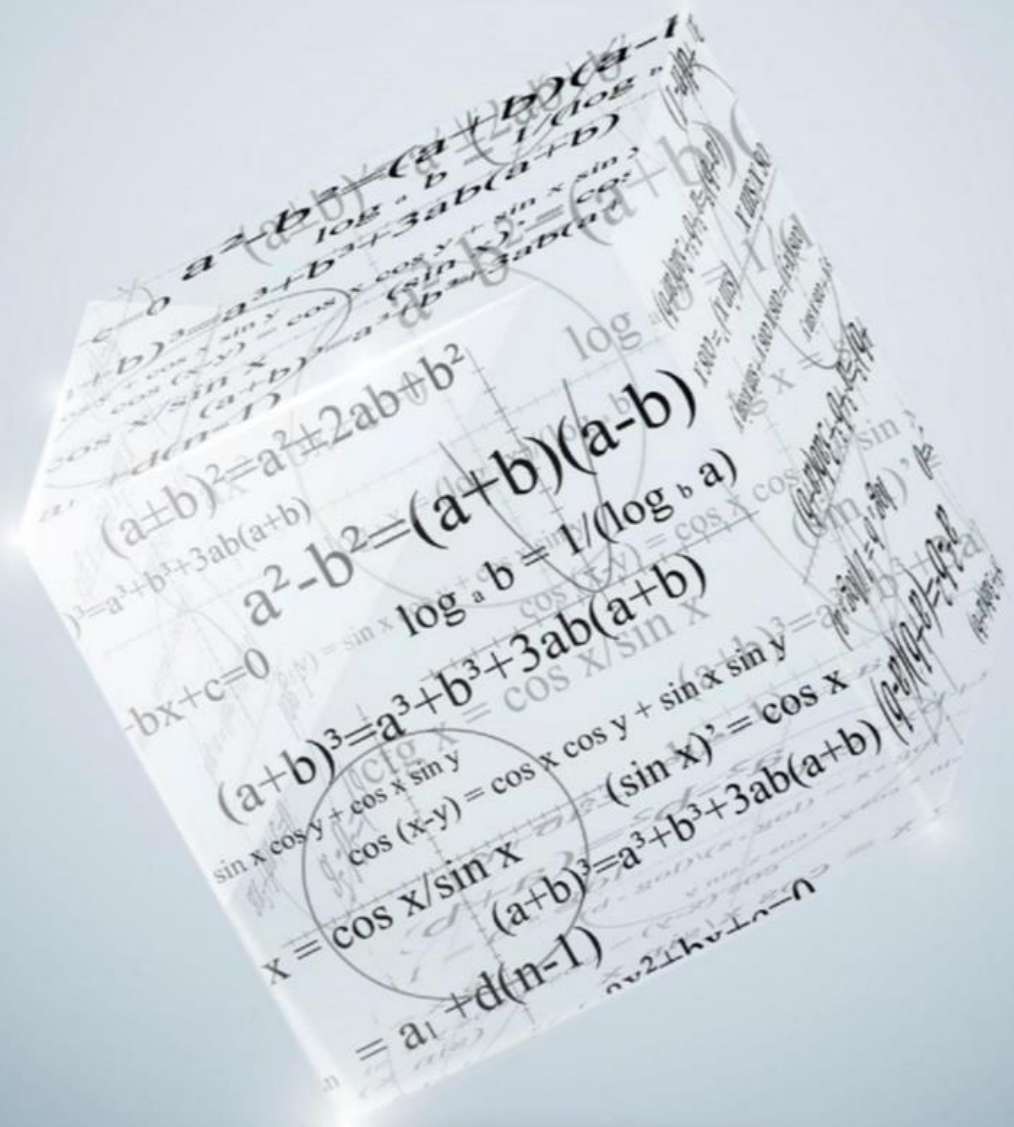
UNIVERSIDADE ABERTA
www.uab.pt

UNIVERSIDADE DE COIMBRA
UNIVERSIDADE ABERTA
EDUCAÇÃO A DISTÂNCIA
DISTANCE EDUCATION

1 2 9 0
UNIVERSIDADE D COIMBRA

Finite Model Enumeration



Doctoral Thesis in Computational Algebra | 2023

Choiwah Chow

Finite Model Enumeration

Choiwah Chow

Doctor's Degree in Computational Algebra

Doctoral thesis supervised by:

Prof. Dr. João Araújo (UNL/UAb, Portugal)

Prof. Dr. Mikoláš Janota (CTU in Prague, Czechia)

Prof. Dr. Gilda Ferreira (UL/UAb, Portugal)

2023

Dedicated to my wife, Vai, my children, Casey, Emily, and Corey

Acknowledgments

I would like to express my deepest gratitude to Professors João Araújo, Mikoláš Janota, and Gilda Ferreira for their guidance, support, encouragement, and help in my wonderful journey into research. Not only did they broaden my horizon in mathematics and computer science, they also showed me how to do research and to present the results in academic papers.

I was fortunate to have many good colleagues in the DAC program to study together. I would like to thank in particular Fernando Ferreira, Rui Barradas Pereira, and Carlos Fernando Mendes Sousa for their supports. Rui has helped me test out some of my GAP packages and provided great feedback. Fernando has shared a copy of the Latex template and the guidelines for the doctoral thesis that satisfied UAb's and UC's requirements. Carlos shared his C++ source code of the Library for Automated Deduction Research (LADR), which was the backbone for both the Prover9 and Mace4 as well as numerous other utility programs built around these two main applications. He updated part of the library at my request at times when he was very busy with his own studies. Moreover, he provided me a copy of the 2017-11A version of the C-based Prover9/Mace4/LADR code base, which I used for conversion to the C++ version of Mace4 and many related utilities such as isofilter.

I would like to thank Springer Nature for giving me the permissions to include in this thesis the paper *Boosting isomorphic model filtering with invariants* [5].

Resumo

Para estudar e compreender diferentes tipos de álgebras relacionais (grupos, semigrupos e suas versões ordenadas, quasigrupos, corpos, anéis, MV-álgebras, reticulados, etc.), os matemáticos recorrem a bibliotecas de todos os modelos, a menos de isomorfismo, de ordem n (para pequenos valores de n) de classes de álgebras de seu interesse. Essas bibliotecas permitem experimentação, como formular e/ou testar conjecturas, etc., para obter intuição sobre as classes de álgebras em questão. Como um isomorfismo divide o conjunto de todos os modelos em classes de equivalência, apenas um elemento representativo em cada classe de equivalência é necessário. Os demais são redundantes e foram removidos para melhor se entender a estrutura das álgebras.

Para construir essas bibliotecas, precisamos encontrar todos os modelos a menos de isomorfismo de classes de álgebras em que estamos interessados. Enumerar todos os modelos finitos de fórmulas de álgebras de primeira ordem a menos de isomorfismo é um problema difícil, mesmo para álgebras de ordens pequenas. Atualmente, a melhor estratégia de enumeração de modelos finitos é dividir o trabalho em duas fases: primeiro, gerir modelos de acordo com as leis estabelecidas pela fórmula de primeira ordem que definem a álgebra, depois, na etapa de pós-processamento, encontrar apenas um modelo representativo de cada classe de equivalência de modelos isomórficos.

Uma álgebra de ordem finita n definida com fórmulas de primeira ordem, pode ser representada por tabelas de operações (também conhecidas como multiplicação) de tamanhos finitos. Os enumeradores de modelos finitos tradicionais, como Mace4, basicamente executam pesquisa combinatórias nessas tabelas de operação para encontrar instâncias que satisfaçam, ou equivalentemente, não violem as regras estabelecidas pela fórmula de primeira ordem. Isto constitui um enorme problema. Por exemplo, existem n^{n^2} possíveis tabelas de operações binárias de tamanho $n \times n$ com n valores possíveis em cada célula. Ou seja, mesmo para enumerar todos os modelos de ordem 4 para uma álgebra definida com apenas uma operação binária em sua fórmula de primeira ordem, estamos diante da tarefa de verificar $4^{4^2} \approx 4.3$ mil milhões de possibilidades. Os enumeradores de modelos finitos tradicionais também produzem muitos modelos isomórficos na medida em que os modelos de isomorfismo geralmente constituem mais de 99% das saídas. Esta é uma característica das álgebras definíveis por FOL: qualquer permutação nos elementos de domínio de um modelo é um modelo. Encontrar modelos não isomórficos é computacionalmente muito dispendioso. Se seguirmos uma abordagem simples de força bruta, então, dados os modelos m, pode ser necessário, no pior caso, $m(m-1)/2$ comparações de modelos para verificar o isomorfismo. Cada comparação de modelos de ordem n pode exigir, no pior caso, $n!$ verificações porque existem $n!$ permutações diferentes nos símbolos n.

Assim, essas duas etapas (gerar modelos de FOL e remover modelos isomórficos) consomem muitos recursos. Somos assim impelidos a encontrar algoritmos melhores que façam uso otimizado dos recursos de computação disponíveis. A questão é ainda mais complicada pelo fato de que, embora os computadores modernos de uso geral sejam predominantemente multi-core, aproveitar o paralelismo para a pesquisa combinatória é surpreendentemente difícil.

Neste tese, propomos o algoritmo de remoção de cubos isomórficos e o algoritmo de busca de cu-

bos paralelos para melhorar a velocidade do enumerador de modelos finitos tradicional. A pesquisa nas tabelas de operações pode ser vista como uma pesquisa numa árvore de pesquisa. Qualquer ramo parcial da raiz (chamado cubo) pode ser pesquisado em paralelo com outros ramos, fazendo pleno uso dos recursos computacionais disponíveis. Além disso, alguns cubos (isomórficos) produzem modelos isomórficos e, portanto, apenas um deles precisa ser pesquisado. A remoção de cubos isomórficos não apenas acelera o processo de busca, mas também reduz o número de modelos isomórficos que precisam ser removidos na etapa de pós-processamento. Esses algoritmos paralelos são muito escaláveis: mais cubos podem ser gerados se mais recursos de computação estiverem disponíveis. Eles também lidam bem com grandes problemas: cubos mais longos são possíveis com ordens mais altas de álgebra; com cubos mais longos, cubos mais isomórficos podem ser removidos.

Para acelerar o processo de remoção de modelos isomórficos, propomos os algoritmos de remoção de modelos isomórficos baseados em invariantes paralelos. Notamos que modelos com diferentes conjuntos de vetores invariantes não podem ser isomórficos. Colocamos modelos em blocos numa tabela de hash usando vetores invariantes como chaves. Modelos em diferentes blocos, portanto, têm diferentes conjuntos de vetores invariantes e, portanto, não podem ser isomórficos. Esses blocos de modelos podem ser verificados quanto ao isomorfismo, separadamente, em paralelo. A melhoria na velocidade obtém-se por (1) tamanho de bloco menor significa menos pares de modelos para comparar em termos de isomorfismo, (2) os blocos podem ser processados em paralelo, fazendo pleno uso dos recursos computacionais disponíveis. Esses algoritmos também lidam bem com grandes problemas: modelos de ordens mais altas fornecem mais vetores invariantes o que permite maior dispersão dos modelos.

A junção desses algoritmos geralmente permite uma acentuada melhoria no desempenho dos enumeradores de modelos finitos tradicionais. Melhor ainda, esses algoritmos podem ser incorporados em muitos enumeradores de modelos finitos comuns com alterações mínimas na seu código base. Além disso, os algoritmos propostos são comprovadamente corretos.

Em qualquer pesquisa matemática, o ser humano é o fator mais importante a ser considerado. Nem todos os matemáticos são especialistas em computação e nem todos querem investir o seu valioso tempo em ferramentas matemáticas. Assim, as ferramentas devem ser projetadas para serem fáceis de usar, sem uma curva de aprendizagem íngreme para que o utilizador possa usar as ferramentas de forma eficaz. Nesse sentido, desenvolvemos um gerador de pacotes GAP para gerar automaticamente pacotes GAP para álgebras de pequenas ordens. O gerador oculta os detalhes técnicos do processo de geração, mas dá aos utilizadores a capacidade de controlar muitos dos aspectos não técnicos, como o nome e a definição (em FOL) da álgebra a ser explorada. O gerador de pacotes GAP faz uso extensivo das facilidades fornecidas pelo sistema GAP, e o pacote gerado está em conformidade com os padrões GAP mais recentes. Um pacote GAP separado e autônomo, `Magmaut`, é desenvolvido para fornecer funções que permitam a manipulação de isomorfismos e automorfismos para álgebras do tipo $(2^m, 1^n)$. Ele complementa o pacote GAP gerado para álgebra de pequenas ordens com funcionalidade relacionadas com o isomorfismo.

Com as ferramentas fornecidas por este projeto, os matemáticos podem gerar os pacotes GAP de

álgebras do tipo $(2^m, 1^n)$ de seu interesse, sem se prenderem à complexidade que muitas vezes vem com as ferramentas que geram tais pacotes GAP.

Nesta tese, propomos ainda muitos algoritmos paralelos para aproveitar os computadores multinúcleos modernos para enumerar modelos finitos de álgebras. Mostramos algumas aplicações importantes desses algoritmos no desenvolvimento de pacotes GAP para calcular diferentes morfismos como isomorfismo e monomorfismo etc., na geração de pacotes GAP de álgebras e na contagem do número de álgebras de pequenas ordens. Também desenvolvemos uma ferramenta de fácil utilização para ajudar os matemáticos a gerar pacotes GAP de álgebras (do tipo $(2^m, 1^n)$) de pequenas ordens.

Palavras-chave Álgebra, CREAM, cubo-e-conquista, enumeração de modelo finito, GAP, monóide inerentemente não finito, invariante, Isofilter, isomorfismo, Magmaut, Mace4, Master Prover, monomorfismo, algoritmo paralelo.

Abstract

To study and get intuition on different types of relational algebras (groups, semigroups, and their ordered versions, quasigroups, fields, rings, MV-algebras, lattices, etc.), mathematicians resort to libraries of all models, up to isomorphism, of order n (for small values of n) of the classes of algebras they are interested in. These libraries allow experimentation such as forming and/or testing conjectures etc., to gain insights into the classes of algebras in question. Since an isomorphism partitions the set of all models into equivalence classes, so, only one representative element in each equivalence class is needed. The rest of them are redundant and are removed to make it easy to discern the structure of the algebras.

To construct these libraries, we need first to find all models up to isomorphism of the classes of algebras of interest. Enumerating all finite models from first-order formulas up to isomorphism is a hard problem, even for algebras of small orders. The current best practice of finite model enumeration is to divide the job into two steps: first generate models according to the laws laid down by the first-order formula defining the class of algebra, then in the post-processing step, find only one representative model of each equivalence class of isomorphic models.

An algebra of finite order n defined by a first-order formula can be represented by operation tables (also known as multiplication tables) of finite sizes. Traditional finite model enumerators such as Mace4 basically perform combinatorial search on these operation tables to find instances that satisfy, or equivalently, not violate, the rules laid down by the first-order formula. This is a hard problem. For example, there are n^{n^2} possible binary operation tables of size $n \times n$ with n possible values in each cell. That is, even to enumerate all models of order 4 defined with just one binary operation in its first-order formula, we are facing the task of checking $4^{4^2} \approx 4.3$ billion possibilities. Traditional finite model enumerators also produce a lot of isomorphic models to the extent that isomorphism models often constitute over 99% of the outputs. This is a characteristics of algebras definable by FOL: any permutation on the domain elements of a model is a model. Finding non-isomorphic models is very computationally intensive. If we follow a simple brute-force approach, then, given m models, it may require in the worst case $m(m-1)/2$ comparisons of models to check for isomorphism. Each comparison of models of order n may require in the worst case $n!$ checks because there are $n!$ different permutations on n symbols.

Thus, both of these two steps (generating models from FOL and removing isomorphic models) are resource-intensive. They prompt us to find better algorithms that also make optimal use of the available computing resources. The matter is further complicated by the fact that while modern-day general-purpose computers are predominantly multi-core, harnessing parallelism for combinatorial search is surprisingly difficult.

In this thesis, we propose the isomorphic cubes removal algorithm and the parallel cubes search algorithm to improve the speed of the traditional finite model enumerator. Searching the operation tables can be seen as searching a search tree. Any partial branch from the root (called a cube) can be searched in parallel with other branches, making full use of the computing resources available. Furthermore, some (isomorphic) cubes produce isomorphic models and hence only one of them needs to be searched.

Removing isomorphic cubes not only speeds up the search process, but also reduces the number of isomorphic models that need to be removed in the post-processing step. These parallel algorithms are very scalable: more cubes can be generated if more computing resources are available. They also cope with big problems well: longer cubes are possible with higher orders of algebra; with longer cubes, more isomorphic cubes can be removed.

To speed up the isomorphic models removal process, we propose the parallel invariant-based isomorphic model removal algorithms. We note that models with different sets of invariant vectors (ordered list of invariants) cannot be isomorphic. We put models into blocks in a hash-table using invariant vectors as the keys. Models across different blocks thus have different sets of invariant vectors and hence cannot be isomorphic. These blocks of models can be checked for isomorphism separately in parallel. The improvement in speed comes from (1) smaller block size means fewer pairs of models to compare for isomorphism, (2) the blocks can be processed in parallel, making full use of the computing resources available. These algorithms also cope with big problems well: models of higher orders give more different invariant vectors to spread out the models.

These algorithms together often give an improvement of orders of magnitude in the performance of the traditional finite model enumerators. Better yet, these algorithms can be incorporated in many common finite model enumerators with minimal change in their code base. Furthermore, the proposed algorithms are proved to be correct.

In any mathematical research, humans are the most important factor to be given considerations. Not all mathematicians are computer experts, and not all of them want to invest their valuable time in the tools of mathematics. Thus, the tools must be designed to be user-friendly, without a steep learning curve for the user to be able to use the tools effectively. We therefore develop a GAP package generator to automatically generate GAP package for algebras of small orders. The generator hides the technical details of the generation process, but gives users the ability to control many of the non-technical aspects such as the name and the definition (in FOL) of the algebra to be explored. The GAP package generator makes extensive use of the facilities provided by the GAP system, and the generated package complies with the latest GAP standards. A separate, standalone GAP package, `Magmaut`, is developed to provide functions to manipulate isomorphism and automorphism for algebras of type $(2^m, 1^n)$. It supplements the generated GAP package of algebra of small orders with isomorphism-related functionality.

With the tools provided by this project, mathematicians can generate the GAP packages of algebras of type $(2^m, 1^n)$ of their interest, without being bogged down by the complexity that often comes with the tools that generate such GAP packages.

In this thesis, we propose many parallel algorithms to take advantage of modern-day multi-core computers to enumerate finite models. We show some important applications of these algorithms in developing GAP packages to calculate different morphisms such as isomorphism and monomorphism etc., in generating GAP packages of algebras, and in counting the number of algebras of small orders. We also develop a user-friendly tool to help mathematicians to generate GAP packages of algebras (of type $(2^m, 1^n)$) of small orders.

Keywords Algebra, CREAM, cube-and-conquer, finite model enumeration, GAP, inherently nonfinitely

based monoid, invariant, Isofilter, isomorphism, Mace4, Magmaut, Master Prover, monomorphism, parallel algorithm.

Contents

[1] Results in this section have been published in articles [4, 5].

List of Tables

List of Figures

Chapter 1

Introduction

One of the important tools that working algebraists need in their research work is libraries of the algebras they are interested in. These libraries allow them to get intuitions, test or refute hypotheses and conjectures, and gain insights into the properties of the algebras (see for example on p. 2891 of [43]). These libraries are so important that the search for them has a long history in mathematics predating many years of the use of computers. (See Appendix B of [28]; and for more recent results see OEIS [64], where many of the sequences are the number of order n non-isomorphic models of a given class of algebras.) Many libraries of algebraic models of small orders such as the smallsemi package [27] for semigroups and the loops package [52] for quasigroups are available in the GAP [31] system. A lot more such libraries are needed, but they often take an inordinate amount of time and computing resources to generate.

Many of these algebras can be defined in first-order logic (FOL) and there are tools to allow mathematicians to encode their algebras and produce a meaningful library. However, enumerating all finite models up to isomorphism is a hard problem, even for algebras of small orders. The current best practice of finite model enumeration from a first-order formula is to divide the job into two major, resource-intensive steps.

The first step is to generate models according to the laws specified by the first-order formula. This is a hard problem in general. For example, a first-order formula such as the one in Equation 1.1 can be represented by a binary operation table of size n where n is the domain size. For a very small domain size of 4, the binary operation table is of size 4×4 with 4 possible values in each table cell. There are $4^{4^2} \approx 4.3$ billion possibilities for such a table. It is infeasible to try out each of these configurations to see which one satisfies the given first-order formula. Worse, this step often results in a huge number of isomorphic models. For example, the following first-order formula defines semigroups.

$$(x * y) * z = x * (y * z) \tag{1.1}$$

For this formula, the traditional finite model enumerator Mace4 [48] generates 1,021,120,198 semigroups of order 7, out of which only 1,627,672 ($\approx 0.16\%$) [64] are pairwise non-isomorphic.

The second step is to find only one representative model of each equivalence class of isomorphic

models generated in the first step. This step is often more resource-intensive than the first step. Given m models, there could be $m(m-1)$ comparisons in the worse scenario if we are to compare each pair of them for isomorphism. If we check for isomorphism between two structures by trying out all possible mappings of domain elements from one to another, there would be $n!$ trials. This is certainly prohibitively expensive except for very small domain sizes.

For computational intensive problems such as enumerating models up to isomorphism, we need to use all the computing resources available. However, while modern-day general-purpose computers are predominantly multi-core, harnessing parallelism for combinatorial search is surprisingly difficult. Consequently, there are few parallel algorithms in constraints programming in general, and in finite model enumeration in particular. Indeed, in satisfiability modulo theories (SMT), even negative results are concluded for cube-and-conquer [37]. A recent literature review concludes that "there is little overall guidance that can be given on how best to exploit multi-core computers to speed up constraint solving" [33]. We aim to help close this gap by devising new parallel algorithms for finite model enumeration.

In this thesis, we propose two novel techniques, the isomorphic cubes removal algorithm and the invariant-based isomorphic models filtering algorithm. Together, they greatly improve the performance of many finite model searchers that search on the first-order formulas. Furthermore, these two algorithms are inherently parallel, allowing them to speed up the model enumeration process by taking full advantages of the multi-core architecture of the modern-day computers.

The isomorphic cubes removal algorithm not only speeds up the first step but also generates fewer isomorphic models. Suppressing the generation of isomorphic models in the first step reduces the workloads of both the first and the second steps. Not only does it make the whole process much faster, the required computing resources (disk space, etc.) are also reduced. The remaining isomorphic models are then dealt with by the invariant-based parallel algorithm, which is designed to greatly reduce the complexity of removing isomorphic models.

1.1 Objectives

In this research project, we improve finite model enumerators toward the following high-level objectives:

1. Research into fast parallel algorithms that can be incorporated to existing finite model enumerators, and for use in new finite model enumerators.

2. Develop tools that Mathematicians can use to quickly generate all models (up to isomorphism) of the algebras of their interests on their multi-core computers.

3. The tool will also take advantage of massively parallel computing architectures to pre-generate models (up to isomorphism) of algebras of general interest.

For the mathematicians' convenience, the non-isomorphic models are put into packages to be loaded into GAP, which is the most popular platform for research in computational algebra.

1.1.1 First Objective

Fast algorithms and fast computers help enumerate models to higher orders. Parallel algorithms can make use of the available computing resources as much as possible and are therefore a focus of our research. There are many existing finite model enumerators with different characteristics and each may be particularly good for a particular subset of problems. Our research devises algorithms that are versatile to be used in different finite model enumerators.

1.1.2 Second Objective

To achieve the objective of empowering mathematicians who are not necessarily computer experts to explore algebras of their choice, we automate the process of finite model enumeration as much as possible. Users of our software can still specify some input parameters, such as the definition and the order of the algebras they are interested in. Also available is a limited set of parameters such as the number of random invariants (see Section 4.4) to affect the efficiency of the finite model generation process.

We choose Mace4 as the finite model enumerator for our experiments because it has many advantages that mathematicians appreciate (see Section 3.1), such as ease of defining algebras in FOL.

The algorithms presented in this thesis do not require the mathematicians to have deep knowledge of the algebras they want to explore, nor extensive computer expertise is expected to use our system. Depending on the order of the algebra wanted, the mathematician can easily generate a GAP package of all models, up to isomorphism, within minutes or hours.

Furthermore, the algorithms presented here apply well to many finite model enumerators other than Mace4. For example, constraints solver such as Minion [32] could benefit from the invariant-based isomorphic filtering algorithm to remove isomorphic models they generate.

1.1.3 Third Objective

In order to achieve the third objective, that is, to take advantage of massively parallel computer architectures such as clusters of computers (homogeneous or heterogeneous), the algorithms employed must be scalable and require as few as possible synchronizations. Synchronization among running jobs often substantially slow down the whole process because some jobs may halt to wait for others to reach a certain milestone. The parallel algorithms in this project break down a big problem into a large number of homogeneous sub-problems of smaller sizes. Each of these sub-problems can be processed independently of others. The results of the sub-problems are collected and put together without further processing to combine them.

In the cube-based parallel search algorithm 5.3, for example, the search tree is divided into many smaller search sub-trees. These sub-trees can be searched simultaneously and independently. Furthermore, when a worker process has finished its job, it can ask running workers to split their jobs and it can then take on some of the unfinished work. The overheads for this kind of operation is small, and synchronization is needed only when there are unused computing resources.

Thus, the algorithms in this project are suitable for both massively parallel computer architectures as well as a single multi-core computer.

1.2 Validation

In this research project, all the algebras defined in the MarcieX [9] database (formerly the Axiomatic Library Finder [8]) are used in benchmarking the algorithms as far as possible (see Appendix A for the list of algebras). In case it is not feasible to do performance measurement on all algebras, we use the following set of representative algebras because they each have a structure that appears in many branches of algebra (see Chapter 9):

1. Semigroups (1 associative binary operation)
2. Loops (1 non-associative binary operation, 1 constant)
3. Inverse semigroups (1 binary operation and 1 unary operation)
4. Quandles (2 binary operations)
5. Semigroup sub-variety var$\{N_2^1 \cap [x^2 = y^2]\}$ (1 associative binary operation)
6. Meadows (2 binary operations, 2 unary operations, 2 constants)

The performance on this subset gives a good representation of our new algorithms.

We generate the GAP package for each of these algebras up to orders that are infeasible with the original Mace4. For example, we generate a GAP package of semigroups up to order 7, which contains more than 1.6 million non-isomorphic models.

We compared the number of models, up to isomorphism, generated by our algorithms against the literature as much as possible. We have not found any discrepancies.

1.3 Thesis Outline

This thesis starts in Chapter 1 with clear statements of the objectives to achieve in this research project, with explanation of, and examples showing, the magnitude of the problems of finite model enumeration.

In Chapter 2, we provide a review of some basic mathematical and computer science terminology and basic facts to help ease into the discussions of the novel algorithms in later chapters. In-depth knowledge of computer science is not expected, so we explain all aspect of computer science in layman's term.

In Chapter 3, we describe the improvements made to Mace4 to support the new parallel algorithms that are proposed in this thesis to improve the enumeration all models, up to isomorphism, of a given first-order formula.

In Chapter 4, two parallel algorithms are introduced to speed up the process of filtering out isomorphic models in a set of models. These algorithms make use of invariants and hash maps to cut down the number of models to compare for isomorphism.

Next, in Chapter 5, two novel cube-based parallel algorithms are introduced to speed up the generation of models by a model enumerator such as Mace4. These cube-based parallel algorithm not only efficiently parallelize the model enumeration process, but also reduces significantly the number of models output by the model enumerator. This is done by selectively pruning off part of the search tree that are known to only produce models isomorphic to the remaining search tree. Pruning search tree not only speed up the enumeration process, but also reduces the work to be done by the postprocessing step that removes isomorphic models.

Then in Chapter 6, we show another application of the idea of invariants in computational algebra: `Magmaut`, a GAP package, that compares algebras of type $(2^m, 1^n)$ for isomorphism. It uses invariants to quickly check for non-isomorphism between models, and to guide the construction of isomorphisms between isomorphic models (to prove the models are isomorphic). It also contains functions that generates automorphism groups for magmas.

Next, in Chapter 7, we show two very interesting applications of our new parallel algorithms and the `Magmaut` package in computational algebra. We count the number of models, up to isomorphism, of many common algebras. We also find the INFB monoids (up to isomorphism) up to order 11.

The next chapter, Chapter 8, connects all the dots. In this chapter, we show how the algorithms and the GAP functions described previously are used to generate all models, up to isomorphism, for algebras of small orders, and how these models are packaged into a GAP package for general distribution. The `Magmaut` package provides some functionality need by these GAP packages of algebras of small orders.

Next, in Chapter 9, we show the experimental results in running the algorithms described throughout this thesis.

In Chapter 10, we discuss how the current work can be extended or enhanced.

Finally, in Chapter 11, we summarize what we have achieved, and give some concluding remarks.

Note that it appears logical to discuss the cube-based search algorithms for enumerating models before introducing the invariant-based algorithms for removing isomorphic models generated by the cube-base search algorithms. However, the former algorithms make use of the latter algorithms, so the algorithms are presented in the order as they are to make them easy to follow.

The reader who are familiar with algebra and basic computer science concepts may skip Chapter 2 and refer back to it for background information only when needed.

Chapter 2

Preliminaries

In this chapter, we will discuss the mathematical, computer science, and other background that is necessary for the complete discussion of this thesis.

2.1 Definitions and Preliminaries

We give a brief overview of the mathematics used in the subsequent chapters; we draw mainly from Chapter 2 and Chapter 5 of [19].

A (finite) relational algebra is a triple (D, Σ_F, Σ_R), where D is an non-empty set and Σ_F is a tuple of operations $(f_1, \cdots, f_k)$, that is, functions $f_i : D^{n_i} \to D$ and Σ_R is a tuple of relations $(R_1, \cdots, R_j)$, i.e., $R_i \in \Sigma_R$ is a subset of D^{n_i}. In other words, Σ_F and Σ_R are indexed by finite index sets. In this case, f_i, is said to be an operation of *arity* n_i. The arity of R_i is defined analogously. Furthermore, at least one of Σ_F and Σ_R is non-empty.

A (first-order) *signature* lists the operations and relations that characterize an algebraic structure. In a structure, an interpretation ties the function and relation symbols to mathematical objects that justify their names.

The simplest algebraic structure is the *magma*:

Definition 2.1.1. *A magma is an algebraic structure that consists of a set equipped with a single binary operation.* □

In this thesis, the algebras under discussion are of type $(2^m, 1^n)$:

Definition 2.1.2. *An algebra of type $(2^m, 1^n)$, where $m, n \geq 0$ and $m + n > 0$, is an algebra defined with m binary operations/relations and n unary operations/relations.* □

The *order* of a relational algebra is the size of its domain D. In this thesis, $D = \{0, \cdots, n-1\}$, where $n \geq 2$, i.e., we exclude the trivial case of searching on domains of size 1. Recall that examples of relational algebras are all imaginable algebras, (di)graphs, etc.; in the following, by algebra we mean relational algebra.

Let π denote an arbitrary permutation on D, π_{id} denote the identity permutation, and $\pi_{(a,b)}$ denote the permutation cycle (a, b). For example, $\pi_{(0,1)}$ is the permutation cycle $(0, 1)$.

Definition 2.1.3. *A generating set G of an algebra A is a subset of A such that every domain element of A can be expressed as combination (under the operations of A) of finitely many elements of G.* □

While the concept of isomorphism is ubiquitous to scientific literature, its definition appears under slight variations. Throughout this dissertation, we rely on the following definition.

Definition 2.1.4. *Let A and B be structures defined on the same signature Σ_F, Σ_R. A function f from A to B is said to be an* isomorphism *if it is a bijection and preserves all operations and relations. This means that if $g_i \in \Sigma_F$, with the respective interpretations g_i^A and g_i^B in A and B, then $f(g_i^A(a_1, \ldots, a_n)) = g_i^B(f(a_1), \ldots, f(a_n))$, for all $a_1, \ldots, a_n \in A$.*

Analogously, f preserves $R_i \in \Sigma_R$ of arity n, with the respective interpretations R_i^A and R_i^B in A and B, when the following is true: $(a_1, \ldots, a_n) \in R_i^A$ iff $(f(a_1), \ldots, f(a_n)) \in R_i^B$, for all $a_1, \ldots, a_n \in A$. □

Definition 2.1.5. *Two structures A and B are said to be isomorphic if there is an isomorphism from A to B, or vice versa.* □

If A and B are the same structure, then the isomorphism between them is called an automorphism.

Definition 2.1.6. *An automorphism is an isomorphism of a structure with itself.* □

An important property of isomorphisms is that they preserve sets defined by some fixed first-order formula. More precisely, suppose we have two finite relational algebras A and B, on a signature Σ, isomorphic under $f : A \to B$. In addition, suppose we have a set S contained in A^k and definable by a first-order formula Φ in the language of Σ. Then $f(S)$ is precisely the subset of B^k that satisfies Φ (*cf.* Theorem 1.1.10 in [47]).

For example, suppose we have two isomorphic finite algebras: $(A, *)$ and $(B, *)$, with $f : A \to B$ an isomorphism. Suppose also that S is the set $\{(x, y) \in A^2 \mid (\exists z \in A)\ (x *^A z) *^A y = x *^A (z *^A y)\}$. As S is the set of all elements in A^2 that satisfy a FOL in a language with the function symbol $*$, then $f(S) := \{(f(x), f(y)) \mid (x, y) \in S\}$ is precisely the set of all pairs $(x, y) \in B^2$ such that $(\exists z \in B)\ (x *^B z) *^B y = x *^B (z *^B y)$.

This idea is usually expressed by saying that sets definable by FOL formulas are invariant (or preserved) under isomorphism. This guarantees that when we split the list of algebras into blocks using invariants based on defining formulas, algebras in different blocks are non-isomorphic; algebras inside the same block might be isomorphic or non-isomorphic. Therefore, to discard the redundant algebras we only have to check within each block. This is the ground for our invariants-based algorithm in Chapter 4. For future reference, we state these considerations as a proposition.

Proposition 2.1.1. *Let A and B be algebras of a signature Σ and $f : A \to B$ be an isomorphism from A to B. Then any ordered tuple $(a_1, \ldots, a_m) \in A^m$ satisfies a first-order formula in the common language Σ if and only if the ordered tuple $(f(a_1), \ldots, f(a_m)) \in B^m$ satisfies the same first-order formula in B.*

2.2 Computer Science Terminology

2.2.1 Hash map

Hash map, also known as hash table, is a very efficient data structure used to implement an associative array that maps keys to values. The basic idea of hashing is to distribute the key/value pairs across an array of buckets. Given a hash table (array) of size m, and a data item such as a character string or a vector of integers, a hash function is used to calculate an integral hash key n, in the range of $[0, m-1]$, from the data item. The data item is placed n^{th} slot of the hash table. The following example will help make this idea clear.

Suppose the hash table HT is defined as an array with 3 buckets, or slots. Each bucket may contain multiple data items. Suppose further that the data items are vectors of non-negative integers. The hash key of a vector is calculated as the remainder of the sum of elements in the vector divided by 3.

For example, for the vector $(1, 0, 3, 4)$, the hash key is the remainder of $1 + 0 + 3 + 4 = 8$ divided by 3, which is 2. The data item will be placed in the $HT[2]$ bucket.

Hash maps can be very efficient in searching. If we have a list of 100 vectors of integers hashed into a hash table of size 3 as in the example, and we want to find out whether the vector $(1, 0, 3, 4)$ is one of them. Instead of searching for the vector in the full set of 100 vectors, we only need to search in the set of vectors in the $HT[2]$ bucket.

Note that hash keys are often not unique among the data items. As in the previous example, the vector $(0, 1, 3, 4)$ also has the hash key of 2. When 2 data items are hashed into the same bucket in a hash table, we say that a *collision* occurs.

The effectiveness of hashing depends on

1. The ratio of amount of data and the size of the hash table. The more slots available in the hash table, the less likely collisions will occur.

2. The ability of the hash function to discriminate data items and assign different hash keys to them to avoid collisions.

Fortunately, hash tables with good hash functions are available in libraries in most mainstream computer languages such as C++ and Python, which are used extensively in this project.

2.2.2 Greedy Algorithm

Definition 2.2.1. *A greedy algorithm is any algorithm that uses some heuristic of making the locally optimal choice at each step.* □

In general, a greedy algorithm does not guarantee to produce an optimal solution, but which often has much lower computational complexity than algorithms that always produce optimal solutions. For example, suppose we want to find a generating set of a magma A of order n. One way is to generate all subsets of A, and find out which ones can be expanded to A by systematically checking the finite combinations of the elements in the subset. Then the minimal generating set can be found. There are

2^n subsets, so this algorithm is quite computationally intensive. An alternative (a greedy algorithm) is to start with an empty set G, then for each unused element a of A, expand $G \cup \{a\}$ to a sub-magma of A (by including finite combinations of elements of $G \cup \{a\}$). Add to G only the element that gives the biggest sub-algebra. Repeat the steps until G expands to A. This G is a generating set of A, but there is no guarantee that it is the minimal generating set. Algorithm 2 and Algorithm 6 are examples of greedy algorithm.

2.3 Finite Model Enumeration

Given a signature Σ and a first-order formula $\mathcal{F}$ on Σ, a traditional finite model finder first expands the FOL to its ground terms by its domain elements in D, then searches for models by applying a depth-first search with backtracking to systematically and exhaustively explore the search space [70]. The domain elements $D = \{0, \ldots, n-1\}$ are seen as special constants not appearing in the original $\mathcal{F}$, c.f. [59].

We will follow the terminology in [70] in describing the finite model enumeration process.

Definition 2.3.1. *The assignment $f_i(a^i_0, \cdots, a^i_{k_i-1}) = v_i$, where f_i is a k_i-ary function symbol in Σ and $a^i_j, v_i \in D$, is called a* value assignment *(VA)* clause*, and $f_i(a^i_0, \cdots, a^i_{k_0-1})$ is called a* cell term*, or simply a* cell*.* □

The above definition follows from the concept that searching for a finite model is done by filling the cells of the multiplication table of f.

To search for finite models in $\mathcal{F}$, the finite model finder employs a *cell selection* strategy to pick cell terms successively, without duplicates, to assign values from D to form VA clauses. If the VA clause causes failure of any axiom in $\mathcal{F}$, then a new value will be tried. If no value can form a VA clause that does not cause the failure of the axioms in $\mathcal{F}$, then the model finder backtracks to the previous cell term to try to assign another value to it. When all cell terms in $\mathcal{F}$ are assigned values without violating the axioms in $\mathcal{F}$, a model, as represented by its VA clauses, is found. After a model is found, the process can continue with backtracking to find other models. Not all models obtained from a finite model finder are non-isomorphic. Some models can be transformed to others by renaming their domain elements. That is, by applying a permutation (more generally, an isomorphism) on the domain D. If there is an isomorphism from one model to another, then the two models are said to be isomorphic.

The search space can be organized as a search tree in which nodes are VA clauses. The root node is an empty VA clause. The cell term in each node is selected by the cell selection strategy. 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 i^{th} position of the sequence and $v_i \in D$, and $t_i \neq t_j$ when $i \neq j$. Furthermore, a search path will be terminated at the first VA clause that causes a violation of any axiom of $\mathcal{F}$.

If the length of a search path is the same as the total number of cell terms in $\mathcal{F}$, then it is a complete search path and its VA clauses represent a model of $\mathcal{F}$. Otherwise, it is an incomplete search path representing *partial assignments* of cell values in $\mathcal{F}$.

The backtracking algorithm in its simplest form is very costly. If all possible value assignments are examined in every cell, the search will have to go through a huge number of possible combinations of values. For example, to search a FOL formula $\mathcal{F}$ with just one binary operation, there will be n^{n^2} possible combinations (n^2 cells with n possible value each). Even the very small domain size of 4 will give over 4 billion combinations of cell values. However, in practice, the number of viable combinations to check is much smaller than the maximum number of theoretical possibilities because of the constraints imposed by $\mathcal{F}$. Furthermore, a finite model finder may infer new VA clauses from existing ones by *propagation*.

Example 2.3.1. *Suppose the first-order formula contains only the equation $f(x, y) = f(y, x)$, that is, the operation f is commutative. After the assignment $f(0, 1) = 0$, the finite model finder can infer $f(1, 0) = 0$. This is referred to as positive propagation.*

On the other hand, if the first-order formula contains the inequality $f(x, y) \neq f(y, x)$, then after the assignment $f(0, 1) = 0$, the finite model finder can exclude 0 as a possible value for the cell $f(1, 0)$. This is referred to as negative propagation.

Many algorithms and heuristics have been devised to speed up the process by cutting down the number of search paths. We will discuss one such algorithm, a very important one, in the following section.

2.3.1 Least Number Heuristic

The least number heuristic (LNH) [11, 71, 72] is a very effective symmetry-breaking algorithm widely implemented in model finders/enumerators, such as Mace4. The main idea of the LNH is that all domain elements that have not yet appeared in any VA clauses and the current cell term in the search are equivalent to each other and therefore only one of them, say, the smallest one, needs to be considered in a cell value assignment.

To ease further discussions of the LNH, we introduce the notation Vals(P) to denote the set of all domain elements appearing in P, where P can be a search path, a VA clause, or a cell term.

Example 2.3.2. *For the cell term $f(1, 1)$: Vals$(f(1, 1)) = \{1\}$.*

For the VA clause $f(1, 1) = 0$: Vals$(f(1, 1) = 0) = \{0, 1\}$.

For the partial search path $S = \{f(0, 0) = 0; f(1, 1) = 0\}$: Vals$(S) = \{0, 1\}$. □

The LNH can now be stated precisely: In adding a VA clause, $t = v$, to extend a search path S, the possible choices of v allowed under the LNH are Vals$(S) \cup$ Vals$(t) \cup \{s\}$ where s is the smallest domain element in $D \setminus ($Vals$(S) \cup$ Vals$(t))$.

Strictly speaking, it is not necessary to set s to be the smallest domain element not seen so far, it could as well be the biggest one, for example. But the rule to set s must be unambiguous — only one value is consistently picked by the rule each time. In this research project, we will always set s to be the the smallest domain element not seen so far.

We prove in Corollary 5.2.1 (Chapter 5, Section 5.2) that any exhaustive search under the LNH will not lose any non-isomorphic models.

Example 2.3.3. *Given the first-order formula* $(x * y) * z = x * (y * z)$*, the model finder Mace4 generates 7,743,056,064 models in 1,010 minutes without the LNH, but only 1,021,120,198 models in 135 minutes with the LNH. That is, the LNH enables Mace4 to generate 7 times fewer (isomorphic) models in one-seventh of the time. This also makes the post-processing step of removing isomorphic models much easier because there are 7 times fewer models to process. The experiments are run on an Intel® Xeon®Silver 4110 CPU 2.0 GHz* $\times$*32 computer, with 64 GB RAM.* □

Definition 2.3.2. *A search path is* LNH-compliant *if it respects the LNH restrictions on the choices of values assigned to its VA clauses.* □

Example 2.3.4. *Suppose the domain size is 4. The complete search path* $\{f(1) = 0; f(0) = 3; f(3) = 1; f(2) = 1\}$ *is not LNH-compliant.*

For the first VA clause in the search path, $S = \emptyset$ *and* $t = f(1)$*. So,* Vals$(S) \cup$ Vals$(t) = \emptyset \cup \{1\} = \{1\}$*, and therefore* $D \setminus ($Vals$(S) \cup$ Vals$(t)) = \{0, 2, 3\}$*. Thus,* $s = min(\{0, 2, 3\}) = 0$*. The LNH limits the choices of the value for* $f(1)$ *to* Vals$(S) \cup$ Vals$(t) \cup \{s\} = \{0, 1\}$*. So the first VA clause* $f(1) = 0$ *is LNH-compliant.*

However, for the second VA clause in the search path, $S = \{f(1) = 0\}$ *and* $t = f(0)$*. So,* Vals$(S) \cup$ Vals$(t) = \{0, 1\} \cup \{0\} = \{0, 1\}$*, and therefore* $D \setminus ($Vals$(S) \cup$ Vals$(t)) = \{2, 3\}$*. Thus,* $s = min(\{2, 3\}) = 2$*. The LNH limits the choices of the value for* $f(0)$ *to* Vals$(S) \cup$ Vals$(t) \cup \{s\} = \{0, 1, 2\}$*, so* $f(0) = 3$ *is not allowed under the LNH. Therefore, the whole search path is not LNH-compliant.* □

Two important observations on the LNH:

1. The LNH is more effective at the beginning than at the end of the search path. Once $n - 1$ domain elements are seen in the beginning of search path, the LNH is rendered useless in subsequent extensions of that search path because all n domain elements are allowed in all subsequent VA clauses.

2. The LNH speeds up the search process by limiting the choices of the values for the cell terms. It does not impose any restrictions on the order of the cell terms in the search path. However, in practice, a number called the *maximal designated number* (*mdn*) is often used to partition the domain into 2 subsets so that $\{0, \cdots, mdn\}$ are domain elements already seen, and $\{mdn + 1, \cdots, n - 1\}$ are domain elements not seen so far [70]. In this case, cell selection strategies that keep the *mdn* small are preferred because the search tree will be kept narrow. The use of mdn simplifies the implementation at the expense of the generality of the LNH.

Example 2.3.5. *The* concentric cell selection strategy *is a simple cell selection strategy to minimize the growth of choices of values in the finite model search with the LNH. This strategy picks the cell* $f(a_0, \cdots, a_{k-1})$ *with the least* $r = max(a_0, \cdots, a_{k-1})$ *from all available cells. Any tie-breaker can be used in case of a tie. For example, in searching models defined by a binary operation, one of the possible orders of the cells selected by this cell selection strategy is* $f(0, 0)$*,* $f(1, 1)$*,* $f(0, 1)$*,* $f(1, 0)$*,* $f(2, 2)$*,* $f(0, 2)$*,* $f(2, 0)$*,* $f(2, 1)$*,* $f(1, 2)$*,* $\ldots$*. That is,* $r = 0$ *for the first cell,* $r = 1$ *for the next 3 cells,* $r = 2$ *for 5 subsequent cells, and so on.* □

2.3.2 Cube

Definition 2.3.3. *A cube is a prefix of a search path.* □

Example 2.3.6. *The complete list of cubes of the search path* $\{f(1) = 0; f(0) = 3; f(3) = 1; f(2) = 1\}$ *given in Example 2.3.4 are:*

$\{f(1) = 0\}$

$\{f(1) = 0; f(0) = 3\}$

$\{f(1) = 0; f(0) = 3; f(3) = 1\}$

$\{f(1) = 0; f(0) = 3; f(3) = 1; f(2) = 1\}$ □

As a prefix of a search path, the cube contains VA clauses. Permutations and isomorphisms can be applied to a cube by applying them to its VA clauses.

Definition 2.3.4. *If* π *is a permutation on* D *and* B *is a cube, then*
$\pi(B) \coloneqq \{f(\pi(a_1), \ldots, \pi(a_k)) = \pi(v) \mid f(a_1, \ldots, a_k) = v$ *is a VA clause in* $B\}$. □

Observe that $\pi_{id}(B)$ is the (unordered) set of all individual VA clauses in the cube B.

Note that predicates in a first-order formula are usually considered functions with two values, T (true) and F (false), which do not affect the LNH because they are not domain elements. For convenience, we consider $I(T) = T$ and $I(F) = F$ for all isomorphisms I so that the same terminology is used for both relations and functions.

Definition 2.3.5. *Two cubes are said to be isomorphic if their VA clauses are isomorphic.* □

In particular, two cubes B_0 and B_1 of the same search tree are isomorphic if there is a permutation π on D such that $\pi(B_0) = \pi_{id}(B_1)$, because in this case, one set of VA clauses can be transformed into another by renaming the domain elements.

Example 2.3.7. *If* $B_0 = \langle f(0) = 0; g(0,0) = 0; f(1) = 0; g(1,1) = 0\rangle$ *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* $\pi_{(0,1)}(B_0) = \{f(1) = 1, g(1,1) = 1, f(0) = 1, g(0,0) = 1\} = \pi_{id}(B_1)$. □

Chapter 3

Mace4

3.1 Overview

Mace4 [48] was first developed as a companion to Prover9, which is a very popular automatic theorem prover among mathematicians. The idea behind the Prover9/Mace4 [49] pair is that Prover9 is used to prove a hypothesis and Mace4 is to refute it by finding a counterexample. They are often run in parallel, side-by-side, so that the process is complete when either Prover9 proves the hypothesis, or Mace4 finds a counterexample to refute it. Both processes end when one of them succeeds or when a preset maximum process time is reached.

Mace4 can also be used as a finite model enumerator or finder without modifications. There is an input parameter to instruct Mace4 to find an exact number of models or find all models. As a traditional finite model enumerator, Mace4 follows the algorithm outlined in Section 2.3, with LNH built in its search engine.

Mace4 compares favorably with many newer model enumerators, especially for mathematicians who may not be computer experts or who wants to concentrate on mathematics and not on the tools of mathematics. Its strengths include:

1. Its input language is FOL, which has a very intuitive syntax and is easy to learn.

2. A lot of utilities are built around it, including programs to extract non-isomorphic models and to translate inputs to feed other systems such as TPTP [67].

3. It has good performance as a model generator. For example, it outperforms Minion and IDP^3 at least when the inputs are not tuned by experts (see Tables 9.2 and 9.3).

4. Prover9/Mace4 has a large user base. They have been used by many mathematicians for many years (since 2003), and the code base is stable and reliable.

3.2 Mace4 Code

Prover9 and Mace4 share a lot of common code provided by the Library for Automated Deduction Research (LADR), which was originally written in the programming language C (C99 standard). The latest known C version of the code base is LADR-2017-11A, which contains the LADR library, Prover9, Mace4 and many other auxiliary programs such as *interpformat* (for transforming models between different formats), *isofilter* (for filtering out isomorphic models), and *interpfilter* (for filtering out interpretations).

The code base of LADR library and the Prover9, along with some auxiliary programs embedded in the LADR library, were re-written in C++ recently, and is available as `https://gitlab.com/cfmsousa/prover9` [66]. Based on this C++ LADR library, we first port the C-based Mace4 program, and many other utility programs, to C++ and then add numerous improvements to them as described in the following sections. The main goal of the upgrades of Mace4 is to make it a modern model finder and enumerator that supports parallel algorithms.

3.3 Porting Mace4 to C++

The first major improvement made to Mace4 is porting it to C++. C is a low-level programming language that is the go-to programming language for writing very efficient code. It allows direct access to memory and to many low-level computer operations. However, it is not easy to write good extensible code quickly, or to model real world objects effectively, in such a low level programming language. It often requires a lot of re-writing, testing, and debugging when new features are added to a library or program written in C. Furthermore, there are also not many libraries built around it.

Fortunately, C++, which is a mature and stable extension of C, is available for developing high performance applications. As an extension of C, C++ maintains its backward compatibility with C, making the transition safe and easy. Low level C constructs remain available in C++. In addition, C++, being an object-oriented programming language, has many new programming constructs built on top of C to allow effective modeling of real world objects and concepts. It also has a lot of programming constructs that makes it easy to write secure and safe code without loss of execution efficiency. Furthermore, there are many useful libraries and data structures, such as hash-map, built in C++ that allow programmers to quickly write new programs with minimal efforts. C++ code is usually much easier to understand, maintain, and verify than C code.

As a model enumerator, the C++ version of Mace4 is found to be about 20% faster than the C version even though the C++ version of Mace4 is nothing more than a simple and direct conversion of the C version of Mace4 (see Table 9.4). There are no differences in the algorithms implemented in the two versions. The performance boost from the C++ version could conceivably be due to the fact that the C++ compiler is a better compiler than the C compiler. More effort is put into improving the C++ compiler than the C compiler. Moreover, libraries are added to the C++ platform at a much faster pace than that to the C platform. This trend is expected to continue in the foreseeable future because the computer industry is betting on the C++ compiler which is gaining popularity.

Mace4 is also upgraded to a 64-bit application, so that it can handle bigger problems.

3.4 Streamlining Mace4 Outputs

The second major improvement made to Mace4 is specific to its use as a model enumerator. Mace4 prints out models as it finds them, in a format easy for human to read, along with a lot of related information such as number of models found, number of various steps to get the models, and the run time etc. The size of the raw outputs of Mace4 could be huge for higher orders. To post-process the models emitted by Mace4, a special auxiliary program, `interpformat`, is used to extract these models from the raw Mace4 output and re-write them into a format that is good for other auxiliary programs such as the `isofilter` which removes isomorphic models. This is a very time- and memory-consuming step except for very small algebras. For example, it takes the original C version of Mace4 2 seconds to generate 90,536 semigroups of order 5 into a file of size of 32.7 MB. Then it takes `interpformat` 157 seconds to extract these semigroups from the Mace4 file to put the models into a file of size 15.7 MB (see Table 9.5). That is, it takes 78 times longer to extract the models from the Mace4 output file than the time to generate the models themselves. (Timing in this section is obtained on an Intel® Xeon®Silver 4110 CPU 2.0 GHz $\times$32 computer, with 64 Gb RAM.)

The problem becomes intractable even for semigroups of order 6. It takes the original C version of Mace4 159 seconds to generate 5,628,898 semigroups of order 6. The Mace4 output file is of size 2.1 Gb. It takes `interpformat` more than 4.5 hours just to process 123 thousand models from the Mace4 output file of over 5.6 million models (the `interpformat` process is terminated since it takes too long).

To save time and space, an input parameter, **-A1**, is added to signal Mace4 that the output models are to be printed out in a separate file and in a format that is compatible with the utility programs in the Prover9/Mace4 microcosm. Mace4 is modified to suppress the printing of most of the non-essential information, and to emit the models separately in a compact format that can be consumed by other auxiliary programs such as the `isofilter` when this new option is turned on. With this option turned on, it takes Mace4 the same 2 seconds to generate the same 90,536 semigroups of order 5 into a separate file of size 13.5 MB. There is no need to run the `interpformat` process because all models are already in a separate file, and are in the format suitable for other postprocessing steps. For semigroups of order 6, the new option allows Mace4 to generate the same 5,628,898 models in shorter time (128 seconds) into a separate file of size 1.0 Gb (see Table 9.5), without the need to use interpformat to extract of models from the Mace4 outputs.

Similar pattern of improvements are seen with other algebras such as the loops (see Table 9.5).

3.5 Support Searching with Cubes

The last but not least major improvement on Mace4 is the support of cubes. A cube is a prefix of a search path, which represents partially filled operation tables. A cube can also be considered the *prefix*

of a search sub-tree. This feature allows the use of Mace4 to efficiently search multiple sub-trees in parallel.

Here we present the brief idea of using cubes to do parallel search of models with Mace4. In-depth discussion of parallelization of model searching with Mace4 will be given in Chapter 5.

First, Mace4 is modified to generate cubes. A new optional input, **-C**n, where n is the length of the desired cubes, tells Mace4 to print out all cubes of length n, and not to complete the search for models. Some pairs of cubes may be isomorphic to each other. We can discard one cube from each pair of isomorphic cubes because isomorphic cubes only lead to isomorphic models (see Section 5.2).

Lastly, Mace4 is modified to read in a *cube.config* file which contain cubes. It then populates the operation tables with a cube in that file, and continue searching from the partially filled operation tables.

These two new features allow us to use Mace4 for more complex search strategies. For example, we can use Mace4 to generate short cubes first. Next, we detect and discard isomorphic cubes from the short cubes, then use Mace4 to read in the non-isomorphic short cubes and generate longer cubes. Again, isomorphic cubes are detected and discarded from the outputs at this stage. The process can be repeated to extend the length of the cubes to the desired length. Then the final set of cubes can be processed in parallel to enumerate all models.

3.6 Integration with Master Prover

The Master Prover[1] is an enhanced version of the Prover9/Mace4 system. One of the powerful features of the Master Prover is its use of portfolio of first-order theorem provers including Prover9, Waldmeister [18], Vampire [42], and E-Prover [63], to solve the same problem simultaneously. That is, given a conjecture, the Master Prover runs all the chosen provers in the portfolio and Mace4 in parallel to prove it or find a counterexample to disprove it. The Master Prover stops all solvers when one of them proves the conjecture or finds a counterexample. The new C++ version of Mace4 is added to this system as one of the solvers for finding counterexamples.

[1] Available as `https://gitlab.com/cfmsousa/master-prover/-/wikis/Master-Prover`

Chapter 4

Filtering Isomorphic Models with Invariants[1]

4.1 Overview

In this chapter, we present:

1. In-depth discussion of the definition and properties of invariants (Section 4.2)).
2. A small basic set of invariants that have high discriminating powers, and yet are inexpensive to compute (Section 4.3).
3. How randomly generated invariants are added to the invariant-based algorithm to help discover invariants of high discriminating powers (see Section 4.4).
4. How a hash-map is used to store models partitioned by the invariant-based algorithm to allow fast storage and retrieval of models having the same set of invariant vectors. It also helps to avoid comparing models with different sets of invariant vectors. (Section 4.5).
5. An invariant-based parallel algorithm that can be applied to enumerate non-isomorphic models of algebras defined by a first-order formula containing at least one binary operation or relation (Section 4.5).

Redundant (with respect to isomorphism) models may either be eliminated during the search phase or filtered out afterwards. Guaranteeing that search never produces isomorphic models is a hard problem and is rarely done in modern finite model enumerators. In this chapter, we tackle the second problem, i.e., the removal of redundant models from an already enumerated set.

In the context of finite model enumeration, the complexity of checking whether two models are isomorphic is only part of the problem. Another source of complexity is the large number of models that need to be checked. If every model is checked against all others, then the performance degrades rapidly as the total number of models increases.

[1] This chapter is adapted from articles [4, 5].

If we assign each domain element in a generated model a vector that is invariant under isomorphism (see Section 4.2) and put all models having the same multiset of invariant vectors into separate blocks, then models across the blocks are guaranteed not to be isomorphic (see Section 4.5). This splits the problem into substantially smaller sub-problems. Moreover, processing of the blocks can easily be done in parallel as models across blocks cannot be isomorphic. Parallel processing is an important facet of our approach since modern-day computers are more often than not equipped with multiple cores.

4.2 Invariants

We define an invariant as a function that accepts an element and a structure of which the function's value is preserved (invariant) under isomorphism. The motivation for this definition is that if two structures' invariants give different multisets of values, then the structures are *not* isomorphic, but not necessarily the other way around.

Definition 4.2.1 (invariant)**.** *Let I be a function that accepts a structure and an element that returns an integer. We say that I is an* invariant *if $I(A, x) = I(B, f(x))$ for all isomorphic structures A and B with isomorphism $f : A \to B$ and $x \in A$.*

The key observation is that if invariants are calculated for all elements, then isomorphic structures result in the same multiset of values.

Proposition 4.2.1. *Let I be an invariant and A, B structures. Define the multisets $M_A = [I(A, x) \mid x \in A]$, and $M_B = [I(B, x) \mid x \in B]$. If A and B are isomorphic then $M_A = M_B$.*

Proof. Let f be an isomorphism from A to B. There is a bijection between M_A and M_B that maps $v = I(A, x)$ to a unique $v \in M_B$ with $v = I(B, f(x))$. □

Example 4.2.1. *Let the invariant I count the number of times an element is obtained in a given structure, i.e., $I(\langle D, \odot \rangle, x) = |\{ d_1, d_2 \in D \mid d_1 \odot d_2 = x \}|$. Consider multiplication and addition modulo 2 on $D = \{0, 1\}$, i.e., $A = \langle D, \odot = * \rangle$ $B = \langle D, \odot = +_{mod\,2} \rangle$. We get the corresponding multisets, $[3, 1]$ and $[2, 2]$, which are different and therefore A and B are* not *isomorphic.*

We combine multiple invariants by calculating a vector of values for each element. This is further illustrated by a detailed example in Section 4.5.

Corollary 4.2.1. *Let $I_1, \ldots, I_k$ be invariants and A, B structures. Define the multisets of vectors of integers $V_A = [(I_1(A, x), \ldots, I_k(A, x)) \mid x \in A]$, and $V_B = [(I_1(A, x), \ldots, I_k(B, x)) \mid x \in B]$. If A and B are isomorphic then $V_A = V_B$.*

4.3 Basic Invariants

The goal of this section is to introduce our list of basic invariants. We start by observing that the axioms satisfied by an algebra might render some invariants useless. For example, if the algebra is a group,

then the invariant that counts the number of inverses of a domain element (see **U3**) would be useless for distinguishing non-isomorphic models because there is exactly one inverse for each domain element. Thus, we need multiple invariants with deep algebraic meaning and high discriminating power in order to target as many different algebraic properties/classes as possible. On the other hand, we should choose properties that are inexpensive to compute and not very many of them as that could slow down the computation.

