Cube-based Isomorph-free Finite Model Finding

Choiwah Chow[a,*], **Mikoláš Janota**[a,**] **and João Araújo**[b,***]

[a]Czech Technical University in Prague, Czech Republic
[b]Universidade Nova de Lisboa, Lisbon, Portugal
ORCID (Choiwah Chow): https://orcid.org/0000-0002-2067-0568, ORCID (Mikoláš Janota): https://orcid.org/0000-0003-3487-784X, ORCID (João Araújo): https://orcid.org/0000-0001-6655-2172

Abstract. Complete enumeration of finite models of first-order logic (FOL) formulas is pivotal to universal algebra, which studies and catalogs algebraic structures. Efficient finite model enumeration is highly challenging because the number of models grows rapidly with their size but at the same time, we are only interested in models modulo isomorphism. While isomorphism cuts down the number of models of interest, it is nontrivial to take that into account computationally.

This paper develops a novel algorithm that achieves isomorphism-free enumeration by employing isomorphic graph detection algorithm nauty, cube-based search space splitting, and compact model representations. We name our algorithm cube-based isomorph-free finite model finding algorithm (CBIF). Our approach contrasts with the traditional two-step algorithms, which first enumerate (possibly isomorphic) models and then filter the isomorphic ones out in the second stage. The experimental results show that CBIF is many orders of magnitude faster than the traditional two-step algorithms. CBIF enables us to calculate new results that are not found in the literature, including the extension of two existing OEIS sequences, thereby advancing the state of the art.

1 Introduction

Enumerating finite algebras is a very important task in universal algebra. E.g., after observing that there is only one group of prime order, it is easy to conjecture, and then prove, that there is only one group of order (size) p, for prime p. Model enumeration has mostly been performed by the two-step direct-search model enumeration procedure, which is plagued by isomorphic models. Intuitively, two models are isomorphic if one model can be transformed to the other by renaming its domain elements. More formally, two models are isomorphic if one can be transformed to the other by an isomorphism, which is a structure preserving bijective function. That is, isomorphism is an equivalence relation that partitions the set of models into equivalence classes. We need only one member from each of these equivalence classes, and the rest are noise that need to be removed. In our example of group of prime order, if isomorphic models are allowed in model enumeration, there will be more than one group for each order. We may miss the hint that there is only one group of any prime order. Furthermore, in many instances, it is crucial for mathematicians to possess not just the number, but the complete list of all non-isomorphic models of a certain size n within a given theory. The guarantee that no models have been omitted makes these libraries invaluable, as it enables mathematicians to draw definitive conclusions from testing conjectures on them. Consequently, significant effort is devoted to creating and maintaining these comprehensive libraries. Libraries such as `SmallGrp` [38], `Smallsemi` [16], and `Loops` [29] provide comprehensive lists of groups, semigroups, and other algebraic structures, ensuring that mathematicians have access to complete datasets. All these libraries have been produced with a combination of deep theorems and dedicated computational tools, which are not always available. Thus, mathematicians need simple and efficient tools that are not dedicated to just a few structures to produce libraries of all non-isomorphic models of a given theory for any given size. However, isomorphic models are often generated in large proportions during the first step of the non-dedicated model search process, and it is difficult to identify which pairs of models are isomorphic. For example, the traditional model generator Mace4 generates over 423 million models for the Involutive Lattices of order 13, belonging to only 8,028 isomorphism classes. That is, less than 0.002% of the models generated are non-isomorphic to each other. It takes the computer in our experiments (see Section 6) 2.5 days to generate these 423+ million models according to the definition of Involutive Lattices (the first step) and many more hours to filter out the isomorphic models (the second step). *In this paper we propose to generate the non-isomorphic models directly without the intermediate outputs, which take time to generate, require disk space to hold, and more time to filter out the redundant isomorphic models.*

If the model search process can be organized as a search tree, then a common strategy is to prune this search tree as much as possible and as early as possible during the search process. We can prune a branch of the search tree if we know that it does not produce any model that is not isomorphic to those already found in other searched branches. If we prune all the isomorphic branches during the search process, then we have an isomorph-free model search algorithm. We will introduce in Section 5 the *cube-based isomorph-free finite model search algorithm* (`CBIF`), which efficiently identifies and removes isomorphic branches and models during the search process.

Fast isomorph-free algorithms are paramount in computational universal algebra. However, as previously pointed out, few algebras have specialized algorithms for generating models. `CBIF`, on the other hand, provides mathematicians with a generic tool that enables enumerating models beyond the state of the art.

The main contributions of this paper are the following:

1. We introduce a novel approach that applies the `nauty` algo-

* Choiwah Chow. Email: choiwah.chow@gmail.com
** Mikoláš Janota. Email: mikolas.janota@gmail.com.
*** João Araújo. Email: jjrsga@gmail.com

rithm [27] in conjunction with a data encoding scheme and a hash table for fast detection of isomorphic cubes (Section 2.1) and models.
2. Based on the new isomorphic cubes/model detection algorithm, we devise a novel cube-based isomorph-free finite model searching algorithm, `CBIF`.
3. We provide an implementation of the algorithm, which enables us to calculate new results that are not yet reported in the literature .

2 Preliminaries

We will be concerned only with models definable in first-order logic (FOL) with equation. We follow well-established terminology, cf. [3, 8, 40], in this paper. Models are defined by a signature Σ and a FOL formula $\mathcal{F}$ on Σ, which may contain k-ary function symbols, and m-ary predicate (or relations) symbols, where $k, m \in \mathbb{N}_0$. Nullary functions are called *constants*. Predicates are treated as functions with the values `True` and `False`. So our description on functions applies to predicates as well. The domain is denoted by the set $D = \{0, \ldots, n-1\}$, where $n \geq 2$. The size of D is called the *order* of the model.

2.1 Finite Model Enumeration

A traditional direct-search finite model finder first expands the FOL formula to its ground representation by its domain elements in D. Then it searches for models by assigning values to the functions and predicates in $\mathcal{F}$, backtracking to the previous step upon failure, or when all values are exhausted, and retrying with different values.

