

The Monoids of Order Eight and Nine

Andreas Distler¹ and Tom Kelsey²

¹ School of Mathematics and Statistics, Mathematical Institute,
North Haugh, St Andrews, KY16 9SS, UK
`andreas@mcs.st-and.ac.uk`

² School of Computer Science, Jack Cole Building,
North Haugh, St Andrews, KY16 9SX, UK
`tom@cs.st-and.ac.uk`

Abstract. We describe the use of symbolic algebraic computation allied with AI search techniques, applied to the problem of the identification, enumeration and storage of all monoids of order 9 or less. Our approach is novel, using computer algebra to break symmetry and constraint satisfaction search to find candidate solutions. We present new results in algebraic combinatorics: up to isomorphism and anti-isomorphism, there are 858,977 monoids of order 8 and 1,844,075,697 monoids of order 9.

1 Introduction

The aim of this paper is to find all solutions to a class of problems in algebraic combinatorics. This is a well-known research area: it is natural when discussing, for example, various types of latin squares to try to resolve the question of how many of each type exist. As well as obtaining the correct answer in terms of number of solutions, we aim to **store** each solution so that they can be analysed in terms of their structure by algebraists. This second aim means that we are not searching for a purely constructive solution to the enumeration problem; we generate and store a canonical example from each equivalence class of solutions.

The On-Line Encyclopedia of Integer Sequences [1] contains numerous examples of known initial sequences of enumerations of algebraic and combinatoric structures. The sequence that this paper extends is A058133: the numbers of monoids of order n , considered to be equivalent when they are isomorphic or anti-isomorphic. Currently values for $n \leq 7$ have been published.

Definition 1. *A monoid is an algebraic structure equipped with a closed and associative binary operator, and an identity element. More formally, a monoid is a tuple $\langle S, *, e \rangle$ where S is a set; $*$: $S \times S \rightarrow S$ satisfies $x * (y * z) = (x * y) * z \quad \forall x, y, z \in S$; and $e \in S$ satisfies $x * e = x = e * x \quad \forall x \in S$.*

*If $\langle S, *, e \rangle$ is a monoid, and $|S| = n$, then $\langle S, *, e \rangle$ has order n .*

Monoids can be thought of as semigroups having a multiplicative identity. Groups are special cases of monoids; each group element has a multiplicative inverse. Throughout this paper we consider only finite monoids, where S is a finite set.

Definition 2. Let $\langle M_1, *_1, e_1 \rangle$ and $\langle M_2, *_2, e_2 \rangle$ be two monoids of the same order. A bijection $g : M_1 \rightarrow M_2$ is an isomorphism if it respects the multiplication – i. e. $(a *_1 b)^g = a^g *_2 b^g$ – and an anti-isomorphism if it inverts the multiplication – i. e. $(a *_1 b)^g = b^g *_2 a^g$. If such a bijection exists the monoids are isomorphic, respectively anti-isomorphic, and are equivalent if either.

In this paper we are interested in monoids up to equivalence. Thus we can choose the underlying set to be $\{1, 2, \dots, n\}$. We then represent monoids by their multiplication tables, with rows and columns indexed from 1 up to n .

Table 1: Example multiplication table, and its image under $(1, 4)$

*	1	2	3	4	*	1	2	3	4
1	1	2	3	4	1	3	1	3	1
2	2	1	3	4	2	1	4	3	2
3	3	3	3	3	3	3	3	3	3
4	4	4	3	3	4	1	2	3	4

Isomorphism of monoids induces an action on tables. Given a permutation g of the members of S , we modify the table by permuting each row according to g , then each column, and finally permuting the values. An anti-isomorphism is the result of an isomorphism action followed by transposing the table. The effect of applying permutation $(1, 4)$ is shown in Table 1. Since permutations of a finite set, permutations of rows and columns of a multiplication table, and table transposition are all invertible, (anti-)isomorphism is a reflexive, symmetric and transitive relation on monoids, and is hence an equivalence relation.

In common with many algebraic enumeration problems, there is a combinatorial explosion as n increases. There are n choices for each of the n^2 positions in a multiplication table. For $n = 9$ and 10 , the number of choices is approximately 1.5×10^{17} and 10^{20} respectively. This increase in problem size effectively rules out the obvious exhaustive search approach of generating each table, checking if the monoid axioms hold, then checking whether or not an (anti-)isomorphic version of the table has already been found.