Our choices of invariants are based on concepts ubiquitous in various kinds of algebras. For example, one of our invariants (**B5**) is based on the fact that idempotents appear in many algebras; in particular, it is well-known that every finite semigroup has at least one idempotent and hence this invariant is useful for a wide range of algebras, especially those that have a semigroup reduct.

As observed above, the overwhelming majority of the most popular algebras are defined using operations of arity at most 2 (see page 26 of [19]), so we design most of our invariants around binary operations. We have 10 invariants for binary operations, 4 for binary relations, 4 for unary operations, and 1 for ternary operations to target different common algebraic structures. Together they have a high discriminating power, and yet are easy and inexpensive to compute. In Section 4.4, we shall discuss how random invariants can be constructed from an arbitrary combination of operations of different arities.

In the following discussions on invariants, the domain of the algebra is denoted by $D = \{0, \ldots, n-1\}$.

4.3.1 Invariants from Unary Operations

For each unary operation g in the algebra, we compute the invariants for each element $x \in D$:

U1 1 if $g(x) = x$, 0 otherwise (fixed point);

U2 1 if $g(x) \neq x$ and $g(g(x)) = x$, 0 otherwise (transposition);

U3 The number of $y \in D$ such that $g(y) = x$ (size of the inverse image);

U4 The number of $y \in D$ such that $g(g(y)) = x$ (size of the inverse image under g^2).

The correctness of **U1** and **U2** as invariants follows readily from Proposition 2.1.1. The correctness **U3** and **U4** also follows from Proposition 2.1.1, with the note that isomorphism is a bijection. Let us illustrate the proof on the invariant **U3**.

Proposition 4.3.1. *The function **U3** is an invariant (Definition 4.2.1).*

Proof. Let f be an isomorphism of structures A and B with a unary function g. We need to prove that **U3**$(A, d) =$ **U3**$(B, f(d))$ for any $d \in A$. Construct the sets of solutions of $g(y) = x$ for both of considered structures, i.e., define $S_C = \{(x, y) \in C \times C \mid g(y) = x\}$, for $C \in \{A, B\}$. For an element $d \in A$ consider the set $S_A^d = \{(d_1, d_2) \in S_A \mid d_1 = d\}$. Analogously, consider the set $S_B^{f(d)} = \{(d_1, d_2) \in S_B \mid d_1 = f(d)\}$. From Proposition 2.1.1, f is a bijection between S_A and S_B, i.e. $(d_1, d_2) \in S_A$ iff $(f(d_1), f(d_2)) \in S_B$. But therefore also f is a bijection between S_A^d and $S_B^{f(d)}$. This means that $|S_A^d| = |S_B^{f(d)}|$, i.e., the values of **U3** for the element d and $f(d)$ are the same. □

Similar arguments can easily be used to directly prove the correctness of the other invariants.

4.3.2 Invariants from Binary Operations

For each domain element $x \in D$, we compute the following invariants for each binary operation in the algebra:

B1 The smallest integer n such that $x^n = x^k, n > k >= 1$ where we define x^n to be $(\ldots(x*x)*x)*x)\ldots$ for n x's (*periodicity*).

B2 The number of $y \in D$ such that $x = (xy)x$ (*number of inverses*).

B3 The number of distinct xy for all $y \in D$ (*size of right ideal*).

B4 The number of distinct yx for all $y \in D$ (*size of left ideal*).

B5 1 if $xx = x$, 0 otherwise (*idempotency*).

B6 The number of $y \in D$ such that $x(yy) = (yy)x$ (*number of commuting squares*).

B7 The number of $y \in D$ such that $x = yy$ (*number of square roots*).

B8 The number of $y \in D$ such that $x(xy) = (xx)y$ (*number of square associatizers*).

B9 The number of pairs of $y, z \in D$ such that $zy = yz = x$ (*number of commuting pairs*).

B10 The number of $y \in D$ such that there exist pairs of $s, t \in D$ where $x = st$ and $y = ts$ (*number of conjugates*).

4.3.3 Invariants from Binary Relations

For each domain element $x \in D$, the following invariants are calculated for each binary relation R:

R1 The number of distinct y such that $R(x, y)$.

R2 The number of distinct y such that $R(y, x)$.

R3 1 if $R(x, x)$, 0 otherwise. (reflexivity)

R4 The number of y such that $R(x, y)$ & $R(y, x)$.

4.3.4 Invariants from Ternary Operations

Ternary operations are very rare in the FOLs defininig the operations of algebras. Indeed, no ternary operation exists in the definition in any of the operations of the 158 algebras listed in the ALF database [8], although a few of them come from the Skolemization of binary operations. Moreover, calculations involving ternary operations are often very expensive as deeply nested loops are involved. Thus, only one simple invariant would be included in the algorithm. For each domain element $x \in D$, we compute one invariant for each ternary operation:

T1 The number of times x appears in the ternary operation table (frequency).

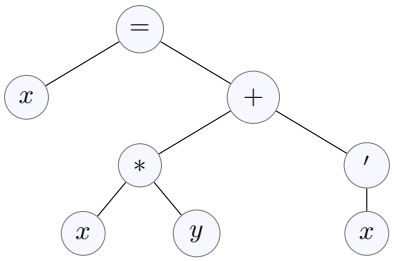


Figure 4.1: Expression tree for $x = (x * y) + x'$.

We call the hand-crafted invariants listed above the *basic invariants* to differentiate them from the randomly generated invariants which will be discussed in Section 4.4. It should be simple to see that the validity of these basic invariants follows from Proposition 4.4.1.

4.4 Random Invariants

As discussed in Section 4.3, we need different invariants to target different algebraic structures. The basic invariants are inspired by our knowledge of the most popular algebras, however, there are many other less common algebraic structures and new ones are continually appearing.

Therefore, we need a general way (adaptable to each class of algebras) of generating invariants with good discriminating power. A practical solution to this problem is to generate a large set of invariants with a random number of operations and a random number of variables, and then automatically discover the best subset to use.

4.4.1 Generation of Random Invariants

A first-order formula can be represented by an expression tree with operations in the internal nodes and variables in the leaves. For example, Figure 4.1 shows the tree representation of the first-order formula

$$x = (x * y) + x' \tag{4.1}$$

It is simple to randomly generate such an expression tree, as shown in Algorithm 1, although it is difficult to guarantee uniform distribution of operations in the expression trees. For simplicity, the root is set to be an operation that evaluates to true or false, and it always has two children. It is the only node that can be assigned the equality relation. It may also be a binary relation if there is one in the algebra.

Proposition 4.4.1. *If we fix a variable in the first-order formula generated by Algorithm 1 as the base variable, then the number of solutions will be an invariant.*

Proof. Correctness is shown as in Proposition 4.3.1. □

We may choose any of the variables in the randomly generated first-order formula as the base variable for the invariant. For example, if we choose x as the base variable for the first-order formula (4.1),

Algorithm 1: Generation of Random First-order Formula

input : A list of binary/unary operations/relations, max depth of tree, list of variables
output: An expression tree representing a first-order formula

```
1  BuildNode (level) begin
       /* recursively build a node                                  */
       /* maxLevel is maximum depth of tree allowed                 */
2      if level = maxLevel then
3          nodeType ← leaf
4      else
5          nodeType ← randomly pick a leaf, unary, or binary operation
6      create newNode
7      if nodeType = leaf then
8          newNode.value ← randomly pick a variable
9          return newNode
10     newNode.op ← randomly pick a unary or binary operation
11     newNode.left ← buildNode (level + 1)
12     if newNode.op is a binary operation then
13         newNode.right ← buildNode (level + 1)
14     return newNode
15 begin
16     create root node R
17     R.op ← randomly pick the equality operation, or one of the binary relations (if exists)
18     R.left ← BuildNode (1)
19     R.right ← BuildNode (1)
20     return R
```

then the invariant would read: The number of $y \in D$ such that $x = (x * y) + x'$, which is a valid invariant by the foregoing argument.

4.4.2 Quality Measure of Random Invariants

After a large set of invariants is generated, a small subset will be selected based on its ability to reduce the work of the next step in filtering out isomorphic models. For a block with m models, the worst-case scenario requires comparing every pair of models for isomorphism. There are $m(m-1)/2$ such comparisons in total. Based on this observation, we measure the quality of invariants by a score as follows: Suppose a set of invariants are applied to a set of models, resulting in n blocks of models in which models in different blocks have different multisets of invariant vectors. Let S_i, where $i \in [n]$, denote these n blocks of models. The score for this set of invariants is computed as

$$\sum_{j=1}^{n} |S_j|(|S_j| - 1) \tag{4.2}$$

The ultimate goal is to find the set of invariants having the minimum score over all possible combinations of randomly generated invariants in conjunction with the basic invariants.

4.4.3 Selecting Random Invariants

We are not aware of any tractable algorithm for finding the optimal subset from a large set of random invariants according to the quality measure stated above.[2] A feasible solution is to apply a greedy algorithm (see page 282 of [57] for the general discussions of using greedy algorithms) to a small sample of the models to find an approximate optimal subset. In practice, we observed that it is sufficient to use a sample size of 0.1–0.2%, or a thousand, whichever is larger, of the original set of models (see Section 9.4 for discussions). The algorithm is detailed in Algorithm 2. The idea of the algorithm is to start with the basic invariants, then add the random invariants and calculate the scores one-by-one, keeping the random invariant only if it gives a better score. Then repeat the process of adding random invariants, calculating the scores, and picking the best random invariant that minimizes the score until it cannot be further improved, or a preset maximum number of trials is reached. This subset of random invariants, which may or may not be truly optimal, will then be used together with the basic invariants for the next step.

Algorithm 2: Selecting Random Invariants with Greedy Algorithm

input : A set of random invariants R, a set of models M, and maximum trials T
output: A set $K \subseteq R$ of random invariants with $|K| < T$

```
1  K ← ∅
2  bestScore ← ∞
3  done ← false
4  while ¬done ∧ |K| < T do
5      a ← ∅
6      foreach r ∈ R ∖ K do
7          trialScore ← score of K ∪ {r} on M according to equation (4.2)
8          if trialScore < bestScore then
9              bestScore ← trialScore
10             a ← r
11     if a ≠ ∅ then
12         K ← K ∪ {a}
13     done ← (a = ∅)
14 return K
```

Note that the main purpose of adding randomly generated invariants is not to divide the models into more blocks in all cases, but to increase the robustness of the algorithm by the automatic discovery of important invariants in cases where the basic ones are insufficient.

4.5 The Invariant Algorithm

Recall from Section 4.2 that an invariant gives us an integer for each element in a structure. Further, multiple invariants are combined into a vector of integers for each element. We refer to this vector as

[2] We conjecture that the problem is NP-hard; it resembles K-means clustering, which is NP-hard [2].

invariant vector. From Corollary 4.2.1, if multisets of invariant vectors differ for two structures, these structures are *not* isomorphic.

In our algorithm, which is a multiset of invariant vectors, is represented as a lexicographically sorted vector of invariant vectors. This means that the original multisets are equal if and only if these sorted vectors are equal. This is briefly demonstrated by the following example.

Example 4.5.1. *Consider some structures C and D with domain elements $\{0, 1, 2\}$ and some invariants I_1, I_2. Suppose applying the invariants to C and D gives the multisets $V_C = [(0, 1), (3, 2), (0, 1)]$ and $V_D = [(3, 3), (0, 1), (0, 1)]$. Sorting them lexicographically gives two vectors of invariant vectors revealing that the multisets are not equal: $[(0, 1), (0, 1), (3, 2)]$ and $[(0, 1), (0, 1), (3, 3)]$, i.e., C and D are not isomorphic.*

The goal is not only to compare two models for isomorphism but to extract all non-isomorphic models from a list of models. In this case, a hash map is set up to store blocks of the models in which models from different blocks are guaranteed not to be isomorphic. We use the sorted invariant vectors for each model to send the model efficiently to the block (in the hash map) to which it belongs as summarized in Algorithm 3. That is, the keys in this hash map are the invariant vectors, and the values are the blocks of the models. After all models are hashed into the hash map, the blocks stored in the hash map can be processed separately, and possibly in parallel, to extract one representative model from each isomorphism class.

Algorithm 3: Hashing Models

input : A list of models M
output: A Hash-map of blocks of models having the same sets of invariant vectors

```
1 begin
2     H ← empty hash-map
3     foreach m ∈ M do
4         v ← sorted invariant vectors of m
5         if v is not already a key in H then
6             H[v] ← {m}
7         else
8             H[v] ← H[v] ∪ {m}
9     return H
```

As a simplified example to show how invariants are used, consider the inverse semigroups of order 2. It is defined by an unary function "/" (the inverse function) and a binary function "*" (the semigroup function). Mace4 generates 4 models from the FOL definition of inverse semigroups (see Table 4.1).

To make the example easy to follow, we use only 4 invariants: **U1**, **B7**, **B8**, and **B9**. These 4 invariants are calculated for each domain element in each model and the results are put into a vector of length 4. So, there is a set of two invariant vectors for each model. Isomorphic models will have identical sets of invariant vectors. While not absolutely necessary, we sort each set of the invariant vectors lexicographically for ease of comparisons, as shown in Figure 4.2. Note that the leftmost column in each figure holds the domain elements, and is not part of the invariant vector.

Table 4.1: Operation tables of Inverse Semigroups A, B, C, and D

Model A

$*_A$	0	1
0	0	0
1	0	1

$'_A$	0	1
	0	1

Model B

$*_B$	0	1
0	0	1
1	1	0

$'_B$	0	1
	0	1

Model C

$*_C$	0	1
0	0	1
1	1	1

$'_C$	0	1
	0	1

Model D

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

$'_D$	0	1
	0	1

Model A

1: 1 1 2 1
0: 1 1 2 3

Model B

1: 1 0 2 2
0: 1 2 2 2

Model C

0: 1 1 2 1
1: 1 1 2 3

Model D

0: 1 0 2 2
1: 1 2 2 2

Figure 4.2: Lexicographically sorted invariant vectors with discerning properties highlighted.

Next, the sorted invariant vectors for each model are concatenated into a single combo vector. For example, the combo invariant vector for model A is "$1, 1, 2, 1, 1, 1, 2, 3$", and that of B is "$1, 0, 2, 2, 1, 2, 2, 2$". By comparing the final combo vectors, we see that the models A and C have the same set of invariant vectors and hence could be isomorphic, but we cannot tell that from the invariant vectors alone. Same for the pair of models B and D. Next, we put these models into a hash-map using their sorted invariant vectors (or the combo vectors) as keys (see Algorithm 3). Models A and C will therefore go to one block in the hash-map, and models B and D will go to another. Now, it is not necessary to compare models across the blocks for isomorphism because models from different blocks have different sets of invariant vectors. Finally, models within each block are compared for isomorphism separately from other blocks, possibly in parallel. The resulting non-isomorphic models are simply put into the same set as no processing is needed to combine them. It turns out that in this case, the models in each of the two blocks are isomorphic. So there are only two non-isomorphic inverse semigroups of order 2.

To add random invariants to the algorithm, a preprocessing step, which aim at selecting an optimal subset of random invariants, is performed before the normal process. As described in Section 4.4.3, we construct a list of random invariants, calculate basic invariants and the random invariants on a small sample of the input models. Then the greedy algorithm is applied to find an optimal subset of random invariants for further processing (see Section 4.4.3), although optimality is not guaranteed. Finally, proceed to normal processing with the basic invariants and the optimal random invariants together.

Note that the invariant-based algorithm does not compare models for isomorphism. It only cuts down the size of the problem to improve the performance of existing isomorphism filters such as Mace4's *isofilter*.

Invariants have the potential to cope with the increasing size of the order of the algebra very well as illustrated in the following example. Suppose an invariant with extremely low discriminating power gives only 2 values, 0 and 1. However, when applied to the models of order 2, it could actually give 4 possible invariant vectors: [0,0], [0,1], [1,0], [1,1]. It is easy to generalize this observation: applying an invariant that gives at most m values to the models of order n could result in a maximum of m^n distinct invariant vectors. Furthermore, if k invariants give $\{m_1, m_2, \ldots, m_k\}$ values, then the maximum number

of invariant vectors would be

$$\prod_{i=1}^{k} {m_i}^n \tag{4.3}$$

There will be fewer distinct sorted invariant vectors than given in the expression 4.3, but the number still goes up with the number of domain elements.

From the above analysis on the intricate interactions between invariants, we can make two more important observations:

1. Combining invariants of low discriminating powers could give an invariant vector of surprisingly high discriminating power if they are targeting different areas of the algebraic structures.

2. In general, the number of non-isomorphic models increases rapidly as the order of the algebra increases, but so does the maximum number of possible invariant vectors. This helps invariants to retain their discriminating powers to some extent as the order of the algebra increases. This explains why the invariant-based algorithm is very scalable.

4.6 Related Work

Classes of algebras can often be defined in first-order formulas. For these algebras, a traditional finite model finder, such as Mace4 [48], SEM [72], and FALCON [71], FMSET [14], etc., can work on finding all their models. A well-known issue with this approach is that first-order formulas introduce symmetries into the problem, which leads to the generation of a huge number of isomorphic models in the outputs [59]. These isomorphic models can either be suppressed in the search phase or be removed in a postprocessing step after the models are generated.

Finite model enumeration can be posed as a constraint programming (CP) task [40]. Some CP solvers, such as Minion [32] and Gecode [53], support parallelization [46]. In CP, the search space can be divided into partitions by adding constraints to rule in and/or out partitions. Each partition can be processed by a separate worker thread/process. Minion further implements a work stealing search scheme that also partitions the search space dynamically by splitting the existing constraint model after the search has started [41]. It has been used to do parallel searches that take over 130 CPU years to complete [26]. However, to effectively add symmetry-breaking constraints such as lex-leaders to a CP solver often requires deep knowledge of the problem at hand (e.g., the semigroups in [26]) which may not be available when mathematicians first define and study a new algebraic structure. Moreover, isomorph-freeness is not guaranteed in CP solvers.

Past work in enumerating non-isomorphic finite models has focused on not generating isomorphic models in the search phase. Symmetry breaking is thus a central focus of their research [22, 23, 59]. An excellent example of a simple, powerful, and general algorithm to break symmetry *dynamically* is the least number heuristic (LNH) [11, 71, 72], which picks the smallest one not used so far when a new domain element is to be selected during the search. This is implemented in many finite model finders such as Mace4, SEM, FMSET, and FALCON. Another symmetry breaker, the eXtended least number

heuristic (XLNH) [11, 12], is based on similar ideas as the LNH, but could give better performance if there is at least one unary operation in the FOL clauses that define the model. It is also implemented in many finite model enumerators such as SEM.

The underlying idea of LNH can also be applied to break symmetries *statically*. This is necessary for approaches where we do not wish to modify the underlying solver. This is the case for finite model finders based on SAT solvers [22, 38, 59]. The issue is the overhead of encoding the LNH in conjunctive normal form (CNF) as well as its complex interaction with the SAT solver. Originally, the LNH was only encoded for constants [22]. Later, with additional effort, it was shown that other terms can also be considered [59].

The addition of symmetry-breaking input clauses could be useful in steering the searcher away from the needless exploration of sub-search space [23]. For example, it is well-known that a finite semigroup has at least one idempotent element, so we may add the clause 0 * 0 = 0 to the list of input clauses to cut off the search of the branch 0 * 0 = 1, etc. However, this kind of symmetry breaking often requires deep knowledge of the algebra in question, which may not be available when the algebra is first studied.

Most importantly, these symmetry-breaking techniques do not guarantee isomorph-freeness. While not generating isomorphic models would be ideal, but to guarantee that isomorphic models are not produced in the search phase is a hard problem that few modern-day solvers attempt to do. Systems that do try to do so, such as SEMK [17, 50] and SEMD [39], are either yet-to-be-completed or are better off allowing some isomorphic models in the outputs for some cases for better efficiency.

When isomorphic models are not totally suppressed in the search phase, they need to be removed in the post-processing steps. Many of them use a limited number of invariants to help speed up the process. Mace4, for example, has a program, *isofilter*, to filter out isomorphic models that it generates in the search phase. It calculates one invariant, frequency of occurrence of domain element, that is, the number of times a domain element appears in the operation tables. It uses this invariant to help separate non-isomorphic models and to help guide the construction of isomorphic functions between potentially isomorphic models. Needless to say, the discriminating power of one single invariant is limited. Indeed, it fails miserably when the operation table is a Latin square, which is the case for quasigroups. To alleviate this issue, Mace4 has another program, *isofilter2*, which does not try to construct isomorphic functions between models, but to convert models to their canonical forms based on the same algorithm [50] used by SEMK. *Isofilter2* works very well with quasigroups, but the overhead in computing the canonical forms of the models is so high that it becomes slower than *isofilter* for many algebraic structures such as semigroups.

The loops package [52] in GAP [31] is not a finite model enumerator but provides a stand-alone function to extract non-isomorphic models from a list of quasigroups. It uses 9 invariants, some of which are expensive to calculate, to help separate non-isomorphic models, and to guide the construction of isomorphic functions between models having the same invariant vectors. These 9 invariants exploit the specific properties of the quasigroup, and may not be effective for other algebraic structures. On the other hand, our invariants target many different areas in the common algebraic structures. Furthermore, their invariant vectors are limited to one binary operation table. Our invariant vectors can be constructed

from multiple unary, binary, and ternary operation tables.

Invariants are sometimes incorporated into the finite-model enumerating algorithm for specific algebras. These invariants are sometimes very simple and easy to compute, such as those in the algorithm for enumerating quandles [29]. But others may be very complicated, not easy to implement, and not cheap to compute as in the case of the algorithm for enumerating inverse semigroups [45] using the constraint solver, Minion [32]. Furthermore, the number of invariants used in these cases is usually very small, often two or fewer. Recall expression 4.3 on page 28 which shows that the number of invariants could increase the discriminating power drastically. Our invariants are cheap to calculate, are applicable to more algebraic structures, and are high in number to provide more opportunities for separating non-isomorphic models. Moreover, they can easily be incorporated into any finite model enumerator.

Neither Mace4's nor loops' isomorphic model filters make use of the hash table to store the models so that non-isomorphic models will never be compared once they are separated by their invariant vectors. This introduces some inefficiencies in their algorithms. Thus, both could benefit immensely from the reduced number of models in the blocks created by our invariant-based algorithm as a preprocessing step.

Another important feature in the invariant-based algorithm is randomization. Using randomization in the search phase in Boolean Satisfiability (SAT) and Constraint Satisfiability Problem (CSP) algorithms is a tried and tested technique [13, 35, 44]. This strategy is built into many SAT solvers such as Chaff [51]. However, using randomly generated invariants to help separate non-isomorphic models in the finite model enumeration is a novel idea. It helps solve the hardest problems in filtering isomorphic models as shown in our experiments, and consequently, increases the robustness of the invariant-based algorithm.

Chapter 5

Symmetries for cube-and-conquer in finite model finding[1]

5.1 Overview

One natural way to parallelize the search of a tree structure is to search in parallel the cubes that span the entire search tree. Recall that the search tree represents all possible assignments of values, in some sequential order, in the multiplication tables of functions in a first-order formula.

Cubes not only provide a natural way for parallelizing the model search process, but also a way for suppressing the generation of a portion of the isomorphic models (see Section 5.2). Suppressing the search of isomorphic models not only speeds up the search process and reduces the need for computing resources such as disk space, but also speeds up the post-processing isomorphic model removal process because fewer isomorphic models need to be processed.

We find inspiration in the well-established cube-and-conquer approach introduced for SAT [36]. In SAT this means splitting the search space by mutually exclusive conjunctions of propositional literals (cubes). In the context of finite model finding, the structure is richer—a decision of the solver corresponds to inserting a point into the graph of one of the considered functions, e.g., $f(0,1) := 3$.

We show that a cube can be excluded from the search if it is isomorphic to an existing one. Effectively, this is breaking symmetries in the search space. However, the task is nontrivial because finite model finders already contain a technique, called the least number heuristic (LNH), to exclude some isomorphic models (see Section 2.3.1 for more details about the LNH). The LNH enables the solver to consider only certain values from the co-domain for a given decision point. Therefore, we identify side conditions that enable us to prune isomorphic cubes in the presence of the LNH. Like so, we can take advantage of the two powerful and complementary techniques and ultimately suppress the generation of a large number of isomorphic models.

[1] This chapter is adapted from the article [6] to appear in the 29th International Conference on Principles and Practice of Constraint Programming (CP 2023)

5.2 Isomorphic Cubes Redundancy

The main objective of this section is to show that one of any pair of isomorphic cubes can be removed from the search. More formally, if cubes B_0 and B_1 are isomorphic, then it is sufficient to explore assignments extending B_0 and ignore all assignments extending B_1. We need to prove that any model of the formula in question that will be lost by discarding B_1 must necessarily be isomorphic to some model obtained from extending B_0 under the LNH.

As a motivational example, consider the cube $\langle f(0,0) = 0\rangle$, which states that f is idempotent in 0. But because 0 does not appear in the original FOL formula, intuitively, the constant 0 cannot play a special role in the formula. Consequently, this cube will search *all* interpretations of f that have at least one idempotent. For instance, the cube $\langle f(1,1) = 1\rangle$ will search the same interpretations, up to isomorphism. Now, we need to show this property formally and also show that it holds when the solver searches with the LNH restriction.

The key idea of the proof is that given a model B_1 with VA clauses A, any cube that is isomorphic to a subset of A can be gradually extended to be a model isomorphic to B_1 . Each extension step of the cube must uphold the following properties: (1) The cube is isomorphic to some subset of A. (2) The cube is LNH-compliant. The extension step is illustrated in Figure 5.1. We are given a cube B_0 that is isomorphic to an $A_0 \subseteq A$. When the finite model finder decides on some empty cell t, we need to show that it is possible to find a value according to the LNH such that the extended cube is isomorphic to some subset of A containing A_0.

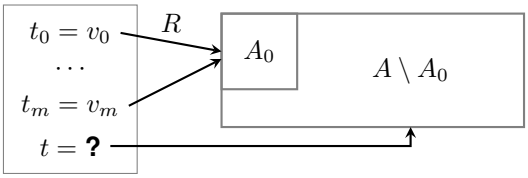


Figure 5.1: Extension of a cube according to the VA clauses A

Notation 1. *For a mapping R on D and a value $d \in D$ we write $\mathcal{E}_R^d$ for a mapping that maps d to $R(d)$ if $d \in$ dom(R) and otherwise maps d to $\min(D \setminus$ rng$(R))$. Note that the domain of $\mathcal{E}_R^d$ is dom$(R) \cup \{d\}$. The co-domain of $\mathcal{E}_R^d$ is $rng(R)$ if $d \in$ dom(R), otherwise it is $rng(R) \cup \min(D \setminus$ rng$(R))$.*

We further write $\mathcal{E}_R^{d_1,\ldots,d_k}$ for successive extensions by $d_1, \ldots, d_k$, i.e. $\mathcal{E}_R^{d_1,d_2} = \mathcal{E}^{d_2}_{\mathcal{E}_R^{d_1}}$ etc. □

Lemma 5.2.1. *If R is a bijection between some $D_0, D_1 \subseteq D$ and $d \in D$ then $\mathcal{E}_R^d$ is well-defined and also a bijection.*

Proof. If $d \in D_0$, then $\mathcal{E}_R^d = R$ and there is nothing to proof. If $d \in D \setminus D_0$, then by definition, $\mathcal{E}_R^d = R \cup \{(d,p)\}$ for some $p \in D \setminus D_1$. Since R is a bijection from D_0 to D_1, $d \notin$ dom(R), and $p \notin$ rng(R), so $\mathcal{E}_R^d$ is well-defined, one-one, and onto. That is, it is a bijection. □

Notation 2. *$B \oplus \langle t = u\rangle$ is the new cube formed by extending the cube B with the VA clause $t = u$.* □

Lemma 5.2.2. *Let B be an LNH-compliant cube and A a model s.t. B is isomorphic to some $A_0 \subseteq A$. Then for any cell term t not appearing in B, there exists a value u and a VA clause $t' = u' \in A \setminus A_0$, s.t. $B \oplus \langle t = u \rangle$ is LNH-compliant and isomorphic to $A_0 \cup \{t' = u'\}$.*

Proof. Let B be isomorphic to A_0 via some isomorphism R and let t denote the cell term $f(a_1, \ldots, a_k)$. Define R_1 as $\mathcal{E}_R^{a_1,\ldots,a_k}$, and let t' denote the cell term $f(R_1(a_1), \ldots, R_1(a_k))$, i.e., map the cell that the solver searches on into a cell in the prescribed model A.

Since A is complete, there must exist a value $u' \in D$ with $t' = u' \in A$, i.e. u' can be found by performing a lookup of t' in A. Since t is not a cell term in B, and R_1 is a bijection, so t' is not a cell term in A_0 and must therefore be in $A \setminus A_0$. Thus, $t' = u'$ is a VA clause in $A \setminus A_0$.

Next, define R_2 as $\mathcal{E}_{R_1^{-1}}^{u'}$, i.e. map u' back into the search by extending the inverse. Recall that the inverse of a bijection is a bijection, so R_1^{-1} is a bijection. It then follows from Lemma 5.2.1 that R_2 is a well defined bijection. Let $u = R_2(u')$. Note that by definition of t', $R_2(t') = R_1^{-1}(t') = t$. Therefore $t' = R_2^{-1}(t)$.

Finally, revert the mapping back by setting $R_3 = R_2^{-1}$. R_3 is a bijection because R_2 is. Furthermore, $R_3(t) = R_2^{-1}(t) = t'$, and $R_3(u) = R_2^{-1}(u) = R_2^{-1}(R_2(u')) = u'$. So, $R_3(\langle t = u \rangle) = \langle t' = u' \rangle$. Thus, u' is a homomorphic image of u under R_3. Therefore, R_3 is the isomorphism, and that $R_3(B \oplus \langle t = u \rangle) = A_0 \cup \{t' = u'\}$. In other words, $B \oplus \langle t = u \rangle$ is isomorphic to $A_0 \cup \{t' = u'\}$.

Finally, by definition of R_3, u either already appears in B or otherwise is the smallest domain element not in B. We therefore conclude that the extension of the cube B with the VA clause $t = u$ is LNH-compliant. □

Theorem 5.2.1. *Suppose we are searching under the LNH with any cell selection strategy on a signature Σ and a FOL formula, $\mathcal{F}$, on Σ. If B_0 and B_1, of length $l \geq 0$, are isomorphic cubes, and if M_1 is a model obtained by completing (not necessarily under the LNH) the search path in B_1, then B_0 can be extended by a search path S under the said LNH and cell selection strategy to a model M_0 which is isomorphic to M_1.*

Proof. We will use mathematical induction on the length of S to prove the theorem.

Let A denote the VA clauses of M_1, and A_0 denote the VA clauses of B_1.

Base case. Suppose the search path of B_0 is S_0 (the initial cube of S) and suppose further that the cell selection strategy selects a term t_0 to extend S_0 in the search. Since B_0 and B_1 are isomorphic, so B_0 is isomorphic to A_0, a subset of A. Therefore, by Lemma 5.2.2, S_0 can be extended by one VA clause with the cell term t_0 to a longer search path S_1 which is LNH-compliant and isomorphic to $A_1 \subseteq A$.

Induction step: Suppose the search path S is extended m times, where $m > 1$, so that S_m is LNH-compliant and isomorphic to a subset $A_m \subseteq A$. Then by Lemma 5.2.2, S_m can be extended by one VA clause with the cell term t_{m+1}, chosen by the said cell selection strategy, to S_{m+1} which is LNH-compliant and isomorphic to $A_{m+1} \subseteq A$.

Note that a model finder may do propagations after a cell value assignment. That is, some cell terms can be assigned values inferred from existing VA clauses. Propagations can be viewed as part of the cell selection strategy and be handled the same way as regular cell value assignments.

We can therefore conclude by mathematical induction that S can be extended to a complete search path when all cell terms in $\mathcal{F}$ are filled with values such that S represents the model M_0, is LNH-compliant, and is isomorphic to $A_s \subseteq A$. Since M_0 and M_1 are of the same size, so A_s and A must necessarily be of the same size and hence must be equal. Therefore, M_0 is isomorphic to M_1. □

Theorem 5.2.1 shows that isomorphic cubes will always extend to isomorphic models. So, one of the isomorphic cubes may be discarded without losing any non-isomorphic model. The idea of the theorem is not entirely new (see for example [11]), but we have given a new direct, simple and intuitive proof.

Corollary 5.2.1. *On searching under the LNH with any cell selection strategy on a signature Σ and a FOL formula $\mathcal{F}$ on Σ, if M_1 is a model in $\mathcal{F}$, then there is a complete search path S under the said LNH that results in a model M_0 which is isomorphic to M_1.*

Proof. The corollary follows directly from Theorem 5.2.1 by setting the length l in Theorem 5.2.1 to 0. □

Corollary 5.2.1 proves the completeness of the LNH in that every model in any search is isomorphic to some model found by searching under the LNH with any (fixed) cell selection strategy. An alternative proof of the same is given in [71], but the proof given here is more intuitive and direct.

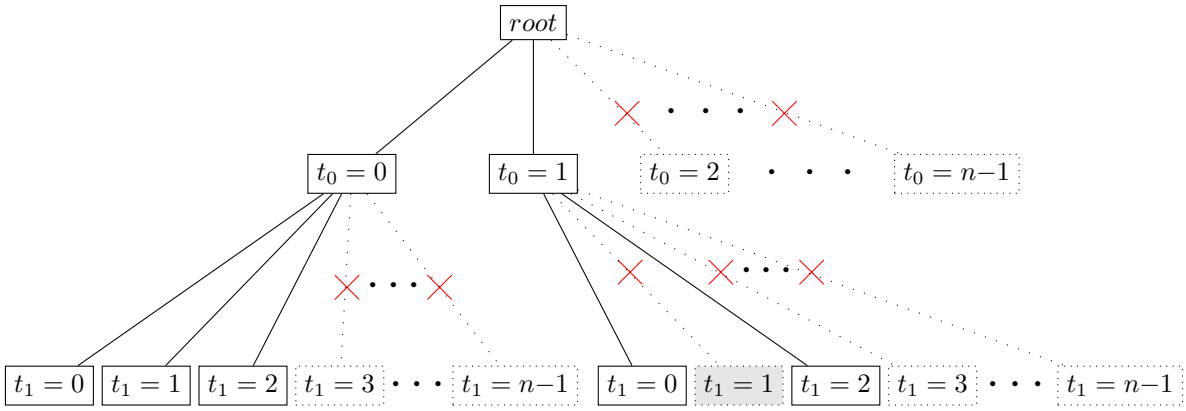
5.3 Searching with Cubes

In this section, algorithms are presented specifically for enumerating all models for a FOL, although they apply equally well to finding just one model.

Cubes can be constructed to partition the search space into non-overlapping subtrees that can be processed in parallel. It is not necessary to search all the subtrees that originate from the collection of cubes that span the entire search space because isomorphic cubes in the same collection can be eliminated without losing non-isomorphic models. For example, suppose we want to search for models of order 3 or more on a function $f : D^2 \to D$ under the LNH with a cell selection strategy that selects $f(0,0)$ then $f(1,1)$ as the first 2 cell terms in the search process. There are at most 6 cubes of length 2 (listed below) under the said LNH and cell selection strategy, so together they must span the whole search space in the sense that every search path that starts with the cell terms $f(0,0)$ then $f(1,1)$ in the search tree must include one of the 6 cubes in it.

1. $\langle f(0,0) = 0; f(1,1) = 0 \rangle$.
2. $\langle f(0,0) = 0; f(1,1) = 1 \rangle$.
3. $\langle f(0,0) = 0; f(1,1) = 2 \rangle$.
4. $\langle f(0,0) = 1; f(1,1) = 0 \rangle$.
5. $\langle f(0,0) = 1; f(1,1) = 1 \rangle$.
6. $\langle f(0,0) = 1; f(1,1) = 2 \rangle$.

Since $\pi_{(0,1)}(Cube\ 1) = \{f(1,1) = 1, f(0,0) = 1\} = \pi_{id}(Cube\ 5)$, so Cubes 1 and 5 are isomorphic and one of them can thus be discarded without losing non-isomorphic models per Theorem 5.2.1. This



Note: t_0 denotes $f(0,0)$ and t_1 denotes $f(1,1)$. A dotted line with a cross is a branch pruned by the LNH, except for the branch ending on the VA clause $t_1 = 1$ (the shaded node), which is pruned by the isomorphic cube removal algorithm.

Figure 5.2: Partial Search Tree Showing Cubes of Length 2

example demonstrates the importance of keeping the LNH in the search - it cuts the search space from *potentially* n^2 cubes down to 6 (some cubes may be cut early because they violate the axioms laid down by the underlying FOL formula, so, there may be fewer than n^2 cubes). Theorem 5.2.1 allows us to further cut the number of cubes down to 5 (see Figure 5.2 for illustration). More isomorphic cubes can be removed with longer cubes (see Table 9.14).

The procedure of removing isomorphic cubes starts with generating a set of short cubes (typically of length 2 for a binary operation) that spans the entire search space. The model finder takes short cubes as inputs and runs with them as if they are generated by itself to generate longer cubes of predefined length l. Specifically, the model finder runs as usual, except that it emits the cubes of length l when the depth of the search tree reaches l. After outputting the cube, the model finder backtracks as if it has reached the bottom of the search tree, and runs on a new branch as usual until all cubes of length l are generated. Some models may be generated in this process due to propagation, and they will be kept as part of the final outputs. Next, the cubes are compared for isomorphism and only one of any pair of isomorphic cubes will be kept. This new set of non-isomorphic cubes of length l will be used as inputs to the model finder in the next round of generation of longer cubes. The process is repeated until the desired length of cubes is reached.

For searching models defined by one operation of arity k, we use the sequence of lengths l: $k, 2^k, 3^k, 4^k, \ldots$. This is to match the concentric cell selection strategy (see Example 2.3.5 for its definition) of the finite model finder such as Mace4.

Example 5.3.1. *In searching for models defined by two binary operations and one unary operation with no constants such as the Involutive Lattices (see Table 9.21 of Chapter 9), the search sequence under the LNH with the concentric cell selection strategy will be* $u(0)$, $b_1(0,0)$, $b_2(0,0)$, $u(1)$, $b_1(1,1)$, $b_2(1,1)$, $b_1(0,1)$, $b_1(1,0)$, $b_2(0,1)$, $b_2(1,0)$, $u(2), \cdots$. *To match this cell selection strategy, the sequence of cube lengths is* $6, 10, 21, 36, 55, \cdots$. *Note that, for example, the cubes with length 10 are closed under permutations on* $\{0,1\}$ *in the sense that applying the cycle* $\pi_{(0,1)}(0,1)$ *on a cell in any cube with length*

10 will not result in any cell not appearing in a cube of length 10. Similarly, the cubes with length 21 are closed under all permutations on $\{0, 1, 2\}$. This property is important in finding isomorphic cubes which have the same cell terms after applying permutations on them.

5.3.1 Invariants

To speed up the isomorphic cubes removal process, the same invariant-based algorithm described in Chapter 4 used to remove isomorphic models can be applied to cubes. First invariant vectors for cubes are calculated and used as hash keys to group cubes having the same invariant vectors into hash buckets (see Algorithm 4). Cubes within the same bucket are tested for isomorphisms (see Algorithm 5). There is no need to test for isomorphisms across buckets because cubes in different buckets have different invariant vectors. This saves tremendous amount of testing time as isomorphic cubes are relatively few. Furthermore, buckets can be processed independently and in parallel, making best use of the available computing resources.

Algorithm 4: Generate Non-Isomorphic Cubes of a Fixed Length

input : List of cubes C of length l_0 and target length of cubes l_1

output: A set of non-isomorphic cubes of length L and a set of models

```
1  K ← generate all cubes under LNH of length l_1 from C of length l_0
2  models ← models generated by propagation when cubes are generated
3  H ← an empty hash-map
4  foreach c ∈ K do
5      v ← sorted invariant vector of c
6      if v is not already a key in H then
7          H[v] ← {c}
8      else
9          H[v] ← H[v] ∪ {c}
10 P ← all permutations on all domain elements appearing in any cell term in K
11 newCubes ← ∅
12 foreach b ∈ H do
13     newCubes ← newCubes ∪ remove_isomorphic_cubes(b, P) /* See Algorithm 5 */
14 return newCubes, models
```

Finally, non-isomorphic cubes of the target length can then be processed independently in parallel and their output models collected separately.

5.3.2 Work Stealing

In the basic form of this cube-based parallel algorithm, cubes are statically generated before the model enumeration process begins. It has the advantage of low runtime overheads as no synchronization

Algorithm 5: Removing Isomorphic Cubes

input : List of cubes C, list of permutations P

output: List of non-isomorphic cubes

```
1  K ← ∅
2  foreach c ∈ C do
3      found ← false
4      foreach p ∈ P do
5          if p(c) ∈ K then
6              found ← true
7              break
8      if ¬found then
9          K ← K ∪ {c}
10 return K
```

among running finite model finders is needed. The preprocessing time for generating the cubes is also small for short to medium-length cubes. The disadvantage is that the workload may be uneven among the parallel processes. Some jobs may take a long time to finish when free workers sitting idle after finishing the rest of the jobs.

This problem can be alleviated with work stealing algorithms. There is a natural and low-cost way to divide the jobs in cube-based finite model searching. A running job can cut out some cubes that are not being worked on to the free workers. For example, suppose a running job is working on a cube $B_0 = \langle f(0,0) = 0 \rangle$, and its cell selection strategy picks the cell $f(1,1)$ to assign value next. Under the LNH, $f(1,1)$ may be assigned a value from $\{0,1,2\}$. If the job is requested to spin out some work for other free workers, then it can generate 3 cubes, $B_0 = \langle f(0,0) = 1; f(1,1) = 0 \rangle$, $B_1 = \langle f(0,0) = 1; f(1,1) = 1 \rangle$, and $B_2 = \langle f(0,0) = 1; f(1,1) = 2 \rangle$. It continues to work on the cube B_0 and releases B_1 and B_2 to the free workers.

To reduce the total overheads of cutting cubes from running workers and initializing free workers with new cubes, we employ the heuristic that shorter cubes represent taller and bigger search trees and hence should be given more processor time. So, the *shortest* cubes that are not being worked on in the running workers are cut out to feed the free workers. This minimizes the total number of work requests. Continuing with the previous example, suppose now the running job has progressed, along B_0, to the cube $\langle f(0,0) = 1; f(1,1) = 0; f(0,1) = 0; f(1,0) = 0 \rangle$ before receiving the first request for spinning out work. It will still cut out the same two cubes, B_1, and B_2, to the free workers because they are the shortest cubes that are not being worked on. The running job remembers that the cubes B_1, and B_2 have been cut out to other workers, and will not work on them when the search backtracks to the cell $f(1,1)$.

5.3.3 Optimal Cube Length

When the cubes are extended longer and longer, more and more isomorphic cubes can be removed, which tends to reduce the run time of the search. However, it also takes more time to find the isomorphism between longer cubes. Moreover, more isomorphic models are generated by propagation during the process of extending the length of the cubes. The optimal cube length to use is system and FOL dependent and can only found empirically. This will be discussed further in Section 9.5.1.

5.4 Related Work

Parallel algorithms can be characterized by how the search is done. There are two main search methods: the embarrassingly parallel search method (EPS) and the work stealing search method [16, 21, 60, 61]. In the former method, the task is decomposed into many sub-tasks that are queued up to be processed by free worker threads/processes. In the latter method, when a worker completes its task, it asks other workers for more work. The busy workers may split their tasks into smaller sub-tasks and pass some of them to the free workers. The main focus of this method is to keep all the CPUs running until all jobs are done, although in some cases, the work stealing scheme can affect efficiency [21]. The EPS method is a natural choice for the cube-based parallelization scheme because preprocessing is done to generate large number of non-isomorphic cubes. However, the work stealing search method can also be used to supplement the search when all cubes are being worked on and there are idle workers (see Section 5.3.2).

Parallel algorithms can help select the best strategy in solving problems with the EPS method [56]. After a problem is decomposed into a large number of sub-tasks, a small number (e.g., 1%) of these sub-tasks are run in parallel using different strategies of the same solver or different solvers. The strategy that gives the best performance on the subset of sub-tasks will be used to run all sub-tasks. The same idea is used in the invariant-based isomorphic models removal algorithm [5]: it randomly generates a large number of invariants, then applies them to a small percentage of models to pick the best performing random invariants to apply to the whole set of models. This idea can be applied to the finite model finders that support multiple cell selection strategies to pick the best function order and cell selection strategy for any specific problem.

Finite model enumeration can be posed as a constraint programming (CP) task [40]. Some CP solvers, such as Minion [32] and Gecode [53], support parallelization [46]. In CP, the search space can be divided into partitions by adding constraints to rule in and/or out partitions. Each partition can be processed by a separate worker thread/process. Minion further implements a work stealing search scheme that also partitions the search space dynamically by splitting the existing constraint model after the search has started [41]. It has been used to do parallel searches that take over 130 CPU years to complete [26]. However, to effectively add symmetry-breaking constraints such as lex-leaders to a CP solver often requires deep knowledge of the solver *and* the problem at hand (e.g., the semigroups in [26]) which may not be available when mathematicians first define and study a new algebraic structure.

Moreover, to use traditional CP solvers for finite model enumerations, mathematicians need to learn a new CP-specific language such as CHR [65] and Savile Row [54]. It is possible to use a translator to translate between languages, but that adds uncertainties to the fidelity and the optimality of the translated specifications. FOL remains one of the most popular languages among mathematicians due to its simple and intuitive syntax. Moreover, a popular automatic theorem prover, Prover9 [49], shares the same input language with Mace4. This adds more than just convenience to the process, as it also reduces the chances of discrepancies between Prover9 and Mace4 on the same problem.

A well-known issue with enumerating models defined with the FOL is the isomorphic models included in the outputs. This is an inherent symmetry property of the FOL [59]. A lot of research efforts are given to symmetry-breaking [11, 22, 23, 59, 68]. Although complete symmetry-breaking is known to be computationally challenging [23, 68], many useful algorithms, such as the LNH and the XLNH [11, 12], have emerged in partial symmetry-breaking. The XLNH is more restrictive as it only works on unary operations. The LNH is implemented in many systems such as Falcon [71], SEM [72], FMSET [14], and Mace4. The isomorphic cubes algorithm, which removes more cubes as the cube length grows, is designed to preserve and complement the effectiveness of the LNH.

Another important symmetry-breaking strategy is to steer the search engine away from the fruitless exploration of sub-search space by adding symmetry-breaking input clauses [23, 68]. The cube-based parallel and the isomorphic cubes removal algorithms are compatible with this kind of strategy, as with any other symmetry-breaking algorithm that works with the LNH.

Some finite model finders such as SEMK [17] and SEMD [39] try to completely suppress isomorphic models in the search process. However, these isomorph-free algorithms are not easy to parallelize as global information is generated and consumed in many steps, requiring high-cost synchronizations between cooperating workers, especially when the workers are run on different computers. The cube-based parallel algorithm, on the other hand, is an EPS method that requires no synchronizations between workers. The static removal of isomorphic cubes done in a preprocessing step is shown to be effective in suppressing isomorphic models even before the actual search begins. The augmented work stealing algorithm is not high in synchronization costs because it does not involve communications between running jobs. The remaining isomorphic models come out of the cube-based algorithms can be efficiently removed by the invariant-based isomorphic model filtering algorithm (as described in Chapter 9.4) as a postprocessing step.

A local symmetry-breaking algorithm, DSYM [11], exploits local symmetries by finding symmetries (synonym to isomorphisms in their terminology) under invariant partial interpretations (which are invariant cubes), and without parallelism. It also works with the LNH and XLNH. DSYM is a predictive algorithm that works at the *parent* level and predicts which of its immediate children will be isomorphic cubes. It can be seen as a special case of the isomorphic cube algorithm because it removes isomorphic cubes having the same immediate parents, while the isomorphic cube algorithm removes all isomorphic cubes, irrespective of their parents. Nevertheless, for the cases that DSYM covers, it does so right before the cubes are generated, while the isomorphic cube algorithm only detects the symmetries right after the cubes are generated. A disadvantage of DSYM is that it is not clear how it

can be effectively parallelized. Furthermore, DSYM only detects symmetries under the same subtree. The isomorphic cubes removal algorithm, on the other hand, detects both global and local symmetries the same way, and hence detects and removes more symmetries than DSYM. Moreover, DSYM uses only two invariants in the process of testing isomorphism between cubes, while in the isomorphic cubes removal process, many invariants that are proven successful in the invariant-based isomorphic model removal algorithm are used. Lastly, the algorithms in this chapter are described in a generic setting without the implementation artifacts such as mdn. The proof of Theorem 5.2.1 applies not only to the bespoke theorem, but also to the correctness of the LNH (see Corollary 5.2.1). Nevertheless, DSYM can be applied to the cube generation process as well as the model generation process. It may speed up the removal of isomorphic cubes and the generation of models by generating fewer isomorphic cubes and fewer isomorphic models, although we have not run experiments to support these claim. Thus, the isomorphic cube removal algorithm is compatible with DSYM, as with many other symmetry breaking algorithm that works with the LNH.

Chapter 6

Magmaut - A GAP Package[1]

6.1 Overview

Invariants have wide applications in computational algebra. It is an indispensable component of the GAP package `Magmaut` (**Magm**a **Aut**omorphisms) (see Appendix C for its manual) [7], which is developed in this project. The package, which is available as `https://gitlab.com/ChoiwahChow/magmaut`, provides functions to generate automorphism group for a given algebra, and some functions to test for isomorphism and monomorphism between algebras of type $(2^m, 1^n)$ where $m > 0, n \geq 0$. The automorphism functions in this package is out of the scope of this project, so they will not be described here. We would only discuss in details the key functions related to isomorphism, such as `IsomorphismMagmas` and `IsomorphismAlgebras`. `monomorphismMagmas` and `MonomorphismAlgebras` are very similar to, and shares a lot of code with, the corresponding isomorphism functions, and hence will not be described in details here.

`Magmaut` handles algebras of type $(2^m, 1^n)$ represented by a list of $m > 0$ binary multiplication tables, followed by $n \geq 0$ unary multiplication tables. A binary multiplication table is represented by a 2-dimensional array, and a unary operation table is represented by a 1-dimensional array. For algebras, such as the semigroups, that are represented by a single binary operation, `Magmaut` also accepts a single binary multiplication table (a 2-dimensional array) as the representation of an algebra. However, to make use of the functionality built into the GAP platform, many functions in `Magmaut` require the input arguments be given as magmas. There is a GAP function, `MagmaByMultiplicationTable`, to convert a 2-dimensional multiplication table into a magma:

```
# After the SmallLoops package is loaded, AllSmallLoops(6) returns
# all loops of order 6, up to isomorphism, as a list of multiplication tables.
multi6 := AllSmallLoops(6);;

# loop6 is the list of all loops of order 6 as magmas
```

[1] Some algorithms, mainly the isomorphism and monomorphism algorithms, in the `Magmaut` package have been incorporated in the CREAM package [10].

```
loop6 := List(multi6, x->MagmaByMultiplicationTable(x));;
```

The advantage of using multiplication table as the basic representation of an operation in algebra is that it can be used uniformly across all types of algebraic structures.

One way to decide whether two finite algebras A and B are isomorphic or not is to examine all possible mappings from A to B. If one of the mappings between them is an isomorphism, that is, a bijective homomorphism, then A and B are isomorphic. Otherwise, they are not. This is indeed the strategy used in the Loops package [52] in GAP to check for isomorphism between two quasigroups.

It is possible, but not efficient, to generate all mappings between two finite algebras A and B of the same order to look for isomorphism. `Magmaut` incorporates some invariant algorithms to expedite the process of finding an isomorphism between a pair of algebras A and B:

1. Compare sets of the invariant vectors of A and B. If they are not the same, then they cannot be isomorphic.
2. Limit the search of isomorphisms to bijections, that is, different elements in one magma must only be mapped to different elements in the other magma.
3. Limit the search to mapping of elements with the same characteristics, or invariant vectors in this case.
4. Instead of trying all possible mappings of elements in A to B, try only all possible mappings of a generating set of A to B.

Indeed, invariants play a key role in this fast isomorphism checking algorithm.

GAP has its own programming language which is built on top of C. GAP's programming language has a lot of convenient built-in functions, but since it is a high-level language, it is not as efficient as C which is closer to the hardware. Furthermore, C is a compiled language which is in general faster than an interpretive language such as GAP. Therefore, some functions, such as the function that calculate invariants, in `Magmaut` are done in C to improve efficiency.

6.2 Isomorphism Algorithms

It is not necessary to try all possible mappings of domain elements from magma A to magma B to see if any of them can be an isomorphism. Since all domain elements of a magma can be expressed as a finite combination of elements in a generating set, we only need to check mappings of a generating set of A to B. To wit, let A, B be magmas, S be a generating set of A, and m be an element in $A \setminus S$ such that $m = s_1 s_2 \dots s_n$ where $s_1, s_2, \dots s_n \in S$, and $h : A \to B$ be a homomorphism, then by definition, $h(m) = h(s_1 s_2 \dots s_n) = h(s_1)h(s_2) \dots h(s_n)$. Thus, the mapping of elements outside S are completely fixed by the mappings of those in S. Not all generating sets are equal, however, some are more *efficient* than others in the sense that fewer mappings from them are possible. Invariants play an important role in finding efficient generating set of a magma. The following example illustrates how invariants are used in this case.

Suppose magmas A and B, of order n, have the same set of invariant vectors. Suppose A has a domain element a_1 that has a unique invariant vector different from those of others. Then B must have a domain element b_1 that has the same unique invariant vector. If there is an isomorphism $\phi : A \to B$, then $\phi(a_1) = b_1$. That is, it cuts down the number of possible values of $\phi(a_1)$ from n to just 1. Likewise, if two elements of A, $\{a_1, a_2\}$, have the same invariant vector, then there must be elements $\{b_1, b_2\}$ of B that have the same invariant vectors. In this case, $\phi(a_1) = b_1$ and $\phi(a_2) = b_2$, or, $\phi(a_1) = b_2$ and $\phi(a_2) = b_1$. The number of possible mappings in case of two elements having the same invariant vector is still quite small, although not as small as the case of domain elements having unique invariant vectors.

If we partition the domain elements of a magma by their invariant vectors, and sort the partitions (also called blocks), in the ascending order of their sizes, then in selecting elements to form a generating set, everything else being equal, elements in the smaller blocks are preferred. This is part of the heuristic used in the `Magmaut` function `MAGAS_EfficientGenerators` that finds an efficient generating set of a magma by a greedy algorithm. In this greedy algorithm (see Algorithm 6), at each step, we add to the generating set the element that increases the size of the sub-magma most. In case of a tie, the element from the smallest block is added to the generating set.

Algorithm 6: Finding an Efficient Generator for a Magma

input : Magma Q, List C (Partition of domain elements of Q in ascending order of block size)

output: A generating set of Q

```
1  G ← ∅                  /* generating set so far                                  */
2  S ← ∅                  /* sub-magma generated by G                               */
   /* domain elements in K are sorted by the block size they reside in             */
3  K ← Concatenation of all blocks of C into a single list
4  while S ≠ Q do
5      foreach k ∈ K do
6          H ← sub-magma generated by G ∪ {k}
7          if Size(H) > Size(bestS) then
8              bestS ← H
9              bestG ← k
10     G ← G ∪ {bestG }
11     S ← bestS
12     K ← K \ {bestG }
13 return G
```

Another important step in constructing an isomorphism from magma $(A, *_A)$ to $(B, *_B)$ is to *extend* a partially constructed isomorphism ϕ. Here is an example to clarify this extension step. Suppose by construction, dom$(\phi) = \{a_1, a_2\}$, $\phi(a_1) = b_1$ and $\phi(a_2) = b_2$. If $a_1 *_A a_2 \notin$ dom(ϕ), then we can make an extension to ϕ by the assignment $\phi(a_1 *_A a_2) = \phi(a_1) *_B \phi(a_2)$. Now dom$(\phi)$ is extended to $\{a_1, a_2, a_1 *_A a_2\}$. More elements such as $a_1 *_A a_1$ can be added to dom(ϕ) by extension. When

dom$(\phi) = A$, we have an isomorphism from A to B, that is, A and B are isomorphic. If in the extension process, some contradictions show up, then we know ϕ cannot be extended to an isomorphism. For example, if $a_1 *_A a_2 = a_1$ and $\phi(a_1 *_A a_2) = \phi(a_1) *_B \phi(a_2) \neq b_1$, then ϕ cannot be a homomorphism and hence cannot be extended to a complete isomorphism.

With the invariant vectors of magmas $(A, *_A)$ and $(B, *_B)$, and the efficient generating set, G, of A as generated by Algorithm 6, the `Magmaut` function `AllIsomorphismMagmasNC` can compute all isomorphisms from A to B by trial-and-error. A generating set is used to construct the domain of the isomorphism because it is always possible to extend it to A unless some contradictions occur in the process. The exhaustive trail-and-error process of extending a generating set to the domain of an isomorphism is summarized in Algorithm 7. It is a recursive algorithm extending one domain element at a time.

Algorithm 7: Extending Isomorphism between Magmas

input : Two Magmas A and B, the partial map so far f, a Generating Set G, Invariant Vectors (of B) P

output: A list of permutations representing the isomorphisms from A to B

```
1  Function Extend(A, B, f, G, P):
2      S ← ∅                 /* the set of all isomorphisms                    */
3      if G = ∅ then
           /* elements in generating set are all mapped                        */
4          Extend the mapping f
5          if there are no contractions then
6              return {f}
7          else
8              return ∅
9      x ← G[1]
10     G ← G \ {x}
11     K ← the set of all domain elements in B having the same invariant vector as x
12     foreach k ∈ K do
13         Q ← f.copy()
14         Q[x] ← k                  /* assign mapping from A to B             */
15         H ← Extend(A, B, Q, G, P)  /* recursively call with smaller G       */
16         S ← S ∪ H
17     return S
```

A slight variation of Algorithm 7 is used for another function `IsomorphismMagmas` which returns just one isomorphism between 2 magmas if they are isomorphic, `fail` otherwise. Algorithm 7 is modified to stop and return the first isomorphism it finds.

The `IsomorphismAlgebras` function in `Magmaut` is the main GAP function used in the GAP packages

of algebras of small orders (see Chapter 8). Internally, it makes use of the `AllIsomorphismMagmasNC` function that finds all the isomorphism between two magmas. The idea is that given two algebras A and B, we find all isomorphisms from the first binary operation of A (which is a magma by definition) to the first binary operation of B (also a magma by definition), then check one-by-one to see if any of these isomorphism is also an isomorphism for all pairs of operations from A to B. The process stops when one of them works for all pairs of operations, in which case, A and B are declared isomorphic. If none of the isomorphism works for all pairs of operations, then A and B are not isomorphic. The procedure is summarized in Algorithm 8.

Algorithm 8: Finding Isomorphism between Algebras of Type $(2^m, 1^n)$

input : Two algebras A and B as lists of multiplication tables

output: A permutation representing the isomorphism from A to B, or fail if A and B are not isomorphic

if A *and* B *are structurally different* **then**
    return fail
$invA \leftarrow$ invariant vector of A [1]
$invB \leftarrow$ invariant vector of B [1]
if $invA \neq invB$ **then**
    return fail
P $\leftarrow$ `AllIsomorphismsMagmas`(A$[1]$, B$[1]$)
K $\leftarrow$ `Length`(A)
$match \leftarrow$ true
foreach $k \in$ P **do**
    foreach $p \in [2..\text{K}]$ **do**
        if $p(\text{A}[k]) \neq \text{B}[k]$ **then**
            $match \leftarrow$ false
            break
    if $match$ **then**
        return p
return fail

6.3 Related Work

Isomorphism is ubiquitous in algebra and there are a plethora of GAP packages, such as the loops package and the semigroups package, that contain some functions to test for isomorphism between algebras.

One main difference between `Magmaut` and other pre-existing GAP packages is that `Magmaut` oper-

ates on algebras of type $(2^m, 1^n)$, and other packages (with the notable exception of the new `CREAM` package, which is available as `https://gitlab.com/rmbper/cream`) operate only on some specific algebras such as semigroups and quasigroups. So, Magmaut handles isomorphism and automorphism on more algebraic structures than, for example, the loops package and the semigroups package.

The `Magmaut` package borrows many algorithms from the `loops` package and extend them to algebras of type $(2^m, 1^n)$. Hence their performance on quasigroups and loops are similar. However, as we pointed out earlier, `Magmaut` handles more than just quasigroups and loops.

The semigroups package uses very specialized data structures to represent a semigroup and uses specialized algorithms for checking semigroups for isomorphisms, so it may perform better than `Magmaut` for some semigroups. But again, they only work for semigroups and nothing else.

There are some commonality between GAP packages `CREAM` and `Magmaut`. They both operate on algebras of type $(2^m, 1^n)$, and represent binary and unary operations by multiplication tables (arrays). They share some common algorithms and complement each other.

