Vol.2018-AL-167 No.7
2018/3/8

Isomorphism Elimination by Zero-Suppressed Binary Decision Diagrams

Takashi Horiyama[2,a)] Masahiro Miyasaka[2,b)] Riku Sasaki[1,c)]

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, both of the given and resulting families of subgraphs are in the form of ZDDs, and the computation are performed on ZDDs. Experimental results show that the proposed method is 300 times faster and 3,000 times less memory than the conventional method in the best case.

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 as in Fig. 1(a), we can obtain the development in Fig. 1(c). When we rotate the positions of cutting edges by 90 degrees, i.e., by cutting along the set of edges $\{e_1, e_3, e_4, e_7, e_9, e_{11}, e_{12}\}$, we can also obtain the development in Fig. 1(c). Are these the same? If we assume the edges are *labeled*, the positions of cutting edges are different, and thus we can say they are different. If we assume 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 (we identify mirror shapes as isomorphic). In [4], a technique for counting the number of nonisomorphic developments of any polyhedron (including nonconvex polyhedron) is given. They also listed the number of labeled and nonisomorphic 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, while a truncated icosahedron (a Buckminsterfullerene, or a soccer ball fullerene) has 375,291,866,372,898,816,000 (approximately 3.75×10^{20}) labeled developments, it has 3,127,432,220,939,473,920 (approximately 3.13×10^{18}) nonisomorphic developments. A truncated icosidodecahedron has $21,789,262,703,685,125,511,464,767,107,171,876,864,000$ (approximately 2.18×10^{40}) labeled developments, and has $181,577,189,197,376,045,928,994,520,239,942,164,480$ (approximately 1.82×10^{38}) nonisomorphic developments. We here note that the technique in [4] counts the number of nonisomorphic developments without enumerating developments.

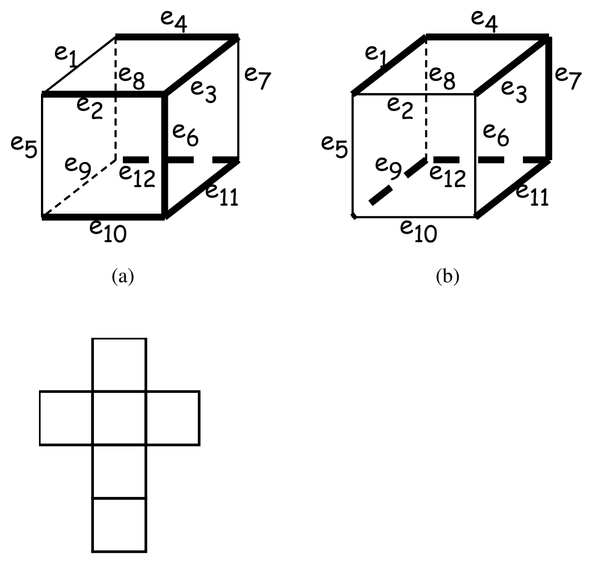

Fig. 1 Different cut edges (a) and (b) have isomorphic developments.

As for the enumeration of nonisomorphic developments, a technique using BDDs (Binary Decision Diagrams) is given in [3]. A BDD [1] is a graph representation of a family of sets. 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 (See, e.g., [[2], Lemma 22.1.1]), and vice versa. In [3], they constructed a BDD corresponding to a family of labeled developments, where each development corresponds to a spanning tree represented by a set of labeled edges. Then, by omitting mutually isomorphic developments, they obtained nonisomorphic developments.

Later, a sophisticated method called a "frontier-based

1 Faculty of Engineering, Saitama University, Japan
2 Graduate School of Science and Engineering, Saitama University, Japan
a) horiyama@al.ics.saitama-u.ac.jp
b) m.miyasaka.020@ms.saitama-u.ac.jp
c) r.sasaki.297@ms.saitama-u.ac.jp

search" [5] is proposed for constructing BDDs/ZDDs representing all constrained subgraphs. A ZDD (Zero-suppressed Binary Decision Diagram) [3] is a variant of BDDs, and also represents a family of sets. The frontier-based search is an extension of Simpath algorithm [6] by Knuth for enumerating all st-paths 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 [3], we can speed-up the construction of the BDD/ZDD representing a family of spanning trees.