A *value assignment (VA) clause* is a term $f(a_1, \ldots, a_k) = v$, where f is a k-ary function symbol in Σ and $a_j, v \in D$. The term $f(a_1, \ldots, a_k)$ is called a *cell term* (or simply *cell*). Conceptually, the finite model finder fills the cells of the operation table of f with values one by one.

To search for models in $\mathcal{F}$, the direct-search finite model finder employs a *function selection* strategy in conjunction with a *cell selection* strategy to pick cell terms successively, without duplicates, to assign values from D to form VA clauses.

Example 1. *Suppose we are searching for models of order n with just one binary operation f. Then a possible cell selection strategy is to order the cells by their first function arguments, then by their second function arguments. So, the cells will be selected in the following order: $f(0,0), f(0,1), \cdots, f(0,n-1), f(1,0), \cdots, f(n-1,n-1)$. The traditional finite model finder assigns values from D to the cells in the list successively. If all cells are assigned values without violating the axioms set forth in $\mathcal{F}$, then a model is found.* □

The cell selection strategy has big impacts on the performance of the finite model finite, but it is beyond the scope of this paper to discuss it in details.

After a value is assigned, a finite model finder may optionally do propagation. There are two kinds of propagation. Positive propagation allows the inference of values for some empty cells. Negative propagation excludes the possibility of some values being assigned to some empty cells. Both kinds of propagation speed up the search process.

Example 2. *(Positive Propagation) Suppose the FOL formula contains the equation $f(x,y) = f(y,x)$, that is, the operation f is commutative. After the assignment $f(a,b) = c$, where $a,b,c \in D$, the finite model finder can infer $f(b,a) = c$ and make the assignment immediately.* □

Example 3. *(Negative Propagation) Suppose the FOL formula contains the inequality $f(x,y) \neq f(y,x)$, then after the assignment $f(a,b) = c$, where $a,b,c \in D$, the finite model finder does not need to try to assign c to the cell $f(b,a)$. If all but one domain elements are eliminated by negative propagation for a cell, then the finite model finder can assign the only remaining value to that cell.* □

After the optional propagation step, the axioms in $\mathcal{F}$ are checked against this new VA clause and any other propagated VA clauses. If any of the axioms are violated, then the VA clause, along with any propagated assignments, are reverted. A new value will be tried for that cell and the process continues. If no value can be assigned to that cell term without failing the axioms in $\mathcal{F}$, then the model finder backtracks to the previous cell to try to assign another value to it. When all cell terms in $\mathcal{F}$ are assigned values and the axioms hold, a model represented by its VA clauses is found. After that, the process may continue with backtracking to find more models.

The models produced this way often contain a lot of redundancies. If a model can be transformed to another model by renaming its domain elements, then one of them is redundant because it offers no new information about the underlying structure. Formally, renaming domain elements is achieved by applying an isomorphism (a structure-preserving bijective function) to the models. Two models are said to be isomorphic to each other if an isomorphism exists from one model to the other. Note that the permutation cycle (a,b), denoted by $\pi_{(a,b)}$, can be used as a function for renaming a to b and vice versa. For example, all three models in Figure 1 are isomorphic. Model A can be transformed to model B by $\pi_{(2,3)}$, and model B can be transformed to model C by $\pi_{(1,3)}$. C turns out to be the lexicographically smallest model in the equivalence class containing these three models, and is usually the preferred model to keep [20].

Figure 1: Isomorphic models A, B, and C

Model A

$*_A$	**0**	**1**	**2**	**3**
0	0	1	0	3
1	1	2	1	2
2	2	1	2	1
3	3	0	3	0

Model B

$*_B$	**0**	**1**	**2**	**3**
0	0	1	2	0
1	1	3	3	1
2	2	0	0	2
3	3	1	1	3

Model C

$*_C$	**0**	**1**	**2**	**3**
0	0	0	2	3
1	1	1	3	3
2	2	2	0	0
3	3	3	1	1

Another way to look at the direct-search model finding process is through a search tree. The search space can be organized as a search tree in which nodes are VA clauses and edges join successive nodes with cell terms in the search order. The root node is an empty VA clause. A *search path* in a search tree is a path from the root to a node in the search tree. It can be represented by a sequence of VA clauses $\langle t_0 = v_0; t_1 = v_1; \cdots \rangle$, where t_i is the cell term in the ith position of the sequence and $v_i \in D$. Furthermore, $t_i \neq t_j$ when $i \neq j$. A search path is terminated at the first VA clause that results in a violation of any axiom of $\mathcal{F}$.

A useful notation here is the *cube*, which is defined as the prefix of a search path. Cubes facilitate massive parallelization (e.g., [3]) although parallelization is not attempted in this version of `CBIF` that follows the recorded object paradigm. Being a prefix of a search path, a cube can be specified by a sequence of VA clauses. Conceptually, the finite model finder extends the cube one VA clause at a time until a model is found. Isomorphisms, and permutations, can be applied to a cube by applying them to its VA clauses. Specifically, if π is a permutation on D, and B is a cube, then $\pi(B) := \{f(\pi(a_1), \ldots, \pi(a_k)) = \pi(v) \mid f(a_1, \ldots, a_k) = v$ is a VA clause in B$\}$.

We define $\pi(b) = b$ for $b \in \{$`True`, `False`$\}$ for any isomor-

phism π since `True` and `False` need to be preserved. We use π_{id} to denote the identity permutation. Observe that $\pi_{id}(B)$ is the set of all individual VA clauses in the cube B.

Cubes are said to be isomorphic to each other if their VA clauses are isomorphic to each other. In particular, two cubes B_0 and B_1 are isomorphic if there is a permutation π on D such that $\pi(B_0) = \pi_{id}(B_1)$.