Chapter 7

Applications

7.1 Overview

The algorithms described in Chapters 4 and 5 can be combined to yield a powerful system to enumerate models from FOL. There are many simple but useful applications of the combined algorithm. We shall describe two such applications in this chapter.

7.2 Numbers of Common Algebras of Small Orders

The number of models is a very important property of an algebra. For example, if a mathematician devises a formula to calculate the number of models, he or she can quickly check the literature for the sequence of number of models for each order, i.e., if such a sequence exists, to check that the new formula contradicts some known facts. For example, Bhatti, Chaudhry & Zaidi [15] state that there are 70 proper BCI-algebra of order 5 through some analyses of its structure. Later, Nisar & Bhatti [55] corrected the number as 31. However, this number is still wrong. The correct number of proper BCI-algebra as calculated by Mace4 is actually 30. Two of the proper BCI-algebras reported on page 26 (Table 11 and 13) of their paper are in fact isomorphic. This kind of situation can be avoided if the sequence of numbers of the algebra in question are available.

To count the number of models of order n defined in the first-order formula is not trivial. We often need to use a finite model enumerator to generate the models of order n according to the rules laid down by the algebra. However, traditional finite model enumerators often generate a huge number of isomorphic models which need to be removed so that the number of non-isomorphic models can be counted. Comparing models for isomorphism is a time-consuming and resource-intensive task. For example, while there are only 1,627,672 semigroups of order 7, the finite model generator Mace4 generates over 1 billion models of order 7. This partly explains why so few sequences of model counts are available in the literature.

In this thesis, we report new integer sequences for the number of models of order n (for small integer values of n) for 100 algebras (see the article in Appendix B 5). These sequences have not been reported

to the *On-Line Encyclopedia of Integer Sequences* (OEIS) [64]. We also increased the length of the existing OEIS integer sequences A305858 and A178432 (see Table 9.23). This is accomplished through the use of the improved model enumeration algorithms described in this thesis and implemented in the finite model enumerator Mace4. We do not store the models generated in the counting process because the storage requirement is too high.

7.3 Inherently Non-finitely Based Monoids

Finding the inherently non-finitely based (INFB) monoids [62], up to isomorphism, is not a trivial problem even for small orders. With the help of GAP's smallsemi package, which has all the pre-computed semigroups up to order 8, one can find the INFB monoids up to order 8. The smallsemi package together with a finite model enumerator such as Mace4 can be used to extend the results to order 9, which was the best result known in the literature. The smallsemi package cannot help beyond that.

We start with monoids of order 6 satisfying the INFB conditions and use our new algorithms to find the two non-isomorphic models. Then proceed to order 7, get all monoids of order 7 satisfying the INFB conditions, use our new algorithm to filter out isomorphic models, and finally use the Magmaut or the CREAM package's monomorphism functions (see Chapter 6) to filter out all models containing the monoids of order 6. The process is repeated for order 8, but this time filter out all models containing the monoids of order 6 and order 7. The process for finding INFB monoids of higher orders are the same: for each order, filter out models containing models of INFB of lower orders.

We are able to extend the results to obtain all INFB monoids of order 11. The numbers of INFB monoids are shown in Table 9.24.

Chapter 8

GAP Package Generator

8.1 Overview

With the algorithms described in the previous chapters, we are ready to show how to fulfill one of the main objectives of this project, which is to develop a tool for the mathematicians to generate all models, up to isomorphism, of algebras (of small orders) of their interest. In this chapter, we describe a tool that takes a first-order formula to generate all models, up to isomorphism, up to a specified order, and put them into a GAP package that complies with the current GAP standards as given in Chapter 76 of the GAP Reference Manual [31]. In particular, the package is complete with documentation and testing code.

More specifically, the auto-generated GAP package of an algebra, A, of small orders contains:

1. A library of functions

   (a) A function to get the list of all models of the algebra of a specific order, if they exist in the package.

   (b) A function to test whether a given model represented by a multiplication table, or a list of multiplication tables in case of an algebra of type $(2^m, 1^n)$, is isomorphic to a model in A.

2. Documentation on

   (a) Definition of the algebra in FOL.

   (b) Usages of the two functions described above.

3. Auto-testing code to do basic tests on the code of package after it is loaded in GAP.

4. Administrative information such as the owner of the package and the location where the package resides.

8.2 The GAP Package Generation Procedure

The GAP package generator[1] for algebras of small orders consists of a collection of Python[2] scripts, templates of GAP code, and templates of documentation in the xml[3] format. In addition, it makes use of the GAP's support for package generation in generating test scripts and documentation.

The GAP package generation starts with generation of models of the algebra using Mace4. The algebra is defined by a first-order formula specified in an input file (see Section 8.3 for an example). Then, the non-isomorphic models are extracted from the Mace4 outputs. The final models are put into a data file which is compressed to save space and time for loading.

Next, a GAP library function `AllSmall`$\langle algebra \rangle$ to access the models are generated from templates using the name of the algebra in question. For example, if the name of the algebra is Loops, then the access function will be named `AllSmallLoops`. The function call `AllSmallLoops(`n`)` returns a list of all loops of order n stored in the package. Another GAP function `IsASmall`$\langle algebra \rangle$ is also automatically generated. This function is used to check if a model is isomorphic to any model in the package.

Before we show the final steps which are generation of documentation and test code, we show an example of using these functions after the `SmallQuandles` (for quandles) and `SmallLoops` (for loops) are loaded:

```
gap> p4 := AllSmallLoops(4);;        # list of all small loops of order 4
gap> Length(p4);
2
# The loops in AllSmallLoops are all non-isomorphic
gap> IsomorphismMagmas(MagmaByMultiplicationTable(p4[1]),
 MagmaByMultiplicationTable(p4[2]));
fail

# The following call succeeds with the identity permutation as the isomorphism
gap> IsomorphismMagmas(MagmaByMultiplicationTable(p4[1]),
     MagmaByMultiplicationTable(p4[1]));
()

# Illustrate how to check something is a small Loop
gap> IsASmallLoop(p4[1]);
true
gap> q4 := AllSmallQuandles(4);
gap> Length(q4);
7
```

[1]The source code for the GAP package generator is available as `https://github.com/ChoiwahChow/gap_package_generator`

[2]Python is a very popular mainstream programming language

[3]The E**X**tensible **M**arkup **L**anguage is a text format used to describe structured information such as data and documents.

```
# A quandle is not one of the small loops
gap> IsASmallLoop(q4[1]);
false
```

Next, test code is generated from a code template file, by replacing the place-holders with the new algebra name. GAP has built-in support for running test functions and comparing the test outcomes with the expected outcomes, and for flagging error when differences are detected. For example, a template test file contains

```
gap> Length($FunctionName($MinDomainSize));
${MinAlgebraSize}
```

During the code generation process, the place-holders *FunctionName*, *AllSmallLoops*, and *MinDomainSize* are replaced by actual values to become

```
gap> Length(AllSmallLoops(2));
1
```

When the test is run, GAP will make sure the outputs of the test `Length(AllSmallLoops(2));` is `1`. Otherwise, it will flag an error.

The final step is to generate the documentation from a list of document template files and code files. First, a list of template files are processed (by Python scripts) to have place-holders replaced by the actual values, such as the algebra name and the definition of the algebra in FOL, from the configuration file. Then, execute the `configure` script provided by GAP to finally generate all the supporting documentation files in `html` and `xml` formats for web display, and in the `pdf` format as a single document (see Appendix C for examples of the pdf documents).

8.2.1 Data Compression

GAP has built-in support for common data compression formats. When it reads in a file with suffix .gz in its name, it will automatically decompress the file. The compression ratio is generally over 90% for large data files, that is, the size of the .gz file is usually less than one-tenth of that of the original file (see Table 9.26). Compressed data also loads faster than uncompressed data because smaller amount of data are transferred from the hard disk to the memory.

8.3 Generating GAP Package

The required inputs to the GAP package generator include

1. A Mace4 input file containing the first-order formula.
2. A free text file containing the definition of the algebra to be included verbatim in the documentation of the package.

3. A configuration file setting the values of the parameters for the generator (see Appendix D for a sample configuration file).

Suppose we are to generate a GAP package of semigroups of orders 2 to 7. The Mace4 input file would simply be:

```
formulas(sos).
    x * (y * z) = (x * y) * z.    % associativity
end_of_list.
```

The text file containing the definition of semigroups could simply be a one-liner:

```
x * (y * z) = (x * y) * z.    % associativity
```

It may contain additional information if the user wants.

The generation of a GAP package for an algebra defined by a first-order formula is done by the Python script *gap_gen.py*, which takes a configuration file $\langle algebra \rangle$.ini as the only input. All configurable parameters used by the script can be specified in this configuration file. Continuing with the example of generating a GAP package for semigroups up to the order 7, the configurations will contain, among the configurable items, the following basic items:

```
[ROOT]
BaseName = Semigroups
SingularAlgebraName = Semigroup
AlgebraName = semigroups
AlgebraDisplayName = Semigroups
AlgebraDisplayNameLowerCase = semigroups
PackageName = SmallSemigroups
Version = 0.1.0
DateOfRelease = 08/20/2022
Status = dev
# Range of domain size, from MinDomainSize to MaxDomainSize, inclusive.
MinDomainSize = 2
MaxDomainSize = 7
[MACE]
# Use ";" to separate input files that form a partition of the search space.
InputFiles = semi.in
```

There are other advanced options in the configuration file for sophisticated users. For example, if some of the models are already generated by some other means (or are generated before in some failed attempts, such as in generation aborted due to running out of memory, etc.), they can be included in the generation process without the need for repetition:

```
# skip the following domain size for Mace4 and/or isomorphic model elimination,
# domain list is comma-separated
SkipMace4 = 7
Mace4Prebuilt = semigroups_7_0.out
# The Prebuilt files are in the [PrebuiltDir]
SkipNonIso = semigroups_7_0.out
NonIsoPrebuilt = semigroups_7_0_1.out.f
```

In the above example, the semigroups of order 7 will not be generated, but instead, taken from the file `semigroups_7_0_1.out.f`. This adds to the robustness and scalability of the generation process.

8.4 Package Verification

To make sure the GAP packages of algebras of small orders are properly generated and can be loaded in GAP, some tests code is included in the packages. To run the tests in the package `SmallLoops`, for example, log into GAP and issue the command

```
ReadPackage("SmallLoops", "tst/testall.g");
```

The test results displayed would be like

```
      84 ms (82 ms GC) and 175KB allocated for SmallLoops.tst
-----------------------------------
total        84 ms (82 ms GC) and 175KB allocated
0 failures in 1 files


#I  No errors detected while testing
```

8.5 Validation

We generated 6 GAP packages as a practical way to validate the GAP package generator (see Appendix C for their manuals)[4]:

1. `SmallSemigroups` for semigroups up to order 7.
2. `SmallLoops` for loops up to order 9.
3. `SmallQuandles` for quandles up to order 9.
4. `SmallInvSemi` for inverse semigroups up to order 9.
5. `SmallMeadows` for meadows up to order 22.
6. `SmallSemiVarN12Idemp` for the semigroup subvarity var$\{N_2^1 \cap [x^2 = y^2]\}$ up to order 9.

[4]These packages are available in `https://github.com/ChoiwahChow/public/tree/main/smallalgebras`

Each of these packages is generated in less than 65 minutes (see Table 9.25). Using the hardware we have, the highest one or two orders in most of the packages listed above could not have been possible without the algorithms developed in Chapters 4 and 5.

The number of models, up to isomorphism, for semigroups, inverse semigroups, loops, and quandles for small orders can be found in the literature (Sequences A027851, A001428, A057771, and A181769 of OEIS [64], respectively). The results we obtain from the generated GAP packages match those in the literature. As pointed on page 680 of [45], "At present there is no explicit formula for $S(n)$, and the only way to compute $S(n)$ is by a careful exhaustive search." ($S(n)$ denotes the number of inverse semigroups of order n.) So, it is important to have multiple researchers, using different algorithms and/or software, to produce the same numbers. It gives credence to the identical results obtained by different algorithms and software systems. Since the same code base is used to generate GAP packages for all algebras of small orders, the fact that we find no errors in those we check against the literature gives us high confidence that the other packages generated by our automatic GAP package generator are also correct.

8.6 Related Work

Automatic generation of GAP packages of algebras of small orders is a novel idea, although there exist many hand-crafted GAP packages of algebras, such as the `smallsemi` (for semigroups), the `loops` (for quasigroups and loops), the `sonata` (for finite nearrings) [1], and the `sotgrps` (for groups whose orders factorize into at most four primes) [25] packages. Specialized algorithms that depend on deep, special properties of the algebra in question are implemented to optimize both the performance and the storage space for the algebras. This GAP package generator, on the other hand, uses a common representation of algebras of type $(2^m, 1^n)$, and does not rely on the properties of individual algebras. It is particularly suitable for exploring new class of algebras of which the properties are barely known, and for testing the interactions between such algebras.

With the ability to efficiently generate models, the GAP package generator can generate all models (up to isomorphism) of any class of algebra of higher orders and package them into GAP libraries. For example, the `loops` package contains only all loops of orders 5 and 6, the `SmallLoops` package has all loops up to the order 7.

Moreover, the small algebra packages are also compatible with similar, existing packages such as the `loops` package:

```
# SmallLoop is a function in the loops package
# IsASmallLoop is a function in the SmallLoops package
gap> IsASmallLoop(MultiplicationTable(SmallLoop(6, 1)));
true
# SmallSemigroup is a function in the smallsemi package
gap> IsASmallLoop(MultiplicationTable(SmallSemigroup(6, 1)));
false
```

Chapter 9

Results

9.1 Computer System

The programming languages used in developing the code base and scripts are C++ and Python. The C++ and C compilers used is gcc, version 9.4.0. The version of Python used is 3.8.

The operating system used is Ubuntu 20.04.4 LTS. Most of the experiments in this research are run on an Intel® Xeon®Silver 4110 CPU 2.0 GHz $\times$32 computer, with 64 Gb RAM. The exact computer used will be noted at the beginning of the section in case other computers are used.

9.2 Mace4 vs. Top Model Searchers/Enumerators

The experiments in this section are run on an Intel®Core™ i7-9850H CPU 2.6 GHz $\times$12 computer, with 32 Gb RAM. To make the comparison fair, we compare the single-threaded C version of Mace4 with the other two model enumerators.

The definitions of the algebras used in the experiments in this section is given in Table 9.1).

Table 9.1: Definitions of Algebras Used in Experiments

Algebra	Definition in FOL
Semigroups	$x * (y * z) = (x * y) * z.$
IP Loops[1]	$x * y = x * z \rightarrow y = z.\ \ x * y = z * y \rightarrow x = z.\ \ x * 0 = x.$ $0 * x = x.\ \ x' * (x * y) = y.\ \ \ (y * x) * x' = y.$
Inv. Semigroups	$x * (y * z) = (x * y) * z.\ \ x'' = x.\ \ (x * x') * x = x.$ $((x * x') * y) * y' = ((y * y') * x) * x'.$
Quasigroups	$x * y = x * z \rightarrow y = z.\ \ y * x = z * x \rightarrow y = z.$

[1] See definition of IP Loops in [40].

9.2.1 Mace4 vs. Minion

Minion inputs for our tests are typically hundreds and thousands of lines long, hence it is not feasible to manually write them. Instead, we write them in the Essence modeling language [30], and use Savile

Row [54] to translate the Essence files to the Minion format. Many of the Essence input files for our tests are given as examples in Savile Row's release package, and these input files are used in our tests whenever available. Results of the tests are summarized in Table 9.2.

Table 9.2: Finite Model Expansion Performance Comparison between Mace4 and Minion.

	Mace4		Minion	
Algebra	Time	#Models	Time	#Models
Semigroups, order 5	1s	19,507	1.1s	82,511 millions
Semigroups, order 6	70s	5.6 millions	224s	17.06 millions
Semigroups, order 7	2.88h	1,021,120,198	Killed after 9 hours	N/A
IP Loops, order 10	0.24s	988	4.1	1,778
IP Loops, order 11	4s	1,178	54.7s	8,385
IP Loops, order 12	2,474s	193,368	3,184s	696,321
Inv. Semigroups, order 6	2s	41,096	50s	95,976
Inv. Semigroups, order 7	57s	968,795	3,290s	2,930,508
Quasigroups, order 5	0.14s	10,944	0.33s	72,576
Quasigroups, order 6	147s	12 millions	857s	183 millions

Granted, special techniques for specific problems may make Minion much faster for those specific problems. However, the results show that Mace4 is much faster than Minion in all out test cases when they are used in the simplest settings. Mace4 also generates far fewer (isomorphic) models than Minion.

9.2.2 Mace4 vs. IDP^3

Up to IDP^2 in 2012, the IDP system was a FOL model expansion system much like Mace4. Since then it has evolved into a knowledge based system. IDP^3 has been extended with inductive definition, aggregates, partial function, and types [20, 24]. These extensions are great, especially in the software front. However, its finite model expansion module, Minisat(ID), is still based on Minisat, which has not been changed since 2008. Minisat(ID) has extended the coverage of Minisat, but it does not seem to have improved its performance. In fact, in our test cases, we find that Mace4 is often one or two orders of magnitude faster than IDP^3, as shown in Table 9.3.

Table 9.3: Finite Model Expansion Performance Comparison between Mace4 and IDP^3.

	Mace4		IDP^3	
Algebra	Time	#Models	Time	#Models
Semigroups, order 6	70s	5.6 millions	854s	0.07 million
Semigroups, order 7	2.88h	1,021,120,198	crashed after 3 hours	N/A
IP Loops, order 10	0.24s	988	30s	2,229
IP Loops, order 11	4s	1,178	842s	64,224
IP Loops, order 12	2,474s	193,368	crashed after 3 hours	N/A

9.3 Improvements in Mace4

The C++ version of Mace4 is often faster than its C version, although the C++ version is a straight port of the original C version (see Table 9.4).

Table 9.4: Running C/++ Versions of Mace4

Algebra	Order	C Mace4 Time (s)	C++ Mace4 Time (s)	% Speedup
Chains	10	63	52	17
Inverse Semigroups	7	70	54	23
Loops	7	7	5	29
Quandles	8	48	37	23
Meadows	24	23	19	17
Semigroups	6	99	75	23
Skew Lattices	7	28	22	21

Mace4 is enhanced so that it outputs models to a separate file for faster downstream processing. This obviates the need to extract models from the outputs and convert the models to the right formats (see Table 9.5).

Table 9.5: Mace4 Performance with and without -A1 Option

			Mace4 without -A1			Mace4 with -A1	
Algebra	Order	#Models	Mace4 Time (s)	Interpformat Time (s)	Total Disk Space	Mace4 Time (s)	Total Disk Space
Semigroups	5	90,536	2	157	45 Mb	2	13.5 Mb
	6	5,628,898	159	killed after 4.5 h	>3 Gb	128	1.0 Gb
Loops	6	624	0	0	0.4 Mb	1	0.1 Mb
	7	223,808	8	105,135	150.9 Mb	7	46.4 Mb

9.4 Invariants[1]

The experiments are run on an Intel® Core™ i7-9850H CPU 2.6 GHz ×12 computer, with 32 Gb RAM.

We have implemented an invariant-based preprocessor to Mace4's isomorphic models filters.

The ALF database [8] (later migrated to the MarcieX database [9]) contains a collection of 158 classes of algebras of high interest to the research community of algebra. Their definitions are conveniently given in first-order formulas that Mace4 can directly process. We use Mace4 to generate models for each algebra of the highest possible order that it can complete within 2 minutes. Mace4 is not able to generate models for 5 of them within that time limit, and they are excluded from the tests. The excluded algebras are: #112 Kleene algebra, #113 Concurrent Kleene algebra, #114 Omega algebra, #137 Steiner quasigroup, and #138 Steiner loop. In addition, Mace4's *isofilter* is not able to handle two of the largest algebras (#8 BL-algebras and #56 Linear Heyting algebras), each has between 1 to 3 million models. These two algebras are also excluded from most of the statistics of the experimental results. So, we end up with 151 classes of algebras in many of our analyses.

[1] Results in this section have been published in articles [4, 5].

When random invariants are used in the experiment, the number of randomly generated invariants is 50, but at most 20 of the best of them will be used. The maximum depth of the expression tree (see Section 4.4.1) is 4, and the maximum number of variables in it is 3. In this section, random invariants are used unless otherwise specified.

Since we run a large comprehensive set of test cases for comparison, the size of each test case is necessarily limited (uniformly and systematically to make comparisons of results meaningful) by the computing resources available. However, even for the small model sizes used in our experiments, the addition of the invariant-based algorithm improves the overall speed by an order of magnitude, without using parallel processing (see Table 9.6).

The overheads of calculating invariants are observed to be on average about 20 to 30% of the total run time in our experimental setting (see the third column in 9.6). However, invariants improve the speed by orders of magnitudes for big algebras. For example, for the longest (in terms of runtime) 10 algebras in our experiment, the invariant-based algorithm improves the overall speed by over 50 times (see Table 9.6). In fact, a very desirable feature of the invariant-based algorithm is that the improvement increases dramatically as the size of the set of models grows. Granted, for algebras with short runtime, the use of invariants may not pay off. But for those cases, the degradation is really insignificant (see Fig. 9.1). Thus, in general, there is no need to have special logic to decide when not to use invariants.

Furthermore, Mace4's *isofilter* is not able to handle two of the largest data sets, but our invariant-based algorithm can partition these models into smaller blocks to fit in Mace4's limits (see Table 9.6).

Table 9.6: Isomorphism Filtering, w/ vs. w/o Invariants

			Total Runtime (s)	
	#Mace4 Outputs	Invariant Calc. Time (s)	With Invariants	Without Invariants
Shortest 10 classes of Algebras	600	0.2	0.5	0.1
Longest 10 classes of Algebras	9,239,818	430	1,982	87,591
All 151 classes of Algebras	33,643,548	1,500	5,030	95,952
2 Isofilter Failed classes of Algebras	4,075,054	208	727.3	N/A

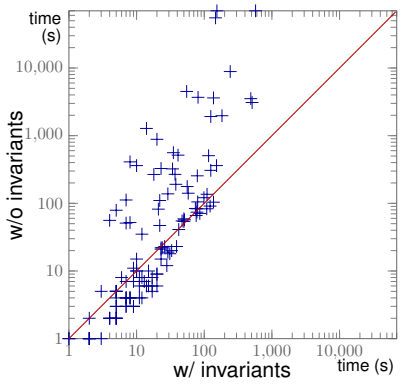


Figure 9.1: Runtimes: w/ vs. w/o Invariants (151 Classes of Algebras in MarcieX)

The performance of the invariant-based algorithm relies heavily on the discriminating power of its invariants. The best possible case is that only 1 non-isomorphic model is in every block, in which case,

only $m-1$ comparisons of models are needed to eliminate all isomorphic models from a block of m models. Our invariants are quite powerful as evidenced by the fact that the average number of non-isomorphic models per block is very close to 1 for the 151 classes of algebras in the experiment (see Table 9.7).

Table 9.7: Discriminating Power (153 classes of Algebras in MarcieX)

	Avg #Non-isomorphic Models per Block	
Percentile	w/ Random Invariants	w/o Random Invariants
95th	1.346	2.677
80th	1.036	1.179
60th	1.003	1.018
40th	1.000	1.005

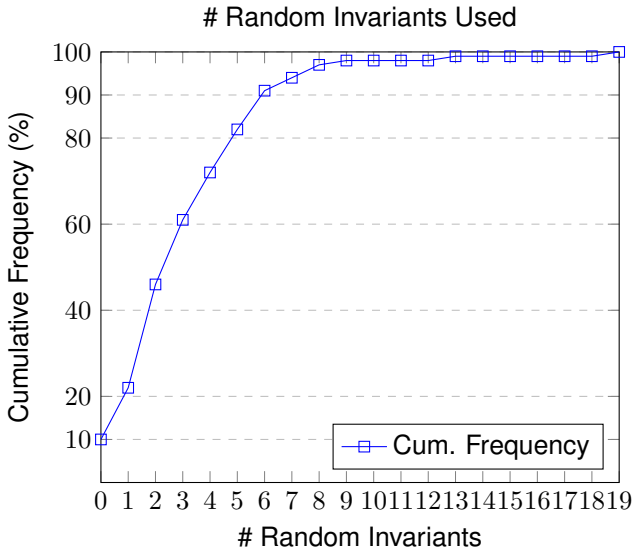


Figure 9.2: # of Random Invariants (153 classes of Algebras in MarcieX)

9.4.1 Basic Invariants vs. Basic Invariants + Random Invariants

As shown in Table 9.7, the hand-crafted basic invariants have very good discriminating power (see the last column in the table). Nevertheless, the addition of random invariants improves the discriminating powers (see the middle column of the table). This increase in discriminating power comes with a small overhead in processing time as the number of random invariants is quite small, usually just a few. For example, 6 or fewer random invariants are used in about 90% of the algebras (see Figure 9.2). For the case when the basic invariants are already doing a very good job, the addition of random invariants may not pay off. But the degradation is minimal because the job would finish fast when the discriminating powers of the invariants are high (See the scatter plot Fig. 9.3). Therefore, there is no need for special logic to decide when not to use random invariants. In our experiment, the overall run time for all 151 classes of algebras is reduced when random invariants are added, with most of the improvements coming from the top 3 classes of algebras, which are among the classes of algebras that take the longest to

finish (see Table 9.8 and Fig. 9.3).

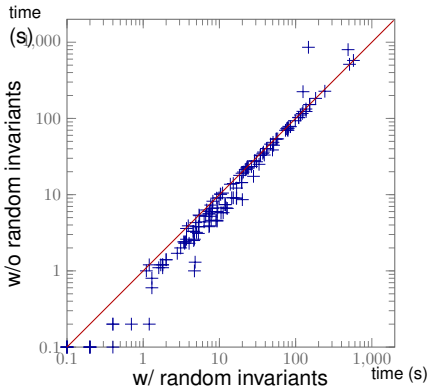


Figure 9.3: Runtimes: w/ vs. w/o Random Invariants

Table 9.8: Isomorphism Filtering Time, w/ and w/o Random Invariants

	Total Time (s)	
	With Random Invariants	Without Random Invariants
Top 3 Improved Algebras	760	1,882
All 151 classes of Algebras	5,758	6,525

9.4.2 Basic Invariants vs. Random Invariants

An interesting question is whether random invariants alone is the fastest way of filtering out isomorphic models. It is conceivable that given enough number of random invariants, all the basic invariants will automatically be covered in it. However, under our experimental setup (use at most 20 best invariants out of 50 random invariants), the basic invariants perform slightly better than the random invariants in general (see Figure 9.4).

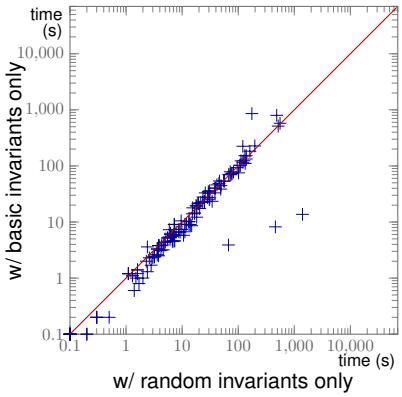


Figure 9.4: Runtimes: w/ only Random Invariants vs. w/ only Basic Invariants

9.4.3 Larger Data Sets

As remarked earlier, to cover a large variety of algebras, we have to limit the order of each algebra in the experiments. Small datasets do not adequately show the true advantage of the invariant-based algorithm. Here we present three examples in which we go two orders higher in each of the algebras, giving us test datasets of over a hundred million models. The first class of algebra (#88 Quasi-MV-algebra) is defined by one binary operation and one unary operation, the second one (#15 Brouwerian semilattices) by two binary operations, and the third one (#4 BCK-join-semilattice) by two binary operations and one relation (see Table 9.9). As shown in Table 9.10, at the baseline when the order of the algebra is small, the invariant algorithm slows down the process very slightly. However, as the order of the algebra goes higher, the invariant-based algorithm improves the speed by orders of magnitudes. In some cases, Mace4 is not able to handle a large number of models. In all the tests, we also observe that the discriminatory powers of the invariants hold up quite well in large datasets (see Table 9.11).

Table 9.9: Definitions of Algebras Used in Experiments

Algebra	Definition in FOL
#88 Quasi-MV-algebra	$x*(y*z)=(x*y)*z.\ \ x''=x.\ \ x+1=1.\ \ (x+0)+0=x+0.$ $(x'+y)'+y=(y'+x)'+x.\ \ (x+0)'=x'+0.\ \ 0'=1.$
#15 Brouwerian semilattices	$x*y=y*x.\ \ (x*y)*z=x*(y*z).\ \ 0*x=x.\ \ x*x=x.$ $x+(y+z)=(x*y)+z.\ \ x+x=0.\ \ (x+y)*x=(y+x)*y.$
#4 BCK-join-semilattice	$(x+y)+((y+z)+(x+z))=1.\ \ 1+x=x.\ \ x+1=1.$ $x+(x*y)=1.\ \ x*((x+y)+y)=((x+y)+y).\ \ x*x=x.$ $x*y=y*x.\ \ (x*y)*z=x*(y*z).\ \ x<y\leftrightarrow x+y=1.$

Table 9.10: Isomorphism Filtering, w/ vs. w/o Invariants for Higher Orders

			Total Runtime (s)	
	Order	#Mace4 Outputs	With Invariants	Without Invariants
#88 Quasi-MV-algebra	7	10,902	1.6	0.7
	8	4,793,924	558	30,701
	9	29,799,618	3,666	N/A[1]
#15 Brouwerian semilattices	6	47,349	12	4
	7	2,247,564	440	1,964
	8	146,875,177	40,017	N/A
#4 BCK-join-semilattice	9	122,754	23	15
	10	1,175,784	305	532
	11	12,307,002	1,213	54,524

[1] Isofilter fails after processing 4.5 million (15% of all) models in 11.5 hours.

9.4.4 Parallel Processing

The invariant-based algorithm is very scalable since the data are divided into blocks that can be processed independently as long as resources are available. In this experiment, we apply parallel processing to the top 3 classes of algebras with the longest runtimes (close to 500s or more). We run each of them with 5 parallel threads and see about a 50 - 60% reduction in run times (see Table 9.12). The

Table 9.11: Discriminating Power of Invariants for Higher Orders

			Non-isomorphic Models	
	Order	#Blocks	Total	Avg per Block
#88 Quasi-MV-algebra	7	567	477	1.19
	8	153,163	55,544	2.76
	9	264,972	141,750	1.87
#15 Brouwerian semilattices	6	745	745	1.00
	7	8,272	8,272	1.00
	8	115,801	114,943	1.01
#4 BCK-join-semilattice	9	26	26	1.00
	10	47	47	1.00
	11	82	82	1.00

Table 9.12: Isomorphism Filtering, Serial vs. Parallel

			Runtime (s)	
	Order	#Blocks	Serial	Parallel
#8 BL-algebras	5	735,820	574	254
#20 Commutative lattice-ordered monoids	7	15,499	510	240
#145 Digroup	12	17	488	177

results would be even better if more resources are available as a large number of blocks are available in these cases.

9.5 Model Generation with Cube

We integrate the cube-based algorithms into the finite model enumerator Mace4, which supports searching on FOL with the LNH and many cell selection strategies [48]. It is considered one of the best finite model finders. Parallelization is controlled outside Mace4. Only minor changes are made to Mace4 to

1. Accept multiple cubes as inputs and continue searching for longer cubes or models from them.

2. Periodically check for signal for work stealing to spin out the shortest available cubes for other workers to work on.

The model searching logic in Mace4 otherwise remains intact. The basic concentric cell selection strategy (see Example 2.3.5 in Section 2.3.1 for its definition) is used in the experiments. A separate program removes isomorphic cubes.

The MarcieX database [9] contains a rich collection of 158 most popular classes of algebras. We pick from it a wide range of representative and challenging problems that involve hundreds of million to over 1.2 billion models. We also draw an interesting example of semigroup subvariety from [3]. The FOL formulas of the algebras used in the experiments in this section are listed in Table 9.13, where $*$ and $-$ are operations, $<$ is a relation, and 0 is a constant and all terms are implicitly universally quantified.

In the tables showing experimental results in this section, the rows with cube length 0 show the results of running Mace4 in a single thread without the cube-based algorithms.

Table 9.13: Definitions of Algebras Used in Experiments

Algebra	FOL Definition
Semigroups	$x * (y * z) = (x * y) * z.$
Semigroups w/ Zero	$x * (y * z) = (x * y) * z. \quad x * 0 = 0. \quad 0 * x = 0.$
var$\{N_2^1 \cap [x^2 = y^2]\}$	$x * (y * z) = (x * y) * z. \ (x * x) * x = x * x. \ x * y = y * x. \ x * x = y * y.$
Tarski Algebras	$(x * y) * y = (y * x) * x. \quad x * (y * z) = y * (x * z). \quad (x * y) * x = x.$
Loops	$x * y = x * z \rightarrow y = z. \quad y * x = z * x \rightarrow y = z. \quad x * 0 = x. \quad 0 * x = x.$
Quasi-ordered Set	$x < y \wedge y < z \rightarrow x < z. \quad x < x.$
Involutive Lattices	$(x * y) * z = x * (y * z). \quad x * y = y * x. \quad (x + y) + z = x + (y + z).$ $x + y = y + x. \quad (x * y) + x = x. \quad (x + y) * x = x.$ $-(x + y) = -x * -y. \quad --x = x.$

Table 9.14 shows the results of applying Theorem 5.2.1 to remove isomorphic cubes for the binary operation of the semigroups of order 7. Observe that the percentage reduction of the number of cubes increases as the cube length increases. The isomorphic cubes removal algorithm is therefore complementary to the LNH because the LNH removes a lot of short cubes but loses its effectiveness as the length of the cubes grows.

Table 9.14: #Cubes for Semigroups of Order 7

	# Cubes		
Cube Length	w/o Isomorphic Cubes Removed	w/ Isomorphic Cubes Removed	% Reduction
2	6	5	16.7
4	34	28	17.6
9	1,568	888	43.4
16	56,206	12,036	78.2
25	1,028,171	59,056	94.3

We run Mace4 to enumerate models of semigroups defined by a single binary operation. The results show a speedup of over 100 times when cubes of length 25 are used, with over 96% of the isomorphic models suppressed (see Table 9.15). The results on semigroups are indicative of the algorithm's usefulness in general to the computational algebraists because algebraic structures related to semigroups are ubiquitous in algebra. Not only are there many well-known semigroup-related algebras, but also many semigroup varieties and subvarieties that are of great research interest [3].

Table 9.15: Running Cubes on Semigroups of Order 7

				Time in min.	
Cube Length	#Cubes	#Models (Millions)	% #Model Reduction	Gen Cubes	Total
0		1,021.1			235.2
2	5	717.7	29.7	0.0	12.5
4	28	611.1	40.2	0.1	9.4
9	888	360.2	64.7	0.1	5.2
16	12,036	158.2	84.5	0.2	2.8
25	59,056	39.5	96.1	0.9	1.7

The semigroup with zero is defined by the same binary operation as semigroups, but with a constant

symbol 0 as the zero element [9]. The constant itself is a symmetry-breaker that reduces the number of cubes and the number of output models. To wit, Mace4 generates over 1 billion semigroups of order 7 (Table 9.15), but less than 353 million semigroups with zero of the same order (Table 9.16). Nevertheless, with a cube length of 25, the cube removal algorithm suppresses the generation of over 65% of the isomorphic models. This shows that the cube removal algorithm works well with other symmetry-breakers.

Table 9.16: Running Cubes on Semigroups w/ Zero of Order 7

				Time in min.	
Cube Length	#Cubes	#Models (Millions)	% #Model Reduction	Gen Cubes	Total
0		353.0			48.4
2	3	353.0	0.0	0.0	3.6
4	3	353.0	0.0	0.1	3.7
9	65	278.3	21.2	0.1	2.7
16	2,723	132.0	62.6	0.2	1.6
25	23,382	32.8	90.7	0.5	1.3

Table 9.17 shows the results of running the algorithms on the semigroup subvariety (see p. 40 of [3] for its definition and discussions). With longer cubes, the algorithms speed up the process by 26 times with 30 threads. The results confirm that the proposed algorithms work remarkably well with semigroup-related algebras.

Table 9.17: Running Cubes on var$\{N_2^1 \cap [x^2 = y^2]\}$ of Order 9

				Time in min.	
Cube Length	#Cubes	#Models (Millions)	% #Model Reduction	Gen Cubes	Total
0		313.0			72.0
2	1	156.5	50.0	0.0	2.9
4	1	156.5	50.0	0.1	2.8
9	2	156.5	50.0	0.1	2.8
16	5	120.9	61.4	0.1	2.3
25	16	55.5	82.3	0.2	1.3
36	70	13.0	95.8	0.3	0.8
49	331	1.5	99.5	1.0	1.1

The Tarski algebra is unlike both the semigroup and the quasigroup in that its multiplication table is not associative and is not a Latin square [9]. It shows the cube-based algorithms perform better and better as the length of the cube increases (see Table 9.18).

Table 9.19 shows the results for loops (a quasigroup-related algebras) defined by a single non-associative binary operation. Here the reduction in the number of the output isomorphic models is not as pronounced. This is expected because the LNH works very well with the Latin square and removes a high percentage of the isomorphic models [69] before the isomorphic cubes removal takes place. For example, while only 0.16% of semigroups of order 7 generated by the LNH are non-isomorphic, 8.7% (106,228,849 out of 1,216,226,816) of the models generated for the loops of order 8 under the LNH

Table 9.18: Running Cubes on Tarski Algebras of Order 13

Cube Length	#Cubes	#Models (Millions)	% #Model Reduction	Time in min. Gen Cubes	Time in min. Total
0		379.6			1,949.9
2	3	189.8	50.0	0.0	70.2
4	1	189.8	50.0	0.1	69.9
9	3	183.3	51.7	0.1	67.7
16	11	158.8	58.2	0.1	58.1
25	55	111.9	70.5	0.2	40.1
36	157	62.1	83.7	0.2	21.8
49	174	24.9	93.4	0.5	8.8
64	171	6.6	98.3	1.0	3.7

alone are non-isomorphic. Nevertheless, the parallel algorithm provides 15 times improvement in speed for cube length of 16.

Table 9.19: Running Cubes on Loops of Order 8

Cube Length	#Cubes	#Models (Millions)	% #Model Reduction	Time in min. Gen Cubes	Time in min. Total
0		1,216			564.0
2	1	1,216	0.0	0.0	47.4
4	2	1,216	0.0	0.1	47.3
9	18	1,216	0.0	0.1	46.2
16	3,583	1,214	0.2	0.1	45.3

The quasi-ordered set is defined by one binary relation. As pointed out in Section 2.3.2, relations can be implemented as functions with two values, T (true) and F (false). The isomorphic cubes algorithms work well on relations just as it works well on functions. As shown in Table 9.20, when cubes of length 36 are used, over 99% of the isomorphic models are suppressed, and the search process is sped up by over 300 times.

Table 9.20: Running Cubes on Quasi-ordered Set of Order 8

Cube Length	#Cubes	#Models (Millions)	% #Model Reduction	Time in min. Gen Cubes	Time in min. Total
0		642.8			59.9
2	1	642.8	0.0	0.0	4.2
4	3	474.6	25.0	0.1	3.2
9	9	209.5	69.0	0.1	1.7
16	33	61.3	90.7	0.1	0.8
25	139	12.6	98.0	0.2	0.3
36	713	2.0	99.7	0.3	0.3

As an example to demonstrate the effectiveness of the algorithms on more complex algebras, consider the *Involutive Lattice* [9], which is defined by two associative binary operations and one unary

operation. For Involutive Lattices of order 13, the search tree has a maximum depth of 351. Using cubes of moderately short length of 105, we obtain a speedup of 300 times, with almost 98% of the isomorphic cubes suppressed (see Table 9.21). The results show that the isomorphic cubes algorithms are highly effective for both simple and complex algebras.

Table 9.21: Running Cubes on Involutive Lattices of Order 13

Cube Length	#Cubes	#Models (Millions)	% #Model Reduction	Time in min. Gen Cubes	Time in min. Total
0		423.0			4,719.7
3	2	423.0	0.0	0.0	432.5
6	3	423.0	37.6	0.1	432.8
10	6	263.9	37.6	0.1	270.0
21	23	178.6	57.8	0.1	180.9
36	108	84.9	79.9	0.2	88.3
55	555	46.0	89.1	0.3	46.2
78	1,710	19.8	95.3	0.5	20.6
105	5,048	8.7	97.9	4.9	14.3

The % reductions in time and number of models (on top of the LNH) are summarized in Figures 9.5 and 9.6. To make the graphs easier to read, the legends are made the same for both graphs, and are shown in Figure 9.6. We observe that with the help of work stealing, the reduction in total time is over 90% even with the short cubes. It is particularly helpful in the cases that there are not many isomorphic cubes to remove, such as the loops. However, the biggest gain in both reduction in time and in isomorphic models is when longer cubes are used. Reduction in isomorphic models also helps tremendously in the post processing step to extract non-isomorphic models.

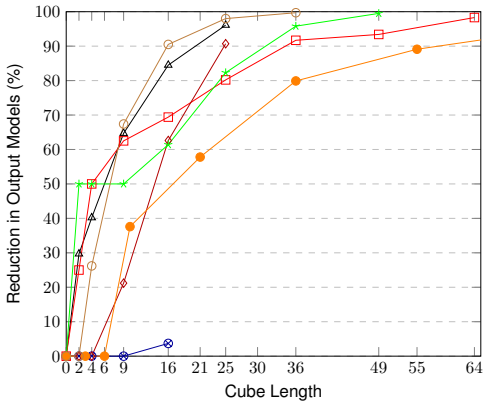


Figure 9.5: Reduction in Number of Output Models

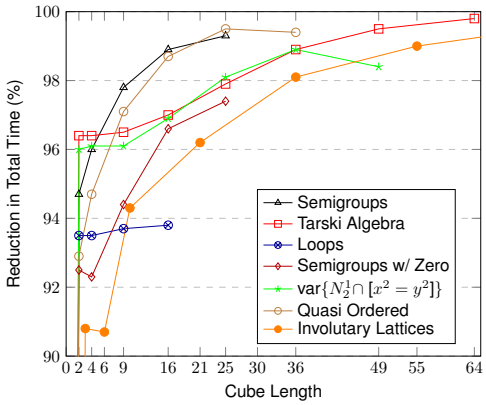


Figure 9.6: Reduction in Total Time with 30 Parallel Processes

Impact on Isomorphic Model Removal

As remarked earlier, reducing the number of Mace4 outputs also reduces the efforts needed to filter out isomorphic models. Table 9.22 shows, using involutive lattices as an example, the out-sized effect of the reduction of Mace4 models on the time to filter out the isomorphic models using the invariant-based

isomorphic model filtering algorithm [5]. For example, for involutive lattices of order 10, the number of Mace4 models is reduced by 40 times (from 575,463 to 13,789) by the isomorphic cubes removal algorithm, the time to filter out the isomorphic models is reduced by 94 times (from 1,405 seconds to 15 seconds). Note also that the higher the order of the algebra, the more the reduction in the isomorphic models removal time.

Table 9.22: Running Invariant-based Isomorphic Models Filter on Involutive Lattices

	w/o Cubes		w/ Cubes		
Order	#Mace4 Output	Isomorphic Model Filter Time (s)	Cube Length	#Mace4 Output	Isomorphic Model Filter Time (s)
9	72,470	224	78	3,670	7
10	575,463	1,405	105	13,789	15
11	4,771,035	71,200	105	97,680	506
12	43,851,030	N/A	105	971,416	10,257

9.5.1 Optimal Cube Length

Up to a certain extent, the search process finishes earlier with fewer isomorphic models with longer cubes. We observe from the experiments that there are three limiting factors on the lengths of the cubes. They are the criteria used in deciding the optimal lengths of cubes to run in the experiments.

First, as the length of the cubes gets longer, more and more models are generated as a result of propagation. This reduces the impact of removing isomorphic cubes because they represent a progressively smaller proportion of the isomorphic models. It is observed that when more than $n-2$ symbols out of the n domain elements are used in the cell terms, the number of (isomorphic) models will be substantial and extending the cube length does not bring enough reduction in isomorphic models to justify the increase in processing time.

Second, even with the help of the invariant vector algorithm, the isomorphic cube removal time grows quite fast as the length of the cube grows. When the isomorphic cubes removal process takes more than a few minutes, further lengthening of the cubes will results in prohibitive overheads in the search process.

Lastly, when the final number of cubes is more than tens of thousands, the overheads in processing them are so high that the search becomes slower (at least for the computers used in the experiments). This factor heavily depends of the number of processors available. More processors mean more parallel processes can be run without slowing down the whole search process.

9.6 Using Cube Algorithms and Invariant Algorithms Together

The Cube algorithms (Chapter 5) and the Invariant algorithms (Chapter 4) are applied at different stages of the model enumeration process and hence can be used together without modifications. Together they are a powerful tool for solving difficult problems. For example, the longest sequence of the numbers of the near-rings [58] is only known up to order 7 (See OEIS [64] sequence A305858), although

longer subsequences are known [34]. We improved the sequence to order 13 by using the Cube algorithms and the Invariant algorithms together. The new results were reported to OEIS and the sequence was updated. We also added the number of models of order 10 for the involutory quandles (OEIS sequence A178432). These results (see Table 9.23) are obtained with many hours of computations using our heavily parallelized algorithms, and the results would otherwise be very difficult to obtain without parallelization.

Table 9.23: New Numbers of Models

Algebra	OEIS Sequence	Order	#Models (up to isomorphism)
Near-Ring	A305858	8	3,856
		9	486
		10	535
		11	139
		12	54,694
		13	454
Involutory Quandles	A178432	10	436,672

Another notable example of using the new algorithm is in finding the models of the INFB monoids. Our results match those reported in the literature up to order 9 and we add new results for order 10 and 11 (Table 9.24).

Table 9.24: Numbers of Models of INFB Monoids

Order	#Models (up to isomorphism)	#Models (up to isomorphism and anti-isomorphism)
6	2	2
7	5	3
8	3	2
9	7	5
10	4	4
11	15	9

9.7 Generation of GAP Packages

An important goal of fast parallel model enumeration algorithms (e.g., the invariant-based isomorphic models filtering algorithms in Chapter 4 and the cube-based model generation algorithms in Chapter 5) is to generate GAP packages of algebras of small orders. Table 9.25 shows the time required to generate packages for various representative classes of algebras.

Data are zipped (compressed) to save space and transfer time. The compression rate is close to 90% or higher for large packages. The data load time in GAP is quite acceptable for practical uses (see Table 9.26).

Table 9.25: GAP Package Generation

Algebra	Orders Range	#Models (up to isomorphism)	Package	
			Size (KB)	Generation Time (min)
Semigroups	2 - 7	1,658,438	6,192	56.9
Inverse Semigroups	2 - 9	32,253	600	52.6
var$\{N_2^1 \cap [x^2 = y^2]\}$[1]	2 - 9	15,843	348	64.7
Loops	2 - 7	23,856	360	3.9
Meadows	2 - 22	32	212	64.6
Quandles	2 - 9	13,064	415	51.8

[1] A semigroup sub-variety. See p. 40 of [3] for its definition.

Table 9.26: GAP Package Data

Algebra	Orders Range	Zipped Data File Size (KB)	Data File Size (KB)	Compression Ratio (%)	GAP Data Load Time (s)
Semigroups	2 - 7	5,848	87,472	93	31.287
Inverse Semigroups	2 - 9	358	2,935	88	1.021
var$\{N_2^1 \cap [x^2 = y^2]\}$[1]	2 - 9	112	1,335	93	0.402
Loops	2 - 7	148	1,264	88	0.632
Meadows	2 - 22	5	14	63	0.014
Quandles	2 - 9	200	2,099	90	0.721

[1] A semigroup sub-variety. See p. 40 of [3] for its definition.

Chapter 10

Future Work

10.1 Mace4

Mace4 has been a popular finite model enumerator and many researchers are advocating its use (see [70] for an example). However, Mace4 must be continually enhanced to remain relevant.

One major improvement would be to incorporate the invariant-based isomorphic models removal algorithm into it, so that isomorphic models can be removed just as they are generated. Note that disk input/output (disk I/O) operations are orders of magnitude slower than CPU operations. Disk I/O should be avoided as much as possible. Removing isomorphic models as they are generated within the same process would obviate the need to write the isomorphic models out to the disk, and read back into an isomorphic model removal (e.g., `isofilter`) process, just to be discarded.

The invariant-based isomorphic models removal algorithm can also be incorporated into standalone isomorphic model removal (e.g., `isofilter`) programs because there are times when models from multiple sources or runs have to be combined and isomorphic models removed from the final outputs.

10.2 Invariants

We have shown that the use of basic invariants improves the performance of isomorphic model filtering, sometimes by orders of magnitudes in the case of large data sets, across a wide range of algebras under active research. We have shown further that addition of random invariants not only speed up the filtering of isomorphic models, but also help make sure important invariants are included.

As pointed out in Section 9.4, the efficiency of the invariant-based algorithm relies heavily on the discriminating powers of both the hand-crafted and the randomly generated invariants. Future work will therefore concentrate on finding powerful invariants to target common algebraic structures, and to find the best parameters to generate optimal random invariants. Some research questions are:

1. What depth and breadth of the expression tree would be most cost-effective in generating random invariants?

2. What is the best range of ratios of binary operations to unary operations in a random invariant?

3. What is the best size of the sample to use in finding the optimal set of random invariants?

10.3 Model Generation with Cubes

There are further promising ways to improve the performance of the cube-based parallel algorithms. The most challenging work is on incorporating different cell selection strategies into the algorithm.

Many cell selection strategies are available for the traditional finite model finders, but there are no simple ways to decide which one will work best in any particular case. We need to resort to empirical methods to help predict the performance each of the cell selection strategy, or a combination of them, for different cases. The finite model finder will also need to be modified to run different cell selection strategies with the cube-based parallel algorithm. Machine-learning algorithms may be applied to predict the best cell selection strategy to use given the characteristics of the problem, such as the domain size, number of functions, and the properties of the functions (e.g., idempotent, commutativity, associativity etc.) in the first-order formula.

10.4 Magmaut

GAP is built on top of the programming language C and supports user packages with embedded C code. Many existing GAP packages have C components to improve their efficiency. Magmaut has a C library to calculate the invariants of algebras represented by their multiplication tables.

One reason that GAP's programming language is slower than C is because it is an interpretive language. Libraries in C are compiled and optimized before they are loaded and run in an interactive GAP session.

Furthermore, GAP's built-in hash-table is not as efficient as those available in other mainstream programming languages such as C/C++ and Python. Magmaut's function `MagmasUpToIsomorphism`, which returns all non-isomorphic algebras from a list of algebras, can certainly benefit from an efficient hash-table, because the hashing algorithm described in Section 4.5 could improve its efficiency immensely. So, a possible improvement to the `MagmasUpToIsomorphism` is to code most of it in C.

However, there are overheads in calling C functions and getting the results back from C functions in the GAP platform. Thus, it would only be beneficial to code in C for functions that run for long time. Moreover, GAP's built-in functionality is not available in the C language. That could make coding new GAP functionality in C very difficult. Care has to be exercised in deciding which parts of what functions should be ported to C. All in all, more extensive use of the low-level language C could improve the efficiency of Magmaut.

10.5 GAP Package Generation

There is no simple way to predict how much time it would take to generate models, up to isomorphism, of an algebra of a particular order. So the user would simply stop the GAP package generation process when it is running for too long, and start with the process again with a smaller order. An improvement to the process would be to add a time-out parameter to the inputs. So, when the system is generating models of order n and the maximum time allowed is reached, then the system would stop the incomplete model generation step and to use the models of orders up to $n-1$ to complete GAP package.

Chapter 11

Conclusions

11.1 Contributions

The major contributions of the present work is on devising and inventing a series of novel parallel algorithms to efficiently enumerate finite non-isomorphic models of classes of algebras of small orders, and on wrapping these algorithms into a system that automatically generates GAP packages with data, code, and documentation. We improve every step of the GAP package generation process for algebras of small orders, starting from using Mace4 to enumerate models from FOL formula, to filtering isomorphic models generated by Mace4, to finally generating the GAP packages. Specifically, we present in this thesis

1. An improved 64-bit version of Mace4 in C++ that is efficient and supports many new parallel algorithms.

2. Parallel algorithms that use invariants both as discriminators and as hash keys to partition a set of models into blocks, in which no models across blocks are isomorphic. The blocks are hashed into a hasp-map so that they will never be processed together. Included in the algorithm is the novel idea of using randomly generated invariants to supplement hand-crafted invariants to make the algorithm more robust. We show that the invariant-based algorithm is simple, efficient, scalable, and parallelizable. It is also compatible with most, if not all, existing finite model enumerators. It can be used as a stand-alone preprocessor to split models into blocks to feed into isomorphic model filters, or it can be directly incorporated into them (see Chapter 4).

3. Parallel algorithms together with a novel symmetry-removal mechanism for enumerating finite algebras (see Chapter 5). The symmetry-removal mechanism (isomorphic cube removal algorithm) is remarkable in that it complements the LNH, which is a well-known powerful symmetry-removal algorithm.

4. The GAP package, `Magmaut`, which contains functions that generate automorphism groups and functions that test for isomorphisms between algebras of type $(2^m, 1^n)$.

5. A library of scripts and programs to automatically generate GAP packages, with documentation, for algebras of small orders. Some sample packages are included in this thesis.

The algorithms proposed in this project have the following desirable characteristics:

1. They are parallel algorithms with no or minimal synchronizations required between parallel jobs. The overheads in parallelizing the tasks are small.
2. They cope well with algebras of increasing size. Longer cubes are possible with algebras of higher orders, and with longer cubes, more isomorphic cubes are removed. Also, more invariant vectors are possible with higher orders of algebras, making more blocks available to spread out the models.
3. They are very scalable.
4. They are compatible with almost all existing algorithms in finite model enumerations. Users do not need to give up, for example, the LNH, or any other symmetry-breaking algorithms to use the isomorphic-cube removal algorithm or the invariant-based algorithm.
5. They require minimal changes to the code base of traditional finite model enumerators to be incorporated into their code base.

Most importantly, the correctness of the algorithms are rigorously proved (see, for example, Theorem 5.2.1).

11.2 Final Remarks

To fulfill an important unmet need for an efficient algorithm for enumerating finite algebraic models in computational algebra, we first enhance the existing finite model enumeration process with the parallel cubes algorithm and the isomorphic cubes removal algorithm that reduce both the runtime and the number of output isomorphic models. These new algorithms are applicable to a wide variety of models definable by the FOL, and are so scalable that they can be used in a mathematician's laptop as well as a cluster of powerful computers. Furthermore, they require minimal efforts to safely integrate into existing finite model finders. Very importantly, these algorithms can be used as a black-box without requiring the uses to have any knowledge about how they work. Mathematicians can then concentrate on mathematics instead of the tools of mathematics. Moreover, during the development of the said algorithms, we discover a novel, intuitive, and direct proof of the LNH.

Next, we use invariants both as discriminators and as hash keys to partition models into blocks, in which no models across blocks are isomorphic. The blocks are hashed into a hasp map so that they will not be processed together. Included in the invariant-based isomorphic model filtering algorithm is the novel idea of using randomly generated invariants to supplement hand-crafted invariants to make the algorithm more robust. We show that the invariant-based algorithm is simple, efficient, scalable, and parallelizable. It is also compatible with most, if not all, existing finite model enumerators. It can be used as a stand-alone preprocessor to split models into blocks to feed into isomorphic model filters, or

it can be directly incorporated into them. Future research will concentrate on finding powerful invariants in different areas of algebraic structures, and on the automatic discovery of optimal random invariants.

Finally, we provide scripts to automatically generate GAP package of algebra of type $(2^m, 1^n)$ for a given first-order formula. Included in the package are functions from the `Magmaut` package that contain functions to test isomorphisms and automorphism for algebras of type $(2^m, 1^n)$. This allows mathematicians to conveniently test their ideas on algebras of interest to them.

In conclusion, with the algorithms and the GAP package-generation scripts presented in this thesis, we fulfill our ultimate goal of helping free mathematicians from spending time on the tools of mathematics so that they have more time to spend on mathematics.

Conclusões

Para atender a uma importante necessidade não atendida de um algoritmo eficiente para enumerar modelos algébricos finitos em álgebra computacional, primeiro aprimoramos o processo de enumeração de modelos finitos existente com o algoritmo de cubos paralelos e o algoritmo de remoção de cubos isomórficos que reduzem o tempo de execução e o número de saídas isomórficas modelos. Esses novos algoritmos são aplicáveis a uma ampla variedade de modelos definidos em FOL e são tão escaláveis que podem ser usados no laptop de um matemático, bem como em um cluster de computadores poderosos. Além disso, eles exigem esforços mínimos para se integrar com segurança aos nos geradores já conhecidos de modelos finitos. Muito importante, esses algoritmos podem ser usados como uma caixa preta sem exigir que os utilizadores tenham qualquer conhecimento sobre como eles funcionam. Os matemáticos podem então se concentrar na matemática em vez das ferramentas da matemática. Além disso, durante o desenvolvimento dos referidos algoritmos, descobrimos uma prova nova, intuitiva e direta do LNH.

Em seguida, usamos invariantes tanto como discriminadores quanto como chaves de hash para particionar modelos em blocos, nos quais nenhum modelo entre blocos é isomórfico. Os blocos são hash em um mapa de hasp para que eles não sejam processados juntos. Incluída no algoritmo de filtragem de modelo isomórfico baseado em invariantes está a nova ideia de usar invariantes gerados aleatoriamente para complementar invariantes feitos à mão para tornar o algoritmo mais robusto. Mostramos que o algoritmo baseado em invariantes é simples, eficiente, escalável e paralelizável. Também é compatível com a maioria, se não todos, os enumeradores de modelos finitos existentes. Ele pode ser usado como um pré-processador autônomo para dividir modelos em blocos para alimentar filtros de modelos isomórficos, ou pode ser incorporado diretamente neles. Pesquisas futuras se concentrarão em encontrar invariantes poderosos em diferentes áreas de estruturas algébricas e na descoberta automática de invariantes aleatórios ótimos.

Finalmente, fornecemos scripts para gerar automaticamente o pacote GAP de álgebra do tipo $(2^m, 1^n)$ para uma determinada fórmula de primeira ordem. Incluídas no pacote estão funções do pacote `Magmaut` que contém funções para testar isomorfismos e automorfismos para álgebras do tipo $(2^m, 1^n)$. Isso permite que os matemáticos testem convenientemente as ideias a propósito do álgebras de seu interesse.

Em conclusão, com os algoritmos e os scripts de geração de pacotes GAP apresentados nesta tese, cumprimos o objetivo final de ajudar os matemáticos a liberarem o tempo gasto nas ferramentas da matemática para que tenham mais tempo para gastar em matemática.

Bibliography

[1] E. Aichinger, F. Binder, J. Ecker, P. Mayr, and C. Nöbauer. SONATA, system of nearrings and their applications, Version 2.9.4. https://gap-packages.github.io/sonata/, Apr 2022. Refereed GAP package.

[2] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. Np-hardness of euclidean sum-of-squares clustering. *Mach. Learn.*, 75(2):245–248, 2009.

[3] J. Araújo, J. P. Araújo, P. J. Cameron, E. W. H. Lee, and J. Raminhos. A survey on varieties generated by small semigroups and a companion website. https://arxiv.org/abs/1911.05817, 2019.

[4] J. Araújo, C. Chow, and M. Janota. Filtering isomorphic models by invariants. In L. D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:9, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] J. Araújo, C. Chow, and M. Janota. Boosting isomorphic model filtering with invariants. *Constraints*, 27(3):360–379, Jul 2022.

[6] J. Araújo, C. Chow, and M. Janota. Symmetries for cube-and-conquer in finite model finding. (in press), 2023.

[7] J. Araújo, M. Janota, and C. Chow. Magmaut manual. https://github.com/ChoiwahChow/magmaut, 2021.

[8] J. Araújo, D. Matos, and J. Ramires. Axiomatic library finder (database). https://axiomaticlibraryfinder.pythonanywhere.com.

[9] J. Araújo, D. Matos, and J. Ramires. MarcieDB: a model and theory database. https://marciedb.pythonanywhere.com, 2022.

[10] J. Araújo, R. B. Pereira, W. Bentz, C. Chow, J. Ramires, L. Sequeira, and C. Sousa. Cream: a package to compute [auto, endo, iso, mono, epi]-morphisms, congruences, divisors and more for algebras of type $(2^n, 1^n)$. https://arxiv.org/abs/2202.00613, 2022.

[11] G. Audemard, B. Benhamou, and L. Henocque. Predicting and detecting symmetries in FOL finite model search. *J. Autom. Reason.*, 36(3):177–212, 2006.

[12] 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 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 427–442, Berlin, Heidelberg, 2001. Springer.

[13] L. Baptista and J. P. M. Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In R. Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 489–494, Berlin, Heidelberg, 2000. Springer.