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 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, 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 compactly represent a family of sets, (2) the enumeration by ZDDs are faster than other methods 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 [8]. Fortunately, however, we can solve the problem in polynomial time if the degrees of vertices in a graph graph are bounded by a constant [7].

Our main issue is to select the lexicographically largest subgraphs on any of the automorphisms. In [3], they constructed BDDs $G_1, G_2, \ldots$, where G_i represents a family of the lexicographically largest subgraphs on automorphism π_i, and then took the intersection of the BDDs for selecting a family of subgraphs that appear in all of the families of $G_1, G_2, \ldots$. Since the method was proposed before the era of the frontier-based search algorithms, similarly to the BDD/ZDD algorithms in those days, it obtains the resulting BDD by the repetition of 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)* [9] is directed acyclic graph that represents a family of sets. As illustrated in Fig. 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 $x_1, x_2, \ldots, x_n$, 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.

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

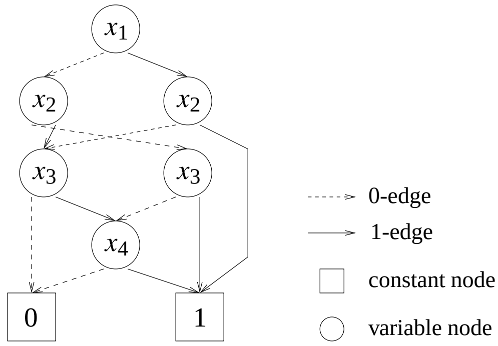

Fig. 2 A ZDD representing $\{\{1,2\},\{1,3,4\},\{2,3,4\},\{3\},\{4\}\}$.

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-*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. Fig. 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 [5] 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-iinstances. Thus, in Section 3, we only show our algorithms as 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 [5].

3. Isomorphism Elimination

Let *pi* 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 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 Aut$G$. Later in this section, we discuss a DP algorithm that outputs 1 if and only if input x satisfies $x \succeq \pi(x)$.

The outline of the algorithm is as follows. The algorithm consists of two phases. In Phase I, $x_n, x_{n-1}, \ldots, x_1$ are given

Algorithm 1: Preparation of Phases I and II

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

Prepare an empty array until[ ]
for $i := n, n-1, \ldots, 1$ **do**
  if $i > \min\{\pi(i), \pi^{-1}(i)\}$ **then**
    // It is necessary to store x_i in the memory
    $k := \begin{cases} \min\{j \mid i \leq \text{until}[j]\} & \text{if } \exists j \text{ s.t. } i \leq \text{until}[j] \\ (\text{cardinality of until[ ]}) + 1 & \text{otherwise} \end{cases}$
    position[i] := k // x_i is stored in $M[k]$
    until[k] := $\min\{\pi(i), \pi^{-1}(i)\}$
    // $M[k]$ should be kept until the level of $x_{\pi(i)}$ or $x_{\pi^{-1}(i)}$
    UpdateMemory[i] := UpdateMemory[i] ∪ {(k, ‘store’)}
    UpdateMemory[until[k]] :=
      UpdateMemory[until[k]] ∪ {(k, ‘erase’)}
cutwidth := cardinality of until[ ]
for $i := n, n-1, \ldots, 1$ **do**
  if $i > \pi(i)$ **then** // x_i is stored until $x_{\pi(i)}$ is given
    Compare[$\pi(i)$] := Compare[$\pi(i)$] ∪ {(i, position[i], ‘input’)}
  else if $i < \pi(i)$ **then** // $x_{\pi(i)}$ is stored until x_i is given
    Compare[i] := Compare[i] ∪ {(i, ‘input’, position[$\pi(i)$])}
  else // x_i and $x_{\pi(i)}$ are the same variable
    Compare[i] := Compare[i] ∪ {(i, ‘input’, ‘input’)}

Algorithm 2: Phase I

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

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

Algorithm 3: Phase II

Input : $(c_n, c_{n-1}, \ldots, c_1)$ and a permutation π'
Output: $\begin{cases} 1: & \text{if } x \geq \pi(x) \\ 0: & \text{otherwise} \end{cases}$