Our approach is to develop algebraic results that allow us to devise algorithms for search-space reduction. We implement these algorithms in symbolic computational algebra; formulate the remaining problems in terms of constraints on solution tables; use advanced AI backtrack search techniques to find solutions; then (in some cases) use computational algebra to decide which canonical representative of (anti-)isomorphic equivalence class to accept as our unique solution.

In the remainder of this introduction we describe the computational algebra tools and techniques used, the basic principles of Constraint Satisfaction and the solver we use, and formalise notions regarding symmetry-breaking in Constraint Satisfaction Problems. In Section 2 we give a detailed derivation of the algebraic and AI search methods used, together with proofs of soundness and – where appropriate – completeness of the algorithms used. We summarise our results in Section 3, and provide concluding remarks and an indication of future avenues of research in Section 4.

1.1 GAP & Computational Algebra

Since our problem domain involves binary operators on finite sets, permutations, identity elements, action homomorphisms and symmetry groups, we use specialist software that provides robust, efficient and extensive implementations of algorithms in abstract algebra. GAP [2] (Groups, Algorithms and Programming) is a system for computational discrete algebra with particular emphasis on, but not restricted to, computational group theory. GAP provides a large library of functions which implement algebraic algorithms.

For our purposes, any advanced computational algebra system could be used. However, we rely heavily on efficient GAP code [3] that tests canonicity of an image of a set of points under the action of a permutation group. This allows isomorphic results to be eliminated efficiently.

1.2 Minion & Constraint Satisfaction

Definition 3. *A Constraint Satisfaction Problem (CSP) L is a set of constraints \mathcal{C} acting on a finite set of variables $\Delta := \{A_1, A_2, \dots, A_n\}$, each of which has a finite domain of possible values $D_i := D(A_i) \subseteq A$. A solution to L is an instantiation of all of the variables in Δ such that no constraint in \mathcal{C} is violated.*

The class of CSPs is a generalisation of propositional satisfiability (SAT), and is therefore NP-complete. Solvers typically proceed by building a search tree, in which the nodes are assignments of values to variables and the edges lead to assignment choices for the next variable. If at any node a constraint is violated, then search backtracks. If a leaf is reached, then no constraints are violated, and the assignments provide a solution. Clearly these search trees are exponential, and for pathological cases each node may have to be constructed. Heuristics exist for choices of variable and value for the next node, and again these need not lead to any reduction in search. The search tree can be pruned by enforcing levels of consistency: it is possible to check the effect of a variable-value instantiation on the domains of other variables. If such a check shows that a domain has become empty, it is safe to backtrack without exploring nodes that would otherwise be created. These checks have a computational cost, and the trade-off is between the effort of making checks – hopefully resulting in a pruned search tree – and the effort of searching a presumably larger tree with less expensive checks. The Handbook of Constraint Programming [4] provides details of CSP techniques.

A recent advance in Constraints is the “model and run” methodology, of users building constraint models and then executing them on a solver with few options. This methodology inspired the development of the constraint solver Minion [5]. A major feature of modern SAT solvers is their optimised use of modern computer architecture. Using this approach, Minion has been designed to minimise memory usage. The result of this is that Minion claims to offer *fast, scalable* constraint solving. Scalability as problem size increases is an important (and also neglected) factor in constraint solver construction. A key aim of our research is to test the claimed scalability of Minion.

1.3 Symmetry Breaking in CSPs

Constraint satisfaction problems (CSPs) are often highly symmetric. Given any solution, there can be others which are equivalent in terms of the underlying problem. Symmetries may be inherent in the problem, or be created in the process of representing the problem as a CSP. Without symmetry breaking (henceforth SB), many symmetrically equivalent solutions may be found and, in some ways more importantly, many symmetric equivalent parts of the search will be explored. An SB method aims to avoid both of these problems.

Permutation groups are the mathematical structures that best encapsulate symmetry. We describe the symmetries of a CSP as a permutation group of the literals (variable-value pairs) of the CSP, and obtain information regarding symmetric equivalence of search states from using GAP. Assignment of the form ($Var = val$) are called *literals*, so a partial assignment is a conjunction of literals. We denote the set of all literals by χ , and adopt the convention of denoting variables by Roman capitals and values by lower case Greek letters.