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

[15] S. Bhatti, C. M.A., and Z. A.H. Characterization of bci-algebras of order 5. *Punjab University Journal of Mathematics*, 25:99–121, 1992.

[16] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[17] T. Boy de la Tour and P. Countcham. An isomorph-free SEM-like enumeration of models. *Electronic Notes in Theoretical Computer Science*, 125(2):91–113, 2005. Proceedings of the 5th International Workshop on Strategies in Automated Deduction (Strategies 2004).

[18] A. Buch and T. Hillenbrand. Waldmeister: Development of a high performance completion–based theorem prover. 06 1998.

[19] S. Burris and H. P. Sankappanavar. *A course in universal algebra*, volume 78 of *Graduate texts in mathematics*. Springer, New York, NY, 1981.

[20] B. D. Cat, B. Bogaerts, M. Bruynooghe, G. Janssens, and M. Denecker. Predicate logic as a modeling language: the IDP system. In M. Kifer and Y. A. Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 279–323. ACM / Morgan & Claypool, 2018.

[21] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-based work stealing in parallel constraint programming. In I. P. Gent, editor, *Principles and Practice of Constraint Programming - CP, 15th International Conference*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2009.

[22] 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.

[23] 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, San Francisco, CA, 1996. Morgan Kaufmann.

[24] M. Denecker. Building the knowledge base system idp3. https://www.cs.nmsu.edu/ALP/wp-content/uploads/2013/10/Denecker.pdf, 2013.

[25] H. Dietrich, B. Eick, and X. Pan. Groups whose orders factorise into at most four primes. *Journal of Symbolic Computation*, 108:23–40, 2022.

[26] A. Distler, C. Jefferson, T. Kelsey, and L. Kotthoff. The semigroups of order 10. In M. Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 883–899. Springer, 2012.

[27] 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.

[28] J. D. Dixon and B. Mortimer. *Permutation Groups*. Springer, New York, NY, 1996.

[29] M. Elhamdadi, J. Macquarrie, and R. Restrepo. Automorphism groups of quandles. *J. Algebra Appl.*, 11(1), 2012.

[30] A. M. Frisch, M. Grum, C. Jefferson, B. M. Hernández, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In M. M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 80–87, 2007.

[31] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.1*, 2021.

[32] 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.

[33] I. P. Gent, I. Miguel, P. Nightingale, C. McCreesh, P. Prosser, N. C. A. Moore, and C. Unsworth. A review of literature on parallel constraint solving. *Theory Pract. Log. Program.*, 18(5-6):725–758, 2018.

[34] A. Golev and A. Rahnev. Computing near rings on finite cyclic groups of order up to 29. *Doklady Bolgarskoi Akademiya Nauk*, 64, 01 2011.

[35] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In J. Mostow and C. Rich, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, pages 431–437, Menlo Park, CA / Cambridge, MA, 1998. AAAI Press / The MIT Press.

[36] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In K. Eder, J. Lourenço, and O. Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC, Revised Selected Papers*, volume 7261, pages 50–65. Springer, 2011.

[37] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina. Lookahead in partitioning SMT. In *Formal Methods in Computer Aided Design, FMCAD*, pages 271–279. IEEE, 2021.

[38] M. Janota and M. Suda. Towards smarter MACE-style model finders. In G. Barthe, G. Sutcliffe, and M. Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 454–470, Manchester, UK, 2018. EasyChair.

[39] 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.

[40] M. A. Khan. Efficient enumeration of higher order algebraic structures. *IEEE Access*, 8:41309–41324, 2020.

[41] L. Kotthoff and N. C. A. Moore. Distributed solving through model splitting. *ArXiv*, abs/1008.4328, 2010.

[42] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[43] K. Kunen. The structure of conjugacy closed loops. *Transactions of the American Mathematical Society*, 352(6):2889–2911, 2000.

[44] I. Lynce, L. Baptista, and J. Marques-Silva. Complete unrestricted backtracking algorithms for satisfiability. In *In Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 214–221, 2002.

[45] M. E. Malandro. Enumeration of finite inverse semigroups. *Semigroup Forum*, 99:679–723, 2019.

[46] A. Malapert, J. Régin, and M. Rezgui. Embarrassingly parallel search in constraint programming. *J. Artif. Intell. Res.*, 57:421–464, 2016.

[47] D. Marker. *Model Theory: An Introduction*. Springer, New York, NY, 2002.

[48] W. McCune. Mace4 reference manual and guide, 2003.

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

[50] B. D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998.

[51] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535, New York, NY, 2001. ACM.

[52] 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.

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

[54] P. Nightingale, Özgür Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017.

[55] F. Nisar and S. A. Bhatti. A note on bci-algebras of order five. *Punjab University Journal of Mathematics*, 38:15–37, 2006.

[56] A. Palmieri, J. Régin, and P. Schaus. Parallel strategies selection. *CoRR*, abs/1604.06484, 2016.

[57] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Upper Saddle River, NY, 1982.

[58] G. Pilz. *Near-rings: What they are and what they are good for*. Univ., 1981.

[59] 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 - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *Lecture Notes in Computer Science*, pages 3–21, Switzerland AG, 2019. Springer.

[60] J. Régin and A. Malapert. Parallel constraint programming. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 337–379. Springer, 2018.

[61] J. Régin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In C. Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 596–610, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[62] O. Sapir. Non-finitely based monoids. *Semigroup Forum*, 90(3):557–586, Jun 2015.

[63] S. Schulz. *The E Equational Theorem Prover – User Manual*. 01 2004.

[64] N. J. A. Sloane and T. O. F. Inc. The on-line encyclopedia of integer sequences, 2020.

[65] J. Sneyers, P. van Weert, T. Schrijvers, and L. de Koninck. As time goes by: Constraint handling rules: A survey of chr research from 1998 to 2007. *Theory and Practice of Logic Programming*, 10(1):1–47, 2010.

[66] C. F. M. Sousa. Prover9. `https://gitlab.com/cfmsousa/prover9`, 2020.

[67] G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022.

[68] 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.

[69] H. Zhang. Combinatorial designs by SAT solvers. *Handbook of Satisfiability*, pages 533–568, 2009.

[70] 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.

[71] J. Zhang. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, 17:1–22, 08 1996.

[72] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *IJCAI*, pages 298–303, 1995.

Appendix A

Algebras Used in Experiments

Following is the list of 158 algebras from MarcieDB used in the experiments.

The MarcieDB does not contain the definition of the semigroups. So, the usual definition of semigroups, $(x * y) * z = x * (y * z)$, is prepended to the list below as *#0 Semigroups* as needed in our experiments.

#1 Almost distributive lattices
#2 BCI-algebras
#3 BCK-algebras
#4 BCK-join-semilattice
#5 BCK-lattices
#6 BCK-meet-semilattice
#7 Bilattice
#8 BL-algebras
#9 Boolean algebra
#10 Boolean monoid
#11 Boolean ring
#12 Bounded lattice
#13 Bounded residuated lattices
#14 Brouwerian algebras
#15 Brouwerian semilattices
#16 Chains
#17 Clifford algebras
#18 Commutative BCK-algebras
#19 Commutative integral ordered monoid
#20 Commutative lattice-ordered monoids
#21 Commutative lattice-ordered rings
#22 Commutative lattice-ordered semigroups
#23 Commutative ordered monoids
#24 Commutative ordered semigroups
#25 Commutative partially ordered monoids
#26 Commutative partially ordered semigroups
#27 Commutative regular rings
#28 Commutative residuated partially ordered monoids
#29 Complemented distributive lattices
#30 Complemented lattices
#31 Complemented modular lattices
#32 Loop
#33 De Morgan monoids
#34 Dense linear orders
#35 Directoids
#36 Distributive lattice ordered semigroups
#37 Division ring
#38 Blanked out
#39 f-rings
#40 Fields
#41 FLcw-algebras
#42 Function rings
#43 Generalized Boolean algebras
#44 Hilbert algebras
#45 Hoop
#46 Implication algebras
#47 Implicative lattices
#48 Integral domains
#49 Integral ordered monoids
#50 Involutive lattices
#51 Join-semilattices
#52 Lattice-ordered pregroups
#53 Lattice-ordered rings
#54 Lattice-ordered semigroups
#55 Lineales
#56 Linear Heyting algebras
#57 Linear orders
#58 m-zeroids
#59 Medial quasigroups
#60 Meet-semilattices
#61 Modular ortholattices
#62 Monadic algebras
#63 Quasigroup
#64 Moufang quasigroups
#65 Multiplicative lattices
#66 Multiplicative semilattices
#67 Near-fields
#68 Near-ring
#69 Near-rings with identity
#70 Neardistributive lattices

#71 Normal bands
#72 Normal valued lattice-ordered groups
#73 Ockham algebras
#74 Order algebras
#75 Ordered monoids
#76 Ordered monoids with zero
#77 Ordered semigroup
#78 Ordered semilattice
#79 Ore domains
#80 Ortholattices
#81 Orthomodular lattices
#82 Partially ordered groups
#83 Partially ordered monoids
#84 Partially ordered semigroups
#85 Pocrims
#86 Posets
#87 Quandles
#88 Quasi-MV-algebra
#89 Quasi-ordered sets
#90 Residuated lattices
#91 Residuated partially ordered monoids
#92 Residuated posets
#93 Right hoops
#94 Semigroups with zero
#95 Semirings with identity
#96 Semirings with identity and zero
#97 Semirings with zero
#98 Sequential algebras
#99 Shells
#100 Skew lattices
#101 Skew-fields
#102 Tarski algebras
#103 Totally ordered monoids
#104 Wajsberg hoops
#105 Near-semiring
#106 MV-algebra
#107 Orthoimplication algebra
#108 Orthomodular implication algebra
#109 Quasi-implication algebra
#110 Idempotent semiring
#111 Concurrent semiring
#112 Kleene algebra
#113 Concurrent Kleene algebra
#114 Omega algebra
#115 Concurrent semiring with invariants
#116 Ground mereotopologies
#117 Domain semigroup
#118 Domain monoid
#119 Left closure semigroup
#120 Left closure monoid
#121 Inverse semigroup
#122 Domain range semigroup
#123 Antidomain monoid
#124 Semilattice pseudo-complemented semigroup
#125 Closable semilattice pseudo-complemented semigroup
#126 Cancellative semigroup
#127 Involutory quandle
#128 Left Bol loop
#129 Right Bol loop
#130 left conjugacy closed loop
#131 Right conjugacy closed loop
#132 Conjugacy closed loop
#133 RIF loop
#134 ARIF loop
#135 Extra loop
#136 F-quasigroup
#137 Steiner quasigroup
#138 Steiner loop
#139 Rectangular quasigroup
#140 Rectangular loop
#141 Trisemigroup
#142 Trigroup
#143 Blanked out
#144 Double Ward quasigroup
#145 Digroup
#146 Left disemigroup
#147 Right disemigroup
#148 Disemigroup
#149 Left I-Semiring
#150 Doppelsemigroup
#151 Strong doppelsemigroup
#152 Dimonoid
#153 Commutative dimonoid
#154 Diassociative semigroup
#155 Duplicial semigroup
#156 Extended duplicial semigroup
#157 Associative dialgebra
#158 Associative trialgebra
#159 Trialgebra
#160 Associative dimonoid

Appendix B

Published and To-be-published Articles

Titles of the articles in this appendix:

1. Filtering Isomorphic Models by Invariants [4] (Published).
2. Boosting isomorphic model filtering with invariants [5] (Published).
3. Symmetries for cube-and-conquer in finite model finding [6] (Accepted by the 29th International Conference on Principles and Practice of Constraint Programming (CP 2023)).
4. CREAM: A PACKAGE TO COMPUTE [AUTO, ENDO, ISO, MONO, EPI]-MORPHISMS, CONGRUENCES, DIVISORS AND MORE FOR ALGEBRAS OF TYPE $(2^n, 1^n)$ [10] (Published).
5. Number of Non-isomorphic Models of Common Algebras (To be submitted to the *Journal of Integer Sequences*).

Filtering Isomorphic Models by Invariants

João Araújo
NOVA University Lisbon, Portugal

Choiwah Chow
Universidade Aberta, Lisbon, Portugal

Mikoláš Janota
Czech Technical University in Prague, Czech Republic

Abstract

The enumeration of finite models of first order logic formulas is an indispensable tool in computational algebra. The task is hindered by the existence of isomorphic models, which are of no use to mathematicians and therefore are typically filtered out a posteriori. This paper proposes a divide-and-conquer approach to speed up and parallelize this process. We design a series of invariant properties that enable us to partition existing models into mutually non-isomorphic blocks, which are then tackled separately. The presented approach is integrated into the popular tool Mace4, where it shows tremendous speed-ups for a variety of algebraic structures.

2012 ACM Subject Classification Computing methodologies; Theory of computation → Constraint and logic programming

Keywords and phrases finite model enumeration, isomorphism, invariants, Mace4

Digital Object Identifier 10.4230/LIPIcs.CP.2021.4

Category Short Paper

Funding *João Araújo*: 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.
Mikoláš Janota: The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

1 Introduction

There are many types of relational algebras (groups, semigroups, quasigroups, fields, rings, MV-algebras, lattices, etc.) using operations and relations of many arities, but the overwhelming majority of the most popular only use operations of arity at most 2; in the words of two famous algebraists, *It is a curious fact that the algebras that have been most extensively studied in conventional (albeit modern!) algebra do not have fundamental operations of arity greater than two.* (See page 26 of [4])

To study and get intuition on them, mathematicians resort to libraries of all order n models of the algebra they are interested in (for small values of n). These libraries allow experiments such as testing and/or forming conjectures etc., to gain insights. Therefore, it comes as no surprise that GAP [8], the most popular computational algebra system, has many such libraries. For groups it has the list of almost all small groups up to order a few thousands and the list of all primitive groups up to degree a few thousands, among others; for semigroups it has the list of all small models up to order 8 [6]; for quasigroups up to order 6 [16]; there is also a library of Lie algebras and many others. These libraries are so important that the search for them has a long history in mathematics predating for many years the use of computers. For example, the search for libraries of degree n primitive groups

started long ago: Jordan (1872) for $n \leq 7$; Burnside (1897) for $n \leq 8$; Manning (1906-1929) for $n \leq 15$; Sims (1970) for $n \leq 20$, Pogorelov (1980) for $n \leq 50$; Dixon and Mortimer (1988) for $n \leq 1000$. (See Appendix B of [7]; and for more recent results in OEIS [17]).

Many more such libraries are needed. For example, SMALLSEMI [6] has the list of semigroups up to order 8 (there are too many semigroups of order 9 to be storable), but if we impose extra properties on the semigroup (such as being inverse, a band, regular, or Clifford, etc. – there are tens of classes of semigroups –) their numbers decrease and hence libraries of models of higher orders could be produced and stored.

Many of these algebras can be defined in first order logic (FOL) and there are tools to allow mathematicians to encode their algebras and produce a meaningful library. The problem is that usually the tools that can be easily learned and used by mathematicians generate too many isomorphic models, thus wasting time generating redundant models and then wasting more time to get rid of them. For example, Mace4 [13], a very popular finite model enumerator among mathematicians due to its very intuitive and user-friendly language, would produce 28,947,734 inverse semigroups of order 8 when given the following simple first-order formulas as input [1] (with binary operation $*$ and unary operation $'$).

$$(x * y) * z = x * (y * z). \qquad (x * x') * x = x. \qquad 0 * 0 = 0.$$
$$((x * x') * y) * y' = ((y * y') * x) * x'. \qquad x'' = x.$$

During the search, the number of output models in this example is already greatly reduced by the Least Number Heuristic (LNH) and the special symmetry breaking input clause $0 * 0 = 0$. Out of the almost 29 millions output models, only 4,637 ($\approx 0.016\%$) are pairwise non-isomorphic. The proportion of non-isomorphic models in the outputs tends to get smaller very fast as the order of the algebraic structure goes higher.

Redundant models may either be eliminated during search or filtered out afterwards. Guaranteeing that search never produces isomorphic models is a hard problem and is rarely seen in modern solvers. This paper therefore tackles the second problem, i.e., the removal of redundant models from an already enumerated set.

In our context, the complexity of checking whether two models are isomorphic is only part of the problem. Another source of complexity is the large number of models that need to be checked. If all pairs of models are checked, the performance degrades rapidly as the total number of models increases (see Section 5).

To tackle this problem, we explored many different strategies eventually concluding that the best one is to assign to every generated model a vector that is invariant under isomorphism. This allows us to partition the output with all the isomorphic models living inside the same block (or part). This splits the problem into substantially smaller sub-problems. Moreover, processing inside each block can easily be done in parallel as models across blocks cannot be isomorphic. This is an important facet of the approach since modern-day computers are more often than not equipped with multiple cores.

What made this project take off was the identification of a large number of general algebra properties invariant under isomorphism coupled with experiments to identify a small subset of these properties without losing discriminating power. This approach will help mathematicians on two levels: first, it provides them with a tool on their desktop that quickly produces a library for the algebra they are working with; second, the tool may be run on a cluster of computers to pre-compute libraries for the most famous classes of algebras, and add them to GAP [8] or a similar system.

Our contributions to the area of isomorphic model elimination are (see Section 3):

- Devise an invariant-based algorithm that can be applied to algebras defined in FOL and containing at least one binary operation.
- Design a small set of invariant properties that in practice have high discriminating power, and yet are inexpensive to compute.
- Use a hash-map to store models partitioned by the invariant-based algorithm to allow fast storage and retrieval of models in the same block.

We apply the proposed partitioning technique to Mace4's isomorphic model filtering programs, and observe orders of magnitude speed-up in its isomorphic model elimination step (see Section 4).

2 Mathematical Background

Algebra is a pair (A, Ω), where A is a set and Ω is a set of operations, that is, functions $f : A^n \to A$ (in this case f is said to be an operation of arity n). Let $A = (D, *_A)$ and $B = (D, *_B)$ be two algebras, each with one binary operation on a finite domain (or universe) D. An isomorphism of these two algebras is a bijective function $f : A \to B$ such that $f(a *_A b) = f(a) *_B f(b)$, for all $a, b \in A$. Two models are said to be isomorphic if there exists an isomorphism between them. The relation *A is isomorphic to B* is clearly an equivalence relation and hence induces a partition of the algebras considered. Only one representative algebra in each block is needed.

The definition of isomorphism can easily be extended to cover algebras with multiple binary operations. Formally, suppose A and B are algebras of type $(2^m, 1^n)$, where m, n are non-negative integers; then we can assume that the binary operations are $(*_1, \ldots, *_m)$ and the unary operations are $(g_1, \ldots, g_n)$. An isomorphism between them is a bijection $f : A \to B$ such that $f(a *_i b) = f(a) *_i f(b)$, for all $a, b \in D$ and every binary operation $*_i$, and for any unary operation g_i, we have $f(g_i(a)) = g_i(f(a))$, for all $a \in D$.

3 Invariant-based Algorithm

Let A and B be two algebras and $f : A \to B$ an isomorphism between them; in addition, suppose $e^2 = e \in A$ is an idempotent. Then $f(ee) = f(e)$ implies that $f(e)f(e) = f(e)$, that is, $f(e)$ under an isomorphism is also an idempotent. As isomorphisms map idempotents onto idempotents, it follows that the number of idempotents in A must be smaller or equal to the number of idempotents on B. Since the inverse of an isomorphism is an isomorphism, A and B must have the same number of idempotents. We call these properties that are preserved by isomorphisms (such as the *number of idempotents*) *invariant properties* or *invariants* for short. These invariant properties are the basis of our proposed algorithm.

Guided by fundamental concepts heavily appearing in different parts of mathematics, we design 10 invariant properties that collectively have high discriminating powers, and yet are inexpensive to compute. For a binary operation in a model with finite domain D, we compute the invariant properties for each domain element x as:

1. The smallest integer n such that $x^n = x^k, n > k >= 1$ where we define x^n to be $(\ldots(x * x) * x) * x) \ldots$ for n x's (*periodicity*).
2. The number of $y \in D$ such that $x = (xy)x$ (*number of inverses*).
3. The number of distinct xy for all $y \in D$ (*size of right ideal*).
4. The number of distinct yx for all $y \in D$ (*size of left ideal*).
5. 1 if $xx = x$, 0 otherwise (*idempotency*).

6. The number of $y \in D$ such that $x(yy) = (yy)x$ (*number of commuting squares*).
7. The number of $y \in D$ such that $x = yy$ (*number of square roots*).
8. The number of $y \in D$ such that $x(xy) = (xx)y$ (*number of square associatizers*).
9. The number of pairs of $y, z \in D$ such that $zy = yz = x$ (*number of symmetries*).
10. The number of $y \in D$ such that there exists pairs of $s, t \in D$ where $x = st$ and $y = ts$ (*number of conjugates*).

Invariant 5 is the idempotent property of the domain element and is preserved by isomorphisms as discussed before. The correctness of invariants in general hinges on the following lemma (folklore). Let F be a FOL formula on the signature of the algebra and M and M' two isomorphic models. It holds that the sets S and S' defined by F in M and M', respectively, are of the same cardinality. This is because the isomorphism induces a bijection between the two sets (*cf.* Theorem 1.1.10 in [12]). In other words, invariants based on solution counting are guaranteed to be correct.

We call the ordered list of invariant properties so calculated the invariant vector of that domain element. Each model with n domain elements will be associated with n invariant vectors. Isomorphic models must have the same set of invariant vectors.

To facilitate comparisons of invariant vectors, we sort the invariant vectors by the lexicographical order of their elements (see the example below for more explanations). It follows that models isomorphic to each other must have the same sorted invariant vectors. If the model has multiple binary operations, then invariant vectors are calculated for each of the binary operations, and all the invariant vectors of the same domain element are concatenated to form a combo invariant vector for that domain element. The combo invariant vectors will then be sorted to yield the final ordered list of invariants.

Often we are not only to compare 2 models for isomorphism, but to extract all non-isomorphic models from a list of models. In that case, we set up a hash map to store the blocks of the models. We use the invariant vectors for each model to send the model quickly to the block (in the hash map) to which it belongs. That is, the keys in this hash map are the invariant vectors, and the values are the blocks of the models. After all models are hashed into the hash map, the blocks stored in the hash map can be processed separately, and possibly in parallel, to extract one representative model from each isomorphism class.

Note that our invariant-based algorithm does not compare models for isomorphism. It only cuts down the size of the problem to improve the speed of existing isomorphism filters such as Mace4's *isofilter*.

As an example to show how invariant vectors are constructed and used, suppose we want to find all non-isomorphic models in a list of 3 quasigroups, A, B, and C, of order 4. Suppose further that their domain is $D = \{0, 1, 2, 3\}$ and their operation tables are given in Table 1.

Table 1 Operation tables of Quasigroups A, B and C.

Model A

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

Model B

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

Model C

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

The 10 invariant properties can easily be calculated for each of the domain elements of these models. Note that while the invariant vector for each domain element is calculated separately, it is not important exactly which domain element gives a particular invariant vector. It is the set of invariant vectors as a whole that matters.

Invariant vectors of Model A

0:	2	1	4	4	1	4	2	4	4	1
1:	3	1	4	4	0	4	2	4	4	1
2:	5	1	4	4	0	4	0	4	4	1
3:	5	1	4	4	0	4	0	4	4	1

Invariant vectors of Model B

0:	2	1	4	4	1	4	2	4	4	1
2:	3	1	4	4	0	4	2	4	4	1
1:	5	1	4	4	0	4	0	4	4	1
3:	5	1	4	4	0	4	0	4	4	1

Invariant vectors of Model C

0:	2	1	4	4	1	4	4	4	4	1
1:	3	1	4	4	0	4	0	4	4	1
2:	3	1	4	4	0	4	0	4	4	1
3:	3	1	4	4	0	4	0	4	4	1

Figure 1 Lexicographically sorted invariant vectors with discerning properties highlighted.

Next we sort the invariant vectors of each model by their elements lexicographically. Invariant vectors of models A and C need no change as they are already in the desired sort order. Invariant vectors of model B will be in sort order by interchanging the invariant vectors of elements 1 and 2, which are the second and third row. The final invariant vectors are shown in the Figure 1. Note that the first column in the tables is the domain element, and the next 10 columns are its invariant properties.

The highlighted numbers in the figure are the discerning invariant properties in the example. All other invariant properties are the same from domain element to domain element. For example, Invariant properties 3, *size of right ideal*, and 4, *size of left ideal*, always equal to the size of the domain D because the operation table of a quasigroup is a Latin square. This highlights the need for multiple invariant properties targeting different areas of algebraic structures to increase their collective discriminating powers. In fact, our algorithm depends more on the orthogonality of the invariants than on the splitting power of any one individual invariant. See Table 2 for the top invariants in different algebras.

It should be easy to see that models A and B have the same sorted invariant vectors, and thus are possibly isomorphic to each other. They are indeed isomorphic to each other because applying the permutation $(1, 2)$ to model B will give model A. However, invariant vectors alone cannot prove that they are isomorphic models. It is also easy to see that the invariant vectors of model C are different from those of the other 2 models, and from this fact alone, we can conclude that model C is not isomorphic to any of A and B.

Finally, for ease of comparison and hashing, we concatenate the sorted invariant vectors into a single string. The string representation of the invariant vectors for the models are:

A, B: 2,1,4,4,1,4,2,4,4,1,3,1,4,4,0,4,2,4,4,1,5,1,4,4,0,4,0,4,4,1,5,1,4,4,0,4,0,4,4,1
C: 2,1,4,4,1,4,4,4,4,1,3,1,4,4,0,4,0,4,4,1,3,1,4,4,0,4,0,4,4,1,3,1,4,4,0,4,0,4,4,1

Since we are to extract all non-isomorphic models from this list of models, we use the string representations of the invariant vectors as the keys for the hash map. Both models A and B will therefore go to the same block in the hash map, but C will go to a different block. Now that all 3 models are deposited in their blocks in the hash map, each block can be processed separately in parallel as we only need to compare models in the same block for isomorphism. This step can be performed by many existing programs such as Mace4's isomorphism filters (see Section 4).

Finally, if the models have multiple binary operations, we compute the unsorted invariant vectors for each binary operation as described above, then concatenate the invariant vectors of the same domain element into one combo invariant vector, sort these combo invariant vectors in lexicographical order, and finally concatenate the sorted invariant vectors into their string format.

It is important to note that the hash map in our algorithm obviates the need to compare invariant vectors among the models during the partitioning process. If we do pairwise comparison of models by their invariant vectors in any step, we would end up with a $O(n^2)$ worst-case scenario.

4 Experimental Results

We have implemented an invariant-based pre-processor to the Mace4's isomorphic models filters. We run the experiments on a 6-core Intel® Core™ i7-9850H CPU computer. We shall show results of tests on 3 algebraic structures, namely, quasigroup, inverse semigroup, and quandle [1]. They are chosen because of their importance in the mathematical world. Quasigroup is the most prominent non-associative algebra, inverse semigroup is probably the most studied associative algebra with a unary operation, and quandles is probably most important algebra with 2 binary operations.

The results show that when the size of the output models is more than just a few hundreds of thousands, the invariant vectors often give an order or two magnitudes of improvements in the speed of the isomorphism elimination process even without running them in parallel. A very desirable feature of our algorithm is that the improvement increases dramatically as the size of the problem grows. Furthermore, Mace4's *isofilter2* is not able to handle input size beyond a few million quasigroups of order 6 (see Table 2), but our invariant-based algorithm can partition the models into smaller blocks of sizes within Mace4's limits.

Table 2 Isomorphism Eliminations.

			Time (s)	
	Order	# of Mace4 Outputs	With Invariants	Without Invariants
Quasigroups	5	10,944	1	1
	6	11,543,040	1,182	N/A
Inverse Semigroups	5	2,151	<1	<1
	6	38,828	3	2
	7	929,923	73	81
	8	28,947,734	2,873	150,703
Quandles	6	1,833	2	1
	7	22,104	6	374
	8	359,859	450	267,463

We show the results of the non-parallel runs to demonstrate the improvements due solely to the invariant vectors. The performance can be improved further if the blocks are processed in parallel. For example, the processing time for the biggest block for quandles of order 8 is only 20 seconds, so if we have enough processors to process all the blocks in parallel, then the processing time can theoretically be cut down close to $24 + 19.937 \approx 44$ seconds from 450 seconds, more than 90% reduction (see Table 3).

Table 3 Isomorphism Eliminations in Parallel.

			Time (s)	
	Order	#Blocks	Generating Invariants	Processing Biggest Block
Quasigroups	6	1,129,129	265	0.0106132
Inverse Semigroups	8	4,582	1,031	2.807
Quandles	8	1,143	24	19.937

One reason for the dramatic improvement in the run-time by our invariant-based algorithm is that the invariant vectors chosen have great discriminating power as shown by the fact that the average number of non-isomorphic models per block is very close to 1 (see Table 4). The top 4 contributing invariants for the highest order of each class are also listed in Table 4.

Table 4 Discriminating Power of Invariant Vectors.

			Non-isomorphic Models		
	Order	#Blocks	Total	Avg per Block	Top 4 Invariants
Quasigroups	5	1,402	1,411	1.01	
	6	1,129,129	1,130,531	1.00	6, 1, 8, 10
Inverse Semigroups	5	52	52	1.00	
	6	208	208	1.00	
	7	908	911	1.02	
	8	4,582	4,637	1.01	9, 3, 2, 1
Quandles	6	66	73	1.11	
	7	250	298	1.19	
	8	1,143	1,581	1.38	8, 3, 6, 10

5 Related Work

The proposed approach falls into the class of *divide-and-conquer* algorithms; most notably Heule et al. [10] recently applied the *cube-and-conquer* approach [9] to solve the Boolean Pythagorean triples problem.

There are a large number of techniques to break symmetries during the search phase [5], such as the Least Number Heuristic (LNH) [18] and the eXtended LNH (XLNH) [2]. The LNH, for example, is a very popular dynamic symmetry breaker implemented in Mace4, FALCON [18], and SEM [19], etc., to help reduce the number of isomorphic models. However, these techniques do not guarantee isomorph-freeness. Systems that try to generate isomorph-free models, such as SEMK [3, 14] and SEMD [11], are either yet to be complete, or are better off allowing some isomorphic models in the outputs for some problem sets. Thus, post-processing tools such as our invariant-based algorithm have an important role in isomorphism elimination as total elimination of isomorphism in the model search phase may not always be the best option.

Invariants are widely used under different guises in many branches of mathematics. For example, in graph theory, node invariants can be used to help detect isomorphic graphs [15]. Interestingly, similar ideas can be seen in Mace4's isomorphism filters. Indeed, Mace4's *isofilter* uses the numbers of occurrences of domain elements in the operation tables as the lone invariant that serves 2 purposes: First is to do quick checks for non-isomorphism, as models having different occurrences of domain elements cannot be isomorphic. Second is to guide the construction of isomorphic functions between potential isomorphic models, as domain elements can only map to domain elements having the same occurrences in the operation tables. This reduces the number of permutations to try in the search of isomorphic functions. However, the lone invariant in *isofilter* would fail miserably if the models are quasigroups for which each domain element would appear the same of times in the operation table. To mitigate this problem, Mace4 provides another isomorphism filter, *isofilter2*, which

transforms the models to their canonical forms based on the same algorithm [14] given by McKay as mentioned above in SEMK. Compared to *isofilter*, *isofilter2* performs much better for quasigroups, but worse on other algebraic structures such as semigroups due to its high overheads in computing canonical forms. Nevertheless, both filters compare every model against the list of non-isomorphic models found so far, and hence their performances degrade rapidly as the number of models increases. Therefore, both filters benefit immensely from the reduced number of models in the blocks created by our invariant-based algorithm.

The *loops* package [16] in GAP [8] uses invariant vectors of 9 invariants in many of its isomorphism-related functions. Like Mace4's *isofilter*, it uses invariant vectors to check for non-isomorphism, and to help guide the construction of isomorphic function between models using sophisticated algorithms that take advantage of other GAP functions. Their invariant vectors work on only one operation table, and exploit heavily specific properties of quasigroups and loops, which may be ineffective in other kinds of algebras. Our invariant-based algorithm targets different aspects of all algebraic structures including quasigroups, semigroups, and more. It also works with multiple binary operations, and does not rely on any built-in functionality of GAP. Moreover, given a list of models to find non-isomorphic models, the *loops* package would compare the invariant vector of every model against those of the list of all non-isomorphic models found so far to get the list of potential isomorphic models. Our hash map-based organization of models eliminates the need to compare invariant vectors repeatedly because all models having the same invariant vectors are already grouped into the same block in the hash map.

6 Future Work and Conclusions

Currently, we only compute invariants based on binary operations, which are by far the most prevalent operations in algebraic structures [4]. However, unary operations are also quite common, and may be even less expensive to manipulate. The discriminating power of the invariant vectors of the model can be enhanced with the addition of invariant vectors based on unary operations, and will be part of our future focus.

The results of our research open a whole new line of research into using invariant properties to eliminate isomorphism in finite model enumeration:

- Identify more invariant properties and the cases for which each of them may be useful.
- Allow dynamic, and preferably automatic, selection of invariant properties to use in any given algebraic structure because different invariants work best for different algebraic structures (see example in Section 3, and also Table 4), so we need to allow dynamic, and preferably automatic, selection of invariant properties.
- Find the best sets of invariant properties to use for various sizes and types of models. A larger set of algebras (usually of higher orders) may need more invariants in the invariant vectors to provide enough discriminating power to separate the models into smaller blocks, but a smaller set of algebras may incur too much overhead in computing the invariant vectors with many invariant properties.

We observe that the invariant-based algorithm is efficient, scalable, and parallelizable. It is also compatible with most, if not all, existing finite model enumerators. The focus of future research will be on finding more good invariant properties, in binary and in unary operations, to be used as partitioning keys, and on adding the capability of dynamic and automatic selection of invariant properties to use.

References

1 João Araújo, David Matos, and João Ramires. Axiomatic library finder. URL: https://axiomaticlibraryfinder.pythonanywhere.com/definitions [cited 15.05.2021].

2 Gilles Audemard and Laurent Henocque. The extended least number heuristic. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning*, pages 427–442, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

3 Thierry Boy de la Tour and Prakash Countcham. An isomorph-free sem-like enumeration of models. *Electronic Notes in Theoretical Computer Science*, 125(2):91–113, 2005. Proceedings of the 5th International Workshop on Strategies in Automated Deduction (Strategies 2004). doi:10.1016/j.entcs.2005.01.003.

4 Stanley Burris and Hanamantagouda P. Sankappanavar. *A course in universal algebra*, volume 78 of *Graduate texts in mathematics*. Springer, 1981.

5 James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 148–159. Morgan Kaufmann, 1996.

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

7 John D. Dixon and Brian Mortimer. *Permutation Groups*. Springer, 1996.

8 The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.1*, 2021. URL: https://www.gap-system.org.

9 Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC, Revised Selected Papers*, volume 7261, pages 50–65. Springer, 2011. doi:10.1007/978-3-642-34188-5_8.

10 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the BooleanPythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing (SAT)*, 2016. doi:10.1007/978-3-319-40970-2_15.

11 Xiangxue Jia and Jian Zhang. A powerful technique to eliminate isomorphism in finite model search. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 318–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

12 David Marker. *Model Theory: An Introduction*. Springer, 2002.

13 William McCune. Mace4 reference manual and guide. Technical Report Technical Memorandum No. 264, Argonne National Laboratory, Argonne, IL, August 2003. URL: https://www.cs.unm.edu/~mccune/prover9/mace4.pdf.

14 Brendan D McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998. doi:10.1006/jagm.1997.0898.

15 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

16 Gábor Nagy and Petr Vojtěchovský. LOOPS, computing with quasigroups and loops in GAP, Version 3.4.1, November 2018. Refereed GAP package. URL: https://gap-packages.github.io/loops/.

17 Neil J. A. Sloane and The OEIS Foundation Inc. The on-line encyclopedia of integer sequences, 2020. URL: http://oeis.org/?language=english.

18 Jian Zhang. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, 17:1–22, August 1996. doi:10.1007/BF00247667.

19 Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In *IJCAI*, pages 298–303, 1995. URL: http://ijcai.org/Proceedings/95-1/Papers/039.pdf.

Constraints (2022) 27:360–379
https://doi.org/10.1007/s10601-022-09336-x

Boosting isomorphic model filtering with invariants

João Araújo[1] · **Choiwah Chow**[2] · **Mikoláš Janota**[3]

Accepted: 16 May 2022 / Published online: 16 June 2022

Abstract
The enumeration of finite models is very important to the working discrete mathematician (algebra, graph theory, etc) and hence the search for effective methods to do this task is a critical goal in discrete computational mathematics. However, it is hindered by the possible existence of many isomorphic models, which usually only add noise. Typically, they are filtered out *a posteriori*, a step that might take a long time just to discard redundant models. This paper proposes a novel approach to split the generated models into mutually non-isomorphic blocks. To do that we use well-designed hand-crafted invariants as well as randomly generated invariants. The blocks are then tackled separately and possibly in parallel. This approach is integrated into Mace4 (the most popular tool among mathematicians) where it shows tremendous speed-ups for a large variety of algebraic structures.

Keywords Computational algebra · Finite model enumeration · Isomorphism · Invariant · Random generation of invariants · Mace4 · Hashing

1 Introduction

To study and get intuition on different types of relational algebras (groups, semigroups, and their ordered versions, quasigroups, fields, rings, MV-algebras, lattices, etc.), mathematicians resort to libraries of all order n models (for small values of n) of the algebra they are interested in. These libraries allow experimentation such as forming and/or testing conjectures etc., to gain insights into the algebras in question. Indeed, GAP [14], the most popular computational algebra system, has many libraries of different algebras (semigroups, quasigroups, etc.), and more are needed. These libraries are so important that the search for them has a long history in mathematics predating for many years the use of computers.

✉ Choiwah Chow
1702603@estudante.uab.pt

João Araújo
jj.araujo@fct.unl.pt

Mikoláš Janota
mikolas.janota@cvut.cz

[1] Universidade Nova de Lisboa, Lisbon, Portugal

[2] Universidade Aberta, Lisbon, Portugal

[3] Czech Technical University in Prague, Prague, Czechia

(See Appendix B of [12]; and for more recent results please see OEIS [28], where many of the sequences are the number of order n non-isomorphic models of a given class of algebras.)

Many of these algebras can be defined in first-order logic (FOL) and there are tools to allow mathematicians to encode their algebras and produce a meaningful library. The problem is that usually the tools, such as Mace4 [22], which can be easily learned and used by mathematicians and hence is very popular among them, generate too many isomorphic models (see Section 2 for the definition of isomorphism) that need to be eliminated [4]. For example, out of the 230,984 models for the implication algebras (see definition in [3]) of order 10 generated by Mace4, only 18 ($\approx 0.0078\%$) of them are pairwise non-isomorphic.

Redundant models may either be eliminated during the search phase or filtered out afterwards. Guaranteeing that search never produces isomorphic models is a hard problem and is rarely seen in modern solvers. This paper, therefore, tackles the second problem, i.e., the removal of redundant models from an already enumerated set.

In the context of finite model enumeration, the complexity of checking whether two models are isomorphic is only part of the problem. Another source of complexity is the large number of models that need to be checked. If every model is checked against all others, then the performance degrades rapidly as the total number of models increases.

If we assign each domain element in a generated model a vector that is invariant under isomorphism (see Section 2) and put all models having the same multiset of invariant vectors into separate blocks, then models across the blocks will not be isomorphic (see Section 6). This splits the problem into substantially smaller sub-problems. Moreover, processing of the blocks can easily be done in parallel as models across blocks cannot be isomorphic. Parallel processing is an important facet of our approach since modern-day computers are more often than not equipped with multiple cores.

Our contributions to the area of isomorphic model elimination are:[1]

1. Devise an invariant-based parallel algorithm that can be applied to algebras defined by first-order logic formulas and containing at least one binary operation or relation (see Section 6).
2. Design a small basic set of invariants that have high discriminating power, and yet are inexpensive to compute (see Section 4).
3. Add randomly generated invariants to the invariant-based algorithm to help discover invariants of high discriminating power (see Section 5).
4. Use a hash-map to store models partitioned by the invariant-based algorithm to allow fast storage and retrieval of models in the same block (see Section 6).

Our goal is to help mathematicians on two levels: first, provide them with a tool on their desktop that quickly produces a library for the algebra they are working on; second, run the tool on a cluster of computers to precompute libraries for the most famous classes of algebras, and add them to GAP [14] or a similar system.

[1] Some preliminary ideas and results have been presented in [2]. This paper adds more invariants including randomly generated invariants and proves their validity. It also reports substantially more experimental results and drills deeper into related work.

2 Definitions and preliminaries

We give a brief overview of the mathematics used in the subsequent sections; we draw mainly from Chapter 2 of [9].

A relational algebra is a triple (D,Σ_F,Σ_R), where D is a set and Σ_F is a set of operations, that is, functions $f : D^n \to D$ and Σ_R is a set of relations, i.e., $R \in \Sigma_R$ is a subset of D^n. The *order* of a relational algebra is the size of its domain D. (Recall that examples of relational algebras are all imaginable algebras, (di)graphs, etc.; in the following, by algebra we mean relational algebra.)

While the concept of isomorphism is ubiquitous to scientific literature, its definition appears under slight variations. Throughout this paper, we rely on the following definition. Let A and B be structures defined on the same signature Σ_F,Σ_R. A function f from A to B is said to be an *isomorphism* if it is a bijection and preserves all operations and relations. This means that if $g \in \Sigma_F$, with the respective interpretations g^A and g^B in A and B, then $f(g^A(a_1, \ldots, a_n)) = g^B(f(a_1), \ldots, f(a_n))$, for all $a_1, \ldots, a_n \in A$. Analogously, f preserves $R \in \Sigma_R$ of arity n, with the respective interpretations R^A and R^B in A and B, if $(a_1,\ldots,a_n) \in R^A$ implies $(f(a_1), \ldots, f(a_n)) \in R^B$, for all $a_1,\ldots,a_n \in A$.

An important property of isomorphisms is that they preserve sets defined by some fixed formula. More precisely, suppose we have two finite relational algebras A and B, on a signature Σ, isomorphic under $f : A \to B$. In addition, suppose we have a set S contained in A^k and definable by a FOL formula Φ in the language of Σ. Then $f(S)$ is precisely the subset of B^k that satisfies Φ (*cf.* Theorem 1.1.10 in [21]).

For example, suppose we have two isomorphic finite algebras: $(A,*)$ and $(B,*)$, with $f : A \to B$ an isomorphism. Suppose also that S is the set $\{(x,y) \in A | (\exists z \in A)\ (x *^A z) *^A y = x *^A (z *^A y)\}$. As S is the set of all elements in A^2 that satisfy a FOL in a language with the function symbol $*$, then $f(S) := \{(f(x),f(y)) | (x,y) \in S\}$ is precisely the set of all pairs $(x,y) \in B^2$ such that $(\exists z \in B)\ (x *^B z) *^B y = x *^B (z *^B y)$.

This idea is usually expressed by saying that sets definable by FOL formulas are invariant (or preserved) under isomorphism. This guarantees that when we split the list of algebras using invariants based on defining formulas, algebras in different blocks are non-isomorphic; algebras inside the same block might be isomorphic or non-isomorphic. Therefore, to discard the redundant algebras we only have to check within each block. This is the ground for our invariants-based algorithm. For future reference, we state these considerations as a proposition.

Proposition 1 Let A and B be algebras of a signature Σ and $f : A \to B$ be an isomorphism from A to B. Then any ordered tuple $(a_1, \ldots, a_m) \in A^m$ satisfies a first-order formula in the common language Σ if and only if the ordered tuple $(f(a_1), \ldots, f(a_m)) \in B^m$ satisfies the same first-order formula in B.

3 Invariants

We define an invariant as a function that accepts an element and a structure whose value is preserved by isomorphism. The motivation for this definition is that if two structures' invariants give different multisets of values, then the structures are *not* isomorphic.

Definition 1 (invariant) Let I be a function that accepts a structure and an element that returns an integer. We say that I is an *invariant* if $I(A,x) = I(B,f(x))$ for all isomorphic structures A and B with isomorphism $f : A \to B$ and $x \in A$.

The key observation is that if invariants are calculated for all elements, then isomorphic structures result in the same multiset of values.

Proposition 2 Let I be an invariant and A, B structures. Define the multisets $M_A = [I(A,x) \mid x \in A]$, and $M_B = [I(B,x) \mid x \in B]$. If A and B are isomorphic then $M_A = M_B$.

Proof Let f be an isomorphism from A to B. There is a bijection between M_A and M_B that maps $v = I(A,x)$ to a unique $v \in M_B$ with $v = I(B,f(x))$.

Example 1 Let the invariant I count the number of times an element is obtained in a given structure, i.e., $I(\langle D, \odot \rangle, x) = |\{d_1, d_2 \in D \mid d_1 \odot d_2 = x\}|$. Consider multiplication and addition modulo 2 on $D = \{0,1\}$, i.e., $A = \langle D,\odot = * \rangle$ $B = \langle D,\odot = +_{\text{mod } 2} \rangle$. We get the corresponding multisets, [3,1] and [2,2], which are different and therefore A and B are *not* isomorphic.

We combine multiple invariants by calculating a vector of values for each element. This is further illustrated by a detailed example in Section 6.

Corollary 1 Let $I_1, \ldots, I_k$ be invariants and A, B structures. Define the multisets of vectors of integers $V_A = \big[(I_1(A,x), \ldots, I_k(A,x)) \mid x \in A\big]$, and $V_B = \big[(I_1(A,x), \ldots, I_k(B,x)) \mid x \in B\big]$. If A and B are isomorphic then $V_A = V_B$.

4 Basic invariants

The goal of this section is to introduce our list of basic invariants. We start by observing that the axioms satisfied by an algebra might render some invariants useless. For example, if the algebra is a group, then the invariant that counts the number of inverses of a domain element (see **U3**) would be useless for distinguishing non-isomorphic models, because there is exactly one inverse for each domain element. Thus, we need multiple invariants with deep algebraic meaning and large discriminating power in order to target as many different algebraic properties/classes as possible. On the other hand, we should choose properties that are inexpensive to compute and not very many as that could slow down the computation.

Our choices of invariants are based on concepts ubiquitous in various kinds of algebras. For example, one of our invariants (4.2) is based on the fact that idempotents appear in many algebras; in particular, it is well-known that every finite semigroup has at least one idempotent and hence this invariant is useful for a wide range of algebras, especially those that have a semigroup reduct.

As observed above, the overwhelming majority of the most popular algebras are defined using operations of arity at most 2 (see page 26 of [9]), so we design most of our invariants around binary operations. We have 10 invariants from binary operations, 4 from binary relations, 4 from unary operations, and 1 from ternary operations to target different common algebraic structures. Together they have high discriminating powers, and yet are easy and inexpensive to compute.

In the following discussions on invariants, the domain of the algebra is denoted by $D = \{0, \ldots, n-1\}$.

4.1 Invariants from unary operations

For each unary operation g in the algebra, we compute the invariants for each element $x \in D$:

U1 1 if $g(x) = x$, 0 otherwise (fixed point);
U2 1 if $g(x) \neq x$ and $g(g(x)) = x$, 0 otherwise (transposition);
U3 The number of $y \in D$ such that $g(y) = x$ (size of the inverse image);
U4 The number of $y \in D$ such that $g(g(y)) = x$ (size of the inverse image under g^2).

The correctness of **U1** and **U2** as invariants follows readily from Proposition 1. The correctness **U3** and **U4** also follows from Proposition 1, with the note that isomorphism is a bijection. Let us illustrate the proof on the invariant **U3**.

Proposition 3 The function **U3** is an invariant (Definition 1).

Proof Let f be an isomorphism of structures A and B with a unary function g. We need to prove that $\mathbf{U3}(A,d) = \mathbf{U3}(B,f(d))$ for any $d \in A$. Construct the sets of solutions of $g(y) = x$ for both of considered structures, i.e., define $S_C = \{(x, y) \in C \times C \mid g(y) = x\}$, for $C \in \{A,B\}$. For an element $d \in A$ consider the set $S_A^d = \{(d_1, d_2) \in S_A \mid d_1 = d\}$. Analogously, consider the set $S_B^{f(d)} = \{(d_1, d_2) \in S_B \mid d_1 = f(d)\}$. From Proposition 1, f is a bijection between S_A and S_B, i.e. $(d_1,d_2) \in S_A$ iff $(f(d_1),f(d_2)) \in S_B$. But therefore also f is a bijection between S_A^d and $S_B^{f(d)}$. This means that $|S_A^d| = |S_B^{f(d)}|$, i.e., the values of **U3** for the element d and $f(d)$ are the same.

Similar arguments can easily be used to directly prove the correctness of the other invariants.

4.2 Invariants from binary operations

For each domain element $x \in D$, we compute the following invariants for each binary operation in the algebra:

B1 The smallest integer n such that $x^n = x^k, n > k >= 1$ where we define x^n to be $(\ldots(x * x) * x) * x)\ldots$ for n x's (*periodicity*).
B2 The number of $y \in D$ such that $x = (xy)x$ (*number of inverses*).
B3 The number of distinct xy for all $y \in D$ (*size of right ideal*).
B4 The number of distinct yx for all $y \in D$ (*size of left ideal*).
B5 1 if $xx = x$, 0 otherwise (*idempotency*).
B6 The number of $y \in D$ such that $x(yy) = (yy)x$ (*number of commuting squares*).
B7 The number of $y \in D$ such that $x = yy$ (*number of square roots*).
B8 The number of $y \in D$ such that $x(xy) = (xx)y$ (*number of square associatizers*).
B9 The number of pairs of $y,z \in D$ such that $zy = yz = x$ (*number of commuting pairs*).
B10 The number of $y \in D$ such that there exist pairs of $s,t \in D$ where $x = st$ and $y = ts$ (*number of conjugates*).

4.3 Invariants from binary relations

For each domain element $x \in D$, the following invariants are calculated for each binary relation R:

R1 The number of distinct y such that $R(x,y)$.
R2 The number of distinct y such that $R(y,x)$.
R3 1 if $R(x,x)$, 0 otherwise. (reflexivity)
R4 The number of y such that $R(x,y)$ & $R(y,x)$.

4.4 Invariants from ternary operations

Ternary operations are very rare. Indeed, no ternary operation exists in the definition in any of the 158 algebras listed in the ALF database [3], although a few of them come from the Skolemization of binary operations. Moreover, calculations involving ternary operations are often very expensive as deeply nested loops are involved. Thus, only one simple invariant would be included in the algorithm. For each domain element $x \in D$, we compute one invariant for each ternary operation:

T1 The number of times x appears in the ternary operation table. (frequency)

We call the hand-crafted invariants listed above the *basic invariants* to differentiate them from the randomly generated invariants which will be discussed in Section 5. It should be simple to see that the validity of these basic invariants follows from Proposition 4.

5 Random invariants

As discussed in Section 4, we need different invariants to target different algebraic structures. The basic invariants are inspired by our knowledge of the most popular algebras, however, there are many other less common algebraic structures and new ones are continually appearing.

Therefore, we need a general way (adaptable to each class of algebras) of generating invariants with good discriminating power. A practical solution to this problem is to generate a large set of invariants with a random number of operations and a random number of variables, and then automatically discover the best subset to use.

5.1 Generation of random invariants

A first-order formula can be represented by an expression tree with operations in the internal nodes and variables in the leaves. For example, Fig. 1 shows the tree representation of the first-order formula

$$x = (x * y) + x' \tag{1}$$

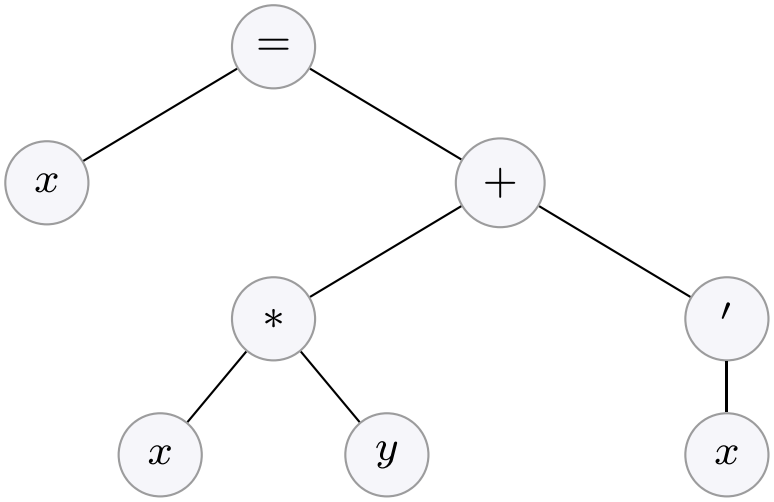


Fig. 1 Expression tree for $x = (x * y) + x'$

It is simple to randomly generate such an expression tree, as shown in Algorithm 1. For simplicity, the root is set to be an operation that evaluates to true or false, and it always has two children. It is the only node that can be assigned the equality relation. It may also be a binary relation if there is one in the algebra.

Algorithm 1: Generation of Random First-order Formula

input : A list of binary/unary operations/relations, max depth of tree, list of variables
output : An expression tree representing a first-order formula

```
Function BuildNode (level) begin
    /*recursively build a node                                   */
    /*maxLevel is maximum depth of tree allowed                  */
    if level = maxLevel then
        nodeType ← leaf
    else
        nodeType ← random pick a leaf, unary, or binary operation
    create newNode
    if nodeType = leaf then
        newNode.value ← randomly pick a variable
        return newNode
    newNode.op ← randomly pick a unary or binary operation
    newNode.left ← buildNode (level +1)
    if newNode.op is a binary operation then
        newNode.right ← buildNode (level +1)
    return newNode
begin
    create root node R
    R.op ← randomly pick the equality operation, or one of the binary relations (if
      exists)
    R.left ← BuildNode (1)
    R.right ← BuildNode (1)
    return R
```

Proposition 4 If we fix a variable in the first-order formula generated by Algorithm 1 as the base variable, then the number of solutions will be an invariant.

Proof Correctness is shown as in Proposition 3.

We may choose any of the variables in the randomly generated first-order formula as the base variable for the invariant. For example, if we choose x as the base variable for the first-order formula (1), then the invariant would read: The number of $y \in D$ such that $x = (x * y) + x'$, which is a valid invariant by the foregoing argument.

5.2 Quality measure of random invariants

After a large set of invariants are generated, a small subset will be selected based on its ability to reduce the work of the next step in filtering out isomorphic models. For a block with m models, the worst-case scenario requires comparing every pair of models for isomorphism. There are $m(m-1)/2$ such comparisons in total. Based on this observation, we measure the quality of invariants by a score as follows: Suppose a set of invariants are applied to a set of models, resulting in n blocks of models in which models in different blocks have different sets of invariant vectors. Let S_i, where $i \in [n]$, denote these n blocks of models. The score for this set of invariants is computed as

$$\sum_{i \in \{1..n\}} |S_j|(|S_j| - 1) \tag{2}$$

The goal is to find the set of invariants having the minimum score over all possible combinations of randomly generated invariants in conjunction with the basic invariants.

5.3 Selecting random invariants

We are not aware of any tractable algorithm for finding the optimal subset from a large set of random invariants according to the quality measure stated above.[2] A feasible solution is to apply a greedy algorithm (see page 282 of [26] for the general discussions) to a small sample of the models to find an approximate optimal subset. In practice, it is sufficient to use a sample size of 0.1–0.2%, or a thousand, whichever is larger, of the original set of models (see Section 7 for discussions). The algorithm is detailed in Algorithm 2. The idea of the algorithm is to start with the basic invariants, then add the random invariants and calculate the scores one-by-one, keeping the random invariant only if it gives a better score. Then repeat the process of adding random invariants, calculating the scores, and picking the best random invariant that minimizes the score until it cannot be further improved, or a preset maximum number of trials is reached. This subset of random invariants, which may or may not be truly optimal, will then be used together with the basic invariants for the next step.

[2] We conjecture that the problem is NP-hard; it resembles K-means clustering, which is NP-hard [1].

Algorithm 2: Selecting Random Invariants with Greedy Algorithm

```
   input  : A set of random invariants R, a set of models M, and maximum trials T
   output : A set K ⊆ R of random invariants with |K| < T
 1 K ← ∅
 2 bestScore ← ∞
 3 done ← false
 4 while ¬done ∧ |K| < T do
 5 |   done ← true
 6 |   a ← ∅
 7 |   foreach r ∈ R ∖ K do
 8 |   |   trialScore ← score of K ∪ {r} on M according to equation (2)
 9 |   |   if trialScore < bestScore then
10 |   |   |   bestScore ← trialScore
11 |   |   |   a ← r
12 |   if a ≠ ∅ then
13 |   |   done ← false
14 |   |   K ← K ∪ {a}
15 return K
```

Note that the main purpose of adding randomly generated invariants is not to divide the models into more blocks in all cases, but to increase the robustness of the algorithm by the automatic discovery of important invariants in some cases.

6 The invariant algorithm

Recall from Section 3 that an invariant gives us a number for each element in a structure. Further, multiple invariants are combined into a vector of numbers for each element. We refer to this vector as *invariant vector*. From Corollary 1, if multisets of invariant vectors differ for two structures, these structures are *not* isomorphic.

In our algorithm, multiset of invariant vectors, is represented as a lexicographically sorted vector of invariant vectors. This means that the original multisets are equal if and only if these sorted vectors are equal. This is briefly demonstrated by the following example.

Example 2 Consider structures A and B with domain elements $\{0,1,2\}$ and some invariants I_1,I_2 giving the multisets $V_A = [(0,1),(3,2),(0,1)]$ and $V_B = [(3,3),(0,1),(0,1)]$. Sorting them lexicographically gives two vectors of invariant vectors revealing that the multisets are not equal: $((0,1),(0,1),(3,2))$ and $((0,1),(0,1),(3,3))$, i.e., A and B are not isomorphic.

The goal is not only to compare two models for isomorphism but to extract all non-isomorphic models from a list of models. In this case, a hash map is set up to store blocks of the models in which models from different blocks are guaranteed not to be isomorphic. We use the sorted invariant vectors for each model to send the model efficiently to the block (in the hash map) to which it belongs as summarized in Algorithm 3. That is, the keys in this hash map are the invariant vectors, and the values are the blocks of the models. After all models are hashed into the hash map, the blocks stored in the hash map can be processed separately, and possibly in parallel, to extract one representative model from each isomorphism class.

Algorithm 3: Hashing Models

```
input  : A list of models M
output : A Hash-map of blocks of models having the same sets of invariant vectors
1 begin
2     H ← an empty hash-map
3     foreach m ∈ M do
4         v ← sorted invariant vectors of m
5         if v is not already a key in H then
6             H[v] ← {m}
7         else
8             H[v] ← H[v] ∪ {m}
9     return H
```

As a simplified example to show how invariants are used, consider the inverse semigroups of order 2. It is defined by an unary function "′" (the inverse function) and a binary function "*" (the semigroup function). Mace4 generates 4 models from the FOL definition of inverse semigroups (see Table 1).

To make the example easy to follow, we use only 4 invariants: **U1**, **B7**, **B8**, and **B9**. These 4 invariants are calculated for each domain element in each model and the results are put into a vector of length 4. So, there is a set of two invariant vectors for each model. Isomorphic models will have identical sets of invariant vectors. While not absolutely necessary, we sort each set of the invariant vectors lexicographically for ease of comparisons, as shown in Fig. 2. Note that the leftmost column in each figure holds the domain elements, and is not part of the invariant vector.

Next, the sorted invariant vectors for each model are concatenated into a single combo vector. For example, the combo invariant vector for model A is "1,1,2,1,1,1,2,3", and that of B is "1,0,2,2,1,2,2,2". By comparing the final combo vectors, we see that the models A and C have the same set of invariant vectors and hence could be isomorphic, but we cannot tell that from the invariant vectors alone. Same for the pair of models B and D. Next, we put these models

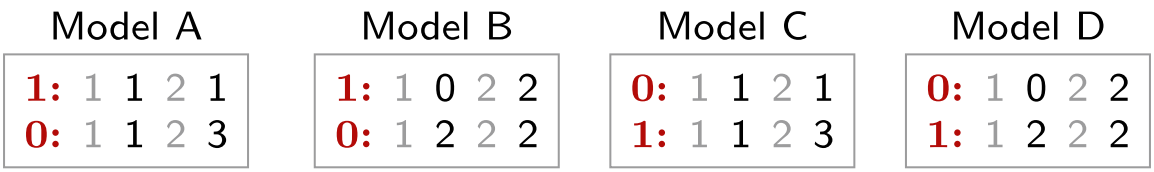


Fig. 2 Lexicographically sorted invariant vectors with discerning properties highlighted

Table 1 Operation tables of Inverse Semigroups A, B, C, and D

Model A

$*_A$	0	1
0	0	0
1	0	1

$'_A$	0	1
	0	1

Model B

$*_B$	0	1
0	0	1
1	1	0

$'_B$	0	1
	0	1

Model C

$*_C$	0	1
0	0	1
1	1	1

$'_C$	0	1
	0	1

Model D

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

$'_D$	0	1
	0	1

into a hash-map using their sorted invariant vectors (or the combo vectors) as keys (see Algorithm 3). Models A and C will therefore go to one block in the hash-map, and models B and D will go to another. Now, it is not necessary to compare models across the blocks for isomorphism because models from different blocks have different sets of invariant vectors. Finally, models within each block are compared for isomorphism separately from other blocks, possibly in parallel. The resulting non-isomorphic models are simply put into the same set as no processing is needed to combine them. It turns out that in this case, the models in each of the two blocks are isomorphic. So there are only two non-isomorphic inverse semigroups of order 2.