Example 4. *If $B_0 = \langle f(0) = 0; g(0,0) = 0; f(1) = 0\rangle$ and $B_1 = \langle f(0) = 1; g(0,0) = 1; f(1) = 1\rangle$, then B_0 and B_1 are not isomorphic because no isomorphism can be found to transform B_0 to B_1. However, extending each cube with one more VA clause s.t. $B_0 = \langle f(0) = 0; g(0,0) = 0; f(1) = 0; g(1,1) = 0\rangle$ and $B_1 = \langle f(0) = 1; g(0,0) = 1; f(1) = 1; g(1,1) = 1\rangle$, then B_0 and B_1 are isomorphic because $\pi_{(0,1)}(B_0) = \{f(1) = 1, g(1,1) = 1, f(0) = 1, g(0,0) = 1\} = \pi_{id}(B_1)$.* □

3 Graph Isomorphism

A cornerstone of our `CBIF` algorithm is the fact that isomorphic cubes extend to isomorphic models [3, Theorem 9]. Thus, a cube can be excluded from the search if it is isomorphic to another cube already covered. We incorporate the graph isomorphism checking tool `nauty` [27] into `CBIF` to speed up the isomorphic cubes/models detection process. Since `nauty` works on graphs, we need to first construct a graph corresponding to the cube/model before `nauty` can be applied. Graph construction algorithm for models are only briefly, and often incompletely, described in the literature, e.g., [22, 28]. So, we give a complete description of our version of the procedure, including how relations are handled, before we describe how to extend it to also handle cubes.

Suppose the model M is defined by a signature Σ and a FOL formula $\mathcal{F}$ on Σ. Suppose further that $\mathcal{F}$ consists of a collection of functions f_i of arity k_i, and relations r_j of arity m_j, where i, j, k_i, and m_j are non-negative integers. Let q be the highest arity of functions and relations in $\mathcal{F}$. G then comprises a collection of sets of vertices and a collection of edges between some of these vertices. Furthermore, each set of vertices is of a different color. `Nauty` does not rename vertices with different colors to each other so as to prevent the renaming of, for example, vertices representing cell terms to vertices representing function values. First, some notations:

Notation 1. *Let X be a set of vertices in G. Then X_d denotes the vertex in X associated with the domain element d.*

Notation 2. *A_p denotes the set of vertices for the pth argument of the functions and relations in $\mathcal{F}$.*

Thus, A_{p_d} denotes the vertex in A_p corresponding to the domain element d.

Notation 3. *Suppose f_i is a function or relation in $\mathcal{F}$ and $f_i(a_1, \cdots, a_{k_i})$ is a cell, then V_{f_i} denotes the set of vertices for the ith function or relation (in any fixed order) in $\mathcal{F}$, and $V_{f_i(a_1,\cdots,a_{k_i})}$ denotes the vertex of the cell $f_i(a_1, \cdots, a_{k_i})$.*

The vertices of G are constructed as follow:

1. For each domain element d add a vertex E_d to the set of vertices E of G.
2. For each domain element d and each p, where $1 \leq p \leq q$, add a vertex A_{p_d} to the set of vertices A_p.
3. For each domain element d, add a vertex R_d to the set of vertices R. These vertices represents function values.
4. Add to G two vertices B_T and B_F, each with a distinct color from the rest of the graph.
5. For each cell term, $f_i(a_1, \cdots, a_{k_i})$, add a vertex $V_{f_i(a_1,\cdots,a_{k_i})}$ to the set of vertices V_{f_i}.
6. For each cell term, $r_j(a_1, \cdots, a_{m_j})$, add a vertex $V_{r_j(a_1,\cdots,a_{m_j})}$ to the set of vertices V_{r_j}.

The purpose of the edges is to connect directly or indirectly all vertices corresponding to the same domain element, so that if a domain element is renamed by an isomorphism, then all its vertices are renamed accordingly because of the connections. So, we add edges to G the following way:

1. Add an edge between each pair of vertices E_d and A_{p_d}, where $1 \leq d \leq n$ and $1 \leq p \leq q$.
2. Add an edge between each pair of vertices E_d and R_d, where $1 \leq d \leq n$.
3. For functions: For each VA clause, $f_i(a_1, \cdots, a_{k_i}) = d$:
   (a) add an edge between $V_{f_i(a_1,\cdots,a_{k_i})}$ and R_d.
   (b) add an edge between each pair of the vertices $V_{f_i(a_1,\cdots,a_{k_i})}$ and A_{p_d}, where $1 \leq d \leq n$.
4. For VA clause $r_j(a_1, \cdots, a_{m_j}) = B$ on predicate r_j:
   (a) add an edge between the vertices $V_{r_j(a_1,\cdots,a_{m_j})}$ and B_T (if $B =$ `True`) or B_F (if $B =$ `False`).
   (b) add an edge between each pair of the vertices $V_{r_j(a_1,\cdots,a_{m_j})}$ and A_{p_d}, where $1 \leq d \leq n$.

Example 5. *Take the model M of order 2 defined by one binary operation f: $\langle f(0,0) = 0, f(0,1) = 1, f(1,0) = 0, f(1,1) = 1\rangle$. Figure 2 shows the operation table for f. Its graph G constructed from the procedure described above is illustrated in Figure 4. `Nauty` uses consecutive non-negative integers (starting from zero) to represent vertices, and edges are represented a pair of vertices. The vertices of the graph generated by the above procedure are: $E = \{0, 1\}$, $A_1 = \{2, 3\}$, $A_2 = \{4, 5\}$, $R = \{6, 7\}$, and $V_f = \{8, 9, 10, 11\}$. Each set E, A_1, A_2, R, and V_f is given a different color. The edges are represented by their end points, as shown in Figure 3. There are 12 vertices and 18 edges in the graph. `Nauty` prints out an undirected edge (v_1, v_2) only when $v_1 < v_2$, and prints v_1 only once for all undirected edges connected to it, as illustrated in Figure 3. For example, the first line on the left side of the figure states that there is an edge between each of the pairs of vertices 0 and 4, 0 and 2, and 0 and 6.*

Let's use the VA clause $f(0,1) = 1$ to illustrate how the edges are constructed. This VA clause is represented by the yellow square (vertex) on the first row, second column of V_f. It has an edge to the vertex marked as "0" in A_1 because the first argument of f is 0. It has an edge to the vertex marked as "1" in A_2 because the second argument of f is 1. Finally, it has an edge to the vertex marked as "1" in R because the value assigned to the cell term is 1. □