Definition 4. *Given a CSP L , with a set of constraints \mathcal{C} , and a set of literals χ , a symmetry of L is a bijection $f : \chi \rightarrow \chi$ such that a full assignment A of L satisfies all constraints in \mathcal{C} if, and only if, $f(A)$ does.*

We denote the image of a literal $(X = \alpha)$ under a symmetry g by $(X = \alpha)^g$. The set of all symmetries of a CSP form a *group*: that is, they are a collection of bijections from the set of all literals to itself that is closed under composition of mappings and under inversion.

Definition 5. *Let G be a group acting on the set Ω . The stabiliser of an element $\omega \in \Omega$ is the set $g \in G$ such that $\omega^g = \omega$. This set is a subgroup of G . The orbit of an element $\omega \in \Omega$ is the set $\{\omega^g | g \in G\}$.*

The stabiliser of a literal $(X = \alpha)$ is the set of all symmetries in G that map $(X = \alpha)$ to itself. The orbit of a literal $(X = \alpha)$, denoted $(X = \alpha)^G$, is the set of all literals that can be mapped to $(X = \alpha)$ by a symmetry in G . That is

$$(X = \alpha)^G := \{(Y = \beta) : \exists g \in G \text{ s.t. } (Y = \beta)^g = (X = \alpha)\}.$$

Given a collection \mathcal{S} of literals, the *pointwise* stabiliser of \mathcal{S} is the subgroup of G which stabilises each element of \mathcal{S} individually. The *setwise* stabiliser of \mathcal{S} is the subgroup of G that consists of symmetries mapping the set \mathcal{S} to itself.

There is a general technique, called “lex-leader”, for generating constraints for any variable symmetry [6]. The idea is essentially simple: For each equivalence class of assignments under our symmetry group, we choose one to be canonical. We then add constraints before search starts which are satisfied only by canonical assignments. We generate canonical assignments by choosing an ordering of the variables and representing assignments as tuples under this variable ordering. Any permutation of variables g maps tuples to tuples, and the lexicographically least of these is our canonical assignment. This gives the set of constraints

$$\forall g \in G, V \preceq_{\text{lex}} V^g$$

where V is the vector of the variables of the CSP, \preceq_{lex} is the standard lexicographic ordering relation, defined by $AD \preceq_{\text{lex}} BC$ iff either $A < B$ or $A = B$ and $D \leq C$, and V^g denotes the permutation of the variables by application of the group element.

Other SB techniques exist, but for this paper we will break symmetries either by lex-leader constraints, or by analysis of properties of monoids.

Definition 6. *Let S be a symmetry breaking technique for CSPs. S is sound if it does not rule out any valid solutions to a CSP. S is complete if it returns exactly one member of each equivalence class of solutions with respect to G .*

Obviously an unsound SB technique is worthless, but there is often a tradeoff in the computational costs involved with incomplete and complete techniques. In the event that no efficient SB method is available for certain symmetries, It may be desirable to break only a subset of the full group of symmetries of a problem. This leaves (partial) SB as a post process to be performed on the solutions obtained.

Remark 1. The above discussion applies to the situation where *all* solutions are required for a given CSP. If only the first solution (if any) is sought, then SB is not always an important consideration. In this paper we are always concerned with finding each symmetrically distinct solution to every CSP posed.

2 Methodology

Our underlying methodology is to use GAP to answer algebraic questions related to monoids. These answers allow us to generate suitable constraints for Minion programs, and eliminate (anti-)isomorphic solutions. In operational terms, GAP is the master process with Minion acting as a black box to provide solutions to carefully formulated CSPs. Our first task is to use the underlying algebra and symmetry of monoids to reduce the search space for the Minion programs.

There are two ways to achieve this. We can give restrictive constraints which will return an unchanged number of solution tables up to (anti-)isomorphism. Here we have to prove that for every monoid there is an equivalent one still in the search space. Secondly we can rule out a certain type of solution for which the number of equivalence classes is known. Here we have to show that the constraints remove all solutions of the specific type. We also have to provide the number of equivalence classes that we rule out, together with a proof where appropriate.

2.1 Reducing the Search Space 1: Identity Elements

The first simplification is to require that the first row and column of each table is the tuple $[1, 2, \dots, n]$. This reduces the number of search variables from n^2 to $(n-1)^2$.

