i]$:= UpdateMemory$[i] \cup \{(k, \text{'store'})\}$
8 UpdateMemory[until$[k]$] := UpdateMemory[until$[k]] \cup \{(k, \text{'erase'})\}$
9 cutwidth := cardinality of until[ ]
10 **for** $i := n, n-1, \ldots, 1$ **do**
11 **if** $i > \pi(i)$ **then** // x_i is stored until $x_{\pi(i)}$ is given
12 Compare$[\pi(i)]$:= Compare$[\pi(i)] \cup \{(i, \text{position}[i], \text{'input'})\}$
13 **else if** $i < \pi(i)$ **then** // $x_{\pi(i)}$ is stored until x_i is given
14 Compare$[i]$:= Compare$[i] \cup \{(i, \text{'input'}, \text{position}[\pi(i)])\}$
15 **else** // x_i and $x_{\pi(i)}$ are the same variable
16 Compare$[i]$:= Compare$[i] \cup \{(i, \text{'input'}, \text{'input'})\}$

tion graph G_π with respect to the variable ordering $x_n, x_{n-1}, \ldots, x_1$.

Algorithm 1 summarizes the preparation necessary for Phases I and II. If $i > \min\{\pi(i), \pi^{-1}(i)\}$ holds in Line 3, we plan to store the value of x_i in $M[k]$ and keep $M[k]$ until $x_{\min\{\pi(i),\pi^{-1}(i)\}}$ is given (Lines 4–6). In Line 4, we assign the position k in the first-fit manner. That is, we set the smallest j as k, where $M[j]$ is not used in time slot i. We use a variable-length array until[ ] to indicate that $M[k]$ should be kept until time slot $\min\{\pi(i), \pi^{-1}(i)\}$ (Lines 1 and 5). In Line 4, if no j satisfies $i \le$ until$[j]$, we prepare a new position. In Line 6, position$[i]$ is used to indicate that x_i is in $M[k]$. In Line 7 (respectively, Line 8), we record the plan for storing x_i in $M[k]$ (respectively, erasing $M[k]$). These plans UpdateMemory[ ] are actually executed in Lines 7–11 of Algorithm 2 (Phase I).

The plan for comparing x_i and $x_{\pi(i)}$ is recorded in Lines 10–16, and it is actually executed in Lines 3–6 of Algorithm 2. Compare$[i]$ is a set of the plans for the comparison in time slot i. In case $i > \pi(i)$ (Lines 11 and 12), we keep x_i in $M[\text{position}[i]]$ until $x_{\pi(i)}$ will be given in time slot $\pi(i)$. Thus, we store our plan in Compare$[\pi(i)]$. Plan $(i, \text{position}[i], \text{'input'})$ indicates that c_i (i.e., the comparison of x_i and $x_{\pi(i)}$) can be obtained by comparing $M[\text{position}[i]]$ and 'input' (i.e., $x_{\pi(i)}$) in time slot $\pi(i)$. In case $i < \pi(i)$ (Lines 13 and 14), since we already have $x_{\pi(i)}$ in $M[\text{position}[\pi(i)]]$, we can compare x_i and $x_{\pi(i)}$ in time slot i. Thus, we store our plan in Compare$[i]$. Plan $(i, \text{'input'}, \text{position}[\pi(i)])$ indicates that c_i is obtained by comparing 'input' in time slot i (i.e., x_i) and $M[\text{position}[\pi(i)]]$. Otherwise, since x_i and $x_{\pi(i)}$ are the same variable, we can compare x_i and $x_{\pi(i)}$ in time slot i. We store our plan $(i, \text{'input'}, \text{'input'})$ in Compare$[i]$, where the plan in-

Algorithm 2: Phase I

Input : UpdateMemory[ ], cutwidth, Compare[ ], $x = (x_n, x_{n-1}, \ldots, x_1)$
Output: $(c_n, c_{n-1}, \ldots, c_1)$

1 Prepare an array $M[\,]$ of size cutwidth
2 **for** $i := n, n-1, \ldots, 1$ **do**
3 **foreach** $(i', p_0, p_1) \in$ Compare$[i]$ **do**
4 $m_0 := \begin{cases} x_i & \text{if } p_0 = \text{'input'} \\ M[p_0] & \text{otherwise} \end{cases}$
5 $m_1 := \begin{cases} x_i & \text{if } p_1 = \text{'input'} \\ M[p_1] & \text{otherwise} \end{cases}$
6 $c_{i'} := \begin{cases} \text{'>'} & \text{if } m_0 > m_1 \\ \text{'<'} & \text{if } m_0 < m_1 \\ \text{'='} & \text{if } m_0 = m_1 \end{cases}$
7 **foreach** $(k, \text{behavior}) \in$ UpdateMemory$[i]$ **do**
8 **if** behavior = *'store'* **then**
9 $M[k] := x_i$ // Store x_i in $M[k]$
10 **else** // In case behavior = 'erase',
11 $M[k] := 0$ // erase $M[k]$