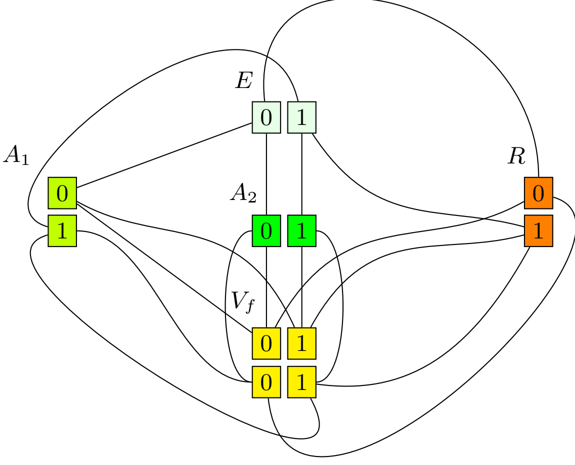
Figure 2: Operation Table of M

f	**0**	**1**
0	0	1
1	0	1

Figure 3: Vertices of M

0 : 4 2 6
1 : 5 3 7
2 : 8 11
3 : 9 10
4 : 8 9
5 : 10 11
6 : 8 10
7 : 9 11
8 :
9 :
10 :
11 :

Figure 4: Graph of Model M



The steps in constructing the graph G for a cube is exactly the same as those for a model, except that an additional vertex U corresponding to the *unassigned* value is needed because not all cells are assigned values. We also need to add an edge between U and each cell, $f_i(a_1, \ldots, a_{k_i})$ or $r_j(a_1, \ldots, a_{m_j})$, that is not yet assigned a value. Note that U does not have any edge to vertices in E, R, B_T, B_F, or A_p because it is not a domain element or a truth value.

Example 6. *For the model M in Example 5, there are exactly 3 edges coming out of each vertex in its graph G (see Figure 4). So there are $3 * 12/2 = 18$ edges for the graph of M of 12 vertices. A complete undirected graph with 12 vertices has $12(12-1)/2 = 66$ edges. So G is a sparse graph. In fact, the graph of the model becomes sparser as the order goes higher. Thus,* `nauty` *has to be run in the sparse mode for the best performance.*

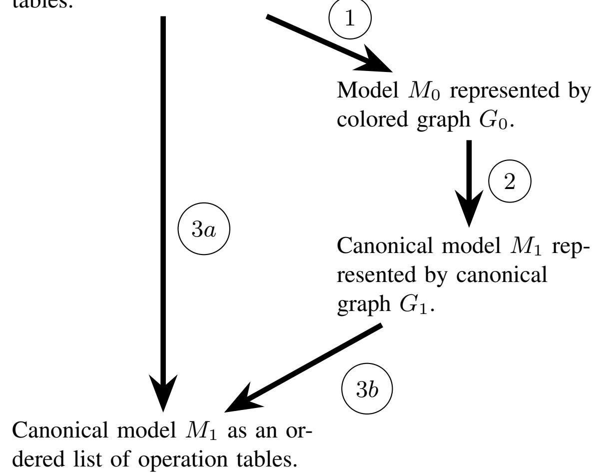
4 Checking for Isomorphisms

Given two models or cubes, we can construct their representations in graphs and apply `nauty` to them to find their canonical graphs. Models/Cubes are isomorphic if and only if they have the same canonical graph. However, a direct comparison of the canonical graphs requires $n(n-1)/2$ comparisons for n graphs in the worse case scenario. The canonical graphs are also quite big, and needs to stay in the memory for fast comparisons. We use a hashing scheme and a special compact representation of models to mitigate these two pitfalls.

Example 5 shows that the vertices (Figure 3) representing a model require much more space than the operations tables. In that example, the binary operation table (Figure 2) can be written out as a sequence of 4 numbers, e.g., as an ASCII string "0101". The graph, on the other hand, contains 30 numbers. There are ways to reduce the size of the graph, e.g., by omitting the left-most column of numbers as they are simply an increasing sequence of consecutive integers. However, it is still much bigger than the 4 numbers in the binary operation table.

To cut down the memory footprint of `CBIF`, we work with the model's entities (functions, relations etc.) in identifying isomorphic models and cubes. The steps are illustrated in Figure 5. Given a model M_0, we construct its corresponding graph G_0 using the algorithm given in Section 3 (Step 1 in Figure 5). We then apply `nauty` to G_0 to get its canonical representation G_1 (Step 2). `Nauty` also outputs the isomorphism I from G_0 to G_1. Let M_1 be the model that gives G_1 when the algorithm given in Section 3 is applied to it. So we can apply I to M_0 to arrive at the model M_1 (the step marked as 3a and 3b). Note that the graph G_1 of model M_1 is in its canonical representation (by construction). We call M_1 the *canonical model* of M_0. All models having the same canonical model are isomorphic to each other. We can therefore use M_1 instead of G_1 to identify isomorphism in models.

Figure 5: Model Dataflow



For any model M_0, we use the encoded operation tables of its canonical model M_1 as its key in hashing. Since cubes are just partial models, so the same algorithm apply in isomorphism of cubes. Now a cube/model can be encoded as an ordered list of operation tables, and each operation table can be encoded as an ordered list of cell values as fixed-width numbers. The output of this encoding scheme is just one long sequence of numbers. For example, the model in Example 5 could be represented by the 4-byte string "0101", or by any other fixed width sequence of numeric values such as a bit vector. This compact encoding scheme greatly reduces the footprint of the memory used for isomorphic models/cubes detection, which is important in recorded object algorithms.

Example 7. *Consider the 2 models D and E as shown in Figure 6. Suppose the graphs of D and E are G_D and G_E, respectively.*

Figure 6: Models D and E

Model D

$*_D$	**0**	**1**
0	0	0
1	0	1

Model E

$*_E$	**0**	**1**
0	0	1
1	1	1

When `nauty` *is applied to G_D, we obtain the canonical graph G, and also the identity permutation. So, the canonical model of D is D itself. When* `nauty` *is applied to the graph G_E, we obtain the same canonical graph G, and the permutation $\pi_{(0,1)}$. Applying $\pi_{(0,1)}$ to the model E gives us the model D. That is, the canonical model of both D and E are the same, and we therefore conclude that D and E are isomorphic. This example shows that we can compare canonical models instead of canonical graphs for isomorphism. The advantages of using canonical models are that they are much smaller and faster to compare than their corresponding canonical graphs.*

5 Isomorph-free Model Finding

`CBIF` searches for models by forming VA clauses one by one as in any direct-search finite model finder. During its traversal of the

search tree, it records the canonical cubes and models in a hash table. It discards the cube or model if its canonical version is already in the hash table. It thus avoids extending any cube that is isomorphic to a cube that has previously been explored. We know that this shortcut does not lose any models because isomorphic cubes only lead to isomorphic models, which have to be eliminated [3].