Proposition 1. *The above restriction is sound.*

Proof. Let M be a monoid on $\{1, 2, \dots, n\}$ with identity $e \neq 1$. Then $M^{(1,e)}$ is a monoid isomorphic to M with 1 as identity. Thus it is sound to choose 1 to be the identity element in any solution. As $1 * x = x * 1 = x$ for all $x \in \{1, 2, \dots, n\}$, the first row and column agree with our restriction, and hence a monoid equivalent to M is in the search space. \square

The second simplification is that we can use existing results for semigroups of order $n - 1$ to obtain monoids of order n . The advantage of this approach is that the number of nonequivalent semigroups is known for $n = 1 \dots 8$, as Integer Sequence A001423 in [1].

Proposition 2. *Let H be a semigroup with underlying set $\{2, \dots, n\}$ and multiplication $*_H$. Define a multiplication $*$ on $\{1, \dots, n\}$ by $1 * x = x * 1 = x \quad \forall x \in \{1, \dots, n\}$, and $x *_H y$ otherwise. Then $\langle \{1, \dots, n\}, *, 1 \rangle$ is a monoid.*

Proof. Element 1 is the required identity by definition of $*$. Consider the products $(a*b)*c$ and $a*(b*c)$. If any of a, b and c is 1, then $(a*b)*c = a*(b*c)$ immediately. Any product not involving 1 is associative, since semigroup multiplication is. Hence $\langle \{1, \dots, n\}, *, 1 \rangle$ is a monoid. \square

By construction, the table for any such monoid contains exactly one 1. Also, two non-equivalent semigroups will give two non-equivalent monoids.

Proposition 3. *Let $M = \langle \{1, \dots, n\}, *, 1 \rangle$ be a monoid with fewer than two 1s in its table. Then $\{2, \dots, n\}$ with multiplication $*$ forms a semigroup.*

Proof. Since $1 * 1 = 1$, the remaining values in the table for M are in $\{2, \dots, n\}$. Therefore multiplication $*$ on $\{2, \dots, n\}$ is closed and associative. \square

Taken together, Propositions 2 and 3 show that the number of non-equivalent semigroups of order $n - 1$ is equal to the number of non-equivalent monoids having exactly one 1 in their table. It remains to search for tables of monoids that contain two or more 1s.

Remark 2. We have computed and stored all non-equivalent semigroups for order n up to 8 [7], using a similar combination of GAP and Minion. We do not report these calculations in detail, since the correct values have already been published.

2.2 Reducing the Search Space 2: Diagonals

To break more symmetries before the search we use an approach which was first introduced in computer search for semigroups [8] of order 6 (and subsequently used in the respective problem of order 8 [9]). The idea is to fix the diagonal entries first and consider no two equivalent diagonals. Observe that every diagonal entry is mapped to a diagonal entry under the action on the table described in Section 1. This yields an induced action on the diagonals and therefore induced equivalence classes of diagonals. As we made the restriction that 1

Algorithm 1 Construct the connected digraphs with N vertices and a K cycle

Require: $K \leq N$ { cycle has not more than all vertices }

- 1: $C \leftarrow \emptyset$
- 2: **for all** p in PARTITIONS(N, K) **do** {the partition specifies the sizes of rooted trees at the vertices of the cycle}
- 3: $F \leftarrow$ FORESTS(p)
- 4: **for** $f \in F$ **do**
- 5: $\hat{f} \leftarrow$ set of all tuples of the elements of f
- 6: **for orbit** in ORBITS(C_K, \hat{f}) **do** {the set of images forms orbits under the cyclic group $\langle(1, 2, \dots, K)\rangle$ }
- 7: $rep \leftarrow$ representative of *orbit* {arbitrary element in the orbit}
- 8: $D \leftarrow$ directed cycle of length K
- 9: **for** $i \in \{1, 2, \dots, K\}$ **do**
- 10: $D \leftarrow D$ merged with $rept_i$ joined at vertex i
- 11: **end for**
- 12: $C \leftarrow C \cup \{D\}$
- 13: **end for**
- 14: **end for**
- 15: **end for**
- 16: **return** C

is to be the identity element of the monoid, the equivalence classes of diagonals will differ from the ones in [8, 9]. Our aim is to find or construct exactly one diagonal from each equivalence class.