To add random invariants to the algorithm, a preprocessing step is added to select an optimal subset of random invariants before the normal process. As described in Section 5.3, we construct a list of random invariants, calculate basic invariants and the random invariants on a small sample of the input models. Then the greedy algorithm is applied to find an optimal subset of random invariants for further processing (see Section 5.3). Finally, proceed to normal processing with the basic invariants and the optimal random invariants together.

Note that the invariant-based algorithm does not compare models for isomorphism. It only cuts down the size of the problem to improve the performance of existing isomorphism filters such as Mace4's *isofilter*.

Invariants have the potential to cope with the increasing size of the order of the algebra very well as illustrated in the following example. Suppose an invariant with extremely low discriminating power gives only 2 values, 0 and 1. However, when applied to the models of an algebra of order 2, it could actually give 4 possible invariant vectors: [0,0], [0,1], [1,0], [1,1]. It is easy to generalize this observation: applying an invariant that gives at most m values to the models of an algebra of order n could result in a maximum of m^n distinct invariant vectors. Furthermore, if k invariants give $\{m_1, m_2, \ldots, m_k\}$ values, then the maximum number of invariant vectors would be

$$\prod_{i=1}^{k} m_i^n \tag{3}$$

From the above analysis on the intricate interactions between invariants, we can make two more important observations:

1. Combining invariants of low discriminating powers could give an invariant vector of surprisingly high discriminating power if they are targeting different areas of the algebraic structures.
2. In general, the number of non-isomorphic models increases rapidly as the order of the algebra increases, but so does the maximum number of possible invariant vectors. This helps invariants to retain their discriminating powers to some extent as the order of the algebra increases. This explains why the invariant-based algorithm is very scalable.

7 Experimental results

We have implemented an invariant-based preprocessor to Mace4's isomorphic models filters. We run experiments on an Intel®;Core™ i7-9850H@2.60GHz x 12 computer, with 32 Gb RAM installed.

The ALF database [3] contains a collection of 158 algebras of high interest to the research community of algebra. Their definitions are conveniently given in first-order

Table 2 Isomorphism Filtering, w/ vs. w/o Invariants

	#Mace4 Outputs	Invariant Calc. Time (s)	Total Runtime (s)	
			With Invariants	Without Invariants
Shortest 10 Algebras	600	0.2	0.5	0.1
Longest 10 Algebras	9,239,818	430	1,982	87,591
All 151 Algebras	33,643,548	1,500	5,030	95,952
2 Isofilter Failed Algebras	4,075,054	208	727.3	N/A

formulas that Mace4 can directly process. We use Mace4 to generate models for each algebra of the highest possible order that it can complete within 2 minutes. Mace4 is not able to generate models for 5 of them within that time limit, and they are excluded from the tests. The excluded algebras are: #112 Kleene algebra, #113 Concurrent Kleene algebra, #114 Omega algebra, #137 Steiner quasigroup, and #138 Steiner loop. In addition, Mace4's *isofilter* is not able to handle two of the largest algebras (#8 BL-algebras and #56 Linear Heyting algebras), each has between 1 to 3 million models. These two algebras are also excluded from most of the statistics of the experimental results. So, we end up with 151 algebras in many of our analyses.

When random invariants are used in the experiment, the number of randomly generated invariants is 50, but at most 20 of the best of them will be used. The maximum depth of the expression tree (see Section 5.1) is 4, and the maximum number of variables in it is 3. In this section, random invariants are used unless otherwise specified.

Since we run a large comprehensive set of test cases for comparison, the size of each test case is necessarily limited (uniformly and systematically to make comparisons of results meaningful) by the computing resources available. However, even for the small model sizes used in our experiments, the addition of the invariant-based algorithm improves the overall speed by an order of magnitude, without using parallel processing (see Table 2).

The overheads of calculating invariants are observed to be on average about 20 to 30% of the total run time in our experimental setting (see the third column in Table 2). However, invariants improve the speed by orders of magnitudes for big algebras. For example, for the longest (in terms of runtime) 10 algebras in our experiment, the invariant-based algorithm improves the overall speed by over 50 times (see Table 2). In fact, a very desirable feature of the invariant-based algorithm is that the improvement increases dramatically as the size of the set of models grows. Granted, for algebras with short runtime, the use of invariants may not pay off. But for those cases, the degradation is really insignificant (see Fig. 3). Thus, in general, there is no need to have special logic to decide when not to use invariants.

Furthermore, Mace4's *isofilter* is not able to handle two of the largest data sets, but our invariant-based algorithm can partition these models into smaller blocks to fit in Mace4's limits (see Table 2).

The performance of the invariant-based algorithm relies heavily on the discriminating power of its invariants. The best possible case is that only 1 non-isomorphic model is in every block, in which case, only $m-1$ comparisons of models are needed to eliminate all isomorphic models from a block of m models. Our invariants are quite powerful as evidenced by the fact that the average number of non-isomorphic models per block is very close to 1 for the 151 algebras in the experiment (see Table 3).

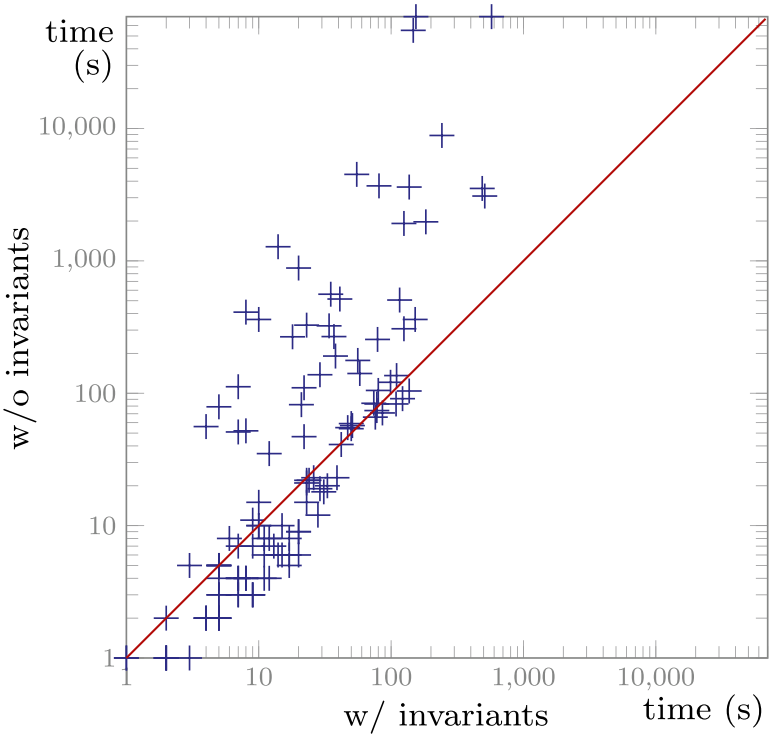


Fig. 3 Runtimes: w/ vs. w/o Invariants (151 ALF Algebras)

Table 3 Discriminating Power (153 ALF Algebras)

Percentile	Avg #Non-isomorphic Models per Block	
	w/ Random Invariants	w/o Random Invariants
95th	1.346	2.677
80th	1.036	1.179
60th	1.003	1.018
40th	1.000	1.005

7.1 Basic invariants vs. basic invariants + random invariants

As shown in Table 3, the hand-crafted basic invariants have very good discriminating power (see the last column in the table). Nevertheless, the addition of random invariants improves the discriminating powers (see the middle column of the table). This increase in discriminating power comes with a small overhead in processing time as the number of random invariants is quite small, usually just a few. For example, 6 or fewer random invariants are used in about 90% of the algebras (see Fig. 4). For the case when the basic invariants are already doing a very good job, the addition of random invariants may not pay off. But the degradation is minimal because the job would finish fast when the discriminating powers of the invariants are high (See the scatter plot Fig. 5). Therefore, there is no need for special logic to decide when not to use random invariants. In our experiment, the overall run time for all 151 algebras is reduced when random invariants are added, with most of the improvements coming from the top 3 algebras, which are among the algebras that take the longest to finish (see Table 4 and Fig. 5).

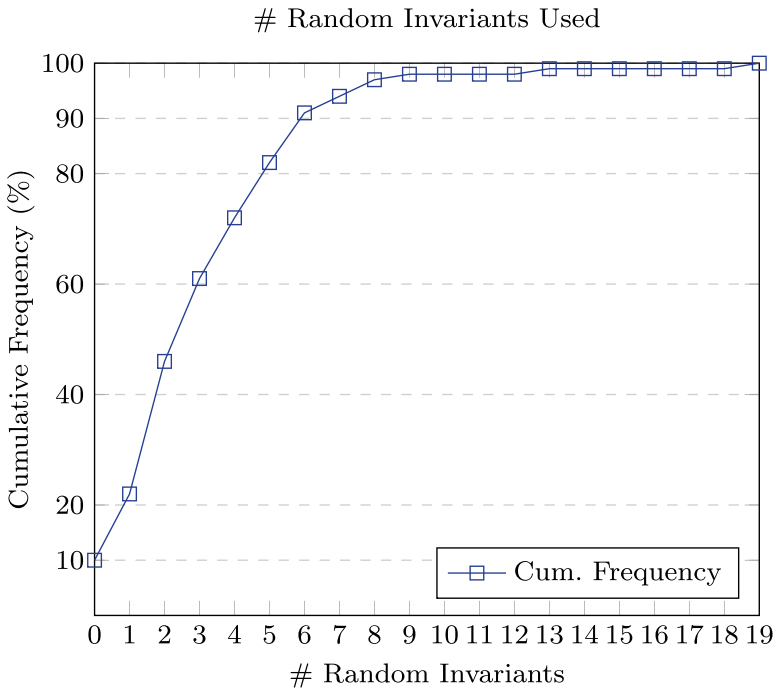


Fig. 4 # of Random Invariants (153 ALF Algebras)

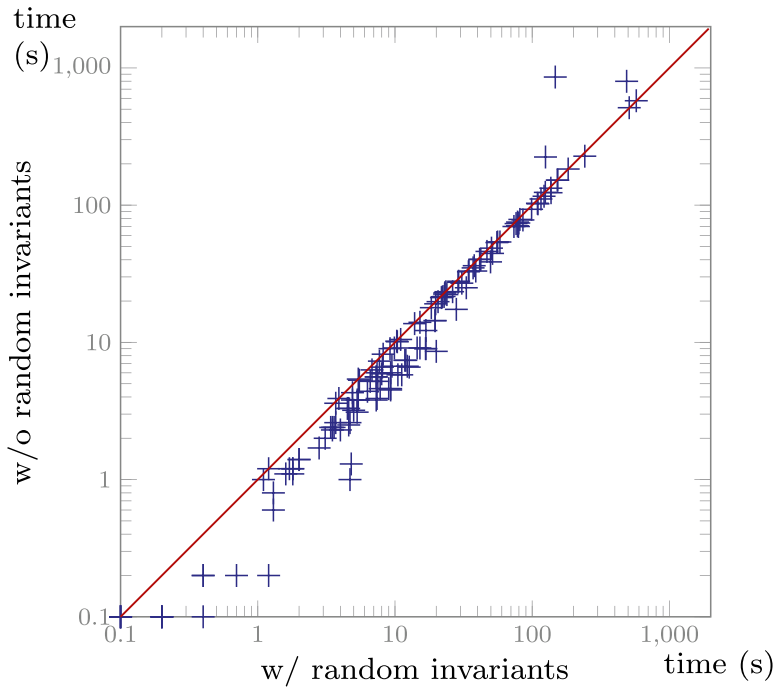


Fig. 5 Runtimes: w/ vs. w/o Random Invariants

7.2 Basic invariants vs. random invariants

An interesting question is whether random invariants alone is the fastest way of filtering out isomorphic models. It is conceivable that given enough number of random invariants, all the basic invariants will automatically be covered in it. However, under our experimental setup (use at most 20 best invariants out of 50 random invariants), the basic invariants perform slightly better than the random invariants in general (see Fig. 6).

Table 4 Isomorphism Filtering Time, w/ and w/o Random Invariants

	Total Time (s)	
	With Random Invariants	Without Random Invariants
Top 3 Improved Algebras	760	1,882
All 151 Algebras	5,758	6,525

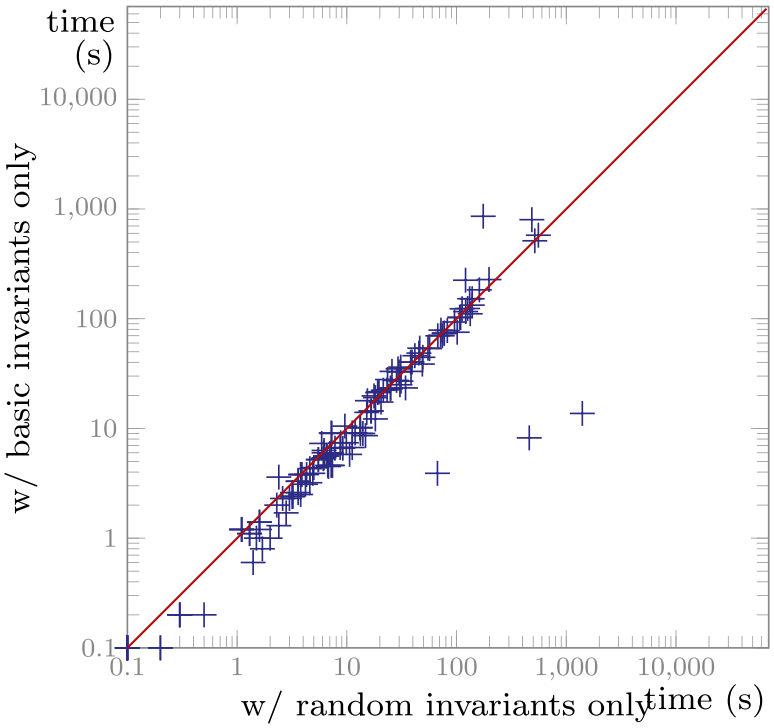


Fig. 6 Runtimes: w/ only Random Invariants vs. w/ only Basic Invariants

7.3 Larger data sets

As remarked earlier, to cover a large variety of algebras, we have to limit the order of each algebra in the experiments. Small datasets do not adequately show the true advantage of the invariant-based algorithm. Here we present three examples in which we go two orders higher in each of the algebras, giving us test datasets of over a hundred million models. The first algebra (#88 Quasi-MV-algebra) is defined by one binary operation and one relation, the second one (#15 Brouwerian semilattices) by two binary operations, and the third one (#4 BCK-join-semilattice) by two binary operations and 1 relation. As shown in Table 5, at the baseline when the order of the algebra is small, the invariant algorithm slows down the process very slightly. However, as the order of the algebra goes higher, the invariant-based algorithm improves the speed by orders of magnitudes. In some cases, Mace4 is not able to handle a large number of models. In all the tests, we also observe that the discriminatory powers of the invariants hold up quite well in large datasets (see Table 6).

7.4 Parallel processing

The invariant-based algorithm is very scalable since the data are divided into blocks that can be processed independently as long as resources are available. In this experiment, we

Table 5 Isomorphism Filtering, w/ vs. w/o Invariants for Higher Orders

			Total Runtime (s)	
	Order	#Mace4 Outputs	With Invariants	Without Invariants
#88 Quasi-MV-algebra	7	10,902	1.6	0.7
	8	4,793,924	558	30,701
	9	29,799,618	3,666	N/A[a]
#15 Brouwerian semilattices	6	47,349	12	4
	7	2,247,564	440	1,964
	8	146,875,177	40,017	N/A
#4 BCK-join-semilattice	9	122,754	23	15
	10	1,175,784	305	532
	11	12,307,002	1,213	54,524

[a] Isofilter fails after processing 4.5 million (15% of all) models in 11.5 hours

Table 6 Discriminating power of invariants for higher orders

			Non-isomorphic Models	
	Order	#Blocks	Total	Avg per Block
#88 Quasi-MV-algebra	7	567	477	1.19
	8	153,163	55,544	2.76
	9	264,972	141,750	1.87
#15 Brouwerian semilattices	6	745	745	1.00
	7	8,272	8,272	1.00
	8	115,801	114,943	1.01
#4 BCK-join-semilattice	9	26	26	1.00
	10	47	47	1.00
	11	82	82	1.00

apply parallel processing to the top 3 algebras with the longest runtimes (close to 500s or more). We run each of them with 5 parallel threads and see about a 50 - 60% reduction in run times (see Table 7). The results would be even better if more resources are available as a large number of blocks are available in these cases.

8 Related work

Classes of algebras can often be defined in first-order formulas. For these algebras, a finite model finder, such as Mace4 [22], SEM [30], and FALCON [29], FMSET [7], etc., can work on finding all their models. A well-known issue with this approach is that first-order formulas introduce symmetries into the problem, which leads to the generation of a huge number of isomorphic models in the outputs [27]. These isomorphic models can either be suppressed in the search phase or be removed in a postprocessing step after the models are generated.

Past work in enumerating non-isomorphic finite models has focused on not generating isomorphic models in the search phase. Symmetry breaking is thus a central focus of their research

Table 7 Isomorphism Filtering, Serial vs. Parallel

	Order	#Blocks	Runtime (s)	
			Serial	Parallel
#8 BL-algebras	5	735,820	574	254
#20 Commutative lattice-ordered monoids	7	15,499	510	240
#145 Digroup	12	17	488	177

[10, 11, 27]. An excellent example of a simple, powerful, and general algorithm to break symmetry *dynamically* is the least number heuristic (LNH) [4, 29, 30], which picks the smallest one not used so far when a new domain element is to be selected during the search. This is implemented in many solvers such as Mace4, SEM, FMSET, and FALCON. Another symmetry breaker, the eXtended least number heuristic (XLNH) [4, 5], is based on similar ideas as the LNH, but could give better performance if there is at least one unary operation in the FOL clauses that define the model. It is also implemented in many finite model enumerators such as SEM.

The underlying idea of LNH can also be applied to break symmetries *statically*. This is necessary for approaches where we do not wish to modify the underlying solver. This is the case for finite model finders based on SAT solvers [10, 17, 27]. The issue is the overhead of encoding LNH in conjunctive normal form (CNF) as well as its complex interaction with the SAT solver. Originally, LNH was only encoded for constants [10]. Later, with additional effort, it was shown that other terms can also be considered [27].

The addition of symmetry-breaking input clauses could be useful in steering the searcher away from the needless exploration of sub-search space [11]. For example, it is well-known that a finite semigroup has at least one idempotent element, so we may add the clause $0 * 0 = 0$ to the list of input clauses to cut off the search of the branch $0 * 0 = 1$, etc. However, this kind of symmetry breaking often requires deep knowledge of the algebra in question, which may not be available when the algebra is first studied.

Most importantly, these symmetry-breaking techniques do not guarantee isomorph-freeness. While not generating isomorphic models would be ideal, but to guarantee that isomorphic models are not produced in the search phase is a hard problem that few modern-day solvers attempt to do. Systems that do try to do so, such as SEMK [8, 23] and SEMD [18], are either yet-to-be-completed or are better off allowing some isomorphic models in the outputs for some cases for better efficiency.

When isomorphic models are not totally suppressed in the search phase, they need to be removed in the post-processing steps. Many of them use a limited number of invariants to help speed up the process. Mace4, for example, has a program, *isofilter*, to filter out isomorphic models that it generates in the search phase. It calculates one invariant, frequency of occurrence of domain element, that is, the number of times a domain element appears in the operation tables. It uses this invariant to help separate non-isomorphic models and to help guide the construction of isomorphic functions between potentially isomorphic models. Needless to say, the discriminating power of one single invariant is limited. Indeed, it fails miserably when the operation table is a Latin square, which is the case for quasigroups. To alleviate this issue, Mace4 has another program, *isofilter2*, which does not try to construct isomorphic functions between models, but to convert models to their canonical forms based on the same algorithm [23] used by SEMK. *Isofilter2* works very well with quasigroups, but the overhead in computing the canonical forms of the models is so high that it becomes slower than *isofilter* for many algebraic structures such as semigroups.

The loops package [25] in GAP [14] is not a finite model enumerator but provides a stand-alone function to extract non-isomorphic models from a list of quasigroups. It uses 9 invariants, some of which are expensive to calculate, to help separate non-isomorphic models, and to guide the construction of isomorphic functions between models having the same invariant vectors. These 9 invariants exploit the specific properties of the quasigroup, and may not be effective for other algebraic structures. On the other hand, our invariants target many different areas in the common algebraic structures. Furthermore, their invariant vectors are limited to one binary operation table. Our invariant vectors can be constructed from multiple unary, binary, and ternary operation tables.

Invariants are sometimes incorporated into the finite-model enumerating algorithm for specific algebras. These invariants are sometimes very simple and easy to compute, such as those in the algorithm for enumerating quandles [13]. But others may be very complicated, not easy to implement, and not cheap to compute as in the case of the algorithm for enumerating inverse semigroups [20] using the constraint solver, Minion [15]. Furthermore, the number of invariants used in these cases is usually very small, often two or fewer. Recall expression (3) on page 33 which shows that the number of invariants could increase the discriminating power drastically. Our invariants are cheap to calculate, are applicable to more algebraic structures, and are high in number to provide more opportunities for separating non-isomorphic models. Moreover, they can easily be incorporated into any finite model enumerator.

Neither Mace4's nor loops' isomorphic model filters make use of the hash table to store the models so that non-isomorphic models will never be compared once they are separated by their invariant vectors. This introduces some inefficiencies in their algorithms. Thus, both could benefit immensely from the reduced number of models in the blocks created by our invariant-based algorithm as a preprocessing step.

Another important feature in the invariant-based algorithm is randomization. Using randomization in the search phase in Boolean Satisfiability (SAT) and Constrained Programming (CSP) algorithms is a tried and tested technique [6, 16, 19]. This strategy is built into many SAT solvers such as Chaff [24]. However, using randomly generated invariants to help separate non-isomorphic models in the finite model enumeration is a novel idea. It helps solve the hardest problems in filtering isomorphic models as shown in our experiments, and consequently, increases the robustness of the invariant-based algorithm.

9 Future work and conclusions

We has shown that the use of basic invariants improves the performance of isomorphic model filtering, sometimes by orders of magnitudes in the case of large data sets, across a wide range of algebras under active research. We have shown further that addition of random invariants not only speed up the filtering of isomorphic models, but also help make sure important invariants are included.

As pointed out in Section 7, the efficiency of the invariant-based algorithm relies heavily on the discriminating powers of both the hand-crafted and the randomly generated invariants. Future work will therefore concentrate on finding powerful invariants to target common algebraic structures, and to find the best parameters to generate optimal random invariants. For example, what depth and breadth of the expression tree would be most cost-effective in generating random invariants? What is the best range of ratios of binary operations to unary operations in a random invariant? What is the best size of the sample to use in finding the optimal set of random invariants?

In summary, we present in this paper an algorithm that uses invariants both as discriminators and as hash keys to partition a set of models into blocks, in which no models across blocks are isomorphic. The blocks are hashed into a hasp map so that they will not be processed together. Included in the algorithm is the novel idea of using randomly generated invariants to supplement hand-crafted invariants to make the algorithm more robust. We show that the invariant-based algorithm is simple, efficient, scalable, and parallelizable. It is also compatible with most, if not all, existing finite model enumerators. It can be used as a stand-alone preprocessor to split models into blocks to feed into isomorphic model filters, or it can be directly incorporated into them. Future research will concentrate on finding powerful invariants in different areas of algebraic structures, and on the automatic discovery of optimal random invariants.

Funding *João Araújo*: The results were supported by the 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.

Mikoláš Janota: The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

1. Aloise, D., Deshpande, A., Hansen, P., & Popat, P. (2009). Np-hardness of euclidean sum-of-squares clustering. *Machine Learning*, *75*(2), 245–248. https://doi.org/10.1007/s10994-009-5103-0.
2. Araújo, J., Chow, C., & Janota, M. (2021). Filtering isomorphic models by invariants. In L.D. Michel (Ed.) *27th International conference on principles and practice of constraint programming (CP 2021), Leibniz international proceedings in informatics (LIPIcs)*. https://doi.org/10.4230/LIPIcs.CP.2021.4, https://drops.dagstuhl.de/opus/volltexte/2021/15295, (Vol. 210 pp. 4:1–4:9). Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
3. Araújo, J., Matos, D., & Ramires, J. Axiomatic library finder (database). https://axiomaticlibraryfinder.pythonanywhere.com.
4. Audemard, G., Benhamou, B., & Henocque, L. (2006). Predicting and detecting symmetries in FOL finite model search. *Journal of Automated Reasoning*, *36*(3), 177–212. https://doi.org/10.1007/s10817-006-9040-3.
5. Audemard, G., & Henocque, L. (2001). The eXtended least number heuristic. In R. Goré, A. Leitsch, & T. Nipkow (Eds.) *Automated reasoning, first international joint conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, proceedings, lecture notes in computer science*. https://doi.org/10.1007/3-540-45744-5/_35, (Vol. 2083 pp. 427–442). Berlin: Springer.
6. Baptista, L., & Silva, J.P.M. (2000). Using randomization and learning to solve hard real-world instances of satisfiability. In R. Dechter (Ed.) *Principles and practice of constraint programming - CP 2000, 6th international conference, Singapore, September 18-21, 2000, proceedings, lecture notes in computer science*. https://doi.org/10.1007/3-540-45349-0/_36, (Vol. 1894 pp. 489–494). Berlin: Springer.
7. Benhamou, B., & Henocque, L. (1999). A hybrid method for finite model search in equational theories. *Fundam Informaticae*, *39*(1-2), 21–38. https://doi.org/10.3233/FI-1999-391202.
8. Boy de la Tour, T., & Countcham, P. (2005). An isomorph-free SEM-like enumeration of models. *Electronic Notes in Theoretical Computer Science*, *125*(2), 91–113. https://doi.org/10.1016/j.entcs.2005.01.003, https://www.sciencedirect.com/science/article/pii/S1571066105000976. Proceedings of the 5th International Workshop on Strategies in Automated Deduction (Strategies 2004).

9. Burris, S., & Sankappanavar, H. P. (1981). *A course in universal algebra. Graduate texts in mathematics* Vol. 78. New York: Springer.
10. Claessen, K., & Sörensson, N. (2003). New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 workshop: model computation - principles, algorithms, applications*.
11. Crawford, J. M., Ginsberg, M. L., Luks, E. M., & Roy, A. (1996). Symmetry-breaking predicates for search problems. In L.C. Aiello, J. Doyle, & S.C. Shapiro (Eds.) *Proceedings of the fifth international conference on principles of knowledge representation and reasoning (KR)* (pp. 148–159). San Francisco: Morgan Kaufmann.
12. Dixon, J. D., & Mortimer, B. (1996). *Permutation groups*. New York: Springer.
13. Elhamdadi, M., Macquarrie, J., & Restrepo, R. (2012). Automorphism groups of quandles. *J. Algebra Appl 11*(1). https://doi.org/10.1142/S0219498812500089.
14. The GAP Group: GAP – Groups, Algorithms, and Programming, Version 4.11.1 (2021). https://www.gap-system.org.
15. Gent, I.P., Jefferson, C., & Miguel, I. (2006). Minion: a fast scalable constraint solver. In G. Brewka, S. Coradeschi, A. Perini, & P. Traverso (Eds.) *ECAI 2006, 17th European conference on artificial intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, including prestigious applications of intelligent systems (PAIS 2006), proceedings, frontiers in artificial intelligence and applications*. http://www.booksonline.iospress.nl/Content/View.aspx?piid=1654, (Vol. 141 pp. 98–102). Amsterdam: IOS Press.
16. Gomes, C.P., Selman, B., & Kautz, H.A. (1998). Boosting combinatorial search through randomization. In J. Mostow C. Rich (Eds.) *Proceedings of the fifteenth national conference on artificial intelligence and tenth innovative applications of artificial intelligence conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*. http://www.aaai.org/Library/AAAI/1998/aaai98-061.php (pp. 431–437). Menlo Park: AAAI Press / The MIT Press.
17. Janota, M., & Suda, M. (2018). Towards smarter MACE-style model finders. In G. Barthe, G. Sutcliffe, & M. Veanes (Eds.) *LPAR-22. 22nd international conference on logic for programming, artificial intelligence and reasoning, EPiC series in computing*. https://doi.org/10.29007/w42s, (Vol. 57 pp. 454–470). Manchester: EasyChair.
18. Jia, X., & Zhang, J. (2006). A powerful technique to eliminate isomorphism in finite model search. In U. Furbach N. Shankar (Eds.) *Automated reasoning* (pp. 318–331). Berlin: Springer.
19. Lynce, I., Baptista, L., & Marques-Silva, J. (2002). Complete unrestricted backtracking algorithms for satisfiability. In *Proceedings of the international symposium on theory and applications of satisfiability testing* (pp. 214–221).
20. Malandro, M. E. (2019). Enumeration of finite inverse semigroups. *Semigroup Forum*, *99*, 679–723.
21. Marker, D. (2002). *Model theory: an introduction*. New York: Springer.
22. McCune, W. (2003). Mace4 reference manual and guide (Technical Memorandum No. 264), 20. https://www.cs.unm.edu/mccune/prover9/mace4.pdf.
23. McKay, B.D. (1998). Isomorph-free exhaustive generation. *Journal of Algorithms*, *26*(2), 306–324. https://doi.org/10.1006/jagm.1997.0898, https://www.sciencedirect.com/science/article/pii/S0196677497908981.
24. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th design automation conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. https://doi.org/10.1145/378239.379017 (pp. 530–535). New York: ACM.
25. Nagy, G., & Vojtěchovský, P. (2018). LOOPS, computing with quasigroups and loops in GAP, Version 3.4.1. https://gap-packages.github.io/loops/. Refereed GAP package.
26. Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial optimization: algorithms and complexity*. Upper Saddle: Prentice-Hall.
27. Reger, G., Riener, M., & Suda, M. (2019). Symmetry avoidance in MACE-style finite model finding. In A. Herzig A. Popescu (Eds.) *Frontiers of combining systems - 12th international symposium, FroCoS 2019, London, UK, September 4-6, 2019, proceedings, lecture notes in computer science*. https://doi.org/10.1007/978-3-030-29007-8/_1, (Vol. 11715 pp. 3–21). Switzerland: Springer.
28. Sloane, N.J.A., & Inc., T.O.F. (2020). The on-line encyclopedia of integer sequences. http://oeis.org/?language=english.
29. Zhang, J. (1996). Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, *17*, 1–22. https://doi.org/10.1007/BF00247667.
30. Zhang, J., & Zhang, H. (1995). SEM: a system for enumerating models. In *IJCAI*. http://ijcai.org/Proceedings/95-1/Papers/039.pdf (pp. 298–303).

Symmetries for cube-and-conquer in finite model finding

João Araújo
Universidade Nova de Lisboa, Lisbon, Portugal

Choiwah Chow
Universidade Aberta, Lisbon, Portugal

Mikoláš Janota
Czech Technical University in Prague, Czechia

Abstract

The cube-and-conquer paradigm enables massive parallelization of SAT solvers, which has proven to be crucial in solving highly combinatorial problems. In this paper, we apply the paradigm in the context of finite model finding, where we show that isomorphic cubes can be discarded since they lead to isomorphic models. However, we are faced with the complication that a well-known technique, the Least Number Heuristic (LNH), already exists in finite model finders to effectively prune (some) isomorphic models from the search. Therefore, it needs to be shown that isomorphic cubes still can be discarded when the LNH is used. The presented ideas are incorporated into the finite model finder Mace4, where we demonstrate significant improvements in model enumeration.

2012 ACM Subject Classification Computing methodologies; Theory of computation → Constraint and logic programming

Keywords and phrases finite model enumeration, cube-and-conquer, symmetry-breaking, parallel algorithm, least number heuristic

Digital Object Identifier 10.4230/LIPIcs..2022.

Funding *João Araújo*: 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.
Mikoláš Janota: 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 European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the grant of National Science Center, Poland, no. 2018/29/N/ST6/02903, and Amazon Research Awards. This article is part of the *RICAIP* project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

1 Introduction

An important tool that working algebraists need in their research is libraries of the algebras they are interested in. These libraries allow them to get intuitions, test or refute hypotheses and conjectures, and gain insights into the properties of the algebras (see examples on p. 2891 of [28]). Many libraries of algebraic models of small orders, such as the smallsemi package [13] for semigroups and the loops package [32] for quasigroups, are available in the GAP [15] system. A lot more such libraries are needed, but they often take an inordinate amount of time and computing resources to generate.

First-order logic (FOL) has been the most popular language to define algebras. There are two major resource-intensive steps in generating non-isomorphic models from a FOL [26]. The first step is to generate models according to the laws specified by the FOL formula. This step often generates a huge number of isomorphic models. For example, given the first-order formula for semigroups, which is $(x * y) * z = x * (y * z)$, Mace4 [31] generates 1,021,120,198

models of order 7, out of which only 1,627,672 ($\approx 0.16\%$) [39] are pairwise non-isomorphic. The second step is to eliminate the isomorphic models generated in the first step. In this paper, we propose a novel efficient and scalable parallel algorithm that not only speeds up the first step but also generates fewer isomorphic models. Suppressing the generation of isomorphic models in the first step reduces the workloads of both the first and the second steps. Not only does it make the whole process much faster, but the required computing resources (disk space, etc.) are also reduced.

While modern-day general-purpose computers are predominantly multi-core, harnessing parallelism for combinatorial search is surprisingly difficult, consequently, there are few parallel algorithms in constraints programming in general, and in finite model enumeration in particular. Indeed, in satisfiability modulo theories (SMT), even negative results are concluded for cube-and-conquer [22]. A recent literature review concludes that "there is little overall guidance that can be given on how best to exploit multi-core computers to speed up constraint solving" [17]. We aim to help close this gap by devising new parallel algorithms for finite model enumeration.

In this paper, we improve finite model enumerators toward the following two high-level objectives:

1. Mathematicians can use the tool to quickly generate all models (up to isomorphism) of the classes of algebras of their interests on their multi-core computers.
2. The tool can also take advantage of massively parallel computing architectures to pre-generate models (up to isomorphism) of the classes of algebras of general interest.

We find inspiration in the well-established cube-and-conquer approach introduced for SAT [21]. In SAT this means splitting the search space by mutually exclusive conjunctions of propositional literals (cubes). In the context of finite model finding, the structure is richer—a decision of the solver corresponds to inserting a point into the graph of one of the considered functions, e.g., $f(0,1) = 3$.

We show that a cube can be excluded from the search if it is isomorphic to an existing one. Effectively, this is breaking symmetries in the search space. However, the task is nontrivial because finite model finders already contain a technique, called the least number heuristic (LNH), to exclude some isomorphic models. The LNH[1] enables the solver to consider only certain values from the co-domain for a given decision point. Therefore, we show that isomorphic cubes can be pruned in the presence of the LNH. Like so, we can take advantage of the two powerful and complementary techniques and ultimately suppress the generation of a large number of isomorphic models.

This paper's contributions are the following:

1. Devise a low runtime overhead parallel algorithm based on the cube-and-conquer approach for finite model enumeration. This scalable algorithm divides finite model enumeration into many independent non-overlapping search jobs to make full use of the available resources.
2. We show that isomorphic cubes can be discarded without losing isomorphic models even in the presence of the well established symmetry breaking technique already present in finite model finders—the least number heuristic (LNH).
3. We extend the model finder Mace4 with the proposed techniques and evaluate it on a large number of problems, where significant speed-up is observed.

[1] Despite the technique being called a heuristic, it does not sacrifice the completeness of the solver.

2 Preliminaries

Familiarity with the general notions of abstract algebra such as groups, semigroups, and quasigroups is assumed, and so is general knowledge about functions and isomorphisms. A good reference is Chapters 2 and 5 of [9].

In this paper, the domain of the search space is denoted by the set $D = \{0, \ldots, n-1\}$, where $n \geq 2$, That is, we exclude the trivial case of searching on domains of size 1.

Let π denote an arbitrary permutation on D, π_{id} denote the identity permutation, and $\pi_{a,b}$ denote the permutation cycle (a, b). For example, $\pi_{0,1}$ is the permutation cycle $(0, 1)$.

2.1 Finite Model Enumeration

Given a signature Σ and a FOL formula $\mathcal{F}$ on Σ, a traditional finite model finder first expands the FOL formula to its ground terms by its domain elements in D, then searches for models by applying a depth-first search with backtracking to exhaustively explore the search space [43]. The domain elements in D are seen as special constants not appearing in the original $\mathcal{F}$, c.f. [36].

Following the terminology of [43], 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 $a_j, v \in D$. We refer to the term $f(a_1, \ldots, a_k)$ as the *cell term* (or simply *cell*) since conceptually the search fills the operation table of f.

To search for finite models in $\mathcal{F}$, the finite model finder employs a *cell selection* strategy to pick cell terms successively, without duplicates, to assign values from D to form VA clauses. If a newly formed VA clause causes any failure in the axioms in $\mathcal{F}$, then a new value will be tried for that cell term. 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 term to try to assign another value to it. When all cell terms in $\mathcal{F}$ are assigned values without violating the axioms in $\mathcal{F}$, a model, as represented by its VA clauses, is found. After a model is found, the process can continue with backtracking to find more models.

A set of models can be partitioned into equivalence classes by isomorphisms. Intuitively, a model can be transformed into any other model in the same equivalence class by renaming its domain elements. Two models are said to be isomorphic to each other if an isomorphism exists from one model to the other.

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. The cell term in each node is selected by the cell selection strategy. 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 i^{th} position of the sequence and $v_i \in D$, and $t_i \neq t_j$ when $i \neq j$. Furthermore, a search path will be terminated at the first VA clause that results in a violation of any axiom of $\mathcal{F}$.

If the length of a search path is the same as the total number of cell terms in $\mathcal{F}$, then it is a complete search path and its VA clauses represent a model. Otherwise, it is an incomplete search path representing *partial assignments* of cell values in $\mathcal{F}$.

The backtracking algorithm in its simplest form is to try every possible value assignment for every cell. For example, to search a FOL formula $\mathcal{F}$ with just one binary operation, there are n^{n^2} possible combinations (n^2 cells with n possible values each). Even the very small domain size of 4 gives over 4 billion combinations of cell values. However, in practice, the number of viable combinations to check is much smaller than the theoretically maximum number because of the constraints imposed by $\mathcal{F}$. Furthermore, a finite model finder may infer new VA clauses from existing ones by *propagation*.

▶ **Example 1.** Suppose the FOL formula contains only the equation $f(x,y) = f(y,x)$, that is, the operation f is commutative. After the assignment $f(0,1) = 0$, the finite model finder can infer $f(1,0) = 0$. This is referred to as positive propagation.

On the other hand, if the FOL formula contains the inequality $f(x,y) \neq f(y,x)$, then after the assignment $f(0,1) = 0$, the finite model finder can exclude 0 from the list of possible values for the cell $f(1,0)$. This is referred to as negative propagation. ◀

2.2 Least Number Heuristic

The least number heuristic (LNH) [4, 44, 45] is a very effective symmetry-breaking algorithm widely implemented in model finders/enumerators such as Mace4. The main idea of the LNH is that all domain elements that have not yet appeared in any VA clauses and the current cell term in the search are indistinguishable to each other and therefore only one of them, say, the smallest one, needs to be considered in a cell value assignment.

To ease discussions of the LNH, we introduce the notation $\mathrm{Vals}(P)$ to denote the set of all domain elements appearing in P, where P can be a search path, a VA clause, or a cell term.

▶ **Example 2.** For the cell term $f(1,1)$: $\mathrm{Vals}(f(1,1)) = \{1\}$. For the VA clause $f(1,1) = 0$: $\mathrm{Vals}(f(1,1) = 0) = \{0,1\}$. For the partial search path $S = \langle f(0,0) = 0; f(1,1) = 0\rangle$: $\mathrm{Vals}(S) = \{0,1\}$. ◀

The LNH can now be stated precisely: In adding a VA clause, $t = v$, to extend a search path S, the possible choices of v allowed under the LNH are $\mathrm{Vals}(S) \cup \mathrm{Vals}(t) \cup \{s\}$ where s is the smallest domain element in $D \setminus (\mathrm{Vals}(S) \cup \mathrm{Vals}(t))$, and they are D if $\mathrm{Vals}(S) \cup \mathrm{Vals}(t) = D$. Furthermore, we say a search path is *LNH-compliant* if it respects the LNH restrictions on the choices of values assigned to its VA clauses.

▶ **Example 3.** Suppose the domain size, $|D|$, is 4. Then the complete search path $\langle f(1) = 0; f(0) = 3; f(3) = 1; f(2) = 1\rangle$ is not LNH-compliant.

For the first VA clause in the search path, $S = \emptyset$ and $t = f(1)$. So, $\mathrm{Vals}(S) \cup \mathrm{Vals}(t) = \emptyset \cup \{1\} = \{1\}$, and therefore $D \setminus (\mathrm{Vals}(S) \cup \mathrm{Vals}(t)) = \{0,2,3\}$. Thus, $s = \min(\{0,2,3\}) = 0$. The LNH limits the choices of the value for $f(1)$ to $\mathrm{Vals}(S) \cup \mathrm{Vals}(t) \cup \{s\} = \{0,1\}$. So the first VA clause $f(1) = 0$ is LNH-compliant. However, for the second VA clause in the search path, $S = \{f(1) = 0\}$ and $t = f(0)$. So, $\mathrm{Vals}(S) \cup \mathrm{Vals}(t) = \{0,1\} \cup \{0\} = \{0,1\}$, and therefore $D \setminus (\mathrm{Vals}(S) \cup \mathrm{Vals}(t)) = \{2,3\}$. Thus, $s = \min(\{2,3\}) = 2$. The LNH limits the choices of the value for $f(0)$ to $\mathrm{Vals}(S) \cup \mathrm{Vals}(t) \cup \{s\} = \{0,1,2\}$, so $f(0) = 3$ is not allowed under the LNH. Therefore, the whole search path is not LNH-compliant. ◀

The LNH does not impose any restrictions on the order of the cell terms in the search path[2]. It speeds up the search by limiting the choices of the values for the cell terms. Therefore, its effectiveness decreases with the length of the search path as more domain elements are used when more VA clauses are added to the search path.

▶ **Example 4.** The *concentric cell selection strategy* is a simple cell selection strategy to minimize the growth of choices of values in the finite model search with the LNH. This strategy picks the cell $f(a_0, \ldots, a_{k-1})$ with the least $r = \max(a_0, \ldots, a_{k-1})$ from all available

[2] In practice, a number called the *maximal designated number* (*mdn*) is often used to partition the domain into 2 subsets so that $\{0, \ldots, \mathrm{mdn}\}$ are domain elements already seen, and $\{mdn + 1, \ldots, n - 1\}$ are domain elements not seen so far [43]. In this case, cell selection strategies that keep the *mdn* small are preferred because the search tree will be kept narrower.

cells. Any fixed tie-breaker can be used in case of a tie. For example, one of the possible orders of the cells by this cell selection strategy for a binary operation is $f(a_0, a_1) < f(b_0, b_1)$ if $a_0 = a_1 \vee a_0 + a_1 < b_0 + b_1 \vee (a_0 + a_1 = b_0 + b_1 \wedge a_0 < b_0)$. This gives the sequence $f(0,0)$, $f(1,1)$, $f(0,1)$, $f(1,0)$, $f(2,2)$, $f(0,2)$, $f(2,0)$, $f(2,1)$, $f(1,2)$, $f(3,3)$ .... ◀

2.3 Cube

A *cube* is a prefix of a search path, and as such, it can be specified by a sequence of VA clauses. Permutations and isomorphisms 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 $\pi(B) \coloneqq \{f(\pi(a_1), \ldots, \pi(a_k)) = \pi(v) \mid f(a_1, \ldots, a_k) = v \text{ is a VA clause in } B\}$. Observe that $\pi_{id}(B)$ is the (unordered) set of all individual VA clauses in the cube B.

Note that predicates in a FOL formula can be implemented as functions with two values, T (true) and F (false), which do not affect the LNH because they are not domain elements. For convenience, we consider $I(T) = T$ and $I(F) = F$ for any isomorphism I so that the same terminology is used for both relations and functions.

Cubes are said to be isomorphic if their VA clauses are isomorphic. 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 5.** If $B_0 = \langle f(0) = 0; g(0,0) = 0; f(1) = 0; g(1,1) = 0\rangle$ 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 Isomorphic Cubes Redundancy

The main objective of this section is to show that isomorphic cubes can be removed from the search. More formally, if cubes B_0 and B_1 are isomorphic, then it is sufficient to explore assignments extending B_0 and ignore *all* assignments extending B_1. We need to prove that any model lost by discarding B_1 must necessarily be isomorphic to some model obtained from extending B_0 under the LNH. This statement is intuitive, but the proof requires some care as effectively, we are dealing with a combination of two symmetry-breaking techniques: LNH and isomorphic cube pruning, under an arbitrary search strategy.

As a motivational example, consider the cube $\langle f(0,0) = 0\rangle$, which states that f is idempotent in 0. But because 0 does not appear in the original FOL formula, intuitively, the constant 0 cannot play a special role in the formula. Consequently, this cube searches *all* interpretations of f that have at least one idempotent. For instance, the cube $\langle f(1,1) = 1\rangle$ will search the same interpretations, up to isomorphism. Now, we need to show this property formally and that it holds when the solver searches with the LNH restriction.

The key idea of the proof is that given a model B_1 with VA clauses A, any cube that is isomorphic to a subset of A can be gradually extended to be a model isomorphic to B_1. Each extension step of the cube must uphold the following properties: (1) The cube is isomorphic to some subset of A. (2) The cube is LNH-compliant. The extension step is illustrated in Figure 1. We are given a cube B_0 that is isomorphic to an $A_0 \subseteq A$. When the finite model finder decides on some empty cell t, we need to show that it is possible to find a value according to the LNH such that the extended cube is isomorphic to some subset of A containing A_0.

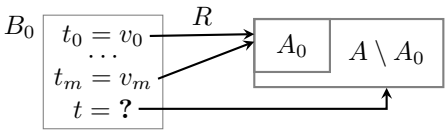


Figure 1 Extension of a cube according to the VA clauses A

▶ **Notation 1.** *For a mapping R from D to D and a value $d \in D$ we write $\mathcal{E}_R^d$ for a mapping that maps d to $R(d)$ if $d \in \mathsf{dom}(R)$ and otherwise maps d to $\min(D \setminus \mathsf{rng}(R))$. We further write $\mathcal{E}_R^{d_1,\ldots,d_k}$ for successive extensions by $d_1, \ldots, d_k$, i.e. $\mathcal{E}_R^{d_1,d_2} = \mathcal{E}^{d_2}_{\mathcal{E}_R^{d_1}}$ etc.* ◀

▶ **Example 6.** Suppose $D = \{1,2,3\}$ and $R : \{1\} \to \{2\}$ s.t. $R(1) = 2$ and $R^{-1}(2) = 1$ (so, R is a bijection). Then $\mathcal{E}_R^2$ s.t. $R(2) = \mathcal{E}_R^2(2) = 1$ is a valid extension of R because $\min(D \setminus \{2\}) = 1$. Furthermore, $\mathcal{E}_R^{2,1}$ s.t. $\mathcal{E}_R^{2,1}(1) = 2$ is a valid (but trivial) extension of $\mathcal{E}_R^2$. ◀

▶ **Lemma 7.** *If R is a bijection between some $D_0, D_1 \subseteq D$ and $d \in D$ then $\mathcal{E}_R^d$ is well-defined and also a bijection.*

Proof. If $d \in D_0$, then $\mathcal{E}_R^d = R$ and there is nothing to proof. If $d \in D \setminus D_0$, then by definition, $\mathcal{E}_R^d = R \cup \{(d,p)\}$ for some $p \in D \setminus D_1$. Since R is a bijection from D_0 to D_1, $d \notin \mathsf{dom}(R)$, and $p \notin \mathsf{rng}(R)$, so $\mathcal{E}_R^d$ is well-defined, one-one, and onto. That is, it is a bijection. ◀

▶ **Notation 2.** *$B \oplus \langle t = u \rangle$ is the new cube formed by extending the cube B with the VA clause $t = u$.* ◀

The following lemma is the core of our proof. We have a cube B isomorphic to some partial assignment A_0 and now we need to prove that for *any* model A completing A_0 and *any* search strategy, we are able to extend B while observing the LNH. Then, the lemma is used to prove that isomorphic cubes can be discarded by induction on cube length (Theorem 9).

▶ **Lemma 8.** *Let B be an LNH-compliant cube and A a model s.t. B is isomorphic to some $A_0 \subseteq A$. Then for any cell term t not appearing in B, there exists a value u and a VA clause $t' = u' \in A \setminus A_0$, s.t. $B \oplus \langle t = u \rangle$ is LNH-compliant and isomorphic to $A_0 \cup \{t' = u'\}$.*

Proof. Let R be an isomorphism mapping B to A_0 and let t be a cell term $f(a_1, \ldots, a_k)$. Define R_1 as $\mathcal{E}_R^{a_1,\ldots,a_k}$, and let t' denote the cell term $f(R_1(a_1), \ldots, R_1(a_k))$, i.e., map the cell that the solver searches on into a cell in the prescribed model A.

Since A is a model, there must exist a value $u' \in D$ with $(t' = u') \in A$, i.e. u' can be found by a lookup of t' in A. Since t is not a cell term in B and R_1 is a bijection, so t' is not a cell term in A_0 and must therefore be in $A \setminus A_0$. Thus, $t' = u'$ is a VA clause in $A \setminus A_0$.

To obtain u (a value for cell t), define R_2 as $\mathcal{E}^{u'}_{R_1^{-1}}$, i.e. map u' back into the search by extending the inverse. Then, set $u = R_2(u')$. By Lemma 7, R_2 is bijection and it is therefore an isomorphism from $A_0 \cup \{t' = u'\}$ to $B \oplus \langle t = u \rangle$. Finally, by definition of R_2, u either already appears in B or otherwise is the smallest domain element not in B. Therefore, the extension of the cube B by the VA clause $t = u$ is LNH-compliant. ◀

▶ **Theorem 9.** *Suppose we are searching under the LNH with any cell selection strategy on a signature Σ and a FOL formula, $\mathcal{F}$, on Σ. If B_0 and B_1, of length $l \geq 0$, are isomorphic cubes, and if M_1 is a model obtained by completing (not necessarily under the LNH) the search path in B_1, then B_0 can be extended by a search path S under the said LNH and cell selection strategy to a model M_0 which is isomorphic to M_1.*

Proof. We will use mathematical induction on the length of the extension, m, on S to prove the theorem. Let A denote the VA clauses of M_1, and A_0 denote the VA clauses of B_1.

Base case is trivial as B_0 and B_1 are given as isomorphic when $m = 0$.

Induction step: Suppose the search path S is extended m times, where $m > 1$, so that S_m is LNH-compliant and isomorphic to a subset $A_m \subseteq A$. Then by Lemma 8, S_m can be extended by one VA clause with the cell term t_{m+1}, chosen by the said cell selection strategy, to S_{m+1} which is LNH-compliant and isomorphic to $A_{m+1} \subseteq A$.

Note that a model finder may do propagations after a cell value assignment. That is, some cell terms can be assigned values inferred from existing VA clauses. Propagations can be viewed as part of the cell selection strategy and be handled the same way as regular cell value assignments.

We can therefore conclude by mathematical induction that S can be extended to a complete search path when all cell terms in $\mathcal{F}$ are filled with values such that S represents the model M_0, is LNH-compliant, and is isomorphic to $A_s \subseteq A$. Since M_0 and M_1 are of the same size, so A_s and A must necessarily be of the same size and hence must be equal. Therefore, M_0 is isomorphic to M_1. ◀

Theorem 9 shows that isomorphic cubes always extend to isomorphic models. So, one of the isomorphic cubes may be discarded without losing any non-isomorphic model.

▶ **Corollary 10.** *On searching under the LNH with any cell selection strategy on a signature Σ and a FOL formula $\mathcal{F}$ on Σ, if M_1 is a model in $\mathcal{F}$, then there is a complete search path S under the said LNH that results in a model M_0 which is isomorphic to M_1.*

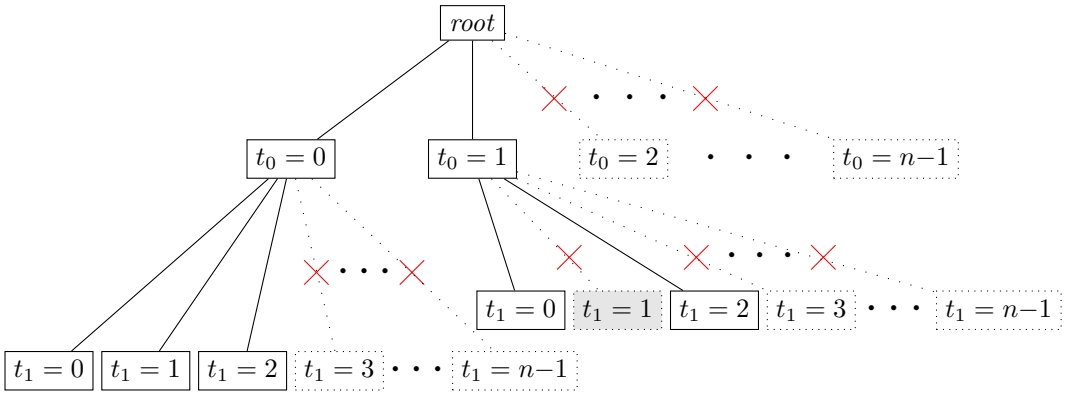
Corollary 10 proves the completeness of the LNH in that every model in any search is isomorphic to some model found by searching under the LNH. An alternative proof of the corollary is given in [44].

4 Searching with Cubes

Cubes can be constructed to partition the search space into non-overlapping subtrees that can be processed in parallel. It is not necessary to search all the subtrees that originate from the collection of cubes that span the entire search space because isomorphic cubes in the same collection can be eliminated without losing non-isomorphic models. For example, suppose we want to search for models of order 3 or more on a function $f : D^2 \to D$ under the LNH with a cell selection strategy that selects $f(0,0)$ then $f(1,1)$ as the first 2 cell terms in the search process. There are at most 6 cubes of length 2 (listed below) under the said LNH and cell selection strategy, so together they must span the whole search space in the sense that every search path that starts with the cell terms $f(0,0)$ then $f(1,1)$ in the search tree must include one of the 6 cubes in it.

1. $\langle f(0,0) = 0; f(1,1) = 0\rangle$.
2. $\langle f(0,0) = 0; f(1,1) = 1\rangle$.
3. $\langle f(0,0) = 0; f(1,1) = 2\rangle$.
4. $\langle f(0,0) = 1; f(1,1) = 0\rangle$.
5. $\langle f(0,0) = 1; f(1,1) = 1\rangle$.
6. $\langle f(0,0) = 1; f(1,1) = 2\rangle$.

Since $\pi_{0,1}(\text{Cube 1}) = \{f(1,1) = 1, f(0,0) = 1\} = \pi_{id}(\text{Cube 5})$, so Cubes 1 and 5 are isomorphic and one of them can thus be discarded without losing non-isomorphic models per Theorem 9. This example demonstrates the importance of keeping the LNH in the search — it cuts the search space from potentially n^2 cubes down to 6. Theorem 9 allows us to further



Note: t_0 denotes $f(0,0)$ and t_1 denotes $f(1,1)$. A dotted line with a cross is a branch pruned by the LNH, except for the branch ending on the VA clause $t_1 = 1$ (the shaded node), which is pruned by the isomorphic cubes removal algorithm.

Figure 2 Partial Search Tree Showing Cubes of Length 2

cut the number of cubes down to 5 (see Figure 2 for illustration). More isomorphic cubes can be removed with longer cubes (see Table 2).

The procedure of removing isomorphic cubes starts with generating a set of short cubes (typically of length 2 for a binary operation) that spans the entire search space. The model finder takes short cubes as inputs and runs with them as if they are generated by itself to generate longer cubes of predefined length l. Specifically, the model finder runs as usual, except that it emits the cubes of length l when the depth of the search tree reaches l. After outputting the cube, the model finder backtracks as if it has reached the bottom of the search tree, and runs on a new branch as usual until all cubes of length l are generated. Some models may be generated in this process due to propagation, and they are kept as part of the final outputs. Next, the cubes are compared for isomorphism and only one of any pair of isomorphic cubes is kept. This new set of non-isomorphic cubes of length l will be used as inputs to the model finder in the next round of generation of longer cubes. The process is repeated until the desired length of cubes is reached.

For searching models defined by one operation of arity k, we use the sequence of lengths l: $k, 2^k, 3^k, 4^k, \ldots$. This is to match the concentric cell selection strategy (see Example 4 for its definition) of the finite model finder such as Mace4. We will discuss the best cube length to use in Section 5.3.

Finally, non-isomorphic cubes of the target length can then be processed independently in parallel and their output models collected separately.

4.1 Invariants

To speed up the isomorphic cubes removal process, the same invariant-based algorithm described in [2] to remove isomorphic models can be applied to cubes. Invariants, such as number of distinct domain elements, are properties that must be identical for cubes to be isomorphic. For example, the cubes $A = \langle f(2,2) = 2; f(2,3) = 4\rangle$ and $B = \langle f(3,3) = 2; f(1,2) = 2\rangle$ cannot be isomorphic because A contains an idempotent 2 but B does not. Powerful and inexpensive invariants for binary operations include (1) Number of y such that $x = (x * y) * x$. (2) number of y such that $y = x * z$ for all $z \in D$. (3) Number of y such

that $y = z * y$ for all $z \in D$. (4) Number of idempotents x (i.e. $x * x = x$) for all $x \in D$. (5) Number of y such that $y * y = x$ for each $x \in D$.

First, invariant vectors (i.e. ordered lists of invariants) for cubes are calculated and used as hash keys to group cubes having the same invariant vectors into hash buckets. Then, cubes within the same bucket are tested for isomorphism. There is no need to test for isomorphism across buckets because isomorphic cubes must have the same invariant vectors. This saves tremendous amounts of testing time. Furthermore, buckets can be processed independently and in parallel to further speed up the process.

4.2 Work Stealing

In the basic form of this cube-based parallel algorithm, cubes are statically generated before the model enumeration process begins. The disadvantage is that the workload may be uneven among the parallel processes. Some jobs may take a long time to finish when free workers sitting idle after finishing their jobs.

This problem can be solved with *work stealing* algorithms (also used in SAT [25]) in which a busy finite model searcher releases cubes that are not currently being worked on. For example, suppose a running model searcher is working on a cube $B_0 = \langle f(0,0) = 0 \rangle$, and its cell selection strategy picks the cell $f(1,1)$ to assign value next. Under the LNH, $f(1,1)$ may be assigned a value from $\{0,1,2\}$. If the model searcher is requested to spin out some work for other free workers, then it generates three cubes, $B_0 = \langle f(0,0) = 1; f(1,1) = 0 \rangle$, $B_1 = \langle f(0,0) = 1; f(1,1) = 1 \rangle$, and $B_2 = \langle f(0,0) = 1; f(1,1) = 2 \rangle$. It continues to work on the cube B_0 and releases B_1 and B_2 to other free workers.

5 Experimental Results

We integrate the cube-based algorithms into the finite model enumerator Mace4, which supports searching on FOL with the LNH and many cell selection strategies [31]. Parallelization is controlled outside Mace4. Only minor changes are made to Mace4 to

1. Accept cubes as inputs and continue searching for longer cubes or models from them.
2. Periodically check for signal for work stealing to spin off cubes for other workers.

The model searching logic in Mace4 remains intact. The concentric cell selection strategy (see Example 4 for its definition) is used in the experiments. A separate program removes isomorphic cubes by separating the cubes with equal invariants then check for isomorphisms (two cubes are isomorphic if one can be transformed to the other by a permutation).

We run the experiments on an Intel® Xeon®Silver 4110 CPU 2.0 GHz ×32 computer, with 64 GB of random access memory (RAM), using 30 parallel processes unless otherwise stated. We pick many disparate and challenging problems from the MarcieX database [3], which contains a collection of 158 most popular algebras. We also draw an example of semigroup subvariety from [1]. 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.

In the tables showing experimental results in this section, the rows with cube length 0 show the results of running Mace4 in a single thread without the cube-based algorithms. All times are wall-clock time.