Algorithm 1 summarizes the approach. For simplicity, it is written as a depth-first search algorithm, but breath-first search or any other traversal order also works. It enumerates all models, but it can be changed to stop after finding one or any specified number of models.

Algorithm 1: Cube-based Isomorph-free Model Generation

```
  input : Axioms F and Domain elements D
  output: A set of non-isomorphic models
  /*NextCell returns the next cell to
    explore (void if none left), and
    Canonicalize returns the canonical
    model/cube                              */
1 H ← empty hash table
2 Function Search(C, t)
    /*C is a cube, t is a cell term       */
3   models ← ∅
4   foreach v ∈ D do
5     E ← C ∪ {t = v}
6     if E does not violate F and
       Canonicalize(E) ∉ H then
7       H ← H ∪ Canonicalize(E)
8       t ← NextCell(E)
9       if t is void then
10        models ← models ∪ E
11      else
12        models ← models ∪ Search(E, t)
13  return models
14 return Search(∅, NextCell(∅))
```

Example 4 in Section 2.1 illustrates an important property of isomorphic cubes: more isomorphic cubes can be found in longer cubes. This means the CBIF retains its efficacy when applied to models of higher orders.

6 Experimental Results

We incorporate CBIF into the finite model enumerator Mace4, which supports searching on FOL [25]. It is considered one of the best finite model finders. The current prototype [9] supports operations up to arity of 2 but it can be extended to support operations of any arities. For comparison, a separate standalone program, Isonaut [11], is developed to use the same procedure as CBIF to remove isomorphic models from a set of models. In our implementation of the two-step algorithm, we use Mace4 to generate models according to the given FOL, and then use Isonaut to remove isomorphic models from the outputs of Mace4. Nauty is run in the *sparse* mode because the graphs from cubes/models are sparse graphs (see Example 6). The experiments[1] are run on an Intel® Xeon®Silver 4110 CPU 2.0 GHz ×32 computer, with 64 GB RAM. All times reported here are CPU times. Except for the IP loops of order 15, we allow 1.5 days for each experiment to run.

[1] Data and scripts are available in [10].

The MarcieDB database [2] contains a rich collection of many popular algebras suitable for our benchmarking purposes. We pick a wide range of challenging algebraic structures with different complexities for our experiments. We run the tests with higher orders to highlight the power of the CBIF algorithm. The definitions of the algebras used in the experiments in this section are listed in Table 1, in which all clauses are implicitly universally quantified.

Table 1: Definitions of Algebras Used in Experiments

Algebra	FOL Definition
Tarski Algebras	$(x*y)*y=(y*x)*x.\ (x*y)*x=x.$ $x*(y*z)=y*(x*z).$
Involutive Lattices	$(x*y)*z=x*(y*z).\ \ x*y=y*x.$ $(x+y)+z=x+(y+z).\ \ --x=x.$ $x+y=y+x.\ \ (x*y)+x=x.$ $(x+y)*x=x.\ \ -(x+y)=-x*-y.$
M-zeroids	$(x+y)+z=x+(y+z).$ $(x*y)*z=x*(y*z).\ \ x+y=y+x.$ $x+0=x.\ \ (x\times y)\times z=x+(y\times z).$ $(x*y)\times x=x.\ \ (x\times y)*x=x.$ $-x=x.\ \ x\times y=y\times x.\ \ x+-x=0.$ $x*y=y*x.\ \ x<y\leftrightarrow 0=-x+y.$ $x+(y*z)=(x+y)*(x+z).$
Near-rings	$(x+y)+z=x+(y+z).\ \ x+0=x.$ $x+-x=0.\ \ (x*y)*z=x*(y*z).$ $(x+y)*z=(x*z)+(y*z).$
Tarski's HSI	$(x+y)+z=x+(y+z).\ x*1=1*x.$ $(x*y)*z=x*(y*z).\ x*y=y*x.$ $x*(y+z)=(x*y)+(x*z).\ \ x^1=x.$ $(x^y)*(x^z)=x^{(y+z)}.\ \ (x*y)^z=x^z*y^z.$ $(x^y)^z=x^{(y*z)}.\ 1^x=x.\ x+y=y+x.$
Loops	$x*y=x*z\rightarrow y=z.\ 0*x=x.$ $y*x=z*x\rightarrow y=z.\ x*0=x.$
C-loops	Loops with $x*(y*(y*z))=((x*y)*y)*z.$
IP loops	Loops with $x'*(x*y)=y.\ \ (y*x)*x'=y.$ Redundant clauses to speed up search: $0*0=0.\ 0'=0.$

We first run tests on 3 algebras with increasing complexities to show the improvements in performance of the CBIF algorithm over the two-step finite model finding algorithm.

Speedup is calculated as the ratio of the time of the two-step model finding algorithm and the time of the CBIF algorithm on the same data set. That is, it is the number of times that CBIF is as fast as the conventional two-step method.

CBIF speeds up the model enumeration process for Tarski Algebras by 3 to 4 orders of magnitude for higher orders. Furthermore, while the two-step model finding procedure maxes out on the computer capacity at order 12, CBIF progresses to order 23 on the same equipment (see Figure 8).

Table 2: Two-step Model Finding vs. CBIF on Tarski Algebras

Order	#Non-isomorphic Models	CPU Time (s)		Speedup (#times)
		Two-step	CBIF	
9	11	10	0.12	80
10	18	103	0.42	245
11	29	1,274	0.65	1,959
12	49	17,912	1.36	13,170

Involutive Lattices's axioms are more complex than those of Tarski Algebras. Nevertheless, CBIF works just as well, giving a speedup of 3 orders of magnitude in higher orders, and finds all models up to order 19 (see Table 3 and Figure 8).