As we describe in Section 3, this does not split the problem into more or less equivalent subproblems. It turns out that many diagonals give no solutions, and others very many. The aim of this approach, therefore, is not to parallelise the computation, but rather to safely reduce the number of diagonals to be tested from n^{n-1} to a more manageable number. Moreover, by fixing a diagonal we reduce the symmetry of the problem.

Proposition 4. *Let C be the family of directed, unlabelled graphs having $n - 1$ vertices, such that the outward degree of any vertex is less than or equal to 1 (and loops are allowed). Then C corresponds to the set of equivalence classes of diagonals with 1 in first position under the full symmetry group on $\{2, \dots, n\}$.*

Proof. For any $c \in C$, every labelling of vertices from $\{2, \dots, n\}$ leads to a diagonal in the following way: the first entry is 1, the entry in position $2 \leq k \leq n$ is the endpoint of the edge from vertex k , and 1 if no such edge exists. Two distinct labellings of c give two diagonals in the same equivalence class, since a re-labelling of c is simply a permutation of elements from $\{2, \dots, n\}$, and hence is an element of the group acting on the diagonals. For any diagonal we can construct a labelled graph with vertices $\{2, \dots, n\}$, and a directed edge from vertex $2 \leq k \leq n$ to the vertex given by the k th entry of the diagonal, unless the entry is 1. The unlabelled graph is in C . \square

Detailed information regarding the connected components of the members of C is given in [10, 3.4]. They can either be rooted trees with the direction of

edges towards the root, or they can be a directed cycle (of length one or more) where every vertex in the cycle is the root of a tree. Algorithm 1 constructs and returns the latter. The former is constructed recursively using the one-one correspondence between rooted trees on n vertices, and forests of rooted trees on $n - 1$ vertices. All sets of forests whose tree sizes are specified by a partition p of $n - 1$ are returned by our function `FORESTS(p)`.

2.3 Constraint Satisfaction Problems in Minion

We can now construct a CSP L for each diagonal D . The set of variables $\Delta := \{A_{1,1}, A_{1,2}, \dots, A_{1,n}, A_{2,1}, \dots, A_{2,n}, \dots, A_{n,1}, \dots, A_{n,n}\}$, consists of each entry in an $n \times n$ table, with each variable having domain $\{1, 2, \dots, n\}$. The constraints are:

1. $a * (b * c) = (a * b) * c$ for all combinations of a, b and c ;
2. the diagonal is fixed as D ;
3. the first row and first column consist of $[1, 2, \dots, n]$;
4. the table contains two or more 1 entries.

The first constraint is associativity, which in Minion is enforced using *element* constraints. The constraint *element(vector, i , val)* specifies that, in any solution, $vector[i] = val$. We add a new variable $A_{a,b,c}$ for each triple (a, b, c) . The pair of constraints

$$element(row(a), b * c, A_{a,b,c}) \text{ and } element(column(c), a * b, A_{a,b,c})$$

then enforce associativity. Constraints 2 and 3 turn $n + 2(n - 1)$ of the n^2 variables into *ground variables*, having domain size exactly one. These implement the reduction in search space described in Sections 2.1 and 2.2, and reduce the symmetries remaining in each problem instance. Constraint 4 is a simple occurrence requirement.

Definition 7. Define Y_D to be set of solutions to the CSP L defined by diagonal D , and Y_n to be the union of the Y_D for all D of length n . Let \hat{Y}_D denote a set of representatives of non-equivalent solutions from Y ; the union of these is \hat{Y}_n .

2.4 Refinements and Optimisation

The CSPs described in Section 2.3 are sufficient to solve our identification and enumeration problem, up to any remaining symmetries. There are, however, a number of improvements that we can make. These involve further restricting the set of diagonals used, and imposing additional constraints for some of the remaining diagonals. We identify diagonals that cannot form part of a table with more than one 1 using the following proposition:

Proposition 5. A monoid with table in Y_n either has an element $x \neq 1$ with $x^2 = 1$ or it contains a sequence of distinct elements x_1, x_2, \dots, x_k with $x_{i+1} = x^2_i$ for $i = 1, 2, \dots, k - 1$, and $x_k^2 = x_1$.