Table 2 shows the results of applying Theorem 9 to remove isomorphic cubes for the binary operation of the semigroups of order 7. Observe that the percentage reduction of the number of cubes increases as the cube length increases. The isomorphic cubes removal algorithm is therefore complementary to the LNH because the LNH removes a lot of short cubes but loses its effectiveness as the length of the cubes grows.

Table 1 Definitions of Algebras Used in Experiments

Algebra	FOL Definition
Semigroups	$x * (y * z) = (x * y) * z.$
Loops	$x * y = x * z \to y = z.$ $y * x = z * x \to y = z.$ $x * 0 = x.$ $0 * x = x.$
var$\{N_2^1 \cap [x^2 = y^2]\}$	$x * (y * z) = (x * y) * z.$ $(x * x) * x = x * x.$ $x * y = y * x.$ $x * x = y * y.$
Tarski Algebras	$(x * y) * y = (y * x) * x.$ $x * (y * z) = y * (x * z).$ $(x * y) * x = x.$
Quasi-ordered Set	$x < y \wedge y < z \to x < z.$ $x < x.$
Involutive Lattices	$(x * y) * z = x * (y * z).$ $x * y = y * x.$ $(x + y) + z = x + (y + z).$ $x + y = y + x.$ $(x * y) + x = x.$ $(x + y) * x = x.$ $-(x + y) = -x * -y.$ $- - x = x.$

Table 2 #Cubes for Semigroups of Order 7

	# Cubes		
Cube Length	w/o Removal of Isomorphic Cubes	w/ Isomorphic Cubes Removed	Reduction (%)
2	6	5	16.7
4	34	28	17.6
9	1,568	888	43.4
16	56,206	12,036	78.2
25	1,028,171	59,056	94.3

We run Mace4 to enumerate models of semigroups defined by a single binary operation. The results show a speedup of over 100 times when cubes of length 25 are used, with over 96% of the isomorphic models suppressed (see Table 3). The results on semigroups are indicative of the algorithm's usefulness in general to the computational algebraists because algebraic structures related to semigroups are ubiquitous in algebra. Not only are there many well-known semigroup-related algebras, but also many semigroup varieties and subvarieties that are of great research interests [1].

Table 4 shows the results for loops (a quasigroup-related algebras) defined by a single non-associative binary operation. Here the reduction in the number of the output isomorphic models is not as pronounced. This is expected because the LNH works very well with the Latin square and removes a high percentage of the isomorphic models [42] before the isomorphic cubes removal takes place. For example, 8.7% (106,228,849 out of 1,216,226,816) of the models generated for the loops of order 8 under the LNH are non-isomorphic. Nevertheless, the parallel algorithm provides 15 times improvement in speed for cube length of 16.

Table 5 shows the results of running the algorithms on the semigroup subvariety var$\{N_2^1 \cap [x^2 = y^2]\}$ (see p. 40 of [1] for its definition and discussions). With longer cubes, the algorithms speed up the process by 26 times with 30 threads. The results confirm that the proposed algorithms work remarkably well with semigroup-related algebras.

The Tarski algebra is unlike both the semigroup and the quasigroup in that its multiplication table is not associative and is not a Latin square [3]. It shows the cube-based algorithms perform better and better as the length of the cube increases (see Table 6).

The quasi-ordered set is defined by one binary relation. The isomorphic cubes algorithms work well on relations just as it works well on functions. As shown in Table 7, when cubes of length 36 are used, over 99% of the isomorphic models are suppressed, and the search

process is sped up by over 200 times.

Table 3 Running Cubes on Semigroups of Order 7

Cube Length	#Cubes	#Models (Millions)	Time in min. Gen.	Time in min. Total Cubes
0		1,021.1		235.2
2	5	717.7	0.0	12.5
4	28	611.1	0.1	9.4
9	888	360.2	0.1	5.2
16	12,036	158.2	0.2	2.8
25	59,056	39.5	0.9	1.7

Table 4 Running Cubes on Loops of Order 8

Cube Length	#Cubes	#Models (Millions)	Time in min. Gen.	Time in min. Total Cubes
0		1,216		564.0
2	1	1,216	0.0	47.4
4	2	1,216	0.1	47.3
9	18	1,216	0.1	46.2
16	3,583	1,214	0.1	45.3

Table 5 Running Cubes on var$\{N_2^1 \cap [x^2 = y^2]\}$ of Order 9

Cube Length	#Cubes	#Models (Millions)	Time in min. Gen.	Time in min. Total Cubes
0		313.0		72.0
2	1	156.5	0.0	2.9
4	1	156.5	0.1	2.8
9	2	156.5	0.1	2.8
16	5	120.9	0.1	2.3
25	16	55.5	0.2	1.3
36	70	13.0	0.3	0.8
49	331	1.5	1.0	1.1

Table 6 Running Cubes on Tarski Algebras of Order 13

Cube Length	#Cubes	#Models (Millions)	Time in min. Gen.	Time in min. Total Cubes
0		379.6		1,949.9
2	3	189.8	0.0	70.2
4	1	189.8	0.1	69.9
9	3	183.3	0.1	67.7
16	11	158.8	0.1	58.1
25	55	111.9	0.2	40.1
36	157	62.1	0.2	21.8
49	174	24.9	0.5	8.8
64	171	6.6	1.0	3.7

Table 7 Running Cubes on Quasi-ordered Set of Order 8

Cube Length	#Cubes	#Models (Millions)	Time in min. Gen.	Time in min. Total Cubes
0		642.8		59.9
2	1	642.8	0.0	4.2
4	3	474.6	0.1	3.2
9	9	209.5	0.1	1.7
16	33	61.3	0.1	0.8
25	139	12.6	0.2	0.3
36	713	2.0	0.3	0.3

Table 8 Running Cubes on Involutive Lattices of Order 13

Cube Length	#Cubes	#Models (Millions)	Time in min. Gen.	Time in min. Total Cubes
0		423.0		4,719.7
3	2	423.0	0.0	432.5
6	3	423.0	0.1	432.8
10	6	263.9	0.1	270.0
21	23	178.6	0.1	180.9
36	108	84.9	0.2	88.3
55	555	46.0	0.3	46.2
78	1,710	19.8	0.5	20.6
105	5,048	8.7	4.9	14.3

As an example to demonstrate the effectiveness of the algorithms on more complex algebras, consider the Involutive Lattice [3], which is defined by two associative binary operations and one unary operation. For Involutive Lattices of order 13, the search tree has a maximum depth of 351. Using cubes of length of 105, we obtain a speedup of 300 times, with almost 98% of the isomorphic cubes suppressed (see Table 8). The results show that the isomorphic cubes algorithms are highly effective for both simple and complex algebras.

The reductions in time and number of models (on top of the LNH) are summarized in Figures 3 and 4. Note that the reduction in total time is over 90% even for short cubes.

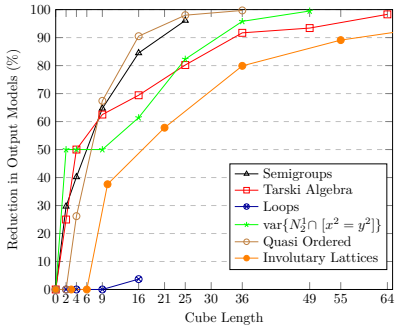


Figure 3 Reduction in Number of Output Models

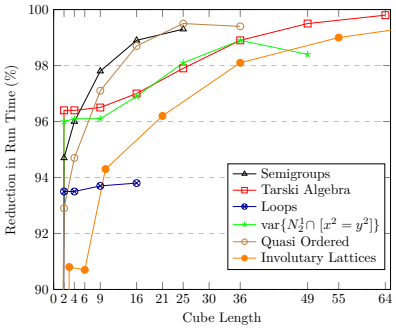


Figure 4 Reduction in Total Time with 30 Parallel Processes

5.1 Speedup of Finite Model Enumeration with Parallelization

Figure 6 shows the performance of the parallel cubes algorithms with 1 to 16 parallel processes, with runtime parameters listed in Figure 5. Here, the reported times do not include isomorphic mode filtering; they are for Mace4 to generate models only. Note that when many processes compete for limited amount of RAM, swapping could slow down the processes substantially. This helps to explain why larger algebras, such as the Involutive Lattice of order 13, have their curves flattened out much faster than small algebras, such as the Semigroups of order 7.

Algebra	Order	Cube Length
Semigroups	7	25
Loops	7	16
Tarski algebras	13	64
var{$N_2^1 \cap [x^2 = y^2]$}	9	49
Quasi Ordered	8	36
Involutive Lattices	12	105

Figure 5 Runtime Parameters

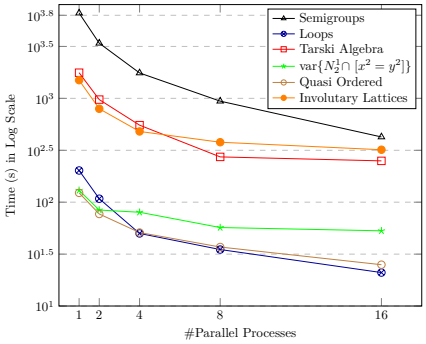


Figure 6 Performance w/ Multiprocessing

5.2 Isomorphic Cubes Removal Speeds up Isomorphic Models Filtering

As pointed out Section 1, reducing the number of Mace4 outputs also reduces the efforts needed to filter out isomorphic models. Table 9 shows, using involutive lattices as an example,

the out-sized effect of the reduction of Mace4 outputs on the time to filter out the isomorphic models using the invariant-based isomorphic model filtering algorithm [2], with 30 parallel processes. With the reduction in number of Mace4 models, the isomorphic model filtering process is sped up by 2 orders of magnitude. The improvement in speed is observed to be better with models of higher orders. We would also point out that the isomorphic model filter generates the same non-isomorphic models with or without the cubes algorithms.

Table 9 Running Invariant-based Isomorphic Models Filter on Involutive Lattices

		w/o Cubes		w/ Cubes		
Order	#Non-iso Models	#Mace4 Output	Isomorphic Model Filter Time (s)	Cube Length	#Mace4 Output	Isomorphic Model Filter Time (s)
9	122	72,470	29	78	3,670	1
10	389	575,463	496	105	13,789	4
11	906	4,771,035	28,424	105	97,680	135
12	3,047	43,851,030	N/A	105	971,416	2,802

5.3 Optimal Cube Length

In general, the search process using longer cubes finishes earlier with fewer isomorphic models. However, we observe that there are three limiting factors on the lengths of the cubes.

First, as the length of the cubes gets longer, more and more models are generated as a result of propagations. This reduces the impact of removing isomorphic cubes because they represent a progressively smaller proportion of the isomorphic models. It is observed that when more than $n-2$ symbols out of the n domain elements are used in the cell terms, the number of (isomorphic) models will be substantial and extending the cube length does not bring enough reduction in isomorphic models to justify the increase in processing time.

Second, the isomorphic cubes removal time grows quite fast as the length of the cube grows. When the isomorphic cubes removal process takes more than a few minutes, further lengthening of the cubes will result in prohibitive overheads in the search process.

Lastly, when the final number of cubes is more than tens of thousands, the overheads in processing them becomes so high that the search becomes slower. This factor depends heavily on the number of processors available. More processors mean more parallel processes can be run without slowing down the whole search process.

One heuristic is to run cube generation until the number of cubes reaches some threshold or the runtime exceeds some threshold, then switch to model generation. The thresholds are system-dependent and can further be fine-tuned by experiments with algebras of interest.

6 Related Work

Parallel algorithms can be characterized by how the search is done. There are two main search methods: the embarrassingly parallel search method (EPS) and the work stealing search method [7, 10, 25, 37, 38]. In the former method, the task is decomposed into many sub-tasks that are queued up to be processed by free worker threads/processes. In the latter method, when a worker completes its task, it asks other workers for more work. The busy workers may split their tasks into smaller sub-tasks and pass some of them to the free workers. The main focus of this method is to keep all the CPUs running until all jobs are done,

although for some cases, the work stealing scheme can affect efficiency [10]. The EPS method is a natural choice for the cube-based parallelization scheme because preprocessing is done to generate large number of non-isomorphic cubes. However, the work stealing search method is also important to supplement the EPS when the workload is uneven (see Section 4.2).

Parallel algorithms can also help select the best strategy in solving a problem with the EPS method [35]. After a problem is decomposed into a large number of sub-tasks, a small number (e.g., 1%) of these sub-tasks are run in parallel using different strategies of the same solver or different solvers. The strategy that gives the best performance on the subset of sub-tasks will be used to run all sub-tasks. The same idea is used in the invariant-based isomorphic models removal algorithm [2]: it randomly generates a large number of invariants, then applies them to a small percentage of models to pick the best performing random invariants to apply to the whole set of models. This idea can be applied to the finite model finders that support multiple cell selection strategies to pick the best function order and cell selection strategy for any specific problem.

Finite model enumeration can be posed as a constraint programming (CP) task [26]. Some CP solvers, such as Minion [16] and Gecode [33], support parallelization [29]. In CP, the search space can be divided into partitions by adding constraints to rule in and/or out partitions. Each partition can be processed by a separate worker thread/process. Minion further implements a work stealing search scheme that also partitions the search space dynamically by splitting the existing constraint model after the search has started [27]. It has been used to do parallel searches that take over 130 CPU years to complete [14]. However, to effectively add symmetry-breaking constraints such as lex-leaders to a CP solver often requires deep knowledge of the solver *and* the problem at hand (e.g., the semigroups in [14]) which may not be available when mathematicians first define and study a new algebraic structure. An alternative to search-space partitioning are portfolios, e.g., ManySAT [18, 19].

Moreover, to use traditional CP solvers for finite model enumerations, mathematicians need to learn a new CP-specific language such as CHR [40] and Savile Row [34]. It is possible to use a translator to translate between languages, but that adds uncertainties to the fidelity and the optimality of the translated specifications. FOL remains one of the most popular languages among mathematicians due to its simple and intuitive syntax. Moreover, a popular automatic theorem prover, Prover9 [30], shares the same input language with Mace4. This adds more than just convenience to the process, as it also reduces the chances of discrepancies between Prover9 and Mace4 on the same problem.

A well-known issue with enumerating models defined with the FOL is the isomorphic models included in the outputs. This is an inherent symmetry property of the FOL [36]. A lot of research efforts are given to symmetry-breaking [4, 11, 12, 36, 41]. Although complete symmetry-breaking is known to be computationally challenging [12, 41], many useful algorithms, such as the LNH and the XLNH [4, 5], have emerged in partial symmetry-breaking. The LNH can be considered a symmetry-breaking with interchangeable values in constraint satisfaction problems (CSP) [20]. The XLNH is more restrictive as it only works on unary operations. The LNH is implemented in many systems such as Falcon [44], SEM [45], FMSET [6], and Mace4. The isomorphic cubes algorithm, which removes more cubes as the cube length grows, complements the LNH.

Another important symmetry-breaking strategy is to steer the search engine away from the fruitless exploration of sub-search space by adding symmetry-breaking input clauses [12, 41]. The cube-based parallel algorithms are compatible with algorithms of this kind of strategy as long as they don't break the LNH.

Some finite model finders, such as SEMK [8] and SEMD [23], try to completely suppress

isomorphic models in the search process. However, these isomorph-free algorithms are not easy to parallelize as global information is generated and consumed in many steps, requiring high-cost synchronizations between cooperating workers, especially when they run on different computers. The cube-based parallel algorithm, on the other hand, is an EPS method that requires no synchronizations between workers. The static removal of isomorphic cubes done in a preprocessing step is shown to be effective in suppressing isomorphic models even before the actual search begins. The augmented work stealing algorithm is not high in synchronization costs because it does not involve communications between running jobs. The remaining isomorphic models from the cube-based algorithms can be efficiently removed by the invariant-based isomorphic model filtering algorithm as a postprocessing step.

The adaptive prefix-assignment technique [24] is a recent addition of symmetry reduction algorithms used in the SAT solvers. Their prefix is equivalent to our cube, and their algorithm also tries to eliminate isomorphic cubes. However, our cubes algorithms work with the LNH that is absent in their algorithm (and in SAT solvers in general).

Another algorithm, DSYM [4], exploits local symmetries by finding symmetries (synonym to isomorphisms in their terminology) under invariant partial interpretations (which are invariant cubes) and without parallelism. It also works with the LNH and XLNH. DSYM is a predictive algorithm that works at the *parent* level and predicts which of its immediate children will be isomorphic cubes. It can be seen as a special case of the isomorphic cube algorithm because it removes isomorphic cubes having the same immediate parents, while the isomorphic cube algorithm removes all isomorphic cubes, irrespective of their parents. Nevertheless, for the cases that DSYM covers, it does so right before the cubes are generated, while the isomorphic cube algorithm only detects the symmetries right after the cubes are generated. A disadvantage of DSYM is that it is not clear how it can be effectively parallelized. Furthermore, DSYM only detects symmetries under the same subtree. The isomorphic cubes removal algorithm, on the other hand, detects both global and local symmetries the same way, and hence detects and removes more symmetries than DSYM. Moreover, DSYM uses only two invariants in testing isomorphism between cubes, while we use many invariants that are proven successful in the invariant-based isomorphic model removal algorithm in our isomorphic cubes removal process. Nevertheless, DSYM can be applied to the cube generation process as well as the final model generation process. That is, the isomorphic cube removal algorithm is compatible with DSYM, as with any other symmetry-breaking algorithm that works with the LNH.

7 Conclusions and Future Work

In this paper, we introduce an efficient parallel algorithm together with a novel symmetry-removal mechanism for enumerating finite models. Future research will focus on improving isomorphic cube removal, on best cell selection strategy, and on predicting of optimal cube length to use.

In conclusion, this paper fulfills an important unmet need for an efficient algorithm for enumerating finite algebraic models in computational algebra by enhancing the existing finite model enumeration process with the parallel cubes algorithm and the isomorphic cubes removal algorithm that reduce both the runtime and the number of output isomorphic models. These new algorithms are so scalable that they can be used in a laptop as well as a cluster of powerful computers, and they require minimal efforts to safely integrate into existing finite model finders. Very importantly, these algorithms can be used as a black-box without requiring the users to have any knowledge about how they work.

References

1. João Araújo, João Pedro Araújo, Peter J. Cameron, Edmond W. H. Lee, and Jorge Raminhos. A survey on varieties generated by small semigroups and a companion website, 2019. arXiv:1911.05817.
2. João Araújo, Choiwah Chow, and Mikoláš Janota. Boosting isomorphic model filtering with invariants. *Constraints*, 27(3):360–379, Jul 2022. doi:10.1007/s10601-022-09336-x.
3. João Araújo, David Matos, and João Ramires. MarcieDB: a model and theory database. https://marciedb.pythonanywhere.com, 2022.
4. Gilles Audemard, Belaid Benhamou, and Laurent 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.
5. Gilles Audemard and Laurent Henocque. The eXtended least number heuristic. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 427–442, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-45744-5_35.
6. Belaid Benhamou and Laurent 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.
7. R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, 1994. doi:10.1109/SFCS.1994.365680.
8. Thierry Boy de la Tour and Prakash Countcham. An isomorph-free SEM-like enumeration of models. *Electronic Notes in Theoretical Computer Science*, 125(2):91–113, 2005. Proceedings of the 5th International Workshop on Strategies in Automated Deduction (Strategies 2004). URL: https://www.sciencedirect.com/science/article/pii/S1571066105000976, doi:10.1016/j.entcs.2005.01.003.
9. Stanley Burris and Hanamantagouda P. Sankappanavar. *A course in universal algebra*, volume 78 of *Graduate texts in mathematics*. Springer, New York, NY, 1981.
10. Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP*, volume 5732, pages 226–241. Springer, 2009. doi:10.1007/978-3-642-04244-7_20.
11. Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
12. James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 148–159, San Francisco, CA, 1996. Morgan Kaufmann.
13. 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.
14. Andreas Distler, Christopher Jefferson, Tom Kelsey, and Lars Kotthoff. The semigroups of order 10. In Michela 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.
15. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.1*, 2021. URL: https://www.gap-system.org.
16. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo 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.

17 Ian P. Gent, Ian Miguel, Peter Nightingale, Ciaran McCreesh, Patrick Prosser, Neil C. A. Moore, and Chris Unsworth. A review of literature on parallel constraint solving. *Theory Pract. Log. Program.*, 18(5-6):725–758, 2018. doi:10.1017/S1471068418000340.

18 Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In David Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2010. doi:10.1007/978-3-642-15396-9_22.

19 Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *J. Satisf. Boolean Model. Comput.*, 6(4):245–262, 2009. doi:10.3233/sat190070.

20 Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren. Tractable symmetry breaking for csps with interchangeable values. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 277–284. Morgan Kaufmann, 2003. URL: http://ijcai.org/Proceedings/03/Papers/041.pdf.

21 Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC, Revised Selected Papers*, volume 7261, pages 50–65. Springer, 2011. doi:10.1007/978-3-642-34188-5_8.

22 Antti E. J. Hyvärinen, Matteo Marescotti, and Natasha Sharygina. Lookahead in partitioning SMT. In *Formal Methods in Computer Aided Design, FMCAD*, pages 271–279. IEEE, 2021. doi:10.34727/2021/isbn.978-3-85448-046-4_37.

23 Xiangxue Jia and Jian Zhang. A powerful technique to eliminate isomorphism in finite model search. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 318–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

24 Tommi Junttila, Matti Karppa, Petteri Kaski, and Jukka Kohonen. An adaptive prefix-assignment technique for symmetry reduction. *Journal of Symbolic Computation*, 99:21–49, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0747717119300288, doi:10.1016/j.jsc.2019.03.002.

25 Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of SAT solvers. *J. Autom. Reason.*, 34(1):73–101, 2005. doi:10.1007/s10817-005-1970-7.

26 Majid Ali Khan. Efficient enumeration of higher order algebraic structures. *IEEE Access*, 8:41309–41324, 2020. doi:10.1109/ACCESS.2020.2976876.

27 Lars Kotthoff and Neil C. A. Moore. Distributed solving through model splitting. *ArXiv*, abs/1008.4328, 2010.

28 Kenneth Kunen. The structure of conjugacy closed loops. *Transactions of the American Mathematical Society*, 352(6):2889–2911, 2000.

29 Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. Embarrassingly parallel search in constraint programming. *J. Artif. Intell. Res.*, 57:421–464, 2016. doi:10.1613/jair.5247.

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

31 William McCune. Mace4 reference manual and guide, August 2003. URL: https://www.cs.unm.edu/~mccune/prover9/mace4.pdf.

32 Gábor Nagy and Petr Vojtěchovský. LOOPS, computing with quasigroups and loops in GAP, Version 3.4.1. https://gap-packages.github.io/loops/, Nov 2018. Refereed GAP package.

33 Morten Nielsen. Parallel search in gecode. *Technical Report, Gecode*, 2006.

34 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial*

Intelligence, 251:35–61, 2017. URL: https://www.sciencedirect.com/science/article/pii/S0004370217300747, doi:10.1016/j.artint.2017.07.001.

35 Anthony Palmieri, Jean-Charles Régin, and Pierre Schaus. Parallel strategies selection. *CoRR*, abs/1604.06484, 2016. URL: http://arxiv.org/abs/1604.06484, arXiv:1604.06484.

36 Giles Reger, Martin Riener, and Martin Suda. Symmetry avoidance in MACE-style finite model finding. In Andreas Herzig and Andrei 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.

37 Jean-Charles Régin and Arnaud Malapert. Parallel constraint programming. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 337–379. Springer, 2018. doi:10.1007/978-3-319-63516-3_9.

38 Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 596–610, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40627-0_45.

39 Neil J. A. Sloane and The OEIS Foundation Inc. The on-line encyclopedia of integer sequences, 2020. URL: http://oeis.org/?language=english.

40 Jon Sneyers, Peter van Weert, Tom Schrijvers, and Leslie de Koninck. As time goes by: Constraint handling rules: A survey of chr research from 1998 to 2007. *Theory and Practice of Logic Programming*, 10(1):1–47, 2010. doi:10.1017/S1471068409990123.

41 Toby Walsh. Symmetry breaking constraints: Recent results. In Jörg Hoffmann and Bart 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.

42 H. Zhang. Combinatorial designs by SAT solvers. *Handbook of Satisfiability*, pages 533–568, 2009. URL: https://cir.nii.ac.jp/crid/1571980076163512448.

43 Hantao Zhang and Jian Zhang. MACE4 and SEM: A comparison of finite model generators. In Maria Paola Bonacina and Mark 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.

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

45 Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In *IJCAI*, pages 298–303, 1995. URL: http://ijcai.org/Proceedings/95-1/Papers/039.pdf.

CREAM: A PACKAGE TO COMPUTE [AUTO, ENDO, ISO, MONO, EPI]-MORPHISMS, CONGRUENCES, DIVISORS AND MORE FOR ALGEBRAS OF TYPE $(2^n, 1^n)$

JOÃO ARAÚJO, RUI BARRADAS PEREIRA, WOLFRAM BENTZ, CHOIWAH CHOW, JOÃO RAMIRES, LUIS SEQUEIRA, AND CARLOS SOUSA

ABSTRACT. The CREAM GAP package computes automorphisms, congruences, endomorphisms and subalgebras of algebras with an arbitrary number of binary and unary operations; it also decides if between two such algebras there exists a monomorphism, an epimorphism, an isomorphism or if one is a divisor of the other. Thus it finds those objects for almost all algebras used in practice (groups, quasigroups in their various signatures, semigroups possibly with many unary operations, fields, semi-rings, quandles, logic algebras, etc).

As a one-size-fits-all package, it only relies on universal algebra theorems, without taking advantage of specific theorems about, eg, groups or semigroups to reduce the search space. Canon and Holt produced very fast code to compute automorphisms of groups that outperform CREAM on orders larger than 128. Similarly, Mitchell et al. take advantage of deep theorems to compute automorphisms and congruences of completely 0-simple semigroups in a very efficient manner. However these domains (groups of order above 128 and completely 0-simple semigroups) are among the very few examples of GAP code faster than our general purpose package CREAM. For the overwhelming majority of other classes of algebras, either ours is the first code computing the above mentioned objects, or the existing algorithms are outperformed by CREAM, in some cases by several orders of magnitude.

To get this performance, CREAM uses a mixture of universal algebra algorithms together with GAP coupled with artificial intelligence theorem proving tools (AITP) and very delicate C implementations. As an example of the latter, we re-implement Freese's very clever algorithm for computing congruences in universal algebras, in a way that outperforms all other known implementations.

1. INTRODUCTION

Investigation of automorphism groups of mathematical structures is one of the classical algebraic problems. A cornerstone was the work of Evariste Galois, but its impact goes far beyond. In the words of P. J. Cameron [22]:

> *In the famous Erlanger Programme, Klein proposed that geometry is the study of symmetry; more precisely, the geometric properties of an object are those which are invariant under all automorphisms of the objects. (. . .) According to Artin, "the investigation of symmetries of a given mathematical structure has always yielded the most powerful results."*

These ideas, expressed by Klein in 1872 [36] and by Artin in 1957 [18], give some insight on why the computation of automorphisms of various mathematical structures has been such an attractive topic for so many

MATHEMATICS DEPARTMENT, FACULTY OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE NOVA DE LISBOA, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
MATHEMATICS DEPARTMENT, UNIVERSIDADE DE LISBOA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
E-mail addresses: jj.araujo@fct.unl.pt, rmbper@gmail.com, wolfram.bentz@uab.pt, choiwah.chow@gmail.com, joao.j.ramires@gmail.com, lfsequeira@fc.ul.pt, cfmsousa@sapo.pt.

decades. It should be observed here that this effort is closely linked to an omnipresent problem in mathematics – stated explicitly by Ulam [60] – of describing the mathematical objects whose endomorphisms encode *all the relevant information* about them:

$$\mathrm{End}(A) \cong \mathrm{End}(B) \Rightarrow A \cong B.$$

The connection between endomorphisms and congruences is well known, but the importance of congruences goes far beyond a tool to get endomorphisms; in fact their importance in modern algebra can hardly be overestimated.

Congruences have close ties to the structure of algebras - for example, the subdirectly irreducible algebras are precisely the ones that possess a unique minimal nontrivial congruence; direct irreducibility is also characterized by properties of congruences. In Universal Algebra, a variety is a class of algebras of the same type, that satisfy a certain set of identities, or equivalently, by a celebrated theorem of Garrett Birkhoff (see [21, Theorem II.11.9]), a class closed under taking subalgebras, homomorphic images and direct products. Each variety is also generated by its subdirectly irreducible algebras. Properties of congruence lattices of algebras over a variety are closely related to the identities satisfied in that variety. This gave rise to what is usually referred to the theory of Maltsev conditions - see, for example, [31]. The commutator of normal subgroups was generalized for algebras, first those in a congruence-permutable variety [55], then for any algebra in a congruence modular variety [33]; see [29]. Thus notions like Abelianness or nilpotency can be applied meaningfully in a much wider context, and many fruitful consequences have been derived, for example in the context of natural dualities [24]. Tame Congruence Theory [34] emerged as a powerful tool to study the structure of finite algebras, and continues to provide a big leverage in the study of locally finite varieties. These examples point to the critical importance of the study of congruences in algebra.

As observed above, congruences are closely related to endomorphisms and the latter form a monoid (semigroup with identity) that very often encodes very relevant information about the original object and can be investigated with the increasingly powerful techniques developed by experts in semigroup theory (especially since they started to massively apply the classification of finite simple groups [1, 3–8, 10–12, 19, 49, 51, 54, 57]).

Everything said above should make the case in favour of a general centralized and very effective GAP tool that computes congruences, automorphisms and endomorphisms of general algebras of type $(2^m, 1^n)$. We chose these since the overwhelming majority of algebras used in practice have this type: it is a very rare occurrence that an algebraic structure actually uses ternary or higher arity operations (see [21]).

The problem is that computing [endo]automorphisms/congruences of general algebras is a difficult task and hence the majority of existing tools target specific classes (groups, semigroups, quasigroups, etc.) in order to take advantage of the domain's theorems. Probably the best example is the GAP code to find automorphisms of groups, devised by J. J. Cannon and D. Holt [23], that takes advantage of many deep theorems and builds upon many past computational optimizations to quickly get auxiliary objects. In contrast, since we want our algorithms to apply for general algebras of type $(2^m, 1^n)$, we cannot rely on theorems that only hold in very specific cases. The challenge of this project is thus to provide an effective tool to compute the objects for finite algebras of type $(2^m, 1^n)$ that works for all algebras of this type and does not fall far behind the more dedicated tools that take advantage of very specific domain theorems.

The final result is that CREAM compares favourably even with the specialized Cannon and Holt tool referred to above: for groups up to order 128, our general purpose tool finds the automorphisms of a group faster than their very optimized code. For most other classes of algebras, our tool outperforms specialized code, in some cases by large margins. This was achieved by a combination of results and ideas from different parts of computational algebra, artificial intelligence theorem proving, and optimized C implementations. Regarding congruences, we take advantage of a new implementation of Freese's algorithm [28], an algorithm that stands out by its quality and generality; for automorphisms, after many different tests and approaches, the most effective way turned out to be a generalization of the ideas developed in [48]. Once possessing a very good tool to compute automorphisms, finding [mono,epi,iso]-morphisms is straightforward.

Finally, our tool also handles divisors. Given two algebras, A and B, we say that B divides A if B is a homomorphic image of a subalgebra of A. In different words, B divides A if there is a subalgebra C of A and a congruence ρ of C such that C/ρ is isomorphic to B. Like congruences, divisors are very important tools. For example, the celebrated Krohn–Rhodes theorem states that every finite semigroup S divides a wreath product of finite simple groups, each dividing S, together with copies of the 3-element monoid $\{1, a, b\}$, where $ba = aa = a$, $ab = bb = b$ (for details see [53]). The key difficulty when finding divisors is to find the subalgebras of a given algebra; CREAM has an algorithm to compute them.

There are numerous papers concerning the automorphism groups of particular classes of algebras, for example, Schreier [52] and Mal'cev [42] described all automorphisms of the semigroup of all mappings from a set to itself. Similar results have been obtained for various other structures such as orders, equivalence relations, graphs, and hypergraphs; see the survey papers [46] and [47]. More examples are provided, among others, by Gluskĭn [32], Araújo and Konieczny [14], [15], and [16], Fitzpatrick and Symons [26], Levi [38] and [39], Liber [40], Magill [41], Schein [50], Sullivan [58], and Šutov [59].

Fast algorithms exist to find the congruences and automorphisms of groups and semigroups. For example, with GAP [30], the package SEMIGROUPS includes the function CongruencesOfSemigroup that returns the congruences of a given semigroup. This function only works for a limited set of semigroups, belonging to the classes Simple, Brandt, Group, Zero Simple or Rectangular Band, and only for Rees Matrix semigroups and Rees Zero Matrix semigroups is it highly efficient. On a different platform, UACalc [27] supports a wider range of algebras. UACalc also implements the Freeze algorithm that is the basis for the calculation of congruences described here.

Regarding automorphisms in GAP, many special properties of groups are used to implement the function *AutomorphismGroup* to efficiently find the automorphism group of a given group. However, this specialized method, while extremely efficient, works only on groups and not on any other more general algebraic structures, such as quasigroups, semigroups and magmas. Likewise, the Loops package [48] in GAP provides another version of the function *AutomorphismGroup* to compute all the automorphisms of a given quasigroup.

To date, there are no known implementations of general functions for finding divisors, congruences and [endo, auto, epi, mono, iso]-morphisms of magmas or algebras of type $(2^m, 1^n)$ in GAP. To fill the void, the CREAM package implements such functions in GAP with part of the code written in C for performance.

This article is composed of 6 sections: this introduction, a short description of the mathematical concepts and background, the description of the algorithms used in the CREAM package, a comparative discussion of the performance of the package, a list of applications of the package and finally a short conclusion.

2. Mathematical Background

2.1. **Algebra of Type (2^m,1^n).** An Algebra in the sense of Universal Algebra is an algebraic structure consisting of a set A together with a collection of operations on A. (If we want to be more precise, an algebra also contains an indexed scheme that references the operations, but we are not going to enter those technical details; for a complete definition see [21].)

An n-ary operation on A is a function that takes an n-tuple of elements from A and returns a single element of A, that is, a function from A^n to A. The number n is called the *arity* of the operation. For the scope of this package we are only considering operations with arity 1 or 2, i.e., unary and binary operations. An algebra of type (2^m,1^n) is a universal algebra with m binary and n unary operations.

The package represents a finite Universal Algebra as a list of operations. An operation of arity 1 is represented by a vector, while an operation of arity 2 is represented by a square matrix. The underlying set A is implicitly specified by the vector or matrix sizes, which need to agree for a valid representation. If this size is d, the algebra is defined on the set $A = \{1, \ldots, d\}$. We can safely ignore the ambiguity this introduces in the (uninteresting) case of an algebra without operations. Each vector or matrix describes the corresponding operation by listing all images in the obvious way. The following is an example of a representation for an algebra with a unary and a binary operation.

```
[ [3, 1, 2], [ [1, 2, 3], [2, 3, 1], [3, 1, 2] ] ]
```

For those CREAM functions that involve more than one algebra, the operations of the algebras need to be aligned by a common index scheme. In our representation, this index is indirectly provided by the order in which the operations are listed. The involved algebras need to be compatible, that is, they have the same number of operations of each arity, and the operations are listed in the same position in the algebra representation.

2.2. **Partition.** A partition of a set A is a collection of non-empty subsets of A, such that every element of A is included in exactly one subset [21]. Each subset in a partition is called a block or part. For example, $\{\{1,3\},\{2,4\}\}$ is a 2-blocks partition of the set $\{1,2,3,4\}$.

A block of a partition is efficiently represented as a tree. A partition is represented by a forest, that is, a disjoint union of trees. The forest is physically presented in the computer memory by an array, whose size will be the size of the set A.

In a tree, each node has a parent node, except for the top node or root. In the array representation, each position of the array represents a node and the value of the node points to its parent node. Top nodes should have a value indicating that the node is the root of the tree. A negative number is used to signal the root, and the absolute value of this number is the number of elements of the block.

A shallow tree is a tree in which all non-root nodes are connected directly to the root. To obtain a unique representation, we adopt the convention that the trees used in presenting a partition are shallow, and that the smallest value in a block will be the root node. A partition representation that respects the above conventions will be called a normalized (representation of a) partition.

Thus, for $l, m \geq 1$, the array

$$\begin{array}{ccccccc} [& \ldots & , -l, & \ldots & , m, & \ldots &] \\ & & \uparrow & & \uparrow & & \\ & & \text{position } i & & \text{position } k & & \end{array}$$

means that i is the root of a block (as its entry is negative) of size l; in addition the element k belongs to the block whose root is m.

Using these conventions, the encoding of the partition `[[1], [2], [3], [4], [5], [6]]` induced by the identity relation is

```
[-1, -1, -1, -1, -1, -1]
```

while the partition `[[1, 2, 3, 4, 5, 6]]` with just one block is encoded as

```
[-6, 1, 1, 1, 1, 1]
```

Other examples are given below.

partition	encoded as
`[[1, 6], [2], [3, 5], [4]]`	`[-2, -1, -2, -1, 3, 1]`
`[[1, 3, 5], [2, 6], [4]]`	`[-3, -2, 1, -1, 1, 2]`
`[[1, 2, 5, 6], [3, 4]]`	`[-4, 1, -2, 3, 1, 1]`

This non-intuitive way of representing partitions will prove its usefulness later on.

2.3. **Congruences.** A congruence of an algebra is an equivalence relation on its underlying set that is compatible with all algebraic operations [21].

Technically, a congruence relation is an equivalence relation $\equiv$ on an algebra that satisfies $\mu(a_1, a_2, ..., a_n) \equiv \mu(a'_1, a'_2, ..., a'_n)$ for every n-ary operation μ and all elements $a_1, ..., a_n, a'_1, ..., a'_n$ such that $a_i \equiv a'_i$ for each $i = 1, ..., n$.

Congruences will be represented in the same way as their corresponding partitions, as detailed in the previous section.

2.4. **Homomorphism, Endomorphism, Isomorphism and Automorphism.** If A, B are two algebras of the same type, then a function $f : A \to B$ is a *homomorphism* from A to B if $\mu(f(a_1), f(a_2), ..., f(a_n)) = f(\mu(a_1, a_2, ..., a_n))$ for every n-ary operation μ and $a_1, \ldots a_n \in A$ (more precisely, μ here stands for the two operations of A and B that are indexed equally by the (common) index scheme).

An *endomorphism* is a *homomorphism* from an algebra to itself, a *monomorphism* is an injective *homomorphism*, while an *isomorphism* is a bijective *homomorphism*. An *automorphism* is a *homomorphism* that is both an *isomorphism* and an *endomorphism*.

3. The Algorithms

3.1. **Congruences algorithm.** The starting point for the computation of congruences are the algorithms described in Freese [28] to calculate the smallest congruence containing a given partition Θ of a finite algebra A; in particular this is used to compute the smallest congruence containing a pair of elements $(a, b) \in A \times A$, called the *principal congruence* generated by $\{a, b\}$. From this base algorithm all congruences of the algebra A are generated in an efficient way. Optimizations to the original algorithm were introduced taking into account that we only allow operations of arity at most 2. In addition, we take advantage of C to get a faster implementation than the original implementations made by Freese and his collaborators. Finally, our implementation is integrated with GAP and hence fully compatible with its other resources.

3.1.1. *Partition Functions.* There are several partition functions that play an important role in the algorithm to compute principal congruences.

The function **CreamRootBlock** is an operation that returns the root node for a node i. The root node of a node is itself if the value of the node representation is negative. If the value of the node representation points to a different node then that node is the parent node. This algorithm could be run recursively until reaching the root node but is implemented iteratively for better performance. This operation will work both for normalized and non-normalized partitions.

Before returning, the node parent of i is set to the found root node in order to make the tree representing the partition as shallow as possible, thus avoiding that in future calls the algorithm needs to transverse several nodes to reach the root node.

Algorithm 1 CreamRootBlock (i, partition)

$j \leftarrow i$
while $partition[j] \geq 0$ **do**
  $j \leftarrow partition[j]$
end while
if $i \neq j$ **then**
  $partition[i] \leftarrow j$
end if
return j

The function **CreamJoinBlocks** is an operation that joins the blocks containing given elements x and y. This operation will work both for normalized and non-normalized partitions. The resulting partition may not in general be normalized even if the original partition is.

In order to keep the tree representing the partition as shallow as possible, the root node of the merged block will be the root node of the larger original block.

Algorithm 2 CreamJoinBlocks (x, y, partition)

```
r ← CreamRootBlock(x, partition)
s ← CreamRootBlock(y, partition)
if r ≠ s then
  if partition[r] < partition[s] then
    partition[r] ← partition[r] + partition[s]
    partition[s] ← r
  else
    partition[s] ← partition[r] + partition[s]
    partition[r] ← s
  end if
end if
```

The **CreamNumberOfBlocks** function returns the number of blocks of a partition. Given the encoding of partitions we simply count the number of positions of the array that have negative values.

Algorithm 3 CreamNumberOfBlocks (partition)

```
nblocks ← 0
dimension ← ArraySize(partition)
for i = 1 to dimension do
  if partition[i] < 0 then
    nblocks ← nblocks + 1
  end if
end for
return nblocks
```

The **CreamNormalizePartition** function normalizes the partition by making it shallow and having the smallest element of each block the root node.

Algorithm 4 CreamNormalizePartition (partition)

```
dimension ← ArraySize(partition)
for i = 1 to dimension do
  r ← CreamRootBlock(i, partition)
  if r ≥ i then
    partition[i] ← −1
    if r > i then
      partition[r] ← i
    end if
  else
    partition[r] ← partition[r] − 1
  end if
end for
```

The **CreamJoinPartition** function computes the join of two partitions, i.e. the smallest partition containing both input partitions.

Algorithm 5 CreamJoinPartition (partition1, partition2)

$dimension \leftarrow ArraySize(partition1)$
for $i = 1$ to $dimension$ **do**
  CreamJoinBlocks(i,CreamRootBlock (i,$partition1$),$partition2$)
end for

The **CreamComparePartitions** function compares normalized partitions. Returns 0 if the partitions are equal, -1 if partition1 $>$ partition2, and 1 if partition1 $<$ partition2. Partitions are ordered by the order of the underlying list. The purpose of this function is to allow binary search in sets of partitions.

Algorithm 6 CreamComparePartitions (partition1, partition2)

$dimension \leftarrow Length(partition1)$
for $i = 1$ to $dimension - 1$ **do**
  if $partition1[i] > partition2[i]$ **then**
    return -1
  end if
  if $partition1[i] < partition2[i]$ **then**
    return 1
  end if
end for
return 0

3.1.2. *Computing Principal Congruences.* The base algorithm in [28] computes the smallest congruence containing Θ, a partition of a finite algebra A. The simplest case is when Θ only contains one nontrivial block and this block has only two elements.

The algorithm only works with unary operations. Since our algebras have binary operations, we have to convert them to a family of unary operations. In this case a binary operation $f(x, y)$ can be converted into $2n$ unary operations (where n is the size of the algebra) by assigning to each element $x \in A$ the unary operations c_x and r_x, which are induced by the corresponding column and row in the Cayley table of f:

$$c_x(y) = f(y, x) \text{ and } r_x(y) = f(x, y).$$

For example, the algebra:

$$[[[2, 1, 1], [1, 2, 2], [1, 3, 2]]]$$

induces the following unary operations (after removing duplicates):

$$[[2, 1, 1], [1, 2, 2], [1, 3, 2], [1, 2, 3]].$$

The principal congruence algorithm takes as input the algebra and the generating pair of elements.

The algorithm joins the pair of elements in the same block and applies the algebra's unary functions to each element of the pair. In this way, it obtains one additional pair per function that will be joined in the same block, repeating this process until the list of pairs is exhausted.

Algorithm 7 CreamPrincipalCongruence (algebra,InitialPair)

$PairList \leftarrow [InitialPair]$
$partition \leftarrow SingletonPartition()$
$partition \leftarrow CreamJoinBlocks(partition, InitialPair[1], InitialPair[2])$
$NFuncs \leftarrow ArraySize(algebra)$
while $PairList \neq empty$ **do**
  $Pair \leftarrow Pop(PairList)$
  for $i = 1$ to $NFuncs$ **do**
    $f \leftarrow algebra(i)$
    $r \leftarrow CreamRootBlock(partition, f(Pair[1]))$
    $s \leftarrow CreamRootBlock(partition, f(Pair[2]))$
    if $r \neq s$ **then**
      $partition \leftarrow CreamJoinBlocks(partition, r, s)$
      $PairList \leftarrow Push(PairList, [r, s])$
    end if
  end for
end while
return $partition$

As described in [28] this algorithm is very efficient showing a moderate growth of execution time with n. Apart from the algorithm itself, several implementation choices were used to increase efficiency as well.

One of these aspects is the use of arrays to encode partitions which allow for a constant and very fast random access to the partition element.

This is combined with the balancing and collapsing of the trees representing the blocks in the partition that are parts of the **CreamJoinBlocks** and **CreamRootBlock** algorithms. This approach aims for trees that are as shallow as possible during the execution of the **CreamPrincipalCongruence** algorithm.

In **CreamJoinBlocks** the tree is balanced by keeping as root of the joined block the root of the bigger original block.

In **CreamRootBlock** the tree is collapsed by setting the parent node of the element on which the function is called equal to its return value, avoiding having to traverse several nodes of the tree in future calls.

Both of these implementation details allow for shallower trees representing the blocks, which will make **CreamRootBlock** faster, given that fewer nodes will have to be transversed to determine the root node of a node.

This is especially important since **CreamRootBlock** is the most called function when calculating a principal congruence of an algebra.

To compute all the principal congruences this function is called for all pairs of elements of A.

Algorithm 8 CreamAllPrincipalCongruences (algebra)

$dimension \leftarrow SizeAlgebra(algebra)$
$allPrincipalCongruences \leftarrow []$
for $i = 1$ to $dimension - 1$ **do**
  for $j = i + 1$ to $dimension$ **do**
    $congruence \leftarrow CreamPrincipalCongruence(algebra, [i, j])$
    $AddCongruence(allPrincipalCongruences, congruence)$
  end for
end for
return $allPrincipalCongruences$

3.1.3. *Calculating All Congruences.* It is known that the congruences of an algebra form a lattice with the usual set inclusion partial order. The meet is the usual intersection; the join of two congruences is the smallest congruence containing both.

The minimal elements of this lattice are precisely the principal congruences, and we can obtain all congruences by computing joins, starting with the principal congruences.

The key part is to find an efficient way to combine minimal congruences to get all congruences of the algebra. The principal congruences are stored in an ordered set and the congruences are combined from start to end by the join operation. Each congruence resulting from these joins will be added to the ordered set (eliminating duplicates).

Algorithm 9 CreamAllCongruences (algebra)

$allPrincipalCongruences \leftarrow CreamAllPrincipalCongruences(algebra)$
$allCongruences \leftarrow allPrincipalCongruences$
$i \leftarrow 1$
while $i < Size(allCongruences)$ **do**
  for $j = 1$ to $Size(allPrincipalCongruences)$ **do**
    if $\neg isContained(allPrincipalCongruences[j], allCongruences[i])$ **then**
      $congruence \leftarrow JoinPartition(allCongruences[i], allPrincipalCongruences[j])$
      $AddCongruence(allCongruences, congruence)$
    end if
  end for
  $i \leftarrow i + 1$
end while
return $allCongruences$

The join operation is only called if the principal congruence is not contained in the congruence that is being joined with. This is optimized by preserving the information about which principal partitions were joined.

3.1.4. *Calculating Monolithic Algebras.* Monolithic Algebras are algebras that have a single minimal congruence that is contained in every other congruence except for the identity congruence.

This is simple to calculate having the list of minimal congruences and a function that compares two partitions and calculates whether one is contained in the other.

The **CreamContainedPartition** function returns whether a partition is contained in another. This function is used internally to determine whether an Algebra is monolithic or not. This algorithm assumes that the partitions are normalized.

Algorithm 10 CreamContainedPartition (partition1, partition2)

```
dimension = ArraySize(partition1)
for i = 1 to dimension do
  if partition1[i] < 0 then
    if partition2[i] < 0 then
      block2 ← i
    else
      block2 ← partition2[i]
    end if
    for j = i + 1 to dimension do
      if partition1[j] = i and not block2 = partition2[j] then
        return false
      end if
    end for
  end if
end for
return true
```

We calculate the principal congruences and compare them eliminating those that contain other congruences. If a single congruence is returned then this congruence is contained in every other congruence (and the algebra is monolithic). This is done with the **CreamIsAlgebraMonolithic** function.

Algorithm 11 CreamIsAlgebraMonolithic (algebra)

```
partitions ← CreamAllPrincipalCongruences(algebra)
npartitions ← Size(partitions)
if npartitions > 1 then
  i ← 2
  repeat
    if CreamContainedPartition(partitions[1], partitions[i]) then
      Remove(partitions, i)
    else if CreamContainedPartition(partitions[i], partitions[1]) then
      partitions[1] ← partitions[i]
      Remove(partitions, i)
      i ← 2
    else
      i ← i + 1
    end if
  until Size (partitions) < i
end if
npartitions ← Size(partitions)
if npartitions > 1 then
  return false
else
  return true
end if
```

3.2. **Automorphism algorithm.** The automorphisms of an algebra A of type (2^m,1^n) can be derived from the calculation of the automorphisms of a set of algebras each containing only one binary operation of

A. The automorphism group of the algebra will be the intersection of all the automorphism groups of these magmas that also commute, function composition-wise, with all the unary operations of the original algebra.

In general, it is difficult to obtain automorphisms efficiently. The authors of the Loops GAP package [48] used an idea that we could apply to general magmas. With some adaptations this allows for very effective computing of the automorphisms of magmas.

3.2.1. *Invariant Vector.* The general idea is to pick a list of properties invariant under homomorphisms and then partition the algebras using these properties. For example, if e is idempotent, then any endomorphism must map e onto an idempotent. This is the basis of the *Discriminator* in the Loops package, which implemented nine such invariants for the domain elements of quasigroups. However, most of these invariants cannot be carried over directly to magmas. We devised a total of seventeen invariants that hold for any magma. In general, unless otherwise said, given an element x in a Magma M and $k \in \mathbb{Z}^+$, we define x^k to be $x^{k-1} * x$ for $k \geq 2$ and $x^1 = x$ (ie. we associate on the left). For each element p in a magma M, CREAM computes the following:

(1) Smallest k such that $p^k = p^n$, with $n > k > 1$.
(2) Number of domain elements for which p is a left identity.
(3) Number of domain elements for which p is a right identity.
(4) Number of elements y such that $p = (py)p$.
(5) Number of distinct elements in row p of the multiplication table.
(6) Number of distinct elements in column p of the multiplication table.
(7) 1 if p is an idempotent, 0 otherwise.
(8) Number of idempotents in column p of the multiplication table.
(9) Number of idempotents in row p of the multiplication table.
(10) 1 if the equality $p(pp) = (pp)p$ holds, 0 otherwise.
(11) Number of domain elements that commute with p.
(12) Number of domain elements s for which $(ss)p = p(ss)$.
(13) Number of domain elements s such that $s^2 = p$.
(14) Number of domain elements s satisfying $p(ps) = (pp)s$
(15) Number of multisets $\{x, y\}$ with $xy = yx = p$.
(16) Number of elements $t \in M$ such that for two idempotents $e, f \in M$, we have $p = et = tf$.
(17) Number of elements $t \in M$ for which there exist two elements $x, y \in M$ such that $p = xy$ and $t = yx$.

Many of these invariants have an algebraic meaning. For example, (17) is the size of the conjugacy class of p when the magma is a group; (16) is the size of the filter generated by p when the magma is a regular semigroup; (13) is the number of square roots of p; (11) is the size of the centralizer of p; (8) and (9) are the number of idempotents in the left or right ideals generated by p; (5) and (6) are the number of elements in the left or right ideals generated by p; (1) deals with the periodicity of p; etc.

The intersection of the blocks induced by these invariants partitions the elements of the algebra, and by a straightforward argument on first order logic any endomorphism must preserve the blocks of this partition.

3.2.2. *Generating Set.* We can cut down the size of the search tree by focusing only on the images of the elements in a generating set. Some generating sets may be more suitable for finding automorphisms than others:

- A smaller generating set is preferred because it would require fewer trials in constructing an isomorphism.
- Generators from partition blocks with fewer elements are preferred since any homomorphism must send the elements of a block into itself.

Our algorithm satisfies both the constraints above and efficiently produces a generating set. It is depicted in Algorithm 12.

Algorithm 12 Constructing Efficient Generating Set

$submagma \leftarrow [\,]$
$generator \leftarrow [\,]$
$candidates \leftarrow$ magma elements sorted by ascending block size
while $submagma \neq magma$ **do**
  $generator \leftarrow generator \cup$ the element that increase the size of $submagma$ most, and if 2 elements increase the size by the same amount, take the one from the smaller block.
  $submagma \leftarrow$ submagma generated by $generator$
  $candidates \leftarrow candidates$ with elements in submagma removed
end while

3.2.3. *The Automorphism Algorithm.* To find the automorphisms we start by partitioning the Magma according to the invariant vectors of each element and then obtain an efficient generating set. We then need to find the images of the generating set taking into account that each element in the generating set can only map to elements having the same invariant vector. Then we extend this partial map to a full map and check that the homomorphism condition is satisfied.

We tested the algorithm on groups, loops, and quasigroups, obtaining the same results as yielded by the existing tools. For magmas of orders 2 and 3, and semigroups of orders 6 and 7 (as there are no general tools to compute automorphisms) we double check our results against the output of Mace4 [44].

3.2.4. *Algebras of Type (2^m,1^n).* Given an algebra of type (2^m,1^n) its automorphism group can be computed by taking the intersection of all the automorphism groups of the binary operations that also commute, function composition-wise, with all the unary operations.

A useful observation is that the intersection of the trivial group and any group is the trivial group. Therefore, if any of the automorphism group generated in the process is a trivial group, then the search can be terminated with the trivial group declared the automorphism group for the algebra.

This algorithm is implemented in the **CreamAutomorphisms** function in the CREAM package, as shown in Algorithm 13.

Algorithm 13 CreamAutomorphisms(algebra)

$binaryOperations \leftarrow AllBinaryOperations(algebra)$
$unaryOperations \leftarrow AllUnaryOperations(algebra)$
$D \leftarrow \emptyset$
repeat
  $B \leftarrow$ Pop a binary operation from $binaryOperations$
  $C \leftarrow$ all automorphisms of B
  $C \leftarrow$ all automorphisms in C that commute with every unary operation in $unaryOperations$
  if $C = \emptyset$ **then**
    return $\emptyset$
  end if
  $D \leftarrow D \cup C$
until $binaryOperations = \emptyset$
return Intersection(D)

3.3. **Endomorphisms algorithm.** The classic approach to calculate endomorphisms would be to use **Mace4** to search for endomorphisms and while this is very effective and fast for low order algebra further optimizations are needed for high order algebras in which the congruences of the algebra can be used to limit the **Mace4** search space for endomorphisms.

To compute all endomorphisms of high order algebras, CREAM does the following:

(1) Compute the congruences of the algebra A;
(2) For each congruence R, compute A/R (except for the trivial congruence);
(3) Compute the subalgebras of A that are isomorphic to A/R (using the finite model builder Mace4);
(4) For each compatible pair subalgebra/congruence derive a corresponding endomorphism;
(5) Add the automorphisms of A.

The congruences of A are calculated with the algorithms described in 3.1. For each congruence R (except for the trivial congruence that will be dealt with later), a representation of the A/R operation can be obtained by replacing the values in A's operation tables by their roots, and eliminating rows and columns not corresponding to roots. If we have the algebra:

```
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],[ 1, 1, 1, 2, 3, 1 ] ] ]
```

and the congruence

```
[ [ 1, 2, 3], [4], [5, 6] ] - [ -3, 1, 1, -1, -2, 5 ],
```

then the A/R operation is:

	1	4	5
1	1	1	1
4	1	1	1
5	1	1	1

and passed to Mace4 as follows:

```
f(a0,a0)=a0.
a0!=a3.
f(a0,a3)=a0.
a0!=a4.
f(a0,a4)=a0.
f(a3,a0)=a0.
f(a3,a3)=a0.
a3!=a4.
f(a3,a4)=a0.
f(a4,a0)=a0.
f(a4,a3)=a0.
f(a4,a4)=a0.
```

where f is the binary operation of the algebra. It is worth observing that Mace4 is zero-based, unlike GAP, which is one-based.

Running Mace4 with the definition of the algebra operation and the above encoding of A/R as assumptions we get 16 possible models each one of them corresponding to a different endomorphism of the algebra A. One of these models would be in Mace4's output:

```
interpretation( 6, [number = 1,seconds = 0], [
    function(a0, [0]),
    function(a3, [1]),
    function(a4, [2]),
    function(fa1(_,_), [
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,1,2,0])]).
```

From this model we get the partial mapping $f : \{1,4,5\} \subseteq A \to \{1,2,3\}$, defined by $f(1) = 1$, $f(4) = 2$ and $f(5) = 3$, that can be represented by $[1,,,2,3,]$. The gaps can be easily filled since elements of the same block must map to the same image. In this example, 2 and 3 have the same image as 1, while 6 and 5 have the same image yielding the endomorphism: $[1,1,1,2,3,3]$.

Repeating this process for all congruences and for all models obtained with Mace4 from each congruence, all the algebra's endomorphisms (except for the automorphisms) can be obtained.

Finally, for the trivial congruence the associated endomorphisms would be also automorphisms and in this case it is more efficient to use the the algorithm described in 3.2 than using Mace4.

3.4. **Monomorphisms algorithms.** Invariants can be used to speed up the process of finding an monomorphism from one magma to another, or from one algebra to another. If f is an injective homomorphism from a magma A to a magma B, then it is a isomorphism from A to B restricted to the range of f. Thus, the same ideas of applying invariants in constructing automorphisms can be used for constructing monomorphisms. Specifically, an monomorphism f can map an element $a \in A$ to $b \in B$ only if a and b have the same invariant vector. This greatly reduces the search space for monomorphisms between A and B.

Algorithm 14 CreamAllMonomorphismMagmas(magma1, magma2)

$genL \leftarrow$ generating set from $magma1$
for i = 1 to $Size(magma2)$ **do**
  $rangeM[i] \leftarrow$ list of elements of $magma2$ that have same invariant vector as $genL[i]$
end for
$monoMaps \leftarrow \emptyset$
for all mappings m from $genL$ to elements of $rangeM$ s.t. $genL[i]$ maps only to elements in $rangeM[i]$ **do**
  if m is an injective homomorphism **then**
    $monoMaps \leftarrow monoMaps \cup m$
  end if
end for
return $monoMaps$

Algorithm 15 CreamAllMonomorphism(algebra1, algebra2)

b1 $\leftarrow$ first binary operation in $algebra1$
b2 $\leftarrow$ first binary operation in $algebra2$
$monoList \leftarrow$ CreamAllMonomorphismMagmas($b1, b2$)
$algebraMonoList \leftarrow []$
for each $monoMap$ in $monoList$ **do**
  $mono \leftarrow true$
  for each corresponding pair of operations $b1 \in algebra1, b2 \in algebra2$ **do**
    if $monoMap$ is not an isomorphism from $b1$ to $b2$ **then**
      $mono \leftarrow false$
      break out of for loop
    end if
  end for
  if $mono$ **then**
    Add $monoMap$ to $algebraMonoList$ if not already in the list
  end if
end for
return $algebraMonoList$

In this section only the function CreamAllMonomorphisms is addressed but the Cream package also includes the functions CreamExistsMonomorphism (returning true if a monomorphism exists) and CreamOneMonomorphism (returning one monomorphism).

3.5. **Epimorphisms algorithm.** CreamAllEpimorphisms(A_1, A_2) returns all epimorphisms from A_1 to A_2, provided the algebras are compatible. The algorithm first finds all congruences φ of A_1 such that A_1/φ and A_2 are isomorphic. For each such φ, the corresponding epimorphisms are obtained by composing the quotient map with an isomorphism to A_2 and all automorphisms of A_2.

For efficiency, a size check is implemented before searching for isomorphisms and the automorphisms of A_2 are only calculated once.

Algorithm 16 CreamAllEpimorphisms(algebra1, algebra2)

$allCongruences \leftarrow CreamAllCongruences(algebra1)$
$dimension1 \leftarrow SizeAlgebra(algebra1)$
$dimension2 \leftarrow SizeAlgebra(algebra2)$
$autoList \leftarrow []$
$epiList \leftarrow []$
for all $cong$ in $allConguences$ **do**
  if $dimension2 = CreamNumberOfBlocks(cong)$ **then**
    $[qalgebra, mapToQalgebra] \leftarrow QuotientAlgebraFromCongruence(algebra1, cong)$
    $iso \leftarrow IsomorphismAlgebras(qalgebra, algebra2)$
    if not $iso = fail$ **then**
      if autoList = [] **then**
        $autoList \leftarrow CreamAutomorphisms(algebra2)$
      end if
      for all $auto$ in $autoList$ **do**
        $epi \leftarrow []$
        for $j = 1$ to $dimension1$ **do**
          $epi[j] \leftarrow auto[iso[mapToQalgebra[i]]]$
        end for
        Add epi to $epiList$ if not already in the list
      end for
    end if
  end if
end for
return $epiList$

In this section only the function CreamAllEpimorphisms is addressed but the Cream package also includes the functions CreamExistsEpimorphism (returning true if a epimorphism exists) and CreamOneEpimorphism (returning one epimorphism).