Table 3: Two-step Model Finding vs. `CBIF` on Involutive Lattices

Order	#Non-isomorphic Models	CPU Time (s) Two-step	CPU Time (s) `CBIF`	Speedup (#times)
9	122	37	0.28	132
10	389	354	1.61	220
11	906	3,523	2.50	1,409
12	3,047	96,519	16.60	5,814

Finally, we run experiments on the M-zeroids, which have the most complex axiomatization among the three. The results as shown in Table 4 and Figure 7 confirm the pattern: The higher the order, the more pronounced the advantage of `CBIF` has over the two-step approach.

Table 4: Two-step Model Finding vs. `CBIF` on M-zeroids

Order	#Non-iso. Models	CPU Time (s) Two-step	CPU Time (s) `CBIF`	Speedup (#times)
7	315	35	1.64	21
8	1,537	980	11.07	89
9	8,583	3,523	92.84	365

While `CBIF` is useful for all problems across the board, its most valuable property is its ability to retain its effectiveness in working with higher orders of complex structures. Our tests show that the processing time by `CBIF` grows much slower than that of the two-step algorithm as the order goes higher. While the more complex structures have steeper slopes and take longer to generate models, as expected, they all show the same growth pattern (see Figure 8).

Figure 7: Speedup w/ `CBIF` over Two-step Algorithm

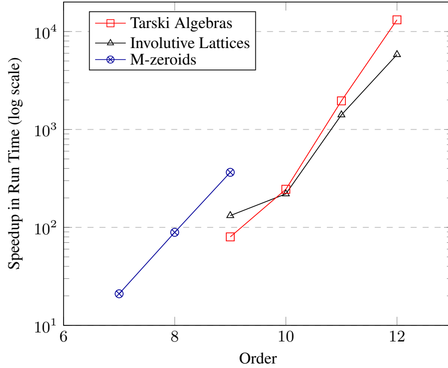
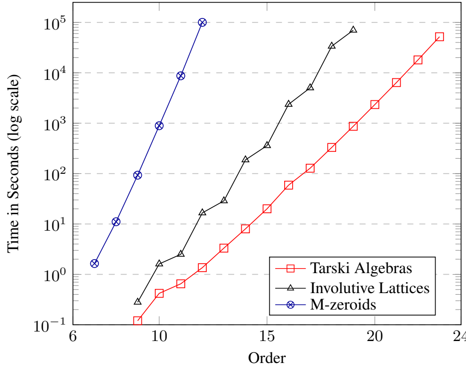


Figure 8: Performance of `CBIF` on Models of Higher Orders



6.1 *New Results*

Here, we present some challenging examples in which `CBIF` contributes new results to the literature. The number of near-rings up to order 13, and the number of Tarski's HSI models up to order 4, have recently been reported in *The On-Line Encyclopedia of Integer Sequences* ® (OEIS) [37] as sequences A305858 and A214396, respectively. Using the `CBIF` algorithm, we extend the sequences to order 15 and order 6, respectively (see Table 5). These new results are achieved solely with the `CBIF` algorithm without any knowledge specific to the algebraic structures in question.

Our last experiments give never-reported-before results on two algebraic structures derived from loops: C-loops and IP loops. Phillips & Vojtěchovský [32] characterize nonassociative C-loops of orders up to 14, but are not able to enumerate all C-loops of order 16 (there are no nonassociative C-loops of order 15). Slaney & Ali [35] think that "significantly extending the search may raise different challenges for automated reasoning." Using `CBIF`, we successfully enumerate C-loops of order 16 in less than 10 minutes (see Table 5).

For IP loops, Slaney & Ali [35] is the first to report successful enumeration of IP loops of order 13 using FINDER [36], which takes a day to crunch out the results. Khan *et al.* [23] point out that "The IP loops having order greater than 13 are not reported in the literature because of the huge search space." Using `CBIF`, we successfully enumerate IP loops of order 14 in less than 2.5 hours, and IP loops of order 15 in 88 hours (see Table 5).

Table 5: New Results Generated by `CBIF`

Algebra	Order	#Non-isomorphic Models	CPU Time (s)
Near-rings	14	4,537	956
	15	3,817	1,524
Tarski's HSI	5	13,577	3
	6	672,740	137
C-loops	16	122	572
IP loops	14	2,104,112	8,570
	15	40,897,240	316,065

The empirical results from `CBIF` match those in the literature, if available. For example, they match the results from the 2-step method for lower orders.

7 Related Work

A practical approach to model finding/enumeration for models of higher orders is the two-step process [1, 3, 22, 26], where most work is on the elimination of isomorphic models in the second step. McCune [26] adds `Isofilter` and `Isofilter2` to the Prover9/Mace4 system to filter out isomorphic models from the outputs of `Mace4` (the first step). `Isofilter` does brute-force comparisons of models for isomorphism. `Isofilter2` calculates canonical forms of models so that isomorphic models must have the same canonical form. Neither program is optimized and does not perform adequately except for very small sets of models of small orders. Khan [22] presents a 2-step model enumeration algorithm in which the first step uses the CP solver *or-tools*[31], and the second step is then tree-based algorithm (TVMC). In TVMC, each isomorphism class is represented by a branch in the tree. Each model is checked against all isomorphism classes by traversing this tree only once. They state that "Our proposed algorithm can enumerate IP loop of order 13 in six hours (21,952 seconds) on an ordinary desktop system." In comparison, `CBIF` takes only 79 seconds to complete the same task on an ordinary computer. Araújo *et al.* [1] observe that some properties of the models are invariant under isomorphisms and devise the invariant-based parallel algorithm to filter out isomorphic models. Nagy and Vojtěchovský [29] also implement a sophisticated quasigroup-specific invariant-based algorithm in the `LOOPS` package

to handle isomorphism in quasigroups. This expensive postprocessing step is obviated in CBIF.

Model searching algorithms can generally be classified according to whether they directly search for models, or are based on some transformations of the inputs [6]. Cube-based algorithms are most applicable to the direct-search approach. In this approach, function selection strategy and cell selection strategy have big impacts on the performance of the finite model finders. See for example [5] on the discussion of various function ordering heuristics. CBIF is agnostic to these strategies and hence can be used in conjunction with them. E.g., by using suitable function and cell selection strategies, CBIF can generate lexicographically smallest models at no extra costs.