Proof. Let $\langle M, *, 1 \rangle$ be a monoid with table in Y_n . If M contains an element $x = 1$ with $x^2 = 1$ we are done. Otherwise there must be at least one pair of distinct elements $x_1, y_1 \in M \setminus \{1\}$ with $x_1 * y_1 = 1$. Define the two sequences of elements $(x_i)_{i=1,2,\dots}, (y_i)_{i=1,2,\dots}$ by

$$x_{i+1} = x_i^2 \text{ and } y_{i+1} = y_i^2, i = 1, 2, \dots$$

Clearly $x_i * y_i = x_1^{2^{i-1}} * y_1^{2^{i-1}} = (x_1 * y_1)^{2^{i-1}} = 1$. As M contains only $n - 1$ elements not equal 1 there must be repetition in the sequence $(x_i)_{i=1,2,\dots}$. Assume $x_{i+1} = x_i$ for some i . Then

$$1 = x_i * y_i = x_{i+1} * y_i = (x_i * x_i) * y_i = x_i * (x_i * y_i) = x_i * 1 = x_i$$

a contradiction. This shows that the period of the repetition is greater than 1 and completes the proof. \square

The conditions in Proposition 5 consider the squares of elements, i. e. the diagonal entries of the multiplication table. If diagonal D satisfies neither of the two conditions then Y_D will be empty, so we can exclude D .

For certain diagonals we can post *implied constraints*, which, although not strictly required, are likely to improve the performance of the solver. These are *all different* constraints which forbid equal values appearing in certain rows and columns. These constraints propagate very efficiently in CSP solvers, and often lead to early backtrack and hence pruning of the search tree.

Proposition 6. *Let a be an element of a monoid $\langle M, *, 1 \rangle$. If $a^k = 1$ for any k , then the values in row and column a of the table for M will be all different.*

Proof. Let $aM = \{a * m \mid m \in M\}$. The following inequalities hold in general

$$|a^k M| \leq |aM| \leq |M|.$$

By hypothesis, $a^k M = M$, so we have equality, and the values in row a of the table for M will be distinct. The case for right multiplication – hence column values – is similar. \square

From the diagonal we can compute $a^{2^{n-1}}$ for any element a . If this power equals 1, we add all different constraints on the row and column of a . In the event that diagonal D gives all different constraints on every row and column, then the solution set Y_D will consist entirely of groups. Since groups of small order are well known, we can safely reject D from our set of diagonals, provided that we add the number of groups not searched for to our final total. This number is obtained by inspection of the diagonals for groups.

2.5 (Anti-)Isomorph Rejection

We now address the problem of ruling out (anti-)isomorphs. There are two approaches available to us. We can either solve the Minion CSP instance, then

Algorithm 2 Enumerate and store Monoids of order n

Require: $n \leftarrow$ order of monoids

Require: $D \leftarrow$ set of inequivalent n -diagonals

Require: $S_N \times C_2 \leftarrow$ the symmetric group acting on n objects

```
1: for  $d \in D$  do
2:    $stab \leftarrow$  the stabilizer of  $d$  in  $S_n \times C_2$ 
3:    $P \leftarrow$  the Minion program for  $d$ 
4:   if  $stab$  is small then
5:     compute lex-leader constraints for  $d$  in GAP
6:     add these constraints to  $P$ 
7:     obtain  $\hat{Y}_d \leftarrow$  solutions of  $P$  from Minion
8:   else
9:     compute signature constraints for  $d$  in GAP
10:    add these constraints to  $P$ 
11:    obtain  $Y_d \leftarrow$  solutions of  $P$  from Minion
12:     $\hat{Y}_d \leftarrow$  (anti-)isomorph rejection of  $Y_d$  in GAP
13:   end if
14: end for
15: return  $\hat{Y} = \cup_d \hat{Y}_d$ 
```

reject (anti-)isomorphs as a post process, or apply constraints which ensure that Minion only returns canonical solutions. Both methods involve GAP computation: the first requires an efficient minimal image test, the second requires the images of each literal ($X = \alpha$) under the symmetry group of the CSP instance. Another key GAP calculation is the stabiliser of each diagonal in $S_n \times C_2$. This is the subgroup of $S_n \times C_2$ which fixes the diagonal entries of a table with respect to the isomorphism operation and transposition, and it represents the remaining symmetry to be broken. The minimum size of such a stabiliser is 2 (occurring whenever the diagonal permits no isomorphic solutions, but table transposition is still allowed), and the maximum size is $2(n-2)!$ (occurring when the diagonal consists of $[1, 1, 3, 4, \dots, n-1, n]$, so that the only symmetry broken is the fixing of 1 at positions $(1, 1)$ and $(2, 2)$). We can apply lex-leader SB constraints for any diagonal, but each such constraint consists of a lexicographic requirement on two vectors – the first our canonical solution, the second its image under a group element – each containing n^3 Boolean variables (one for each literal). Posting a factorial number of such constraints is likely to slow Minion down; each constraint may be checked at each node in the search tree. However, most diagonal stabilisers are small, and posting lex-leader constraints is likely to be highly efficient.