3.6. **SubUniverses algorithm.** The SubUniverses algorithm returns a list of all underlying sets of all subalgebras of the input algebra. It first generates all 1-generated subalgebras, then iteratively expands each i-generated algebra by adding another generator.

For performance enhancement, these expansions are limited to one element from each orbit of the algebra's automorphism group. At the end of each cycle, the remaining $(i + 1)$-generated subuniverses are obtained by applying the automorphism group. The algorithm stops when an iteration produces only one subuniverse with the same size of the algebra.

The algorithm relies on the SubUniverseFromElement routine, which calculates the subalgebra generated by a subalgebra and an additional element. This routine assumes that the input subuniverse is closed under the algebra's operations. The algorithm adds the element to the updated subuniverse, calculates all directly generated additional elements, and places them in a temporary list. These elements are then filtered for those that are already in the subuniverse, and the remaining elements are put into a second list. Elements from the second list are then added to the subuniverse iteratively.

Algorithm 17 CreamAllSubUniverses(algebra)

$dimension \leftarrow CreamSizeAlgebra(algebra)$
$autoList \leftarrow CreamAutomorphisms(algebra)$
for $i = 1$ to $dimension$ **do**
  $sigma[i] \leftarrow []$
  if $i = 1$ **then**
    $sigmaMinus \leftarrow [[]]$
  else
    $sigmaMinus \leftarrow sigmaExpanded[i-1]$
  end if
  for all $cSigma$ in $sigmaMinus$ **do**
    $elemList \leftarrow [1..dimension]$
    Remove all elements in $cSigma$ from $elemList$
    while $Size(elemList) <> 0$ **do**
      $j \leftarrow elemList[1]$
      $sUniverse \leftarrow SubUniverseFromElement(algebra, cSigma, j)$
      Add $sUniverse$ to $sigma[i]$ if not already in the list
      $autoEs \leftarrow$ the orbit of j under $autoList$
      Remove all elements in $autoEs$ from $elemList$
    end while
  end for
  $sigmaExpanded[i] \leftarrow []$
  for all $sUniverse$ in $sigma[i]$ **do**
    $autoUniverses \leftarrow$ the orbit of $sUniverse$ under $autoList$
    Add all elements of $autoUniverses$ to $sigmaExpanded[i]$ if not already in the list
  end for
  if $Size(sigma[i]) = 1$ and $Size(sigma[i][1]) = dimension$ **then**
    $break$
  end if
end for
$sUniverses \leftarrow \cup_{j=1}^{i} sigmaExpended[j]$
return $sUniverses$

Algorithm 18 SubUniverseFromElement(algebra, initSubUniverse, element)

$dimension \leftarrow CreamSizeAlgebra(algebra)$
$elemList \leftarrow [element]$
$newSubUniverse \leftarrow initSubUniverse$
$nElem \leftarrow Size(elemList)$
while $nElem > 0$ **do**
  $currentElem \leftarrow elemList[nElem]$
  Remove element in position $nElem$ from $elemList$
  $newElemList \leftarrow []$
  for all operations op of the $algebra$ **do**
    if op is binary **then**
      Add $op[currentElem][currentElem]$ to $newElemList$ if not already in the list
      for all $subUniverseElem$ in $newSubUniverse$ **do**
        Add $op[currentElem][subUniverseElem]$ to $newElemList$ if not already in the list
        Add $op[subUniverseElem][currentElem]$ to $newElemList$ if not already in the list
      end for
    else
      Add $op[currentElem]$ to $newElemList$ if not already in the list
    end if
  end for
  Add $currentElem$ to $newSubUniverse$ if not already in the list
  $elemList \leftarrow elemList \cup (newElemList \setminus newSubUniverse)$
  $nElem \leftarrow Size(elements)$
  if $Size(newSubUniverse) + nElem = dimension$ **then**
    $newSubUniverse \leftarrow [1..dimension]$
    $nElem \leftarrow 0$
  end if
end while
return $newSubUniverse$

3.7. **DivisorUniverses algorithm.** CreamAllDivisorUniverses(A_1, A_2) checks if A_1 has a divisor that is isomorphic to A_2, where A_1 and A_2 are compatible algebras. Its return is a list of all pairs (B, φ), such that B is a subuniverse of A_1, φ is a congruence of the subalgebra B, and B/φ is isomorphic to A_2.

The algorithm first calculates the subalgebras of A_1, then their quotients and then checks those for isomorphisms to A_2, in each case using the corresponding algorithms of the Cream package. A size condition is used to prune unnecessary calculations.

Algorithm 19 CreamAllDivisorUniverses(algebra1, algebra2)

$dimension \leftarrow CreamSizeAlgebra(algebra2)$
$subUniverseList \leftarrow CreamSubUniverses(algebra1)$
$divisorList \leftarrow []$
for all $subUniverse$ in $subUniverseList$ **do**
  if $dimension <= Size(subUniverse)$ **then**
    $subAlgebra \leftarrow CreamSubUniverse2Algebra(algebra1, subUniverse)$
    $congList \leftarrow CreamAllCongruences(subAlgebra)$
    $rcongList \leftarrow []$
    for all $cong$ in $congList$ **do**
      if $dimension = CreamNumberOfBlocks(cong)$ **then**
        Add $cong$ to $rcongList$ if not already in the list
      end if
    end for
    for all $cong$ in $rcongList$ **do**
      $[qAlgebra, mapToQalgebra] \leftarrow QuotientAlgebraFromCongruence(subAlgebra, cong)$
      if $CreamAreAlgebrasIsomorphic(qAlgebra, algebra2)$ **then**
        Add $[subUniverse, cong]$ to $divisorList$ if not already in the list
      end if
    end for
  end if
end for
return $divisorList$

In this section only the function CreamAllDivisorUniverses is addressed but the Cream package also includes the functions CreamExistsDivisor (returning true if a divisor between the algebras exists) and CreamOneDivisorUniverse (returning one divisor between the algebras).

3.8. **DirectlyReducible algorithm.** CreamAllDirectlyReducible(A) looks for two (non-trivial) commuting congruences ϕ, ψ of A, whose meet is trivial and whose join is $A \times A$. If those exist, A is isomorphic to the direct product $A/\phi \times A/\psi$.

Because all algebras are finite, it suffices to check that the congruences have a trivial meet, and that the corresponding product algebra has the correct size. The algorithm uses numerical constraints to avoid unnecessary calculations.

The return of the algorithm is a list of pairs of congruences. These pairs should be consider as sets, i.e. if (α, β) is listed the (equally valid) pair (β, α) is not listed separately.

Algorithm 20 CreamAllDirectlyReducible(algebra)

$dimension \leftarrow CreamSizeAlgebra(algebra)$
$congList \leftarrow CreamAllCongruences(algebra)$
$pCongList \leftarrow []$
for all $cong$ in $congList$ **do**
  $cNBlocks \leftarrow CreamNumberOfBlocks(cong)$
  if $cNBlocks <> 1$ and $cNBlocks <> dimension$ and $IsInt(dimension/cNBlocks)$ **then**
    Add $cong$ to $pCongList$
  end if
end for
$pairs \leftarrow []$
for all 2-element subsets $\{i, j\}$ of $pCongsList$ **do**
  $iNBlocks \leftarrow CreamNumberOfBlocks(i)$
  $jNBlocks \leftarrow CreamNumberOfBlocks(j)$
  if $iNBlocks * jNBlocks = dimension$ **then**
    $cMeet \leftarrow MeetPartition(i, j)$
    $cMeetNBlocks \leftarrow CreamNumberOfBlocks(cMeet)$
    if $cMeetNBlocks = dimension$ **then**
      Add $[i, j]$ to $pairs$
    end if
  end if
end for
return $pairs$

In this section only the function CreamAllDirectlyReducible is addressed but the Cream package also includes the functions CreamExistsDirectlyReducible (returning true if there are commuting congruences R, T where A/R x A/T is isomorphic to A) and CreamOneDirectlyReducible (returning one pair of commuting congruences R, T where A/R x A/T is isomorphic to A).

4. Performance

4.1. **Congruences algorithm performance.** The **semigroup** GAP package [45] includes the function **CongruencesOfSemigroup** that returns the congruences of a semigroup, for some classes of semigroups. For Rees matrix semigroups the function **CongruencesOfSemigroup** takes advantage of theoretical results and hence achieves better results than **CreamAllCongruences**. But to achieve this kind of performance the semigroup needs to be created using the GAP functions **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**. If instead an isomorphic copy of a Rees Matrix (Zero) semigroup is created using a multiplication table the performance will be much worse.

For Rees Matrix Semigroups or Rees Zero Matrix Semigroups (generated with **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**) the function **CongruencesOfSemigroup** takes advantage of their particular structure by applying the efficient linked triple algorithm [35]. However, this efficiency can only be realized in this restricted setting.

UACalc also implements Freese's algorithm, and was done under his supervision in Java and Jython. It includes a GUI version in Java and a command line interface in Jython. The performance of the GUI version is poor and it will not be used for comparison. On the other hand the command line interface has a performance that is comparable to **CreamAllCongruences**. The same random Rees Matrix Semigroups that were used with **CongruencesOfSemigroup** and **CreamAllCongruences** were written into files readable by **UACalc** and the congruences were calculated in **UACalc**.

Table 1. Performance Comparison on calculating all congruences of Rees Matrix Semigroups.

Size	CongruencesOfSemigroup (ReesMatrixSemigroup) (ms)	CongruencesOfSemigroup (MultiplicationTable) (ms)	CreamAllCongruences (ms)	UACalc (ms)
12	1	15	< 1	1
18	2	28	< 1	3
24	5	55	2	7
30	1	92	4	32
36	4	139	10	34
42	11	203	16	55
48	8	296	24	77
54	5	416	46	138
60	5	553	60	220
66	1	671	82	291
72	9	873	136	409
78	2	1 098	180	555
84	5	1 341	233	743
90	4	1 622	307	961

CreamAllCongruences is consistently more than 3 times faster than **UACalc**. For these algebras with one binary operation the runtime rises roughly proportionally to n^4 (where n is the size of the algebra) or t^2 (being t the number cells of the multiplication matrix of the algebra operation).

4.2. **Automorphisms algorithm performance.** The CREAM package automorphism function was run on algebras of Type (2^m,1^n) to compare the timings against the Loops package. All experiments were run 7 times, with the lowest and highest times discarded. The averages of the remaining 5 are reported in Table 2. We see that the speeds from both packages are quite close.

Table 2. Performance Comparison between Loops and CREAM on Automorphism Group Generation.

Algebraic Structure	Loops Package(sec)	CREAM Package(sec)
Quasigroups, order 5	0.588	0.562
Quasigroups, order 6	520	541
Loops, order 6	0.101	0.086
Loops, order 7	14.623	13.945
Groups, order 32	0.854	0.935
Groups, order 64	39.98	41.05
Groups, order 128	12,031	12,160

4.3. **Endomorphisms algorithm performance.** To evaluate the performance of the algorithms to calculate endomorphisms, we tested both the classic algorithm and the congruence algorithm running with **Mace4**, as described in 3.3, on several types of Semigroups, Monoids and Groups. The only GAP function to calculate endomorphisms that came to our attention is the function Endomorphisms from the package SONATA. This function can calculate endomorphisms for a very narrow set of algebras, namely groups and near-rings. The groups in the list algebras were also run with SONATA.

Table 3. Performance Comparison between endomorphisms calculation algorithms

Algebra	Size	Number of Endomorphisms	Classic (ms)	Congruences (ms)	SONATA (ms)
GossipMonoid(3)	11	66	30	1 635	NA
PlanarPartitionMonoid(2)	14	72	56	233	NA
JonesMonoid(4)	14	72	45	232	NA
BrauerMonoid(3)	15	28	38	160	NA
PartitionMonoid(2)	15	89	50	313	NA
FullPBRMonoid(1)	16	1 426	585	5 134	NA
SymmetricGroup(4)	24	58	101	142	166
FullTransformationSemigroup(3)	27	40	116	364	NA
FullTransformationMonoid(3)	27	40	112	360	NA
SymmetricInverseMonoid(3)	34	54	274	504	NA
JonesMonoid(5)	42	113	746	777	NA
MotzkinMonoid(3)	51	98	1 972	2 400	NA
PartialTransformationMonoid(3)	64	138	2 586	1 963	NA
PartialBrauerMonoid(3)	76	165	7 897	7 796	NA
BrauerMonoid(4)	105	274	50 164	19 875	NA
SymmetricGroup(5)	120	146	32 861	2 979	201
PlanarPartitionMonoid(3)	132	393	95 460	25 803	NA
JonesMonoid(6)	132	393	131 472	22 889	NA
PartitionMonoid(3)	203	687	741 441	105 421	NA
SymmetricInverseMonoid(4)	209	282	470 901	75 940	NA

In this test the classic algorithm not using congruences is faster than using congruences for algebras up to size 60 but for larger algebras its runtime raises very fast and the algorithm using congruences becomes faster.

In view of these results, the **CreamEndomorphisms** function uses the classic algorithm for algebras with size up to 60 and the congruences algorithm for larger algebras.

The SONATA package achieves very good results for groups but it only works on a very narrow set of algebras.

5. Conclusion

The CREAM package provides efficient algorithms for the calculation of congruences, automorphisms and endomorphisms. These algorithms work with algebras of type $(2^m, 1^n)$, covering a very wide set of algebras and being consistently fast.

The integration with Mace4 allowed us to combine efficient algorithms such as [28] with the wide set of possibilities provided by a first order axiom-based model searcher.

The appendices of this article include a detailed performance discussion of the main algorithms and applications of the CREAM package to different algebra classes.

The CREAM package can be found and downloaded from `https://gitlab.com/rmbper/cream` and the outputs from the test runs described in section B can be found in `https://gitlab.com/rmbper/cream_dat`

Appendices

A. Detailed Performance

A.1. **Congruences algorithm performance.** The **semigroup** GAP package [45] includes the function **CongruencesOfSemigroup** that returns the congruences of a semigroup. This function doesn't work for all semigroups, but for those in which it works we have run the **CongruencesOfSemigroup** function against our **CreamAllCongruences** 5 times and calculated the average runtime after removing the best and worst runtimes getting the following results:

Table 4. Performance Comparison between **CongruencesOfSemigroup** and CREAM on calculating all congruences.

Size	Number of Algebras	CongruencesOfSemigroup(ms)	CreamAllCongruences(ms)
1	1	7	< 1
2	3	20	< 1
3	4	29	< 1
4	7	41	< 1
5	9	53	< 1
6	8	71	< 1
7	12	153	4
8	14	484	35
1-8	58	858	40

The number of semigroups for which **CongruencesOfSemigroup** works is very limited, yet even for those algebras **CreamAllCongruences** is more than 20 times faster.

The tests show that there is a very specific type of semigroup in which the function **CongruencesOfSemigroup** takes advantage of theoretical results and hence achieves better results than **CreamAllCongruences**. These are the Rees Matrix Semigroups or the Rees Zero Matrix Semigroups. But to achieve this kind of performance the semigroup needs to be created using the GAP functions **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**. If instead an isomorphic copy of a Rees Matrix (Zero) semigroup is created using a multiplication table the performance will be similar to what was found above. See in the following table the average runtime for 10 runs with random Rees Matrix Semigroups: For Rees Matrix Semigroups or Rees Zero Matrix Semigroups (generated with **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**) the function **CongruencesOfSemigroup** takes advantage of their particular structure by applying the efficient linked triple algorithm [35]. However, this efficiency can only be realized in this restricted setting.

UACalc also implements Freese's algorithm, and was done under his supervision in Java and Jython. It includes a GUI version in Java and a command line interface in Jython. The performance of the GUI version is poor and it will not be used for comparison. On the other hand the command line interface has a performance that is comparable to **CreamAllCongruences**. The same random Rees Matrix Semigroups that were used with **CongruencesOfSemigroup** and **CreamAllCongruences** were written into files readable by **UACalc** and the congruences were calculated in **UACalc**. **CreamAllCongruences** is consistently more than 3 times faster than **UACalc**. For these algebras with one binary operation the runtime rises roughly proportionally to n^4 (where n is the size of the algebra) or t^2 (being t the number cells of the multiplication matrix of the algebra operation).

To further compare the performance of **CreamAllCongruences** with **UACalc**, tests with several specific types of Semigroups, Monoids and Groups were run.

Table 5. Performance Comparison on calculating all congruences of Rees Matrix Semigroup.

Size	CongruencesOfSemigroup (ReesMatrixSemigroup) (ms)	CongruencesOfSemigroup (MultiplicationTable) (ms)	CreamAllCongruences (ms)	UACalc (ms)
12	1	15	< 1	1
18	2	28	< 1	3
24	5	55	2	7
30	1	92	4	32
36	4	139	10	34
42	11	203	16	55
48	8	296	24	77
54	5	416	46	138
60	5	553	60	220
66	1	671	82	291
72	9	873	136	409
78	2	1 098	180	555
84	5	1 341	233	743
90	4	1 622	307	961

Table 6. Performance Comparison between CREAM and UACalc

Algebra	Size	Number of Congruences	CreamAllCongruences (ms)	UACalc (ms)
GossipMonoid(3)	11	84	1	2
PlanarPartitionMonoid(2)	14	9	1	1
JonesMonoid(4)	14	9	< 1	1
BrauerMonoid(3)	15	7	< 1	2
PartitionMonoid(2)	15	13	< 1	1
FullPBRMonoid(1)	16	167	3	4
SymmetricGroup(4)	24	4	3	10
FullTransformationSemigroup(3)	27	7	3	14
FullTransformationMonoid(3)	27	7	3	7
SymmetricInverseMonoid(3)	34	7	6	25
JonesMonoid(5)	42	6	12	36
MotzkinMonoid(3)	51	10	23	73
PartialTransformationMonoid(3)	64	7	63	223
PartialBrauerMonoid(3)	76	16	117	351
BrauerMonoid(4)	105	19	430	1 420
SymmetricGroup(5)	120	3	812	2 700
PlanarPartitionMonoid(3)	132	10	1 094	3 490
JonesMonoid(6)	132	10	950	3 444
PartitionMonoid(3)	203	16	6 500	19 979
SymmetricInverseMonoid(4)	209	11	6 970	22 318
FullTransformationSemigroup(4)	256	11	22 529	53 557
FullTransformationMonoid(4)	256	11	22 385	52 361
MotzkinMonoid(4)	323	11	47 593	134 945
JonesMonoid(7)	429	7	161 987	485 004

The performance for these algebras is mostly similar to what was found for the Rees Matrix Semigroups, with a bigger variance in the results. On average **CreamAllCongruences** is between 3 and 3,5 times faster

than **UACalc** but, depending on the algebras, we get improvements spanning from 2 times to 4,5 times. The runtime also rises roughly proportionally to n^4 (although with more variance).

A.2. **Automorphisms algorithm performance.** The CREAM package automorphism function was run on algebras of Type (2^m,1^n) to compare the timings against the Loops package. All experiments were run 7 times, with the lowest and highest times discarded. The averages of the remaining 5 are reported in Table 7. We see that the speeds from both packages are quite close.

Table 7. Performance Comparison between Loops and CREAM on Automorphism Group Generation.

Algebraic Structure	Loops Package(sec)	CREAM Package(sec)
Quasigroups, order 5	0.588	0.562
Quasigroups, order 6	520	541
Loops, order 6	0.101	0.086
Loops, order 7	14.623	13.945
Groups, order 32	0.854	0.935
Groups, order 64	39.98	41.05
Groups, order 128	12,031	12,160

We have also run experiments on generating automorphism groups for magmas and semigroups. Results are displayed on Table 8. For these experiments, we do not have results from other GAP packages to compare with. Therefore the following results can be taken as the benchmark to beat in future improvements.

Table 8. Performance of CREAM on Automorphism Group Generation for Magmas and Semigroups.

Algebraic Structure	Time(sec)
Magmas, order 2	0.016
Magmas, order 3	1.253
Semigroups, order 6	2.316
Semigroups, order 7	45.947

The CREAMAutomorphisms runs substantially faster than the simple intersection of automorphism groups, especially when the automorphism group is the trivial group. Sample results are shown in Table 9.

Table 9. Performance Comparison between CREAMAutomorphisms and Intersection of Automorphism Groups.

Algebra of type (2, 2, 2, ...)	Algebra AutomorphismGroup (sec)	Intersection of Automorphism Groups (sec)	Is Trival Group the Automorphism Group?
All 2,328 non-isomorphic groups of order 128	153.8	201.5	No
All 67 non-isomorphic groups of order 243	10.3	56.3	No
All 15 non-isomorphic groups of order 625	11.3	122.9	No
All 15 non-isomorphic groups of order 999	2.1	168.9	Yes

A.3. **Endomorphisms algorithm performance.** To evaluate the performance of the algorithms to calculate endomorphisms, we tested both the classic algorithm and the congruence algorithm running with **Mace4**, as described in 3.3, on several types of Semigroups, Monoids and Groups. The only GAP function to calculate endomorphisms that came to our attention is the function Endomorphisms from the package SONATA. This function can calculate endomorphisms for a very narrow set of algebras, namely groups and near-rings. The groups in the list algebras were also run with SONATA.

Table 10. Performance Comparison between endomorphisms calculation algorithms

Algebra	Size	Number of Endomorphisms	Classic (ms)	Congruences (ms)	SONATA (ms)
GossipMonoid(3)	11	66	30	1 635	NA
PlanarPartitionMonoid(2)	14	72	56	233	NA
JonesMonoid(4)	14	72	45	232	NA
BrauerMonoid(3)	15	28	38	160	NA
PartitionMonoid(2)	15	89	50	313	NA
FullPBRMonoid(1)	16	1 426	585	5 134	NA
SymmetricGroup(4)	24	58	101	142	166
FullTransformationSemigroup(3)	27	40	116	364	NA
FullTransformationMonoid(3)	27	40	112	360	NA
SymmetricInverseMonoid(3)	34	54	274	504	NA
JonesMonoid(5)	42	113	746	777	NA
MotzkinMonoid(3)	51	98	1 972	2 400	NA
PartialTransformationMonoid(3)	64	138	2 586	1 963	NA
PartialBrauerMonoid(3)	76	165	7 897	7 796	NA
BrauerMonoid(4)	105	274	50 164	19 875	NA
SymmetricGroup(5)	120	146	32 861	2 979	201
PlanarPartitionMonoid(3)	132	393	95 460	25 803	NA
JonesMonoid(6)	132	393	131 472	22 889	NA
PartitionMonoid(3)	203	687	741 441	105 421	NA
SymmetricInverseMonoid(4)	209	282	470 901	75 940	NA

In this test the classic algorithm not using congruences is faster than using congruences for algebras up to size 60 but for larger algebras its runtime raises very fast and the algorithm using congruences becomes faster.

In view of these results, the **CreamEndomorphisms** function uses the classic algorithm for algebras with size up to 60 and the congruences algorithm for larger algebras.

B. APPLICATIONS

B.1. **Monolithic Algebras.** Some examples of the use of the package to calculate whether an algebra is monolithic can be seen next:

```
gap> algebra := [RecoverMultiplicationTable(6,1)];
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
      [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ] ] ]
gap> CreamAllPrincipalCongruences(algebra);
[ [ -1, -1, -1, -1, -2, 5 ], [ -1, -1, -1, -2, 4, -1 ],
  [ -1, -1, -1, -2, -1, 4 ], [ -1, -1, -2, 3, -1, -1 ],
  [ -1, -1, -2, -1, 3, -1 ], [ -1, -1, -2, -1, -1, 3 ],
  [ -1, -2, 2, -1, -1, -1 ], [ -1, -2, -1, 2, -1, -1 ],
  [ -1, -2, -1, -1, 2, -1 ], [ -1, -2, -1, -1, -1, 2 ],
  [ -2, 1, -1, -1, -1, -1 ], [ -2, -1, 1, -1, -1, -1 ],
  [ -2, -1, -1, 1, -1, -1 ], [ -2, -1, -1, -1, 1, -1 ],
  [ -2, -1, -1, -1, -1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
false
gap> algebra := [RecoverMultiplicationTable(6,19)];
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
      [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 2, 1, 1 ], [ 1, 1, 2, 1, 1, 1 ] ] ]
gap> CreamAllPrincipalCongruences(algebra);
[ [ -2, 1, -1, -1, -1, -1 ], [ -2, 1, -1, -1, -2, 5 ],
  [ -2, 1, -1, -2, 4, -1 ], [ -2, 1, -1, -2, -1, 4 ],
  [ -2, 1, -2, 3, -1, -1 ], [ -2, 1, -2, -1, 3, -1 ],
  [ -2, 1, -2, -1, -1, 3 ], [ -3, 1, 1, -1, -1, -1 ],
  [ -3, 1, -1, 1, -1, -1 ], [ -3, 1, -1, -1, 1, -1 ],
  [ -3, 1, -1, -1, -1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
true
```

B.2. **Small Semigroups.** For all small semigroups up to size 6, all its congruences and endomorphisms were calculated. It was moreover determined whether the semigroups were monolithic (this was calculated up to size 8).

Table 11. Determination of all monolithic semigroups up to size 8

Size	Number of semigroups	Number of monoliths	Performance (ms)
1	1	0	0
2	4	4	0
3	18	7	0
4	126	16	8
5	1 160	103	108
6	15 973	1 823	1 712
7	836 021	149 020	127 356
8	1 843 120 128	48 438 046	462 897 348

All automorphisms for semigroups up to size 7 were also calculated separately and the automorphism group and its ID was calculated for each of these semigroups. The group ID is the identification of a group in the Small Group GAP library [20].

The CREAM library doesn't calculate the automorphisms of an algebra as an automorphism group but as a list of bijective mappings, but there is an easy way to calculate the automorphism group from the list of mappings. To do this, all the mappings are converted into permutations using the **PermList** function and each of these permutations is used as generator for the group. Having this we can get the Group Id using the **IdGroup** function.

The following table gives the automorphism of semigroups up to size 7 (and reproduces the results in [9]):

Table 12. Computation of the automorphism groups of semigroups up to size 7

Size	2	3	4	5	6	7
# Semigroups	4	18	126	1 160	15 973	836 021
[1, 1] trivial	3	12	78	746	10 965	746 277
[2, 1] C_2	1	5	39	342	4 121	76 704
[3, 1] C_3				2	26	412
[4, 1] C_4				1	7	82
[4, 2] $C_2 \times C_2$			3	26	441	7 314
[5, 1] C_5						6
[6, 1] S_3		1	5	33	300	3 638
[6, 2] C_6						37
[8, 2] $C_4 \times C_2$						4
[8, 3] D_8				1	17	169
[8, 5] $C_2 \times C_2 \times C_2$					6	172
[10, 1] D_{10}						2
[12, 4] D_{12}				4	49	790
[16, 11] $C_2 \times D_8$						10
[24, 12] S_4			1	4	30	277
[24, 14] $C_2 \times C_2 \times S_3$						14
[36, 10] $S_3 \times S_3$					2	24
[48, 48] $C_2 \times S_4$					4	45
[72, 40] $(S_3 \times S_3) \wr C_2$						1
[120, 34] S_5				1	4	30
[144, 183] $S_3 \times S_4$						4
[240, 189] $C_2 \times S_5$						4
[720, 763] S_6					1	4
[5040, -] S_7						1

B.3. **Small Groups.** Using the Small Group GAP library [20] the groups from order 2 to 96 were extracted and the Cream functions were used with them:

- Order 2-31 - CreamAllCongruences, CreamAllEndomorphisms and CreamIsMonolithic
- Order 32-96 - CreamAllCongruences and CreamIsMonolithic

A summary from these runs is provided in Table 13.

Table 13. Summary for Small Group Runs

Size	**Number of Groups**	**Number of Monolithic**
2-7	8	6
8-15	19	9
16	14	6
17-23	17	7
24	15	2
25-31	19	7
32	50	16
33-47	54	10
48	52	4
49-63	69	16
64	52	22
65-71	17	3
72	50	3
73-79	18	5
80	52	1
81-95	70	13
96	231	13

B.4. **Groups, Semigroups and Monoids.** For a selection of larger groups, semigroups and monoids, all congruences and endomorphisms were calculated. It was also calculated whether these algebras were monolithic. A summary from these runs is provided in Table 14.

Table 14. # of Congruences and Endomorphisms for a selection of larger algebras

Algebra	Size	Number of Congruences	Number of Endomorphisms	Is Monolithic
GossipMonoid(3)	11	84	66	No
PlanarPartitionMonoid(2)	14	9	72	No
JonesMonoid(4)	14	9	72	No
BrauerMonoid(3)	15	7	28	No
PartitionMonoid(2)	15	13	89	No
FullPBRMonoid(1)	16	167	1426	No
SymmetricGroup(4)	24	4	58	Yes
FullTransformationSemigroup(3)	27	7	40	Yes
FullTransformationMonoid(3)	27	7	40	Yes
SymmetricInverseMonoid(3)	34	7	54	Yes
JonesMonoid(5)	42	6	113	No
MotzkinMonoid(3)	51	10	98	No
PartialTransformationMonoid(3)	64	7	138	Yes
PartialBrauerMonoid(3)	76	16	165	No
BrauerMonoid(4)	105	19	274	No
SymmetricGroup(5)	120	3	146	Yes
PlanarPartitionMonoid(3)	132	10	393	No
JonesMonoid(6)	132	10	393	No
PartitionMonoid(3)	203	16	687	No
SymmetricInverseMonoid(4)	209	11	282	Yes
FullTransformationSemigroup(4)	256	11	345	Yes
FullTransformationMonoid(4)	256	11	345	Yes

B.5. **Automorphisms of Specific Algebra Classes.** The following table contains the groups that appear as automorphims groups of associative rings up to order 15.

Table 15. Computation of the automorphism groups of rings up to size 15

Size	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# Rings	2	2	11	2	4	2	52	11	4	2	22	2	4	4
[1, 1] trivial	2	1	3		2		4				3			
[2, 1] C_2		1	6	1		1	12	1	2		4		2	1
[4, 1] C_4										1				
[4, 2] $C_2 \times C_2$					1		6	3				1		
[6, 1] S_3			2				1							
[6, 2] C_6							1							
[8, 3] D_8							3							
[8, 5] $C_2 \times C_2 \times C_2$							4	1			2			
[12, 4] D_{12}								2						
[16, 11] $C_2 \times D_8$							3							
[16, 14] $C_2 \times C_2 \times C_2 \times C_2$											2			
[24, 12] S_4					1									
[24, 13] $C_2 \times A_4 \times S_3$							2							
[32, 46] $C_2 \times C_2 \times D_8$											4			
[36, 10] $S_3 \times S_3$							4							
[48, 48] $C_2 \times S_4$							7							
[64, 261] $C_2 \times C_2 \times C_2 \times D_8$														1
[72, 40] $(S_3 \times S_3) \wr C_2$								2						
[120, 34] S_5					1									
[144, 183] $S_3 \times S_4$							2							
[216, 162] $S_3 \times S_3 \times S_3$											2			
[576, 8653] $S_4 \times S_4$									1					
[720, 763] S_6						1								
[5040, -] S_7							3							
[13824, -] $S_4 \times S_4 \times S_4$														1
[14400, -] $S_5 \times S_5$											1			
[17280, -] $S_6 \times S_4$											2			
[40320, -] S_8								2						
[362880, -] S_{10}									1	1				
[518400, -] $S_6 \times S_6$													1	
[39916800, -] S_{11}											2			
[479001600, -] S_{12}												1		
[6227020800, -] S_{13}													1	
[87178291200, -] S_{14}														1

The variety of *Quasi-MV-algebras* [37] is defined by the following identities (in ProverX syntax [13,17]):

```
(x + z) + y = x + (y + z).
x'' = x.
x + 1 = 1.
(x' + y)' + y = (y' + x)' + x.
(x + 0)' = x' + 0.
(x + y) + 0 = x + y.
0' = 1.
```

Table 16. Computation of the automorphism groups of Quasi-MV-algebras of size up to 12

Size	2	3	4	5	6	7	8	9	10	11	12
# Quasi-MV-algebras	1	1	4	4	11	11	27	27	60	62	131
[1, 1] trivial	1	1	3	2	7	4	16	8	35	17	76
[2, 1] C_2			1	2	3	4	5	9	10	18	19
[4, 2] $C_2 \times C_2$						2	1	4	2	10	5
[6, 1] S_3					1		3		5		9
[8, 5] $C_2 \times C_2 \times C_2$								2	1	4	2
[12, 4] D_{12}							1		2		5
[16, 14] $C_2 \times C_2 \times C_2 \times C_2$										2	1
[24, 12] S_4						1		2		4	
[24, 14] $C_2 \times C_2 \times C_3$									1		2
[48, 48] $C_2 \times S_4$								1		2	
[48, 51] $C_2 \times C_2 \times C_2 \times S_3$											1
[96, 226] $C_2 \times C_2 \times S_4$										1	
[120, 34] S_5							1	1	2		4
[240, 189] $C_2 \times S_5$									1		2
[480, 1186] $C_2 \times C_2 \times S_5$											1
[720, 763] S_6										2	
[1440, 5842] $C_2 \times S_6$										1	
[5040, -] S_7									1		2
[10080, -] $C_2 \times S_7$											1
[40320, -] S_8										1	
[362880, -] S_9											1

The quasivariety of *BCI algebras* [2] is defined by the following identities (in ProverX syntax [13, 17]):

```
((x * y) * (x * z)) * (z * y) = 0.
(x * (x * y)) * y = 0.
x * x = 0.
((x * y = 0) & (y * x = 0)) -> (x = y).
(x * 0 = 0) -> (x = 0).
```

Table 17. Computation of the automorphism groups of BCI algebras of size up to 7

Size	2	3	4	5	6	7
# BCI algebras	2	5	22	118	974	10,834
[1, 1] trivial	2	3	13	78	679	7,970
[2, 1] C_2		2	7	31	241	2,384
[4, 1] C_4				1	1	2
[4, 2] $C_2 \times C_2$				2	20	207
[6, 1] S_3			2	5	25	195
[6, 2] C_6						1
[8, 2] $C_4 \times C_2$						1
[8, 3] D_8						4
[8, 5] $C_2 \times C_2 \times C_2$						3
[12, 4] D_{12}					4	35
[24, 12] S_4				1	4	22
[36, 10] $S_3 \times S_3$						2
[48, 48] $C_2 \times S_4$						3
[120, 34] S_5					1	4
[720, 763] S_6						1

The variety of *quandles* [43] is defined by the following identities (in ProverX syntax [13, 17]):

```
x v (y v z) = (x v y) v (x v z).
(x ^ y) ^ z = (x ^ z) ^ (y ^ z).
(x v y) ^ x = y.
x v (y ^ x) = y.
x v x = x.
```

Table 18. Computation of the automorphism groups of quandles of size up to 6

Size	2	3	4	5	6
# quandles	1	3	7	22	73
[2, 1] C_2	1	1	1		
[3, 1] C_3			1	1	
[4, 1] C_4				1	2
[4, 2] $C_2 \times C_2$			1	5	8
[5, 1] C_5					1
[6, 1] S_3		2	1	2	3
[6, 2] C_6				2	15
[8, 2] $C_4 \times C_2$					2
[8, 3] D_8			1	3	8
[8, 5] $C_2 \times C_2 \times C_2$					7
[12, 3] A_4			1	1	
[12, 4] D_{12}				3	6
[16, 11] $C_2 \times D_8$					5
[18, 3] $C_3 \times S_3$					3
[20, 3] $C_5 \times C_4$				3	3
[24, 12] S_4			1		2
[24, 13] $C_2 \times A_4 \times S_3$					3
[36, 10] $S_3 \times S_3$					1
[48, 48] $C_2 \times S_4$					2
[72, 40] $(S_3 \times S_3) \wr C_2$					1
[120, 34] S_5				1	
[720, 763] S_6					1

B.6. **Algebra Classes.** The CREAM package allows for the generation of lists of algebras of specific types based on its axiomatic definition. This is done using the function **CreamAlgebrasFromAxioms** that uses Mace4 to generate algebras according with this axiomatic definition. This function takes as input a string with the axioms in Mace4 format and a positive integer with the size of the generated algebras. The generated algebras can then be used with the CREAM functions that calculate congruences, endomorphisms and whether the algebra is monolithic. CREAM was applied in this way to the following classes of algebras:

- Almost distributive lattices
- BCI-algebras
- Bilattices
- Boolean algebras
- Boolean rings
- Commutative lattice-ordered monoids
- Commutative regular rings
- Complemented distributive lattices
- Complemented modular lattices
- Distributive lattice ordered semigroups
- Ockham algebras
- Ortholattices
- Orthomodular lattices
- Idempotent semirings
- Extra loops
- Digroups
- Commutative dimonoids

A summary from these runs is provided in Table 19.

Table 19. Summary for Algebra Class Runs

Algebra Class	Size	Algebra Type	Number of Algebras	Number of Monolithic
Almost distributive lattices	4	(2^2)	2	0
BCI-algebras	5	(2^1)	118	80
Bilattice	6	$(2^4, 1^1)$	32	32
Boolean algebra	8	$(2^2, 1^1)$	1	0
Boolean algebra	16	$(2^2, 1^1)$	1	0
Boolean ring	16	(2^2)	1	0
Commutative lattice-ordered monoids	5	(2^3)	199	97
Commutative regular rings	6	$(2^2, 1^2)$	72	66
Complemented distributive lattices	16	$(2^2, 1^1)$	1	0
Complemented distributive lattices	32	$(2^2, 1^1)$	1	0
Complemented modular lattices	8	$(2^2, 1^1)$	41	40
Distributive lattice ordered semigroups	4	(2^3)	479	170
Ockham algebras	6	$(2^2, 1^1)$	197	20
Ortholattices	7	$(2^2, 1^1)$	46	12
Orthomodular lattices	14	$(2^2, 1^1)$	33	31
Idempotent semiring	5	(2^2)	149	42
Extra loop	10	(2^3)	2	1
Digroup	8	$(2^2, 1^1)$	10	3
Commutative dimonoid	4	(2^2)	101	37

REFERENCES

[1] J. André, J. Araújo, P. J. Cameron, The classification of partition homogeneous groups with applications to semigroup theory. *J. Algebra* **452** (2016), 288–310.

[2] Y. Arai, K. Iséki and S. Tanaka, Characterizations of BCI, BCK-algebras, *Proc. Japan Acad.* (1966), **42** (2), 105–107.

[3] J. Araújo, J.P. Araújo, W. Bentz, P. J. Cameron, P. Spiga, A transversal property for permutation groups motivated by partial transformations. *J. Algebra* **573** (2021), 741–759.

[4] J. Araújo, W. Bentz, P. J. Cameron, The existential transversal property: a generalization of homogeneity and its impact on semigroups. *Trans. Amer. Math. Soc.* **374** (2021), no. 2, 1155–1195.

[5] J. Araújo, W. Bentz, P. J. Cameron, Primitive permutation groups and strongly factorizable transformation semigroups. *J. Algebra* **565** (2021), 513–530.

[6] J. Araújo, W. Bentz, P. J. Cameron, Orbits of primitive k-homogenous groups on (n?k)-partitions with applications to semigroups. *Trans. Amer. Math. Soc. 371* (2019), no. 1, 105–136.

[7] J. Araújo, W. Bentz, P. J. Cameron, G. Royle, A. Schaefer, Primitive groups, graph endomorphisms and synchronization. *Proc. Lond. Math. Soc.* (3) **113** (2016), no. 6, 829–867.

[8] J. Araújo, W. Bentz, E. Dobson, J. Konieczny, J. Morris, Automorphism groups of circulant digraphs with applications to semigroup theory. *Combinatorica* **38** (2018), no. 1, 1–28.

[9] J. Araújo, P.V. Bünau, J.D. Mitchell, M. Neunhöffer, Computing automorphisms of semigroups, J. Symbolic Comput., 45 (3) (2010), pp. 373-392

[10] J. Araújo, P. J. Cameron, B. Steinberg, Between primitive and 2-transitive: synchronization and its friends. *EMS Surv. Math. Sci.* **4** (2017), no. 2, 101–184.

[11] J. Araújo, P. J. Cameron, Two generalizations of homogeneity in groups with applications to regular semigroups. *Trans. Amer. Math. Soc.* **368** (2016), no. 2, 1159–1188.

[12] J. Araújo, P. J. Cameron, Primitive groups synchronize non-uniform maps of extreme ranks. *J. Combin. Theory Ser. B* **106** (2014), 98–114.

[13] J. Araújo, M. Kinyon and Yves Robert, Varieties of regular semigroups with uniquely defined inversion. Port. Math. 76 (2019), 205–228.

[14] J. Araújo and J. Konieczny, Automorphism groups of centralizers of idempotents, *J. Algebra* **269** (2003), 227–239.

[15] J. Araújo, and J. Konieczny, A Method of Finding Automorphism Groups of Endomorphism Monoids of Relational Systems, *Discrete Math.* **307** (2007) 1609–1620.

[16] J. Araújo and J. Konieczny, Automorphisms of Endomorphism Monoids of Relatively Free Bands, *Proc. Edinburgh Math. Soc.* **50** (2007) 1–21

[17] J. Araújo and Yves Robert, The System ProverX, www.proverx.com.

[18] E. Artin, "Geometric Algebra," Interscience, New York, 1957.

[19] M. V. Berlinkov, R. Ferens, M. Szykua, Preimage problems for deterministic finite automata. *J. Comput. System Sci.* **115** (2021), 214–234.

[20] Hans Ulrich Besche, Bettina Eick & Eamonn O'Brien, *GAP package SmallGrp - The GAP Small Groups Library, Version 1.4.2*; 2020, `https://www.gap-system.org/Packages/smallgrp.html`.

[21] Stanley N. Burris & H.P. Sankappanavar, *A Course in Universal Algebra*; 1981, `http://www.math.uwaterloo.ca/~snburris/htdocs/UALG/univ-algebra2012.pdf`.

[22] P.J. Cameron, Automorphism groups of graphs, "Selected Topics in Graph Theory. 2," 89–127, Academic Press, London, 1983.

[23] J. J. Cannon and D. Holt, Automorphism group computation and isomorphism testing in finite groups, *J. Symbolic Comput.* **35** (2003), 241–267.

[24] David Clark and Brian Davey, *Natural dualities for the working algebraist*, Cambridge Studies in Advanced Mathematics 57, 1998

[25] Andreas Distler & James Mitchel, *GAP package Smallsemi - A library of small semigroups, Version 0.6.12*; 2019, `https://www.gap-system.org/Packages/smallsemi.html`.

[26] S. P. Fitzpatrick and J. S. Symons, Automorphisms of transformation semigroups, *Proc. Edinburgh Math. Soc.* **19** (1974/75), 327-329.

[27] Ralph Freese, *UACalc, A Universal Algebra Calculator*; 2015, `http://www.uacalc.org/`.

[28] Ralph Freese, Computing congruences efficiently, *Algebra Universalis* **59(3)** (2008), 337–343. DOI:

[29] Ralph Freese, and Ralph McKenzie, *Commutator Theory for congruence modular varieties*, London Mathematical Society Lecture Notes, Vol. 125, 1987

[30] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.11.0*; 2020, `https://www.gap-system.org`.

[31] O. Garcia and Walter Taylor, *The lattice of interpretability types of varieties*, Memoirs of the American Mathematical Society, Vol. 50, no. 305, 1984

[32] L. M. Gluskĭn, Semigroups and rings of endomorphisms of linear spaces I, *Amer. Math. Soc. Transl.* **45** (1965), 105–137.
[33] J. Hagemmann and C. Herrmann, A concrete ideal multiplication for algebraic systems and its relation to congruence distributivity, *Arch. Math.* **32**, 234-245, 1979
[34] David Hobby and Ralph McKenzie, *The structure of finite algebras*, Contemporary Mathematics, Vol. 76, 1988
[35] J. M. Howie, *Fundamentals of Semigroup Theory*, Oxford University Press, New York, 1995.
[36] F. Klein, "Vergleichende Betrachtungen über neuere geometrische Forschungen," Andreas Diechert, Erlanger, 1872.
[37] A. Ledda, M. Konig, F. Paoli, and R. Giuntini. MV-algebras and quantum computation. *Studia Logica* (2006) **82** (2):245–270.
[38] I. Levi, Automorphisms of normal transformation semigroups, *Proc. Edinburgh Math. Soc. (2)* **28** (1985), 185–205.
[39] I. Levi, Automorphisms of normal partial transformation semigroups, *Glasgow Math. J.* **29** (1987), 149–157.
[40] A. E. Liber, On symmetric generalized groups, *Mat. Sbornik N. S.* **33** (1953), 531–544. (Russian)
[41] K. D. Magill, Semigroup structures for families of functions, I. Some homomorphism theorems, *J. Austral. Math. Soc.* **7** (1967), 81–94.
[42] A. I. Mal'cev, Symmetric groupoids, *Mat. Sbornik N.S.* **31** (1952), 136–151. (Russian)
[43] S. Matveev, Distributive groupoids in knot theory. *Math. USSR Sbornik* (1984) **47**, 73–83.
[44] W. McCune, *Mace4, Version 2009-02A*; 2009 `https://www.cs.unm.edu/~mccune/prover9/manual/2009-02A/mace4.html`.
[45] J. D. Mitchell and others, Semigroups - GAP package, Version 2.6, (2015). `https://gap-packages.github.io/Semigroups/`
[46] V. A. Molchanov, Semigroups of mappings on graphs, *Semigroup Forum* **27** (1983), 155–199.
[47] A. V. Molchanov, On definability of hypergraphs by their semigroups of homomorphisms, *Semigroup Forum* **62** (2001), 53–65.
[48] Gábor Nagy & Petr Vojtěchovský, *The LOOPS Package - a GAP package, Version 3.4.1*; 2018, `https://gap-packages.github.io/loops/`.
[49] P. M. Neumann, Primitive permutation groups and their section-regular partitions, *Michigan Math. J.*, **58** (2009), 309–322.
[50] B. M. Schein, Ordered sets, semilattices, distributive lattices and Boolean algebras with homomorphic endomorphism semigroups, *Fund. Math.* **68** (1970), 31–50.
[51] C. Schneider and A. C. Silva, Cliques and colorings in generalized Paley graphs and an approach to synchronization, *J. Algebra Appl.*, **14** (2015), no. 6, 13 pp.
[52] J. Schreier, Über Abbildungen einer abstrakten Menge Auf ihre Teilmengen, *Fund. Math.* **28** (1936), 261–264.
[53] J. Rhodes and B. Steinberg, The q-theory of finite semigroups. Springer Verlag. (2008).
[54] Shahzamanian, B. Steinberg, Simplicity of augmentation submodules for transformation monoids. *Algebr. Represent. Theory* **24** (2021), no. 4, 1029–1051.
[55] Smith, JDH, *Mal'cev varieties*, Lecture Notes in Mathematics, Vol. 554, 1976
[56] L. Soicher, *GRAPE: Graph algorithms using permutation groups*; version 4.2, `http://www.maths.qmul.ac.uk/~leonard/grape/`.
[57] M. R. Sorouhesh, On ideals of quasi-commutative semigroups. *Bull. Iranian Math. Soc.* **45** (2019), no. 2, 447–453.
[58] R. P. Sullivan, Automorphisms of transformation semigroups, *J. Austral. Math. Soc.* **20** (1975), 77–84.
[59] È. G. Šutov, Homomorphisms of the semigroup of all partial transformations, *Izv. Vysš. Učebn. Zaved. Matematika* **3** (1961), 177–184. (Russian)
[60] S.M. Ulam, "A Collection of Mathematical Problems," Interscience Publishers, New York, 1960.

Number of Non-isomorphic Models of Common Classes of Algebras

João Araújo
Universidade Nova de Lisboa
Lisbon, Portugal
https://docentes.fct.unl.pt/jj-araujo
jj.araujo@fct.unl.pt

Choiwah Chow
Universidade Aberta
Lisbon, Portugal
1702603@estudante.uab.pt

Mikoláš Janota
Czech Technical University in Prague
Czech Republic
http://people.ciirc.cvut.cz/~janotmik/
mikolas.janota@cvut.cz

João Ramires
Universidade Aberta
Lisbon, Portugal

Abstract

The number of models of order n, up to isomorphism, of a class of algebras is an important property of that class of algebras. To date, only a handful of such sequences of numbers are produced and available online in websites such as the oeis.org. We use the parallel model enumerator Mace4 to generate models of the classes of algebras listed in the MarcieX database, count their numbers, and report them in this paper.

1 Introduction

The number of order n non-isomorphic models of a given class of algebras is a very important piece of information. It is also very difficult to find. For example, the number of non-isomorphic groupoids (or magmas) has been found [1]. Similar ideas can be applied to special classes of algebras. Closed formulas are known for some particular classes of groups [2], but finding them often requires very deep results, including the classification of finite simple

groups. For groups in general, only bounds are known. The sequence of order n groups was chosen to be the first one in the *On-Line Encyclopedia of Integer Sequences* (OEIS) [3].

But the results above, along with some of the same flavor, stand out as exceptions. The rule is that there is no closed formula for the order n non-isomorphic models of an important class of algebras (say, rings, semigroups, quasigroups, etc.). Therefore, the best we have now is some computation of the first terms of the sequence. For many classes of algebras not even the first values are computed for any meaningful n. The reason is that the task is far from trivial. We need to use some finite model enumerator to generate the models of order n according to the rules laid down by the axiomatic definition of that class of algebras. These enumerators use delicate symmetry breaking algorithms to reduce the number of redundant isomorphic models, but usually a large number of unnecessary models is generated. This has two adverse effects: time and memory are lost to generate them and afterwards a new algorithm and more time are needed to clean up the list. It often happens that the biggest bottleneck is precisely in this second part (get rid of the redundant generated models). For example, while there are only 1,627,672 semigroups of order 7, the finite model generator Mace4 [4] generates over 1 billion models of order 7; afterwards, its companion program Isofilter is not able to filter out all the isomorphic models in reasonable time (over 2 days) when the program had to be terminated.

In order to push the limits of the known sequences for general classes of algebras, we developed two new algorithms that apply to the two steps of the process. Our first symmetry breaking algorithm (to be published) dramatically reduces the number of redundant models generated; our second algorithm [5] reduces by orders of magnitude the time needed to discard the redundant models. For example, Mace4 generates 28.9 million models for the inverse semigroups of order 8. Without invariants, it takes Isofilter 150,703 seconds to filter out the isomorphic models to get 4,637 non-isomorphic models, but it takes less than 2% (2,873 seconds) of the original time for Isofilter to accomplish the same task with invariants (see Table 2 in [6]). The general idea is the following: to each model generated we attach a vector of parameters that are preserved under isomorphism. This induces a partition of he original list into many blocks (or parts) of non-isomorphic models (that is, isomorphic models belong to the same block). Therefore, rather than testing a given model against all others, we only test it against the models in the same block; in addition, as checking each block is independent from checking the others, the task can be parallelized.

With these two tools we were able to find the initial terms of the sequence of order n models of a given class of algebras for many classes whose sequence was unknown so far; we also were able to confirm and extend the sequences in the cases some terms were already known.

2 MarcieX

MarcieX [7] is a library of theories and theorems in the syntax of ProverX [8]. It has the advantage of being noiseless, unlike many other databases (such as TPTP [9], etc.)

that computer scientists use to benchmark their research in Artificial Intelligence Theorem Proving (AITP). In addition, it provides some routines that allow the user to perform the following operations:

1. given a theorem, MarcieX scans its database to identify in which other classes of algebras of the same type, the analogue holds; for example, given the theorem *If in a group every element is the inverse of itself, then the group is commutative*, MarcieX will check all classes of algebras that have a binary operation and a unary operation, and check if the same result is true (eg, the same result holds for inverse semigroups);

2. given a class of algebras by the user, MarcieX will find the classes of algebras that contain it or are contained in it;

3. given some concept C that in some theory T has some property P (say, a binary relation that in semigroups is a congruence), MarcieX will find other classes of algebras not contained in T in which the concept C keeps property P;

4. given some model (say, a counter-example for some conjecture), MarcieX will identify pages of papers where a model isomorphic to that one appears.

In this paper we use MarcieX as a database of classes of algebras so that we can apply to them our finite model builder algorithms.

3 Counting Models

We use a 24-core Intel® Xeon®CPU E5-2630 v2 2.60GHz ×24 computer, with 64 GB RAM and 600 GB disk space for generating models and for filtering out isomorphic models. Once isomorphic models for an order of an algebra are filtered out, we count the remaining non-isomorphic models and report the number.

We run finite model enumeration on algebras in the MarcieX database. We allocate at least 5 hours to the search, in parallel with all 24 cores of the computer, of models of each order of each algebras. The search will be aborted if the time limit is exceeded and if the search is deemed not likely to finish anytime soon. Another limitation of the hardware used is disk space. We stop the search of models of a particular order if the size of the file containing the models exceeds 500 GB. The search for an sequence of number of models starts from order 2, then proceed incrementally (by 1 each time) up to, but not including, the order in which the time limit or the disk space limit is reached.

We exclude in this paper algebras such as semigroups, quasigroups, and loops, etc., of which the number of models of small orders are well-known, or that they can easily be found in online resources such as the OEIS. We also exclude algebras that we are not able to find their numbers of models at least up to order 4.

We compare our results with the integer sequences in the OEIS. For OEIS sequences A000112, A001930, A084965, A087729, A181769, A226193, A346414, our results match

theirs although we are not able to generate longer sequences than theirs, so, we do not report our results here. For sequences A178432 (see Section 3.76) and A305858 (see Section 3.41), we not only match, but also extend, their results. We found 100 new sequences that have not been reported to the OEIS that are reported in the following sections.

3.1 BCI-algebras

BCI-algebras appear in [10] and the number of non-isomorphic models from order 2 to order 8 is given in Table 1.

Table 1: Number of BCI-algebras

Order:	2	3	4	5	6	7	8
#Models:	2	5	22	118	974	10,834	162,555

3.2 BCK-algebras

BCK-algebras appear in [11] and the number of non-isomorphic models from order 2 to order 8 is given in Table 2.

Table 2: Number of BCK-algebras

Order:	2	3	4	5	6	7	8
#Models:	1	3	14	88	775	9,170	143,866

3.3 BCK-join-semilattice

BCK-join-semilattice appear in [12] and the number of non-isomorphic models from order 2 to order 7 is given in Table 3.

Table 3: Number of BCK-join-semilattice

Order:	2	3	4	5	6	7
#Models:	1	3	14	87	745	8,282

3.4 BCK-lattices

BCK-lattices appear in [13] and the number of non-isomorphic models from order 2 to order 8 is given in Table 4.

Table 4: Number of BCK-lattices

Order:	2	3	4	5	6	7	8
#Models:	1	2	8	38	265	2,391	27,485

3.5 BCK-meet-semilattices

BCK-meet-semilattices appear in [14] and the number of non-isomorphic models from order 2 to order 7 is given in Table 5.

Table 5: Number of BCK-meet-semilattices

Order:	2	3	4	5	6	7
#Models:	1	2	8	38	265	2,391

3.6 Bilattices

Bilattices appear in [15] and the number of non-isomorphic models from order 2 to order 9 is given in Table 6.

Table 6: Number of Bilattices

Order:	2	3	4	5	6	7	8	9
#Models:	0	0	1	3	32	284	3,839	62,735

3.7 BL-algebras

BL-algebras appear in [16] and the number of non-isomorphic models from order 2 to order 5 is given in Table 7.

Table 7: Number of BL-algebras

Order:	2	3	4	5
#Models:	6	34	606	854,822

3.8 Boolean monoids

Boolean monoids appear in [17] and the number of non-isomorphic models from order 2 to order 16 is given in Table 8.

Table 8: Number of Boolean monoids

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#Models:	1	0	5	0	0	0	83	0	0	0	0	0	0	0	242,547

3.9 Brouwerian algebras

Brouwerian algebras appear in [18] and the number of non-isomorphic models from order 2 to order 5 is given in Table 9.

Table 9: Number of Brouwerian algebras

Order:	2	3	4	5
#Models:	4	8	58	8,806

3.10 Clifford algebras

Clifford algebras appear in [19] and the number of non-isomorphic models from order 2 to order 9 is given in Table 10.

Table 10: Number of Clifford algebras

Order:	2	3	4	5	6	7	8	9
#Models:	2	5	16	51	202	879	4,454	25,322

3.11 Commutative BCK-algebras

Commutative BCK-algebras appear in [20] and the number of non-isomorphic models from order 2 to order 10 is given in Table 11.

Table 11: Number of Commutative BCK-algebras

Order:	2	3	4	5	6	7	8	9	10
#Models:	1	2	5	11	28	72	192	515	1,426

3.12 Commutative lattice-ordered monoids

Commutative lattice-ordered monoids appear in [21] and the number of non-isomorphic models from order 2 to order 8 is given in Table 12.

Table 12: Number of Commutative lattice-ordered monoids

Order:	2	3	4	5	6	7	8
#Models:	2	6	31	199	1,598	15,515	181,538

3.13 Commutative lattice-ordered rings

Commutative lattice-ordered rings appear in [22] and the number of non-isomorphic models from order 2 to order 7 is given in Table 13.

Table 13: Number of Commutative lattice-ordered rings

Order:	2	3	4	5	6	7
#Models:	14	51	1,794	4,655	492,120	3,290,875

3.14 Commutative lattice-ordered semigroups

Commutative lattice-ordered semigroups appear in [23] and the number of non-isomorphic models from order 2 to order 7 is given in Table 14.

Table 14: Number of Commutative lattice-ordered semigroups

Order:	2	3	4	5	6	7
#Models:	4	20	149	1,427	16,683	230,688

3.15 Commutative ordered monoids

Commutative ordered monoids appear in [24] and the number of non-isomorphic models from order 2 to order 9 is given in Table 15.

Table 15: Number of Commutative ordered monoids

Order:	2	3	4	5	6	7	8	9
#Models:	2	6	22	92	426	2,146	11,634	67,472

3.16 Commutative partially ordered monoids

Commutative partially ordered monoids appear in [25] and the number of non-isomorphic models from order 2 to order 7 is given in Table 16.

Table 16: Number of Commutative partially ordered monoids

Order:	2	3	4	5	6	7
#Models:	4	27	301	4,887	113,358	3,763,121

3.17 Commutative partially ordered semigroups

Commutative partially ordered semigroups appear in [26] and the number of non-isomorphic models from order 2 to order 5 is given in Table 17.

Table 17: Number of Commutative partially ordered semigroups

Order:	2	3	4	5
#Models:	7	83	1,468	37,248

3.18 Commutative regular rings

Commutative regular rings appear in [27] and the number of non-isomorphic models from order 2 to order 13 is given in Table 18.

Table 18: Number of Commutative regular rings

Order:	2	3	4	5	6	7	8	9	10	11	12	13
#Models:	2	3	13	5	72	7	900	384	800	11	169,452	13

3.19 Commutative residuated partially ordered monoids

Commutative residuated partially ordered monoids appear in [28] and the number of non-isomorphic models from order 2 to order 7 is given in Table 19.

Table 19: Number of Commutative residuated partially ordered monoids

Order:	2	3	4	5	6	7
#Models:	2	5	24	131	1,001	9,248

3.20 Complemented lattices

Complemented lattices appear in [29] and the number of non-isomorphic models from order 2 to order 8 is given in Table 20.

Table 20: Number of Complemented lattices

Order:	2	3	4	5	6	7	8
#Models:	1	0	1	4	41	600	13,941

3.21 Complemented modular lattices

Complemented modular lattices appear in [30] and the number of non-isomorphic models from order 2 to order 11 is given in Table 21.

Table 21: Number of Complemented modular lattices

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	0	1	2	6	13	41	100	303	797

3.22 De Morgan monoids

De Morgan monoids appear in [31] and the number of non-isomorphic models from order 2 to order 4 is given in Table 22.

Table 22: Number of De Morgan monoids

Order:	2	3	4
#Models:	18	882	174,062

3.23 Dense linear orders

Dense linear orders appear in [32] and the number of non-isomorphic models from order 2 to order 4 is given in Table 23.

Table 23: Number of Dense linear orders

Order:	2	3	4
#Models:	4	324	1,179,648

3.24 Directoids

Directoids appear in [33] and the number of non-isomorphic models from order 2 to order 7 is given in Table 24.

Table 24: Number of Directoids

Order:	2	3	4	5	6	7
#Models:	1	2	7	61	2,297	517,919

3.25 Distributive lattice ordered semigroups

Distributive lattice ordered semigroups appear in [34] and the number of non-isomorphic models from order 2 to order 5 is given in Table 25.