As noted, the first step in the two-step model finding procedure often produces many isomorphic models that the second step must filter out. This is particularly true for models defined with FOL because it is an inherent symmetry property of FOL [33]. To mitigate this problem, symmetry-breaking algorithms are applied in this step. There is a bevy of research results on symmetry-breaking [5, 12, 13, 15, 24, 33, 34, 39]. Many useful partial symmetry-breaking algorithms, such as the LNH and the extended LNH (XLNH) [4, 5], have been widely used. The LNH can be considered as symmetry-breaking with interchangeable values in constraint satisfaction problems (CSP) [19]. The XLNH is less flexible as it requires the existence of at least one unary operation. The LNH is implemented in many systems such as Mace4, SEM [42], Falcon [41], and FMSET [7]. Algorithms that remove isomorphic cubes, including the CBIF, are compatible with the LNH [3]. CBIF, which can be classified as a solution symmetry-breaker (as defined in [14]), is a complete symmetry-breaking algorithm and can therefore work without other symmetry-breaking algorithms such as the LNH. However, the LNH is a low-overhead predictive algorithm that is particularly effective in removing short isomorphic cubes and hence nicely complements the CBIF.

Another important symmetry-breaking strategy is to prevent the search engine from the fruitless exploration by adding symmetry-breaking input clauses [15, 39]. CBIF is agnostic to the additional inputs and hence can be used in conjunction with them.

The parallel cube algorithm [3] is yet another effective symmetry-breaking algorithm. Both CBIF and the parallel cube algorithm are based on the fact that isomorphic cubes lead to isomorphic models and hence redundant isomorphic cubes need not be explored. The parallel cube algorithm is only part of the first step of the two-step algorithm. It only removes isomorphic cubes at some synchronization points, and hence incurs synchronization overheads. Some redundant branches in the search tree may still be searched. It also requires disk space to hold the intermediate results. Furthermore, unlike CBIF which avails on the power of the nauty algorithm, it uses less efficient mechanisms such as invariants to detect isomorphic cubes. It can, however, improve the search speed by doing the searches in parallel, in addition to removing redundant cubes.

Another local symmetry-breaker, DSYM [5], exploits local symmetries by finding symmetries under invariant partial interpretations (which are invariant cubes). DSYM is a predictive algorithm that works at the *parent* level and predicts which of its immediate children will be isomorphic cubes. Consequently, it only detects symmetries under the same subtree. It is thus not an isomorph-free algorithm, although it generates much fewer isomorphic models than using the LNH alone. CBIF, on the other hand, detects both global and local symmetries the same way, and removes all symmetries. Nevertheless, CBIF is compatible with DSYM, so they can be used together.

Finite model enumeration can be posed as a *constraint programming (CP)* task [22]. CP solvers abound, e.g., Minion [18], Gecode [30], and Google's CP-SAT [31]. However, they are usually not isomorph-free model generators. As pointed out in [3], to effectively add *symmetry-breaking* constraints such as lex-leaders to a CP solver often requires deep knowledge of the solver, including its input language, *and* the problem at hand (e.g., the semigroups with Minion in [17]). The CBIF, on the other hand, can be applied to searching in any structures as long as the finite model finder does direct search.

Some finite model finders, such as SEMD [21], also completely suppress isomorphic models in the search process. Like CBIF, SEMD also follows the recorded objects paradigm. In SEMD, decision sequences for branches of the search tree are recorded. The search branch can be discarded if its decision sequence has already appeared in some other search branches. It is not clear how well it works with the complex search problems such as those presented in this paper. They report results up to order 4 for the Tarski's HSI Problem, but we find all models of order 6 for the same problem in just 137 seconds (see Table 5).

8 Conclusions and Future Work

In this paper, we introduce the efficient isomorph-free model enumeration algorithm, CBIF, which has wide applications in many research areas such as computational algebra. It enables us to add many new results in computational algebra to the literature (Section 6.1). It exploits the powers hidden in different representations (operation tables and graphs) of the structures. Not only does it run much faster than the traditional two-step model finders, it also generates no intermediate outputs that take up disk space, which often exceed the limits of the system. This greatly enhances the capability of the finite model searcher in dealing with large search spaces. Moreover, it handles exceptionally well structures with complex axiomatization and retains its power in higher domain sizes. Last but not least, it is a complete symmetry breaker that also works with other symmetry-breaking algorithms such as the LNH, and it is easy to be incorporated into any traditional direct-search finite model finder such as Mace4.

Future research will focus on finding the best cell selection strategy, on fast conversion of models to graphs, and on compact representation of canonical models.

Acknowledgements

The authors would like to thank Prof. Brendan D. McKay for advising us how to construct the graphs of models.
The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project *POSTMAN* no. LL1902. The research was co-funded by the European Union under the project *ROBOPROX* (reg. no. CZ.02.01.01/00/22_008/0004590) and Fundação para a Ciência e a Tecnologia, through the projects UIDB/00297-/2020 (CMA), PTDC/MAT-PUR/31174/2017, UIDB/04621/2020 and UIDP/04621/2020.

References

[1] J. Araújo, C. Chow, and M. Janota. Boosting isomorphic model filtering with invariants. *Constraints*, 27(3):360–379, Jul 2022. ISSN 1572-9354. doi: 10.1007/s10601-022-09336-x.
[2] J. Araújo, D. Matos, and J. Ramires. MarcieDB: a model and theory database. https://marciedb.pythonanywhere.com, 2022.

[3] J. a. Araújo, C. Chow, and M. Janota. Symmetries for Cube-And-Conquer in Finite Model Finding. In R. H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-300-3. doi: 10.4230/LIPIcs.CP.2023.8. URL https://drops.dagstuhl.de/opus/volltexte/2023/19045.

[4] G. Audemard and L. Henocque. The eXtended least number heuristic. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 427–442, Berlin, Heidelberg, 2001. Springer. doi: 10.1007/3-540-45744-5_35.

[5] G. Audemard, B. Benhamou, and L. Henocque. Predicting and detecting symmetries in FOL finite model search. *J. Autom. Reason.*, 36(3): 177–212, 2006. doi: 10.1007/s10817-006-9040-3.