Proposition 7. *Adding lex-leader SB constraints to a Minion CSP instance is both sound and complete.*

Proof. This is proved in [6].

The other approach, post-hoc isomorph rejection, is a simple concept. We store all solutions obtained by Minion, then check each solution to see if it is

minimal in the stabiliser of the diagonal. If it is, we keep it; if not, we reject it. There are however some refinements and options to consider.

We define the *signature* of a solution table to be the n -tuple containing the number of occurrences of value k , in position k , for $k = 1, \dots, n$. We use signatures to break more symmetries by posting constraints prior to search. We analyse the directed graph associated with each diagonal, as detailed in Section 2.2. There are two cases to consider. The first is to identify completely interchangeable vertices in rooted trees. We first look for roots of isomorphic trees, and then recurse down each tree. If we identify symmetric vertices, we post a linear ordering on the signature list of the labels of the vertices: we restrict the numbers of occurrences of these values in any solution. Each level in the rooted tree structure can provide zero or more such constraints. The second is to break any cyclic symmetry by fixing a minimal vertex in a cycle. Again, we recurse through the structure of the graph to identify any symmetry at a given level, posting linear ordering constraints whenever symmetries are found. We use the built in methods for Orbits and Stabilizers in GAP to find sets of equivalent values. At each level of the recursion we stabilise the vertices of the levels above.

Proposition 8. *Posting these ordering constraints on signatures is sound.*

Proof. If Y_D is empty there are no solutions to lose. Let $S \in Y_D$ be a solution of the original problem. If S violates a constraint on the first level then there is a symmetry $g \in \text{Stab}(D)$ – the stabiliser of the diagonal – such that S^g satisfies the constraint. Assume inductively that S satisfies all constraints up to level $l - 1$. If S violates a constraint on level l then there is a symmetry $g \in \text{Stab}(D)$ which fixes all the constraints of the levels above l such that S^g satisfies the constraint on level l . \square

As examples, we first consider the diagonal $[1, 1, 3, 4, \dots, n - 1, n]$. Its graph consists of 8 vertices labelled $2, \dots, n$, with each vertex labelled 3 or higher forming a cycle of length one, with the empty rooted subtree. Since these vertices are indistinguishable when the labels are removed, it is safe to impose a linear order on the occurrences of values 3 through n in any solution table. We can then (anti-)isomorph reject in the stabilisers of the signatures of solutions: if g is a permutation with $k^g = l$ which maps solution S to solution T , then the signature of k has to equal the signature of l because of the linear ordering constraint posted. For an example that illustrates our extension of this simple linear ordering, consider diagonal $[1, 1, 3, 3, 4, 4, 3, 7, 7]$ which has graph with edges $4 \rightarrow 3, 7 \rightarrow 3, 5 \rightarrow 4, 6 \rightarrow 4, 8 \rightarrow 7, 9 \rightarrow 7$. We impose on the signatures the constraints $[4, 5, 6] \preceq_{\text{lex}} [7, 8, 9]$, $5 \leq 6$ and $8 \leq 9$. The first breaks the symmetry of the two equivalent sub-trees connected to vertex 3; the others break the symmetry in the equivalent leaves.

Posting signature constraints therefore has two advantages: we reduce the number of solutions returned by Minion, and we (anti-)isomorph reject in a smaller group. There is still the cost of computing the signature stabiliser, and this is non-trivial.

Our algorithm for (anti-)isomorph rejection is to order the returned Minion solutions lexicographically by signature, then only compute the stabiliser when the signature changes. This minimises the number of stabiliser calculations, but requires a potentially expensive sorting preprocess.