Algorithm 3: Phase II

Input : $(c_n, c_{n-1}, \ldots, c_1)$ and a permutation π'

Output: $\begin{cases} 1: & \text{if } x \succeq \pi(x) \\ 0: & \text{otherwise} \end{cases}$

```
1  (i_s, c_{i_s}) := (∞, '=')        // Set the initial state
2  for j := n, n − 1, ..., 1 do
3      i' := π'(j)
4      if i' > i_s then
           // The position of c_{i'} is higher than that of c_{i_s}
5          if c_{i'} ≠ '=' then
6              (i_s, c_{i_s}) := (i', c_{i'})
7      else
           // The position of c_{i_s} is higher than that of c_{i'}
8          if c_{i_s} = '=' and c_{i'} ≠ '=' then
9              (i_s, c_{i_s}) := (i', c_{i'})
10 if c_{i_s} = '>' or '=' then
11     Output 1                    // x ⪰ π(x) holds
12 else
13     Output 0                    // x ⋡ π(x) holds
```

dicates that c_i is obtained by comparing 'input' and 'input' (i.e., both are x_i's) in time slot i.

Algorithm 2 executes the plans in Compare[i] and UpdateMemory[i] in each time slot i. In Line 6, $c_{i'}$ = '>' means $x_{i'} > x_{\pi(i')}$. The notions $c_{i'}$ = '<' and '=' are also defined similarly.

Algorithm 3 describes Phase II. Recall that c_n, c_{n-1}, $\ldots, c_1$ may not be given in this order. For convenience, we introduce permutation π' denoting that c_i's are given in the order of $c_{\pi'(n)}, c_{\pi'(n-1)}, \ldots, c_{\pi'(1)}$. (This ordering is implicitly given by Lines 2 and 3 of Algorithm 2, and thus, it is just for convenience, and we will avoid it later by combining Phases I and II.)

Suppose $i' = \pi'(j)$ as in Line 3 of Algorithm 3. At this moment, we are checking $c_{i'}$. Note that some of the already checked $c_{\pi'(n)}, c_{\pi'(n-1)}, \ldots, c_{\pi'(j)}$ may be in the higher position than $c_{i'}$, and others may be in the lower position than $c_{i'}$. To avoid storing all of them, we use (i_s, c_{i_s}) as an internal state. In case c_{i_s} = '>' (respectively, '<'), all of the already checked c_k's satisfying $k > i_s$ are '=' and already checked c_{i_s} is '>' (respectively, '<'). In this case, if c_k is in the lower position than c_{i_s} (i.e., $k < i_s$), it does not affect the result in the comparison of x and $\pi(x)$. In case c_{i_s} = '=', all of the already checked c_k's are '=', and thus, they do not affect the result in the comparison of x and $\pi(x)$. In Line 1 of Algorithm 3, we set $(i_s, c_{i_s}) := (\infty, '=')$ as an initial state. ($i_s = \infty$ means no c_k's are checked.)

By checking $c_{i'}$, we update (i_s, c_{i_s}): If $c_{i'}$ is in the higher position than c_{i_s} (i.e., $i' > i_s$ holds), $c_{i'}$ is prior to c_{i_s}. Thus, in case $c_{i'}$ is not '=', we store $(i', c_{i'})$ as a new state (Lines 4–6). If c_{i_s} is in the higher position than $c_{i'}$, c_{i_s} is prior to $c_{i'}$. Thus, only in case c_{i_s} is '=' and $c_{i'}$ is not '=', we have a chance to store $(i', c_{i'})$ as a new state (Lines 7–9). After all $c_{i'}$ are checked, we can conclude whether $x \succeq \pi(x)$ holds or not according to the final c_{i_s} (Lines 10–13).

Now, we combine Phases I and II. Line 1 of Algorithm 3 is an initialization of state (i_s, c_{i_s}), and it should be inserted in the beginning of Algorithm 2. Lines 4–9 of Algorithm 3 receive $c_{i'}$, and thus they should be inserted just after Line 6 in the foreach-loop of Algorithm 2. As we mentioned above, we do not need π' since i' in Algorithm 3 is given as the i' in Algorithm 2. Lines 10–13 of Algorithm 3 decide the output according to the final c_{i_s}, and thus they should be inserted just after the last part of Algorithm 2. Given $x_n, x_{n-1}, \ldots, x_1$ on-the-fly, by Algorithm 1 and Algorithm 2+3 (i.e., combined version of Algorithms 2 and 3), we can conclude whether $x \succeq \pi(x)$ holds or not.