[6] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 7(1):58–74, 2009. ISSN 1570-8683. doi: https://doi.org/10.1016/j.jal.2007.07.005. URL https://www.sciencedirect.com/science/article/pii/S1570868307000638. Special Issue: Empirically Successful Computerized Reasoning.

[7] B. Benhamou and L. Henocque. A hybrid method for finite model search in equational theories. *Fundam. Informaticae*, 39(1-2):21–38, 1999. doi: 10.3233/FI-1999-391202.

[8] S. Burris and S. Lee. Tarski's high school identities. *The American Mathematical Monthly*, 100(3):231–236, 1993. ISSN 00029890, 19300972. URL http://www.jstor.org/stable/2324454.

[9] Choiwah Chow. Mace4c, 2023. URL https://github.com/ChoiwahChow/p9m4.

[10] Choiwah Chow. Experiments with mace4c, 2024. URL https://github.com/ChoiwahChow/public/blob/main/ecai24_cbif.zip.

[11] Choiwah Chow. Isonaut, 2024. URL https://github.com/ChoiwahChow/isonaut.

[12] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[13] M. Codish, A. Miller, P. Prosser, and P. J. Stuckey. Breaking symmetries in graph representation. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pages 510–516. IJCAI/AAAI, 2013. URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6480.

[14] D. A. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints An Int. J.*, 11(2-3):115–137, 2006. doi: 10.1007/s10601-006-8059-8.

[15] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 148–159. Morgan Kaufmann, 1996.

[16] A. Distler and J. Mitchell. Smallsemi, a library of small semigroups in GAP, Version 0.6.12. https://gap-packages.github.io/smallsemi/, 2019. GAP package.

[17] A. Distler, C. Jefferson, T. Kelsey, and L. Kotthoff. The semigroups of order 10. In M. Milano, editor, *Principles and Practice of Constraint Programming - CP*, volume 7514, pages 883–899. Springer, 2012. doi: 10.1007/978-3-642-33558-7_63. URL https://doi.org/10.1007/978-3-642-33558-7_63.

[18] I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *ECAI, 17th European Conference on Artificial Intelligence, Including Prestigious Applications of Intelligent Systems (PAIS), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102, Amsterdam, Netherlands, 2006. IOS Press. URL http://www.booksonline.iospress.nl/Content/View.aspx?piid=1654.

[19] P. V. Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In G. Gottlob and T. Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 277–284. Morgan Kaufmann, 2003. URL http://ijcai.org/Proceedings/03/Papers/041.pdf.

[20] M. Janota, C. Chow, J. Araújo, M. Codish, and P. Vojtechovský. Sat-based techniques for lexicographically smallest finite models. In M. J. Wooldridge, J. G. Dy, and S. Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 8048–8056. AAAI Press, 2024. doi: 10.1609/AAAI.V38I8.28643. URL https://doi.org/10.1609/aaai.v38i8.28643.

[21] X. Jia and J. Zhang. A powerful technique to eliminate isomorphism in finite model search. In U. Furbach and N. Shankar, editors, *Automated Reasoning*, pages 318–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37188-5.

[22] M. A. Khan. Efficient enumeration of higher order algebraic structures. *IEEE Access*, 8:41309–41324, 2020. doi: 10.1109/ACCESS.2020.2976876. URL https://doi.org/10.1109/ACCESS.2020.2976876.

[23] M. A. Khan, S. Muhammad, N. Mohammad, and A. Ali. Enumeration of exponent three ip loops. *Quasigroups & Related Systems*, 25(1), 2017.

[24] M. Kirchweger and S. Szeider. SAT modulo symmetries for graph generation. In L. D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP*, volume 210 of *LIPIcs*, pages 34:1–34:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.CP.2021.34. URL https://doi.org/10.4230/LIPIcs.CP.2021.34.

[25] W. McCune. Mace4 reference manual and guide, August 2003. URL https://www.cs.unm.edu/~mccune/prover9/mace4.pdf.

[26] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[27] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi: 10.1016/j.jsc.2013.09.003. URL https://doi.org/10.1016/j.jsc.2013.09.003.

[28] R. Miček. Automated model building. Master's thesis, Charles University, Czech Republic, 2015. http://hdl.handle.net/20.500.11956/81163.

[29] G. Nagy and P. Vojtěchovský. LOOPS, computing with quasigroups and loops in GAP, Version 3.4.1. https://gap-packages.github.io/loops/, Nov 2018. Refereed GAP package.

[30] M. Nielsen. Parallel search in gecode. *Technical Report, Gecode*, 2006.

[31] L. Perron and F. Didier. CP-SAT, 2023. URL https://developers.google.com/optimization/cp/cp_solver/.

[32] J. D. Phillips and P. Vojtěchovský. C-loops: An introduction, 2007.

[33] G. Reger, M. Riener, and M. Suda. Symmetry avoidance in MACE-style finite model finding. In A. Herzig and A. Popescu, editors, *Frontiers of Combining Systems FroCoS*, volume 11715, pages 3–21, Switzerland AG, 2019. Springer. doi: 10.1007/978-3-030-29007-8_1.

[34] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4.

[35] J. Slaney and A. Ali. Generating loops with the inverse property. *Proc. of ESARM*, pages 55–66, 2008.

[36] J. K. Slaney. FINDER: finite domain enumerator - system description. In A. Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 798–801. Springer, 1994. doi: 10.1007/3-540-58156-1_63.

[37] N. J. A. Sloane and T. O. F. Inc. The on-line encyclopedia of integer sequences, 2020. URL http://oeis.org/?language=english.

[38] The GAP Group. GAP – Groups, Algorithms, and Programming, Version 4.10.2, 2019. URL https://www.gap-system.org/Packages/smallgrp.html. Accessed: yyyy-mm-dd.

[39] T. Walsh. Symmetry breaking constraints: Recent results. In J. Hoffmann and B. Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4974.

[40] H. Zhang and J. Zhang. MACE4 and SEM: A comparison of finite model generators. In M. P. Bonacina and M. E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 101–130. Springer, 2013. doi: 10.1007/978-3-642-36675-8_5.

[41] J. Zhang. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, 17:1–22, 08 1996. doi: 10.1007/BF00247667.

[42] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *IJCAI*, pages 298–303, 1995. URL http://ijcai.org/Proceedings/95-1/Papers/039.pdf.