Table 25: Number of Distributive lattice ordered semigroups

Order:	2	3	4	5
#Models:	6	44	479	5,492

3.26 Division rings

Division rings appear in [35] and the number of non-isomorphic models from order 2 to order 15 is given in Table 26.

Table 26: Number of Division rings

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14	15
#Models:	2	3	3	5	0	7	4	6	0	11	0	13	0	0

3.27 Generalized Boolean algebras

Generalized Boolean algebras appear in [36] and the number of non-isomorphic models from order 2 to order 9 is given in Table 27.

Table 27: Number of Generalized Boolean algebras

Order:	2	3	4	5	6	7	8	9
#Models:	4	0	12	0	60	0	1,171,417	0

3.28 Hilbert algebras

Hilbert algebras appear in [37] and the number of non-isomorphic models from order 2 to order 9 is given in Table 28.

Table 28: Number of Hilbert algebras

Order:	2	3	4	5	6	7	8	9
#Models:	1	2	6	21	95	550	4,036	37,603

3.29 Hoops

Hoops appear in [38] and the number of non-isomorphic models from order 2 to order 13 is given in Table 29.

Table 29: Number of Hoops

Order:	2	3	4	5	6	7	8	9	10	11	12	13
#Models:	1	2	5	10	23	49	111	244	545	1,203	2,697	5,974

3.30 Involutive lattices

Involutive lattices appear in [39] and the number of non-isomorphic models from order 2 to order 13 is given in Table 30.

Table 30: Number of Involutive lattices

Order:	2	3	4	5	6	7	8	9	10	11	12	13
#Models:	1	1	3	4	12	20	61	122	389	906	3,047	8,028

3.31 Lattice-ordered rings

Lattice-ordered rings appear in [40] and the number of non-isomorphic models from order 2 to order 6 is given in Table 31.

Table 31: Number of Lattice-ordered rings

Order:	2	3	4	5	6
#Models:	14	51	2,238	4,655	492,120

3.32 Lattice-ordered semigroups

Lattice-ordered semigroups appear in [41] and the number of non-isomorphic models from order 2 to order 6 is given in Table 32.

Table 32: Number of Lattice-ordered semigroups

Order:	2	3	4	5	6
#Models:	6	44	479	6,738	117,028

3.33 Lineales

Lineales, also known as commutative residuated partially ordered monoids, appear in [42] and the number of non-isomorphic models from order 2 to order 7 is given in Table 33.

Table 33: Number of Lineales

Order:	2	3	4	5	6	7
#Models:	2	5	24	131	1,001	9,248

3.34 Linear Heyting algebras

Linear Heyting algebras, also known as Goedel algebras, appear in [43] and the number of non-isomorphic models from order 2 to order 4 is given in Table 34.

Table 34: Number of Linear Heyting algebras

Order:	2	3	4
#Models:	8	302	835,006

3.35 M-zeroids

M-zeroids appear in [44] and the number of non-isomorphic models from order 2 to order 9 is given in Table 35.

Table 35: Number of M-zeroids

Order:	2	3	4	5	6	7	8	9
#Models:	2	3	7	21	75	315	1,537	8,583

3.36 Modular ortholattices

Modular ortholattices appear in [45] and the number of non-isomorphic models from order 2 to order 5 is given in Table 36.

Table 36: Number of Modular ortholattices

Order:	2	3	4	5
#Models:	6	0	306	44,274

3.37 Monadic algebras

Monadic algebras appear in [46] and the number of non-isomorphic models from order 2 to order 7 is given in Table 37.

Table 37: Number of Monadic algebras

Order:	2	3	4	5	6	7
#Models:	8	0	10,432	0	0	0

3.38 Moufang quasigroups

Moufang quasigroups appear in [47] and the number of non-isomorphic models from order 2 to order 5 is given in Table 38.

Table 38: Number of Moufang quasigroups

Order:	2	3	4	5
#Models:	1	5	29	1,351

3.39 Multiplicative lattices

Multiplicative lattices appear in [48] and the number of non-isomorphic models from order 2 to order 4 is given in Table 39.

Table 39: Number of Multiplicative lattices

Order:	2	3	4
#Models:	6	175	25,872

3.40 Multiplicative semilattices

Multiplicative semilattices appear in [49] and the number of non-isomorphic models from order 2 to order 4 is given in Table 40.

Table 40: Number of Multiplicative semilattices

Order:	2	3	4
#Models:	6	220	32,234

3.41 Near-rings

Near-rings appear in [50] and the number of non-isomorphic models from order 2 to order 13 is given in Table 41. The numbers of near-rings up to order 7 were previously reported in the OEIS sequence A305858. We extend this sequence to order 13.

Table 41: Number of Near-rings

Order:	2	3	4	5	6	7	8	9	10	11	12	13
#Models:	3	5	35	10	99	24	3,856	486	535	139	54,694	454

3.42 Near-rings with identity

Near-rings with identity appear in [51] and the number of non-isomorphic models from order 2 to order 19 is given in Table 42.

Table 42: Number of Near-rings with identity

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
#Models:	1	1	6	1	1	1	53	11	1	1	11	1	1	1	4,274	1	26	1

3.43 Neardistributive lattices

Neardistributive lattices appear in [52] and the number of non-isomorphic models from order 2 to order 11 is given in Table 43.

Table 43: Number of Neardistributive lattices

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	1	2	4	9	22	60	174	532	1,700

3.44 Normal bands

Normal bands appear in [53] and the number of non-isomorphic models from order 2 to order 9 is given in Table 44.

Table 44: Number of Normal bands

Order:	2	3	4	5	6	7	8	9
#Models:	3	8	30	114	536	2,698	15,441	96,868

3.45 Normal valued lattice-ordered groups

Normal valued lattice-ordered groups appear in [54] and the number of non-isomorphic models from order 2 to order 7 is given in Table 45.

Table 45: Number of Normal valued lattice-ordered groups

Order:	2	3	4	5	6	7
#Models:	2	3	30	155	5,528	69,332

3.46 Ockham algebras

Ockham algebras appear in [55] and the number of non-isomorphic models from order 2 to order 10 is given in Table 46.

Table 46: Number of Ockham algebras

Order:	2	3	4	5	6	7	8	9	10
#Models:	1	3	13	45	197	827	3,654	16,058	71,709

3.47 Order algebras

Order algebras appear in [56] and the number of non-isomorphic models from order 2 to order 7 is given in Table 47.

3.48 Ordered monoids with zero

Ordered monoids with zero appear in [57] and the number of non-isomorphic models from order 2 to order 9 is given in Table 48.

Table 47: Number of Order algebras

Order:	2	3	4	5	6	7
#Models:	2	7	36	251	2,306	26,568

Table 48: Number of Ordered monoids with zero

Order:	2	3	4	5	6	7	8	9
#Models:	2	6	30	168	1,150	9,374	90,446	1,033,764

3.49 Ortholattices

Ortholattices appear in [58] and the number of non-isomorphic models from order 2 to order 12 is given in Table 49.

Table 49: Number of Ortholattices

Order:	2	3	4	5	6	7	8	9	10	11	12
#Models:	1	0	1	3	13	46	226	1,047	5,432	29,002	163,783

3.50 Orthomodular lattices

Orthomodular lattices appear in [59] and the number of non-isomorphic models from order 2 to order 14 is given in Table 50.

Table 50: Number of Orthomodular lattices

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14
#Models:	1	0	1	1	2	2	5	4	9	9	17	17	33

3.51 Partially ordered monoids

Partially ordered monoids appear in [60] and the number of non-isomorphic models from order 2 to order 6 is given in Table 51.

3.52 Partially ordered semigroups

Partially ordered semigroups appear in [61] and the number of non-isomorphic models from order 2 to order 5 is given in Table 52.

Table 51: Number of Partially ordered monoids

Order:	2	3	4	5	6
#Models:	4	37	549	13,371	504,634

Table 52: Number of Partially ordered semigroups

Order:	2	3	4	5
#Models:	11	173	4,753	198,838

3.53 Pocrims

Pocrims appear in [62] and the number of non-isomorphic models from order 2 to order 8 is given in Table 53.

Table 53: Number of Pocrims

Order:	2	3	4	5	6	7	8
#Models:	2	4	14	45	217	1,159	7,457

3.54 Quasi-MV-algebras

Quasi-MV-algebras appear in [63] and the number of non-isomorphic models from order 2 to order 8 is given in Table 54.

Table 54: Number of Quasi-MV-algebras

Order:	2	3	4	5	6	7	8
#Models:	1	1	9	9	467	567	153,163

3.55 Residuated partially ordered monoids

Residuated partially ordered monoids appear in [64] and the number of non-isomorphic models from order 2 to order 7 is given in Table 55.

3.56 Right hoops

Right hoops appear in [65] and the number of non-isomorphic models from order 2 to order 11 is given in Table 56.

Table 55: Number of Residuated partially ordered monoids

Order:	2	3	4	5	6	7
#Models:	2	5	28	186	1,795	21,904

Table 56: Number of Right hoops

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	2	5	10	23	49	111	244	545	1,203

3.57 Semigroups with zero

Semigroups with zero appear in [66] and the number of non-isomorphic models from order 2 to order 6 is given in Table 57.

Table 57: Number of Semigroups with zero

Order:	2	3	4	5	6
#Models:	2	12	90	960	15,759

3.58 Semirings with identity

Semirings with identity appear in [67] and the number of non-isomorphic models from order 2 to order 5 is given in Table 58.

Table 58: Number of Semirings with identity

Order:	2	3	4	5
#Models:	4	46	663	35,451

3.59 Semirings with identity and zero

Semirings with identity and zero appear in [68] and the number of non-isomorphic models from order 2 to order 5 is given in Table 59.

3.60 Semirings with zero

Semirings with zero appear in [69] and the number of non-isomorphic models from order 2 to order 5 is given in Table 60.

Table 59: Number of Semirings with identity and zero

Order:	2	3	4	5
#Models:	4	39	628	34,786

Table 60: Number of Semirings with zero

Order:	2	3	4	5
#Models:	8	246	628	55,315

3.61 Skew lattices

Skew lattices appear in [70] and the number of non-isomorphic models from order 2 to order 10 is given in Table 61.

Table 61: Number of Skew lattices

Order:	2	3	4	5	6	7	8	9	10
#Models:	3	7	21	53	173	531	1,971	7,903	37,390

3.62 Skew-fields

Skew-fields, also known as division ring, appear in [71] and the number of non-isomorphic models from order 2 to order 12 is given in Table 62.

Table 62: Number of Skew-fields

Order:	2	3	4	5	6	7	8	9	10	11	12
#Models:	2	3	3	5	0	7	4	6	0	11	0

3.63 Near-semirings

Near-semirings appear on p. 2 of [72] and the number of non-isomorphic models from order 2 to order 4 is given in Table 63.

3.64 Orthoimplication algebras

Orthoimplication algebras appear on p. 4 of [73] and the number of non-isomorphic models from order 2 to order 11 is given in Table 64.

Table 63: Number of Near-semirings

Order:	2	3	4
#Models:	22	531	25,364

Table 64: Number of Orthoimplication algebras

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	1	2	2	4	6	11	17	33	63

3.65 Orthomodular implication algebras

Orthomodular implication algebras appear on p. 4 of [73] and the number of non-isomorphic models from order 2 to order 12 is given in Table 65.

Table 65: Number of Orthomodular implication algebras

Order:	2	3	4	5	6	7	8	9	10	11	12
#Models:	1	1	2	2	4	6	11	17	33	63	137

3.66 Quasi-implication algebras

Quasi-implication algebras appear on p. 8 of [73] and the number of non-isomorphic models from order 2 to order 12 is given in Table 66.

Table 66: Number of Quasi-implication algebras

Order:	2	3	4	5	6	7	8	9	10	11	12
#Models:	1	1	2	3	5	8	13	20	33	53	88

3.67 Concurrent semirings

Concurrent semirings appear on p. 37 of [74] and the number of non-isomorphic models from order 2 to order 5 is given in Table 67.

3.68 Concurrent semiring with invariants

Concurrent semiring with invariants appear on p. 37 of [74] and the number of non-isomorphic models from order 2 to order 5 is given in Table 68.

Table 67: Number of Concurrent semirings

Order:	2	3	4	5
#Models:	1	4	43	14,029

Table 68: Number of Concurrent semiring with invariants

Order:	2	3	4	5
#Models:	1	8	199	10,260

3.69 Ground mereotopologies

Ground mereotopologies appear on p. 5 of [75] and the number of non-isomorphic models from order 2 to order 7 is given in Table 69.

Table 69: Number of Ground mereotopologies

Order:	2	3	4	5	6	7
#Models:	3	11	63	518	6,789	141,499

3.70 Left closure semigroups

Left closure semigroups appear on p. 76 of [76] and the number of non-isomorphic models from order 2 to order 7 is given in Table 70.

Table 70: Number of Left closure semigroups

Order:	2	3	4	5	6	7
#Models:	4	23	196	2,179	32,278	599,168

3.71 Left closure monoids

Left closure monoids appear on p. 76 of [76] and the number of non-isomorphic models from order 2 to order 7 is given in Table 71.

3.72 Domain range semigroups

Domain range semigroups appear on p. 81 of [76] and the number of non-isomorphic models from order 2 to order 4 is given in Table 72.

Table 71: Number of Left closure monoids

Order:	2	3	4	5	6	7
#Models:	3	15	109	1,071	14,571	255,798

Table 72: Number of Domain range semigroups

Order:	2	3	4
#Models:	3	66	48,498

3.73 Antidomain monoids

Antidomain monoids appear on p. 82 of [76] and the number of non-isomorphic models from order 2 to order 9 is given in Table 73.

Table 73: Number of Antidomain monoids

Order:	2	3	4	5	6	7	8	9
#Models:	1	2	8	40	261	2,444	33,055	1,682,176

3.74 Semilattice pseudo-complemented semigroups

Semilattice pseudo-complemented semigroups appear on p. 84 of [76] and the number of non-isomorphic models from order 2 to order 8 is given in Table 74.

Table 74: Number of Semilattice pseudo-complemented semigroups

Order:	2	3	4	5	6	7	8
#Models:	2	8	54	540	7,252	124,712	3,680,984

3.75 Closable semilattice pseudo-complemented semigroups

Closable semilattice pseudo-complemented semigroups appear in [76] and the number of non-isomorphic models from order 2 to order 8 is given in Table 75.

3.76 Involutory quandles

Involutory quandles appear on p. 2 of [77] and the number of non-isomorphic models from order 2 to order 11 is given in Table 76. The numbers of involutory quandles up to order

Table 75: Number of Closable semilattice pseudo-complemented semigroups

Order:	2	3	4	5	6	7	8
#Models:	1	4	24	191	2,046	30,693	1,661,964

10 were previously reported in the OEIS sequence A178432. We have added the number of involutory quandles of order 11.

Table 76: Number of Involutory quandles

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	3	5	13	41	142	665	4,288	36,455	436,672

3.77 Left Bol loops

Left Bol loops appear on p. 13 of [78] and the number of non-isomorphic models from order 2 to order 15 is given in Table 77.

Table 77: Number of Left Bol loops

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14	15
#Models:	1	1	2	1	2	1	11	2	2	1	8	1	2	3

3.78 Right Bol loops

Right Bol loops appear on p. 4 of [78] and the number of non-isomorphic models from order 2 to order 15 is given in Table 78.

Table 78: Number of Right Bol loops

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14	15
#Models:	1	1	2	1	2	1	11	2	2	1	8	1	2	3

3.79 Left conjugacy closed loops

Left conjugacy closed loops appear on p. 6 of [78] and the number of non-isomorphic models from order 2 to order 11 is given in Table 79.

Table 79: Number of Left conjugacy closed loops

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	1	2	1	5	1	24	7	18	1

3.80 Right conjugacy closed loops

Right conjugacy closed loops appear on p. 6 of [78] and the number of non-isomorphic models from order 2 to order 11 is given in Table 80.

Table 80: Number of Right conjugacy closed loops

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	1	2	1	5	1	24	7	18	1

3.81 Conjugacy closed loops

Conjugacy closed loops appear on p. 14 of [79] and the number of non-isomorphic models from order 2 to order 11 is given in Table 81.

Table 81: Number of Conjugacy closed loops

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	1	2	1	3	1	7	5	3	1

3.82 RIF loops

RIF loops appear on p. 6 of [78] and the number of non-isomorphic models from order 2 to order 11 is given in Table 82.

Table 82: Number of RIF loops

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	1	2	2	2	1	11	2	23	242

3.83 ARIF loops

ARIF loops appear on p. 6 of [78] and the number of non-isomorphic models from order 2 to order 11 is given in Table 83.

Table 83: Number of ARIF loops

Order:	2	3	4	5	6	7	8	9	10	11
#Models:	1	1	2	1	2	1	11	3	3	1

3.84 Extra loops

Extra loops appear on p. 14 of [79] and the number of non-isomorphic models from order 2 to order 15 is given in Table 84.

Table 84: Number of Extra loops

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14	15
#Models:	1	1	2	1	2	1	5	2	2	1	5	1	2	1

3.85 F-quasigroups

F-quasigroups appear on p. 8 of [78] and the number of non-isomorphic models from order 2 to order 12 is given in Table 85.

Table 85: Number of F-quasigroups

Order:	2	3	4	5	6	7	8	9	10	11	12
#Models:	1	5	13	19	6	41	88	116	20	109	155

3.86 Rectangular quasigroups

Rectangular quasigroups appear on p. 71 of [80] and the number of non-isomorphic models from order 2 to order 5 is given in Table 86.

Table 86: Number of Rectangular quasigroups

Order:	2	3	4	5
#Models:	3	7	40	1,413

3.87 Rectangular loops

Rectangular loops appear on p. 72 of [80] and the number of non-isomorphic models from order 2 to order 6 is given in Table 87.

Table 87: Number of Rectangular loops

Order:	2	3	4	5	6
#Models:	3	3	7	8	117

3.88 Trigroups

Trigroups appear in [81] and the number of non-isomorphic models from order 2 to order 4 is given in Table 88.

Table 88: Number of Trigroups

Order:	2	3	4
#Models:	3	46	43,973

3.89 Digroups

Digroups appear in [82] and the number of non-isomorphic models from order 2 to order 15 is given in Table 89.

Table 89: Number of Digroups

Order:	2	3	4	5	6	7	8	9	10	11	12	13	14	15
#Models:	2	2	4	2	6	2	10	4	7	2	17	2	8	5

3.90 Left disemigroups

Left disemigroups appear on p. 2 of [83] and the number of non-isomorphic models from order 2 to order 5 is given in Table 90.

Table 90: Number of Left disemigroups

Order:	2	3	4	5
#Models:	11	117	2,252	100,461

3.91 Right disemigroups

Right disemigroups appear on p. 2 of [83] and the number of non-isomorphic models from order 2 to order 5 is given in Table 91.

Table 91: Number of Right disemigroups

Order:	2	3	4	5
#Models:	11	117	2,252	100,461

3.92 Disemigroups

Disemigroups appear on p. 3 of [83] and the number of non-isomorphic models from order 2 to order 5 is given in Table 92.

Table 92: Number of Disemigroups

Order:	2	3	4	5
#Models:	6	41	513	51,391

3.93 Left I-Semirings

Left I-Semirings, also known as lazy semirings, appear in [84] and the number of non-isomorphic models from order 2 to order 8 is given in Table 93.

Table 93: Number of Left I-Semirings

Order:	2	3	4	5	6	7	8
#Models:	1	4	24	195	2,146	30,471	567,060

3.94 Doppelsemigroups

Doppelsemigroups appear on p. 11 of [85] and the number of non-isomorphic models from order 2 to order 5 is given in Table 94.

Table 94: Number of Doppelsemigroups

Order:	2	3	4	5
#Models:	8	77	1,217	68,177

3.95 Strong doppelsemigroups

Strong doppelsemigroups appear on p. 12 of [85] and the number of non-isomorphic models from order 2 to order 5 is given in Table 95.

Table 95: Number of Strong doppelsemigroups

Order:	2	3	4	5
#Models:	8	65	841	56,428

3.96 Dimonoids

Dimonoids appear on p. 12 of [85] and the number of non-isomorphic models from order 2 to order 5 is given in Table 96.

Table 96: Number of Dimonoids

Order:	2	3	4	5
#Models:	10	74	1,246	84,836

3.97 Commutative dimonoids

Commutative dimonoids appear on p. 12 of [85] and the number of non-isomorphic models from order 2 to order 6 is given in Table 97.

Table 97: Number of Commutative dimonoids

Order:	2	3	4	5	6
#Models:	3	14	101	1,495	102,268

3.98 Diassociative semigroups

Diassociative semigroups appear on p. 3 of [86] and the number of non-isomorphic models from order 2 to order 5 is given in Table 98.

Table 98: Number of Diassociative semigroups

Order:	2	3	4	5
#Models:	8	52	734	55,883

3.99 Duplicial semigroups

Duplicial semigroups appear on p. 5 of [86] and the number of non-isomorphic models from order 2 to order 4 is given in Table 99.

Table 99: Number of Duplicial semigroups

Order:	2	3	4
#Models:	15	251	8,898

3.100 Associative dialgebras

Associative dialgebras appear on p. 3 of [87] and the number of non-isomorphic models from order 2 to order 5 is given in Table 100.

Table 100: Number of Associative dialgebras

Order:	2	3	4	5
#Models:	8	52	734	55,883

3.101 Associative trialgebras

Associative trialgebras appear on p. 5 of [87] and the number of non-isomorphic models from order 2 to order 4 is given in Table 101.

Table 101: Number of Associative trialgebras

Order:	2	3	4
#Models:	15	185	5,914

3.102 Associative dimonoids

Associative dimonoids appear in [88] and the number of non-isomorphic models from order 2 to order 5 is given in Table 102.

Table 102: Number of Associative dimonoids

Order:	2	3	4	5
#Models:	8	52	734	55,883

4 Conclusions

In this paper, we report new integer sequences for the number of models of order n (for small integer values of n) for 100 algebras. These sequences have not been reported to the OEIS. We also increased the length of the existing OEIS integer sequences A178432 and A305858. This is accomplished with improved model enumeration algorithms implemented in the parallel finite model enumerator Mace4, running on a 24-cores computer.

5 Acknowledgments

João Araújo The results were supported by 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.
Mikoláš Janota The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

References

[1] Harrison, M. A. The Number of Isomorphism Types of Finite Algebras. *Proceedings of the American Mathematical Society*, vol. 17, no. 3, 1966, pp. 731–37. JSTOR, available at https://doi.org/10.2307/2035402. Accessed 10 Jan. 2023.

[2] European Mathematical Society. *Enumeration of Finite Groups*, available at https://euro-math-soc.eu/review/enumeration-finite-groups, 2011

[3] N. J. A. Sloane et al., *The On-Line Encyclopedia of Integer Sequences*, available at https://oeis.org, 2022.

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

[5] Araújo, J., Chow, C. and Janota, M. Boosting isomorphic model filtering with invariants. *Constraints* 27, 360–379 (2022). https://doi.org/10.1007/s10601-022-09336-x

[6] Araújo, J., Chow, C. and Janota, M. Filtering Isomorphic Models by Invariants. *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)* 4:1 – 4:9 (2021). https://drops.dagstuhl.de/opus/volltexte/2021/15295

[7] Araújo, J., Ramires, J., *MarcieX a model / theory / theorem database*, available at https://marciedb.pythonanywhere.com/, 2022.

[8] Sutcliffe, G., The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning* 59 (2017), no. 4, pp. 483-502.

[9] Araújo, J., Ramires, J., *MarcieX a model / theory / theorem database*, available at https://marciedb.pythonanywhere.com/, 2022.

[10] Anonymous Contributors. bci-algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=bci-algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[11] Anonymous Contributors. bck-algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=bck-algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[12] Anonymous Contributors. bck-join-semilattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=bck-join-semilattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[13] Anonymous Contributors. bck-lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=bck-lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[14] Anonymous Contributors. bck-meet-semilattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=bck-meet-semilattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[15] Anonymous Contributors. bilattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=bilattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[16] Anonymous Contributors. basic_logic_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=basic_logic_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[17] Anonymous Contributors. boolean_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=boolean_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[18] Anonymous Contributors. brouwerian_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=brouwerian_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[19] Chisolm, E. (2012). Geometric Algebra. *arXiv*. https://doi.org/10.48550/arXiv.1205.5935

[20] Anonymous Contributors. commutative_bck-algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_bck-algebras&rev=1665985443, 2022. [Online; accessed 21-December-2022].

[21] Anonymous Contributors. commutative_lattice-ordered_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_lattice-ordered_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[22] Anonymous Contributors. commutative_lattice-ordered_rings — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_lattice-ordered_rings&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[23] Anonymous Contributors. commutative_lattice-ordered_semigroups — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_lattice-ordered_semigroups&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[24] Anonymous Contributors. commutative_ordered_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_ordered_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[25] Anonymous Contributors. commutative_partially_ordered_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_partially_ordered_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[26] Anonymous Contributors. commutative_partially_ordered_semigroups — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_partially_ordered_semigroups&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[27] Anonymous Contributors. commutative_regular_rings — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_regular_rings&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[28] Anonymous Contributors. commutative_residuated_partially_ordered_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_residuated_partially_ordered_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[29] Anonymous Contributors. complemented_lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=complemented_lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[30] Anonymous Contributors. complemented_modular_lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=complemented_modular_lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[31] Anonymous Contributors. de_morgan_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=de_morgan_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[32] Anonymous Contributors. dense_linear_orders — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=dense_linear_orders&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[33] Anonymous Contributors. directoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=directoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[34] Anonymous Contributors. distributive_lattice_ordered_semigroups — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=distributive_lattice_ordered_semigroups&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[35] Anonymous Contributors. division_rings — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=division_rings&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[36] Anonymous Contributors. generalized_boolean_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=generalized_boolean_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[37] Anonymous Contributors. hilbert_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=hilbert_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[38] Anonymous Contributors. hoops — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=hoops&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[39] Anonymous Contributors. involutive_lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=involutive_lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[40] Anonymous Contributors. lattice-ordered_rings — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=lattice-ordered_rings&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[41] Anonymous Contributors. lattice-ordered_semigroups — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=lattice-ordered_semigroups&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[42] Anonymous Contributors. commutative_residuated_partially_ordered_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=commutative_residuated_partially_ordered_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[43] Anonymous Contributors. goedel_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=goedel_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[44] Anonymous Contributors. m-zeroid — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=m-zeroid&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[45] Anonymous Contributors. modular_ortholattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=modular_ortholattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[46] Anonymous Contributors. monadic_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=monadic_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[47] Anonymous Contributors. moufang_quasigroups — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=moufang_quasigroups&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[48] Anonymous Contributors. multiplicative_lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=multiplicative_lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[49] Anonymous Contributors. multiplicative_semilattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=multiplicative_semilattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[50] Anonymous Contributors. near-rings — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=near-rings&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[51] Anonymous Contributors. near-rings_with_identity — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=near-rings_with_identity&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[52] Anonymous Contributors. neardistributive_lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=neardistributive_lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[53] Anonymous Contributors. normal_bands — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=normal_bands&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[54] Anonymous Contributors. normal_valued_lattice-ordered_groups — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=normal_valued_lattice-ordered_groups&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[55] Anonymous Contributors. ockham_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=ockham_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[56] Anonymous Contributors. order_algebras — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=order_algebras&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[57] Anonymous Contributors. ordered_monoids_with_zero — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=ordered_monoids_with_zero&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[58] Anonymous Contributors. ortholattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=ortholattices&rev=1646193123, 2022. [Online; accessed 21-December-2022].

[59] Anonymous Contributors. orthomodular_lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=orthomodular_lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[60] Anonymous Contributors. partially_ordered_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=partially_ordered_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[61] Anonymous Contributors. partially_ordered_semigroups — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=partially_ordered_semigroups&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[62] Anonymous Contributors. pocrims — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=pocrims&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[63] Anonymous Contributors. quasi-mv-algebra — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=quasi-mv-algebra&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[64] Anonymous Contributors. residuated_partially_ordered_monoids — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=residuated_partially_ordered_monoids&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[65] Anonymous Contributors. right_hoops — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=right_hoops&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[66] Anonymous Contributors. semigroups_with_zero — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=semigroups_with_zero&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[67] Anonymous Contributors. semirings_with_identity — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=semirings_with_identity&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[68] Anonymous Contributors. semirings_with_identity_and_zero — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=semirings_with_identity_and_zero&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[69] Anonymous Contributors. semirings_with_zero — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=semirings_with_zero&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[70] Anonymous Contributors. skew_lattices — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=skew_lattices&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[71] Anonymous Contributors. division_rings — math-structures. https://math.chapman.edu/~jipsen/structures/doku.php?id=division_rings&rev=1614028295, 2021. [Online; accessed 21-December-2022].

[72] Kumar, J., and Krishna, K. V. (2013). Affine Near-Semirings over Brandt Semigroups. *arXiv*. https://doi.org/10.1080/00927872.2013.833211

[73] Megill, N. D., and Pavicic, M. (2003). Quantum Implication Algebras. *arXiv*. https://doi.org/10.1023/B:IJTP.0000006007.58191.da

[74] Hoare, T., and Möller, B., and Struth, G., and Wehrman, I. (2011). Concurrent Kleene Algebra and its Foundations. *J. Log. Algebr. Program.*. 80. 266-296. 10.1016/j.jlap.2011.04.005.

[75] Hahmann, Torsten and Grüninger, Michael. (2013). Complementation in Representable Theories of Region-Based Space. *Notre Dame J. of Formal Logic.* 54. 10.1215/00294527-1731344.

[76] Desharnais, J., Jipsen, P., Struth, G. (2009). Domain and Antidomain Semigroups. In: Berghammer, R., Jaoua, A.M., Möller, B. (eds) Relations and Kleene Algebra in Computer Science. RelMiCS 2009. *Lecture Notes in Computer Science,* vol 5827. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04639-1_6

[77] Lisitsa, A., and Vernitski, A. (2017). Automated Reasoning for Knot Semigroups and π-orbifold Groups of Knots. 10.1007/978-3-319-72453-9_1.

[78] Phillips, J. and Stanovský, David. (2010). Automated theorem proving in quasigroup and loop theory. *AI Commun.*. 23. 267-283. 10.3233/AIC-2010-0460.

[79] Niebrzydowski, M. (2013). On some ternary operations in knot theory. *arXiv.* https://doi.org/10.48550/arXiv.1301.0391

[80] Krapež, Aleksandar. (2012). Varieties of rectangular quasigroups. *Quasigroups and Related Systems.* 20. 71-80.

[81] Biyogmam, G., and Tchamna, S., and Tcheka, C. (2020). From quotient trigroups to groups. *Quasigroups and Related Systems.* 28. 29-41.

[82] Zhang, G., and Chen, Y. (2018). A construction of free digroup. *arXiv.* https://doi.org/10.1007/s00233-021-10161-6

[83] Biyogmam, G. R., and Tcheka, C. (2019). From Trigroups To Leibniz 3-Algebras. *arXiv.* https://doi.org/10.48550/arXiv.1904.12030

[84] Höfner, P., Laws for Left I-Semiring, http://www.kleenealgebra.de/kleene_db/lemmas/single.php?algebra=Left%20I-Semiring, 2007. [Online; accessed 21-December-2022].

[85] Zhuchok, Anatolii. (2018). Relatively free doppelsemigroups, Potsdam University Press, 2018.

[86] Foissy, L., Manchon, D., and Zhang, Y. (2020). Families of algebraic structures. *arXiv.* https://doi.org/10.48550/arXiv.2005.05116

[87] Loday, J., and Ronco, M. O. (2002). Trialgebras and families of polytopes. *arXiv.* https://doi.org/10.48550/arXiv.math/0205043.

[88] Loday, J. (2001). Dialgebras. *arXiv.* https://doi.org/10.48550/arXiv.math/0102053

(Concerned with sequences A000112, A001930, A084965, A087729, A178432, A181769, A226193, A305858, and A346414.)

2020 *Mathematics Subject Classification*: Primary 08-08; Secondary 68W99.
Keywords: Mace4, finite model enumeration, algebra.

Appendix C

Manuals of GAP Packages

Manuals of GAP packages included in this appendix:

1. `Magmaut`, which is available in the git repository https://gitlab.com/ChoiwahChow/magmaut.
2. `SmallSemigroups`
3. `SmallLoops`
4. `SmallQuandles`
5. `SmallMeadows`
6. `SmallInverseSemigroups`
7. `SmallSemiVarN12Idemps`

Magmaut

Version 1.0.1

João Araújo
Mikoláš Janota
Choiwah Chow

João Araújo Email: jjrsga@gmail.com
Homepage: https://docentes.fct.unl.pt/jj-araujo/

Mikoláš Janota Email: mikolas.janota@gmail.com
Homepage: http://sat.inesc-id.pt/~mikolas/

Choiwah Chow Email: choiwah.chow@gmail.com
Homepage: https://orcid.org/0000-0002-8383-8057

Abstract

The Magmaut package is a GAP package containing methods for calculating automorphism groups for magmas and algebras of type (2, 2, 2, ..., 1, 1, 1, ...), and methods for finding the isomorphism between two magmas. The algorithm employed in this package is particularly efficient for calculating the automorphism groups for semigroups, Rees zero-matrix semigroups, quasigroups, loops, and magmas.

Copyright

Contents

Chapter 1

The Magmaut package

1.1 Introduction

This is the manual for the Magmaut version 1.0.1 a package for GAP.

Magmaut 1.0.1 provides efficient tools to compute the automorphism groups of magmas and algebra of type (2, 2, ..., 1, 1,...), and methods to check whether two magmas are isomorphic.

The automorphism group function in this package works for magma and all its subtypes including groups, semigroups, monoids, quasigroups and loops, etc. The underlying algorithm is very efficient for many algebraic structures. It's elegance lies in its simplicity and extensibility, and it handles all algebraic structures the same way.

1.2 Overview

This manual starts with a brief description of the basic mathematical terminology about magmas (Chapter 3.1), homomorphism, isomorphism, and automorphisms (Chapter 3.2, Chapter 3.3). Then the magma automorphism and isomorphism functions are discussion (Chapter 4).

Chapter 2

Installing Magmaut

2.1 Package dependencies

The Magmaut package is written in GAP and C++. It runs in GAP version 4.10 or above, with C++ run-time support. Other than that, it has no other dependencies.

If you do not see the subfolder pkg/magmaut in the main directory of GAP then download the Magmaut package from the distribution website https://gap-packages.github.io/magmaut/ and unpack the downloaded file into the pkg subfolder.

2.2 Compiling the kernel module

C/C++ is used extensively in the Magmaut package for best performances. This GAP kernel module written in C/C++ must be compiled. That is, it is not possible to use the Magmaut package without first compiling it.

To compile the kernel component inside the `pkg/magmaut-0.1.0` directory, type

```
./configure
make
```

The package has been tested with GCC compiler versions 7, 8 and 9 in the 64-bit Linux platform.

2.3 Building the documentation

The documentation of Magmaut can be built with the simple command inside the `pkg/magmaut-0.1.0` directory:

```
make doc
```

2.4 Testing your installation

Test files conforming to GAP standards are provided for Magmaut and are located in the folder named tst. The following command runs all tests for the Magmaut package:

```
ReadPackage("magmaut","tst/testall.g");
```

Chapter 3

Mathematical Background

In this Chapter, we will lay down the terminologies used for the rest of the manual.

3.1 Magma

A set with one closed binary operation (denoted by $\cdot$ here) is called a magma. Magma is also known as groupoid, and is the most basic type of algebraic structures because it requires no further axiomization. Clearly, common algebraic structures including semigroups, quasigroups, loops, and groups are all magmas. Consequently, the methods in this package apply to them. However, methods developed specifically for specific algebraic structures may conceivably out-perform the general methods provided in this package. Nonetheless, the methods in this package perform very well in all cases. For example, it compares quite well against the loops package for finding the automorphism group of a quasigroup or loop.

A generating set of a magma is a subset such that every element of the magma can be expressed as a combination, under the magma operation, of finitely many elements of the subset.

3.2 Homomorphism, Isomorphism and Automorphism

If M, N are two magmas, and the function $f : M \to N$ respects the binary operations, that is, $f(m) \cdot f(n) = f(m \cdot n)$ for every $m \in M$ and $n \in N$, then f is called a *homomorphism* from M to N. If f is a bijection, then it is called an *isomorphism*. If an isomorphism exists between two magmas, then the two magmas are said to be *isomorphic*. Finally, an *automorphism* is simply an isomorphism from a magma onto itself.

3.3 Algebra of Type (2, 2, ..., 1, 1, 1, ...)

An algebra $A$of type (2, 2, ..., 1, 1, 1, ...) on a domain D consists of a list of binary operations and unary operations on D. An automorphism of A is an automorphism for each of the binary operations, and which commutes with each of the unary operation in function compositions.

Chapter 4

Magma Automorphism/Isomorphism

4.1 Invariants

Every element in a magma has some properties that any homomorphic mapping of the magma must respect. For example, if e is the identity element of a magma M , and f is an automorphism of M, then f must map e to e. Another common example is idempotent: if x is an idempotent, and if f maps x to y, then y must also be an idempotent. Thus, if we are searching for automorphisms, then we only need to look into those bijections that respect these *invariant* properties. This cuts down the search space dramatically and consequently makes the search much faster.

In case of Algebras consisting of a list of binary operations $b_1, b_2, ..., b_n$, The invariants for all the binary operations for each domain element can be pooled together to increase their discrimination power to further cut down the search space.

4.1.1 MAGMAS_Invariants

▷ `MAGMAS_Invariants(Q)` (operation)

Returns: The invariants vector of the magma `Q`.

This operation returns the Invariant vector of a Magma `Q`, which is the list[`A`, `B`], where A is a list of the form [[I_1, n_1], [I_2, n_2],..], i.e. the invariant I_i occurs n_i times in `Q`, and where `B`[i] is a subset of [1..Size(`Q`)] corresponding to elements of `Q` with invariant I_i. Note that domain elements with different invariant vectors cannot be mapped to each other in isomorphism, and that magmas with different invariant vectors cannot be isomorphic.

Example

```
gap> MAGMAS_Invariants(AllSmallGroups(16)[1]);
[ [ [ [ 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 2, 0, 16, 0, 0, 1 ], 1 ],
      [ [ 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 2, 0, 16, 0, 0, 1 ], 1 ],
      [ [ 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 2, 0, 16, 0, 0, 1 ], 2 ],
      [ [ 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 2, 0, 16, 0, 0, 1 ], 4 ],
      [ [ 17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 16, 0, 0, 1 ], 8 ] ],
  [ [ 1 ], [ 5 ], [ 4, 11 ], [ 3, 9, 10, 15 ], [ 2, 6, 7, 8, 12, 13, 14, 16 ] ] ]
```

In the example above, the domain elements 1 and 5 have unique invariant vectors, and the domain elements 4 and 11 have the same invariant vectors. Thus, in finding possible automorphism mappings, there is no need to consider mapping 1 with another other domain elements other than 1 itself, and for 4, we only need to consider mapping it to 4 and 11.

4.1.2 AreEqualMAGMA_Invariants

▷ `AreEqualMAGMA_Invariants(S, T)` (operation)

Returns: `true` or `false`.

If S and T are invariant vectors, then this operation will show whether S and T are equivalent invariants generated by using the operation `MAGMAS_Invariants` (4.1.1).

Example

```
gap> S := MAGMAS_Invariants(AllSmallGroups(4)[2]);;
gap> T := MAGMAS_Invariants(AllSmallGroups(4)[1]);;
gap> AreEqualMAGMA_Invariants(S, T);
false
gap> IsomorphismGroups(AllSmallGroups(4)[1], AllSmallGroups(4)[2]);
fail
```

Example

```
gap> a := Group((1,2), (3,4));;
gap> a = AllSmallGroups(4)[2];
false
gap> S := MAGMAS_Invariants(a);;
gap> T := MAGMAS_Invariants(AllSmallGroups(4)[2]);;
gap> AreEqualMAGMA_Invariants(S, T);
true
```

4.2 Automorphism Group

4.2.1 Magmaut_AutomorphismGroup

▷ `Magmaut_AutomorphismGroup(Q)` (operation)

▷ `AutomorphismGroup(Q)` (operation)

Returns: The automorphism group of Q.

This operation makes use of the operation `MAGMAS_Invariants` (4.1.1) to find the invariant vectors, with respect to the magma Q, of each elements in the domain, and use these invariant vectors to limit the search space for an isomorphic mapping from Q into itself.

Note that `AutomorphismGroup` is already defined in GAP and the `AutomorphismGroup` in ths package has lower priority than it. To use the automorphism in the package for declared groups, either convert them into magmas first, only use `Magmaut_AutomorphismGroup` instead.

Example

```
gap> AutomorphismGroup(MagmaByMultiplicationTable(MultiplicationTable(AllSmallGroups(16)[1])));
<permutation group with 7 generators>
```

Example

```
gap> Magmaut_AutomorphismGroup(AllSmallGroups(16)[1]);
<permutation group with 7 generators>
```

4.2.2 Algebra_AutomorphismGroup

▷ `Algebra_AutomorphismGroup(L)` (operation)

Returns: The automorphism group of L.

L is an algebra of type (2, 2, 2, ..., 1, 1, 1, ...). It is represented as a list of binary operations (as multiplication tables, i.e. 2-dimensional arrays), followed by unary operations (as lists): [b_1, b_2,

b_3, ..., u_1, u_2, u_3, ...]. An automorphism A for an algebra This operation makes use of the operation MAGMAS_Invariants (4.1.1) to find the invariant vectors of each elements in the domain with respect to each of the binary operations and the invariant vectors for each elements are pooled together to increase their discriminating powers. This helps limit the search space for the automorphism group.

Example

```
gap> a := AllSmallGroups(4);;
gap> b := List(a, t->MultiplicationTable(t));;   # algebra with 2 binary operations
gap> Add(b, [3, 2, 1, 4]);;    # add unary operation
gap> Algebra_AutomorphismGroup(b);
Group([ (2,4) ])
```

4.3 Magmas Isomorphism

4.3.1 IsomorphismMagmas

▷ IsomorphismMagmas(*S*, *T*) (operation)

Returns: An isomorphism between *S* and *T*, or fail if they are not isomorphic.

This operation will try to find an isomorphism between magmas *S* and *T* are magmas. It will return the isomorphism found if they are indeed isomorphic, and returns fail otherwise.

This operation will make use of the operation MAGMAS_Invariants (4.1.1) and AreEqualMAGMA_Invariants (4.1.2) to quickly detect those input magmas that cannot possibly be isomorphic if they have different invariant vectors. It will also use the invariant vectors to limit the search space for an isomorphic mapping between the two magmas.

Example

```
gap> IsomorphismMagmas(AllSmallGroups(4)[1], AllSmallGroups(4)[2]);
fail
```

Example

```
gap> a := Group((1,2), (3,4));;
gap> a = AllSmallGroups(4)[2];
false
gap> IsomorphismMagmas(a, AllSmallGroups(4)[2]);
()
```

Index

SmallSemigroups

Version 0.1.0

Joao Araújo
Mikoláš Janota
Choiwah Chow

Joao Araújo Email: jjrsga@gmail.com
Homepage: https://docentes.fct.unl.pt/p191/

Mikoláš Janota Email: mailto://mikolas.janota@gmail.com
Homepage: http://sat.inesc-id.pt/ mikolas/

Choiwah Chow Email: choiwah.chow@gmail.com
Homepage: https://orcid.org/my-orcid?orcid=0000-0002-2067-0568/

Abstract

The SmallSemigroups package is a GAP package containing methods and data to provide the complete list of non-isomorphic models for:

Semigroups of orders 2 to 7.

Copyright

Contents

Chapter 1

Small Semigroups Package

1.1 Accessing Small Semigroups

AllSmallSemigroups is an accessor function for accessing the complete list of non-isomorphic semigroups models from orders 2 to 7.

The models are generated by Mace4 [**?**], using the definitions in Section 1.2.

1.2 Definition of Semigroups

x * (y * z) = (x * y) * z.

1.3 Dependency

This package has dependencies on the Magmaut package.

Chapter 2

Small Algebras

2.1 Accessing Small Semigroups

AllSmallSemigroups is an accessor function for accessing the complete list of non-isomorphic Semigroups of orders 2 to 7.

2.1.1 AllSmallSemigroups

▷ `AllSmallSemigroups(`*m*`)` (function)

This function returns all the semigroups of order *m* from the library.
No check is performed to verify that *m* is a valid argument.
In `AllSmallSemigroups`, an error will be signaled if the semigroups of order *m* are not available.

Example

```
gap> Length(AllSmallSemigroups(2));
5
```

The output semigroups are represented by its operation table for *, as given in Section 1.2.

2.2 Checking Models of Small Semigroups

"IsSemigroup" is a function for checking a model against the complete list of non-isomorphic Semigroups of orders 2 to 7. It returns true if the model is a Small Semigroups, false otherwise.

2.2.1 IsSemigroup

▷ `IsSemigroup(`*m*`)` (function)

This function checks whether *m*, which must be a multiplication table, is isomorphic to a Semigroup in the library.
No check is performed to verify that *m* is a valid argument.

Example

```
gap> IsSemigroup(AllSmallSemigroups(2)[1]);
true
```

References

SmallLoops

Version 0.1.0

Joao Araújo
Mikoláš Janota
Choiwah Chow

Joao Araújo Email: jjrsga@gmail.com
Homepage: https://docentes.fct.unl.pt/p191/

Mikoláš Janota Email: mailto://mikolas.janota@gmail.com
Homepage: http://sat.inesc-id.pt/ mikolas/

Choiwah Chow Email: choiwah.chow@gmail.com
Homepage: https://orcid.org/my-orcid?orcid=0000-0002-2067-0568/

Abstract

The SmallLoops package is a GAP package containing methods and data to provide the complete list of non-isomorphic models for:

Loops of orders 2 to 7.

Copyright

Contents

Chapter 1

Small Loops Package

1.1 Accessing Small Loops

AllSmallLoops is an accessor function for accessing the complete list of non-isomorphic loops models from orders 2 to 7.

The models are generated by Mace4 [**?**], using the definitions in Section 1.2.

1.2 Definition of Loops

x * y = x * z -> y = z.
y * x = z * x -> y = z.
x * 0 = x.
0 * x = x.

1.3 Dependency

This package has dependencies on the Magmaut package.

Chapter 2

Small Algebras

2.1 Accessing Small Loops

AllSmallLoops is an accessor function for accessing the complete list of non-isomorphic Loops of orders 2 to 7.

2.1.1 AllSmallLoops

▷ `AllSmallLoops(`*m*`)` (function)

This function returns all the loops of order *m* from the library.
No check is performed to verify that *m* is a valid argument.
In `AllSmallLoops`, an error will be signaled if the loops of order *m* are not available.

Example

```
gap> Length(AllSmallLoops(2));
1
```

The output loops are represented by its operation table for *, as given in Section 1.2.

2.2 Checking Models of Small Loops

"IsASmallLoop" is a function for checking a model against the complete list of non-isomorphic Loops of orders 2 to 7. It returns true if the model is a Small Loops, false otherwise.

2.2.1 IsASmallLoop

▷ `IsASmallLoop(`*m*`)` (function)

This function checks whether *m*, which must be a multiplication table, is isomorphic to a Loop in the library.
No check is performed to verify that *m* is a valid argument.

Example

```
gap> IsASmallLoop(AllSmallLoops(2)[1]);
true
```

References

SmallQuandles

Version 0.1.0

Joao Araújo
Mikoláš Janota
Choiwah Chow

Joao Araújo Email: jjrsga@gmail.com
Homepage: https://docentes.fct.unl.pt/p191/

Mikoláš Janota Email: mailto://mikolas.janota@gmail.com
Homepage: http://sat.inesc-id.pt/ mikolas/

Choiwah Chow Email: choiwah.chow@gmail.com
Homepage: https://orcid.org/my-orcid?orcid=0000-0002-2067-0568/

Abstract

The SmallQuandles package is a GAP package containing methods and data to provide the complete list of non-isomorphic models for:

Quandles of orders 2 to 9.

Copyright

Contents

Chapter 1

Small Quandles Package

1.1 Accessing Small Quandles

AllSmallQuandles is an accessor function for accessing the complete list of non-isomorphic quandles models from orders 2 to 9.

The models are generated by Mace4 [**?**], using the definitions in Section 1.2.

1.2 Definition of Quandles

% left-selfdistributive
x v (y v z) = (x v y) v (x v z).
% right-selfdistributive
(x ^ y) ^ z = (x ^ z) ^ (y ^ z).
%
(x v y) ^ x = y.
x v (y ^ x) = y.
x v x = x.

1.3 Dependency

This package has dependencies on the Magmaut package.

Chapter 2

Small Algebras

2.1 Accessing Small Quandles

AllSmallQuandles is an accessor function for accessing the complete list of non-isomorphic Quandles of orders 2 to 9.

2.1.1 AllSmallQuandles

▷ `AllSmallQuandles(m)` (function)

This function returns all the quandles of order m from the library.
No check is performed to verify that m is a valid argument.
In `AllSmallQuandles`, an error will be signaled if the quandles of order m are not available.

Example

```
gap> Length(AllSmallQuandles(2));
1
```

The output quandles are represented by its operation table for ^, v, as given in Section 1.2.

2.2 Checking Models of Small Quandles

"IsASmallQuandle" is a function for checking a model against the complete list of non-isomorphic Quandles of orders 2 to 9. It returns true if the model is a Small Quandles, false otherwise.

2.2.1 IsASmallQuandle

▷ `IsASmallQuandle(m)` (function)

This function checks whether m, which must be a multiplication table, is isomorphic to a Quandle in the library.
No check is performed to verify that m is a valid argument.

Example

```
gap> IsASmallQuandle(AllSmallQuandles(2)[1]);
true
```

References

SmallMeadows

Version 0.1.0

Joao Araújo
Mikoláš Janota
Choiwah Chow

Joao Araújo Email: jjrsga@gmail.com
Homepage: https://docentes.fct.unl.pt/p191/

Mikoláš Janota Email: mailto://mikolas.janota@gmail.com
Homepage: http://sat.inesc-id.pt/ mikolas/

Choiwah Chow Email: choiwah.chow@gmail.com
Homepage: https://orcid.org/my-orcid?orcid=0000-0002-2067-0568/

Abstract

The SmallMeadows package is a GAP package containing methods and data to provide the complete list of non-isomorphic models for:

Meadows of orders 2 to 22.

Copyright

Contents

Chapter 1

Small Meadows Package

1.1 Accessing Small Meadows

AllSmallMeadows is an accessor function for accessing the complete list of non-isomorphic meadows models from orders 2 to 22.

The models are generated by Mace4 [**?**], using the definitions in Section 1.2.

1.2 Definition of Meadows

```
% definition of meadows
    % introduction info: https://meadowsite.wordpress.com/#:~:text=A%20meadow%20is%20a%20commutative,equat
    % definition of meadows from
    % https://www.sciencedirect.com/science/article/pii/S2352220814000546?via%3Dihub
    % A meadow is a commutative ring with a total inverse operator satisfying 0^{-1} = 0.
    % https://www.sciencedirect.com/science/article/pii/S2352220814000546?via%3Dihub
    % commutative ring Definition 2.1
    %Addition
    ( x + y ) + z = x + ( y + z ). %associativity
    x + y = y + x. %commutativity
    x + 0 = x. %neutral element
    x + -x = 0. %inverse element
    %Multiplication
    ( x * y ) * z = x * ( y * z ). %associativity
    x * y = y * x. %commutativity
    x * 1 = x. %neutral element
    %Distributive Law
    x * ( y + z ) = ( x * y ) + ( x * z ).
    % definition 2.7, generalized inverse.
    x * (x * x') = x.
    x' * (x' * x) = x'.
    % symmetry breaking
    0' = 0.
    1' = 1.
```

1.3 Dependency

This package has dependencies on the Magmaut package.

Chapter 2

Small Algebras

2.1 Accessing Small Meadows

AllSmallMeadows is an accessor function for accessing the complete list of non-isomorphic Meadows of orders 2 to 22.

2.1.1 AllSmallMeadows

▷ `AllSmallMeadows(m)` (function)

This function returns all the meadows of order *m* from the library.
No check is performed to verify that *m* is a valid argument.
In `AllSmallMeadows`, an error will be signaled if the meadows of order *m* are not available.

Example

```
gap> Length(AllSmallMeadows(2));
1
```

The output meadows are represented by its operation table for +, *, -, as given in Section 1.2.

2.2 Checking Models of Small Meadows

"IsASmallMeadow" is a function for checking a model against the complete list of non-isomorphic Meadows of orders 2 to 22. It returns true if the model is a Small Meadows, false otherwise.

2.2.1 IsASmallMeadow

▷ `IsASmallMeadow(m)` (function)

This function checks whether *m*, which must be a multiplication table, is isomorphic to a Meadow in the library.
No check is performed to verify that *m* is a valid argument.

Example

```
gap> IsASmallMeadow(AllSmallMeadows(2)[1]);
true
```

References

SmallInverseSemigrou

Version 0.1.0

Joao Araújo
Mikoláš Janota
Choiwah Chow

Joao Araújo Email: jjrsga@gmail.com
Homepage: https://docentes.fct.unl.pt/p191/

Mikoláš Janota Email: mailto://mikolas.janota@gmail.com
Homepage: http://sat.inesc-id.pt/ mikolas/

Choiwah Chow Email: choiwah.chow@gmail.com
Homepage: https://orcid.org/my-orcid?orcid=0000-0002-2067-0568/

Abstract

The SmallInverseSemigroups package is a GAP package containing methods and data to provide the complete list of non-isomorphic models for:

Inverse Semigroups of orders 2 to 9.

Copyright

Contents

Chapter 1

Small Inverse Semigroups Package

1.1 Accessing Small Inverse Semigroups

AllSmallInverseSemigroup is an accessor function for accessing the complete list of non-isomorphic inverse semigroups models from orders 2 to 9.

The models are generated by Mace4 [**?**], using the definitions in Section 1.2.

1.2 Definition of Inverse Semigroups

Binary operation *

Unary operation ’

(x * y) * z = x * (y * z).

x” = x.

(x * x’) * x = x.

((x * x’) * y) * y’ = ((y * y’) * x) * x’.

1.3 Dependency

This package has dependencies on the Magmaut package.

Chapter 2

Small Algebras

2.1 Accessing Small Inverse Semigroups

AllSmallInverseSemigroup is an accessor function for accessing the complete list of non-isomorphic Inverse Semigroups of orders 2 to 9.

2.1.1 AllSmallInverseSemigroup

▷ `AllSmallInverseSemigroup(`*m*`)` (function)

This function returns all the inverse semigroups of order *m* from the library.

No check is performed to verify that *m* is a valid argument.

In `AllSmallInverseSemigroup`, an error will be signaled if the inverse semigroups of order *m* are not available.

Example

```
gap> Length(AllSmallInverseSemigroup(2));
2
```

The output inverse semigroups are represented by its operation table for *, as given in Section 1.2.

2.2 Checking Models of Small Inverse Semigroups

"IsASmallInverseSemigroup" is a function for checking a model against the complete list of non-isomorphic Inverse Semigroups of orders 2 to 9. It returns true if the model is a Small Inverse Semigroups, false otherwise.

2.2.1 IsASmallInverseSemigroup

▷ `IsASmallInverseSemigroup(`*m*`)` (function)

This function checks whether *m*, which must be a multiplication table, is isomorphic to a InverseSemigroup in the library.

No check is performed to verify that *m* is a valid argument.

Example

```
gap> IsASmallInverseSemigroup(AllSmallInverseSemigroup(2)[1]);
true
```

References

SmallSemiVarN12Iden

Version 0.1.0

Joao Araújo
Choiwah Chow

Joao Araújo Email: jjrsga@gmail.com
Homepage: https://docentes.fct.unl.pt/p191/

Choiwah Chow Email: choiwah.chow@gmail.com
Homepage: https://www.linkedin.com/in/choiwah-chow-cfa-8ba8554/

Abstract

The SmallSemiVarN12Idemps package is a GAP package containing methods and data to provide the complete list of non-isomorphic models for:

Semigroup Variety N12 Idempotents of orders 2 to 9.

Copyright

Contents

Chapter 1

Small Semigroup Variety N12 Idempotents Package

1.1 Accessing Small Semigroup Variety N12 Idempotents

AllSmallSemiVarN12Idemps is an accessor function for accessing the complete list of non-isomorphic semigroup variety N12 idempotents models from orders 2 to 9.

The models are generated by Mace4 [McC09], using the definitions in Section 1.2.

1.2 Definition of Semigroup Variety N12 Idempotents

x * (y * z) = (x * y) * z.
(x * x) * x = x * x.
x * y = y * x.
x * x = y * y.

1.3 Dependency

This package has dependencies on the Magmaut package.

Chapter 2

Small Algebras

2.1 Accessing Small Semigroup Variety N12 Idempotents

AllSmallSemiVarN12Idemps is an accessor function for accessing the complete list of non-isomorphic Semigroup Variety N12 Idempotents of orders 2 to 9.

2.1.1 AllSmallSemiVarN12Idemps

▷ `AllSmallSemiVarN12Idemps(`*m*`)` (function)

This function returns all the semigroup variety N12 idempotents of order *m* from the library.
No check is performed to verify that *m* is a valid argument.
In `AllSmallSemiVarN12Idemps`, an error will be signaled if the semigroup variety N12 idempotents of order *m* are not available.

Example

```
gap> Length(AllSmallSemiVarN12Idemps(2));
1
```

The output semigroup variety N12 idempotents are represented by its operation table for *, as given in Section 1.2.

2.2 Checking Models of Small Semigroup Variety N12 Idempotents

"IsASmallSemiVarN12Idemps" is a function for checking a model against the complete list of non-isomorphic Semigroup Variety N12 Idempotents of orders 2 to 9. It returns true if the model is a Small Semigroup Variety N12 Idempotents, false otherwise.

2.2.1 IsASmallSemiVarN12Idemps

▷ `IsASmallSemiVarN12Idemps(`*m*`)` (function)

This function checks whether *m*, which must be a multiplication table, is isomorphic to a SemiVarN12Idemps in the library.
No check is performed to verify that *m* is a valid argument.

Example

```
gap> IsASmallSemiVarN12Idemps(AllSmallSemiVarN12Idemps(2)[1]);
true
```

References

[McC09] W. McCune. Prover9, prover9 manual, version 2009-02a. 2009. 4

Index

Appendix D

Configuration File of GAP Package Generator

A sample configuration file for generating loops of orders 2 to 7 is attached here.

```
[ROOT]
# There must be at least one author, and is the main author
Author1 = Joao Araújo
Email1 = jjrsga@gmail.com
Homepage1 = https://docentes.fct.unl.pt/p191/

# List of other authors separated by semi-colon
OtherAuthors=Choiwah
Chow;choiwah.chow@gmail.com;https://www.linkedin.com/in/choiwah-chow-cfa-8ba8554/
CopyrightYear = 2022

BaseName = Loops

# Algebra names must be one word, no spaces.  Display names can be free text
SingularAlgebraName = Loop
AlgebraName = loops
AlgebraDisplayName = Loops
AlgebraDisplayNameLowerCase = loops
PackageName = SmallLoops
Version = 0.1.0
DateOfRelease = 09/20/2022
Status = dev
PackageWWWHome = https://gap-packages.github.io/SmallLoops/
SourceRepository = https://github.com/gap-packages/SmallLoops/

# The file path for the GAP binary (i.e. directory in which the gap executable can
be found)
GAPPATH = /home/choiwah/tp/gap-4.10.2

NonIsoDir = non_iso_outputs

PrebuiltDir = old

[MACE]
# Use ";" to separate input files that form a partition of the search space.
InputFiles = 32_loop.in

InputDir = inputs
OutputDir = outputs

# Range of domain size, from MinDomainSize to MaxDomainSize, inclusive.
MinDomainSize = 2
MaxDomainSize = 7

# skip the following domain size for Mace4 and/or isomorphic model elimination,
# domain list is comma-separated
# SkipMace4 = 8
# Mace4Prebuilt = loops_8_0.out

AdditionalFilters = | egrep "^-\*\["
# Mace4exe = ../LADR-2017-11A-V2/bin/mace4
Mace4exe = bin/mace4

# Number of threads to run MACE4 in parallel jobs.
Threads = 30


[NONISO]
Filter = ../bin/isofilter
```

```
MinModelsInFile = 1
NumRandom = 50
MaxRandom = 20
SamplingFreq = 100
WorkingDir = outputs/outputs_py
OutputDir = outputs/outputs_py
SplitModelexe = bin/splitModels

# The Prebuilt files are in the [PrebuiltDir]
# SkipNonIso = loops_8_0.out
# NonIsoPrebuilt = loops_8_0_1.out.f


[GAP_PACKAGE]
TemplatesDir = templates

AlgebraDefinitionFile = 32_loop.def
Prefix = SMALLLOOPS
```