In the frontier-based search, we construct ZDDs in a top-down manner. Each node of the resulting ZDD has its internal state (i_s, c_{i_s}) and $M[\,]$. The root node of the resulting ZDD is prepared with $(i_s, c_{i_s}) := (\infty, '=')$. We do not care about $M[\,]$ since we are not given any input x_i. The label of the root node is x_n, which indicates that we are checking x_n. For each i in $\{n, n-1, \ldots, 1\}$ and for each node v labeled x_i, we try both of the cases $x_i = 0$ and 1. In case $x_i = 0$, from node v, we prepare node 0-$succ(v)$. The internal state of 0-$succ(v)$ can be obtained by applying Lines 3–11 of Algorithm 2 (combined with Lines 4–9 of Algorithm 3) to the state of v. We can perform similarly in case $x_i = 1$. If two nodes have the same label x_i and the same internal state, by following the definition of ZDD, we merge the two nodes. From this observation, we can estimate the upper bound on the size of the resulting ZDD. Furthermore, since the execution of Lines 3–11 of Algorithm 2 and Lines 4–9 of Algorithm 3 for each node can be done in constant time, we can also evaluate the time complexity.

Theorem 4 *The size of the resulting ZDD is $O(n^2 2^w)$, where w is the cut width of the propagation graph G_π with respect to the variable ordering $x_n, x_{n-1}, \ldots, x_1$. The computation time for the construction is proportional to the size of the resulting ZDD.*

4 Experimental Results

Experimental results are given in Tables 1 and 2. The computation time is measured on Intel(R) Xeon(R) E7-2830 2.13GHz, 2TB Memory, Red Hat Enterprise Linux Server release 6.6.

In table 1, the developments of 5 Platonic solids and 5 out of 13 Archimedean solids (a cuboctahedron, a truncatedtetrahedron, a truncatedoctahedron, a truncatedcube, and a rhombicuboctahedron) are enumer-

Table 1: Summary of the results for Platonic and Archimedian solids.

Polyhedron	$\|E\|$	|Aut|	# (Labeled Developments)	#Developments	Computation Time (s)		Required Memory (MB)	
					Conventional	Proposed	Conventional	Proposed
Tetrahedron	6	24	16	1	0.01	0.00	30	2
Cube	12	48	384	11	0.02	0.01	30	2
Octahedron	12	48	384	11	0.02	0.01	30	2
Dodecahedron	30	120	5,184,000	43,380	9.10	0.54	529	5
Icosahedron	30	120	5,184,000	43,380	5.73	0.51	282	10
Cuboctahedron	24	48	331,776	6,912	0.35	0.06	36	3
Truncatedtetrahedron	18	24	6,000	261	0.03	0.01	30	2
Truncatedoctahedron	36	48	101,154,816	2,108,512	75.59	2.67	11,192	23
Truncatedcube	36	48	32,400,000	675,585	133.63	2.10	2,078	35
Rhombicuboctahedron	48	48	301,056,000,000	6,272,012,000	$>$ 3 H	1,913.97		11,182

Table 2: Summary of the results for d-dimensional hypercubes.

d	$\|E\|$	|Aut|	# (Labeled Developments)	#Developments	Computation Time (s)		Required Memory (MB)	
					Conventional	Proposed	Conventional	Proposed
2	4	8	4	1	0.02	0.00	36	2
3	12	48	384	11	0.10	0.01	36	2
4	24	384	82,944	261	3.00	0.09	150	2
5	40	3,840	32,768,000	9,694	1166.52	3.96	36,036	10
6	60	46,080	20,736,000,000	502,110	$>$ 3 H	478.39	$>$ 140,000	208

ated. The second column $|E|$ in Table 1 gives the number of edges in the 1-skeleton of a polyhedron. The third column |Aut| gives the number of automorphisms of a polyhedron. The fourth and fifth columns give the number of labeled and nonisomorphic (i.e., unlabeled) developments, respectively. For example, as for a rhombicuboctahedron, we have 301,056,000,000 labeled developments. By checking the graph isomorphism for all of these labeled developments among 48 automorphisms, we obtained 6,272,012,000 nonisomorphic developments. The size of the required memory is summarized in the eighth and ninth column.

As the conventional method, we used the algorithm in [8] combined with the frontier-based search [11] for enumerating labeled developments. The difference between the conventional and our proposed methods is as follows: The algorithm in [8] was proposed before the era of the frontier-based search algorithms. Thus it is necessary to construct the ZDDs of $\mathcal{F}_{\pi_1}, \mathcal{F}_{\pi_2}, \ldots$ completely and then make an intersection of the ZDDs. On the other hand, in our proposed method, we can directly construct the ZDD of the intersection without constructing the intermediate ZDDs of $\mathcal{F}_{\pi_1}, \mathcal{F}_{\pi_2}, \ldots$. The proposed method requires less memory than the conventional method in many cases.