$(i_s, c_{i_s}) := (\infty, \text{‘}=\text{’})$
// Set the initial state
for $j := n, n-1, \ldots, 1$ **do**
  $i' := \pi'(j)$
  if $i' > i_s$ **then**
    // The position of $c_{i'}$ is higher than that of c_{i_s}
    if $c_{i'}$ = ‘>’ *or* ‘<’ **then**
      $(i_s, c_{i_s}) := (i', c_{i'})$
  else
    // The position of c_{i_s} is higher than that of $c_{i'}$
    if c_{i_s} = ‘=’ **then**
      $(i_s, c_{i_s}) := (i', c_{i'})$
if c_{i_s} = ‘>’ *or* ‘=’ **then**
  Output 1
  // $x \geq \pi(x)$
else
  Output 0
  // $x \ngeq \pi(x)$

on-the-fly. In other words, x_i is given in time slot $n + 1 - i$ ($i = n, n-1, \ldots, 1$). 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 of the comparison c_i to Phase II. In case $i = \pi(i)$, we compare x_i and $x_{\pi(i)}$, and transfer c_i is '=' (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 $x \geq \pi(x)$ 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, precesely 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. The amount of memory to store x_i's is the cut width of the graph $G = (V, E)$ where $V = \{x_n, x_{n-1}, \ldots, x_1\}$ and $(x_i, x_{\pi(i)}) \in E$.

Algorithm 1 summarizes the preparation necessary for Phases I and II. If $i > \min\{\pi(i), \pi^{-1}(i)\}$ holds in Line 3, we plann to store x_i in $M[k]$ and keep $M[k]$ until $x_{\min\{\pi(i),\pi^{-1}(i)\}}$ is given (Lines 4–6). In Lines 7 and 8, we record the plan for storing/erasing $M[k]$, and the plan UpdateMemory[[]i] is 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.

Algorithm 3 describes Phase II. Recall that $c_n, c_{n-1}, \ldots, c_1$ may not be given in this order. For convinience, we introduce permutation π' for denoting the ordering $c_{\pi'(n)}, c_{\pi'(n-1)}, \ldots, c_{\pi'(1)}$. (The ordering is implicitly given by Lines 2 and 3 of Algorithm 2, and thus, it is just for convinience, and we will avoid it by combining Phases I and II.) By updating (i_s, c_{i_s}) in Lines 4–9, we can check whether $x \geq \pi(x)$ holds or not in Lines 10–13 (Details are omitted).

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 begining 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. Lines 10–13 of Algorithm 3 decide the output according to the final c_{i_s}, and thus they should be inserted after the last part of Algorithm 2.

4. Experimental Results

Experimental results are given in Tables 4 and 4. In table 4, the developments of 5 Platonic solids and 5 out of 13 Archimedean solids (a cuboctahedron, a truncatedtetrahedron, a truncatedoctahedron, a truncatedcube, a rhombicuboctahedron) are enumerated. A development of a polyhedron is a simple polygon obtained by cutting along the edges of the polyhedron and unfolding it into a plane. 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 (See, e.g., [[2], Lemma 22.1.1]). The second column $|E|$ in Table 4 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 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 unlabeled developments. The size of the required memory is summarized in the eigth and ninth column. The proposed method requires less memory than the conventional method. As for a rhombicuboctahedron, however, requires much memory even for the proposed method.

In table 4, the developments of n-dimensional hypercubes are enumerated. As in the case of a cube (i.e., $n = 3$), given an unfolding of a n-dimensional hypercube, we can define its dual whose vertices and edges corresponds to the $(n-1)$-dimensional hypercubes and their adjacency of the original hypercube. The dual has $2n$ vertices and $4\binom{n}{2}$ edges. The automorphism Aut has $2^n\binom{n}{2}$ permutations.

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 300 times faster and 3,000 times less memory than the conventional method in the best case.

References

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

[2] E. D. Demaine, J. O 'Rourk, Geometric Folding Algorithms: Linkages, Origami, Polyhedra, Cambridge University Press (2007).

[3] 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.

[4] 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.

[5] 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).

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

[7] 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).

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

[9] 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.

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

Polyhedron	$\|E\|$	$\|Aut\|$	#(Labeled Unfoldings)	#Unfoldings	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		1,913.97		11,182

Table 2 Summary of the results for n-dimensional hypercubes.

n	$\|E\|$	$\|Aut\|$	#(Labeled Unfoldings)	#Unfoldings	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