In table 2, the developments of d-dimensional hypercubes are enumerated. Similarly to the case of taking dual of a 3-dimensional polyhedron, we prepare the facet-adjacency graph whose vertices and edges corresponds to the $(d-1)$-dimensional hypercubes and their adjacency of the original hypercube. The facet-adjacency graph of d-dimensional hypercube is a complete d-partite graph with $2d$ vertices and $4\binom{d}{2}$ edges. The automorphism Aut has $2^d\binom{d}{2}$ permutations. Table 2 tells that the proposed method is more than 300 times faster and 3,000 times less memory than the conventional method in case $d = 5$. As for the case $d \geq 6$, we believe the speed-up ratio is more than 300.

5 Conclusion

We have address the issue of the isomorphism elimination by proposing the top-down construction method for the ZDDs of lexicographically largest instances. Experimental results show that the proposed method is more than 300 times faster and 3,000 times less memory than the conventional method in the best case. Our algorithms are applicable to many other enumeration problems with eliminating isomorphic instances.

References

[1] T. J. N. Brown, R. B. Mallion, P. Pollak, B. R. M. de Castro, J. A. N. F. Gomes, The number of spanning trees in buckminsterfullerene, *Journal of Computational Chemistry*, vol. 12, pp. 1118–1124, 1991.

[2] T. J. N. Brown, R. B. Mallion, P. Pollak, A. Roth, Some Methods for Counting the Spanning Trees in Labelled Molecular Graphs, examined in Relation to Certain Fullerenes, *Discrete Applied Mathematics*, vol. 67, pp. 51–66, 1996.

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

[4] F. Buekenhout, M. Parker, The Number of Nets of the Regular Convex Polytopes in Dimension ≤ 4, *Discrete Mathematics*, vol. 186, pp. 69–94, 1998.

[5] E. D. Demaine, J. ORourk, Geometric Folding Algorithms: Linkages, Origami, Polyhedra, Cambridge University Press (2007).

[6] M. Gardner, Mathematical Games: Is It Possible to Visualize a Four-Dimensional Figure?, *Scientific American*, 214, pp. 138–143, 1966.

[7] C. Hippenmeyer, Die Anzahl der inkongruenten ebenen Netze eines regulären Ikosaeders, *Elemente der Mathematik*, vol. 34, pp. 61–63, 1979.

[8] T. Horiyama and W. Shoji, Edge Unfoldings of Platonic Solids Never Overlap, In *Proc. of the 23rd Canadian Conference on Computational Geometry (CCCG 2011)*, pp. 65–70, 2011.

[9] T. Horiyama and W. Shoji, The Number of Different Unfoldings of Polyhedra, In *Proc. of the 24th International Symposium on Algorithms and Computation (ISAAC 2013)*, *Lecture Notes in Computer Science*, 8283, pp. 623–633, Springer-Verlag, 2013.

[10] M. Jeger, Über die Anzahl der inkongruenten ebenen Netze des Würfels und des regulären Oktaeders, *Elemente der Mathematik*, vol. 30, pp. 73–83, 1975.

[11] J. Kawahara, T. Inoue, H. Iwashita, and S. Minato. Frontier-based Search for Enumerating All Constrained Subgraphs with Compressed Representation. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol. E100-A, no. 9, pp. 1773–1784, 2017.

[12] G. Kirchhoff, Über die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Verteilung galvanischer Ströme gefuhrt wird, Annalen der Physik und Chemie, 72, pp. 497–508, 1847.

[13] D. E. Knuth, The Art of Computer Programming, vol. 4, fascicle 1, Bitwise Tricks & Techniques, Binary Decision Diagrams, Addison-Wesley (2009).

[14] E. M. Luks, Isomorphism of graphs of bounded valence can be tested in polynomial time, Journal of Computer and System Sciences, 25 (1), pp. 42–65, 1982.

[15] A. Lubiw, Some NP-complete problems similar to graph isomorphism, SIAM Journal on Computing, 10 (1), pp. 11–21, 1981.

[16] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of the 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 272–277, 1993.

[17] G. Pólya, Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen, Acta Mathematica, 68 (1), pp. 145–254, 1937.

[18] TdZdd: A top-down/breadth-first decision diagram manipulation framework, `https://github.com/kunisura/TdZdd`

[19] P. D. Turney, Unfolding the Tesseract, *Journal of Recreational Mathematics*, 17 (1), pp. 1–16, 1984–85.