Table 2: Solutions & Timings

Method	n :	5	6	7	8	9
	Diagonals	27	81	242	699	2,026
Complete SB	Solutions	30	213	1,757	22,951	◇
	GAP cpu (s)	3	17	119	856	◇
	Minion cpu (s)	0	1	24	6,225	◇
	Total cpu (s)	3	18	143	7,081	◇
Isomorph rejection	Solutions	30	213	1,757	22,951	955,569
	GAP cpu (s)	2	13	96	989	197,587
	Minion cpu (s)	0	1	5	63	4,106
	Total cpu (s)	2	14	101	1,052	201,693
Combined 48	Solutions	30	213	1,757	22,951	955,569
	GAP cpu (s)	2	13	96	916	70,381
	Minion cpu (s)	0	1	5	68	3,150
	Total cpu (s)	2	14	101	984	73,531
Combined 240	Solutions	30	213	1,757	22,951	955,569
	GAP cpu (s)	2	13	96	720	24,386
	Minion cpu (s)	0	1	5	139	12,746
	Total cpu (s)	2	14	101	859	37,132

Legend: ◇ denotes timeout; *Combined 48* denotes complete SB only if the stabiliser of the diagonal in $S_n \times C_2$ has size ≤ 48 for $n = 9$, and ≤ 12 for $n = 5 \dots 8$. *Combined 240* denotes complete SB only if the stabiliser of the diagonal in $S_n \times C_2$ has size ≤ 240 for $n > 8$, and ≤ 12 for $n = 5 \dots 7$.

2.6 Enumeration & Storage of Monoids

Algorithm 2 describes our computational method. For each diagonal we generate a Minion instance that models associative multiplication tables having fixed first row, first column and diagonal values, and having at least two occurrences of value 1. The SB method used to break the remaining symmetries depends on the size of the stabiliser of the diagonal. A small stabiliser requires few complete SB constraints, whereas a large stabilizer indicates that signature constraints followed by (anti-)isomorph rejection may perform better. The definition of “small” is unclear *a priori* – we discuss suitable values obtained after experimentation in Section 3.

3 Results

Table 2 contains the timings for computations. We tested three approaches: only using SB constraints, only using post-solution (anti-)isomorph rejection, and the

Table 3: Pathological diagonals for $n = 9$

No.	DIAGONAL	STABILISER	$ Y_D $	$ \hat{Y}_D $
1	113456789	10,080	60,169	9,824
2	113333333	1,440	1,508,566	13,731
3	113344444	240	1,182,180	6,517
4	113335555	48	1,675,952	39,984
5	113333338	48	1,678,602	76,213
6	113333377	24	1,647,092	123,025
7	113333666	24	1,648,105	98,536
8	113344377	16	306,497	35,291
9	113343666	12	542,892	47,123
10	113335377	4	546,794	139,119

Legend: STABILISER is the size of the stabiliser of the diagonal in $S_n \times C_2$; $|Y_D|$ is the number of Minion solutions returned using signature constraints; $|\hat{Y}_D|$ is the number of solutions after (anti-)isomorph rejection.

combined approach set out in Algorithm 2. The first approach suffers badly in terms of Minion effort with increasing n . This is due to the large numbers of SB constraints posted for diagonals with large stabiliser. We were unable to obtain solutions for $n = 9$ in under two weeks, since Minion slowed markedly for certain diagonals. The number of diagonals shown is the number after applying the techniques described in Section 4; for $n = 9$ this reduced the number of diagonals from 2,598 to 2,026.

(Anti-)isomorph rejection is sufficiently efficient for $n = 9$. The combined approach works best, as expected. Smaller stabiliser diagonals mean faster Minion instances, with (anti-)isomorph rejection used when the stabiliser is too large. Both GAP and Minion times improve in the combined approach for $n = 9$. This is due to Minion returning fewer solutions for small stabiliser diagonals: the Minion search tree is pruned heavily by the SB constraints, and GAP has no (anti-)isomorph rejection to perform. We tested the combined approach with different cut-offs for the size of diagonal stabilizer, beyond which (anti-)isomorph rejection would be used.

The best trade-off between GAP and Minion computation was achieved when SB constraints were applied for diagonals having stabilisers with 240 or fewer elements. This roughly halved the total time of performing SB on stabilisers with 48 or fewer elements for $n = 9$. The optimal trade-off – with GAP performing roughly as much (anti-)isomorph rejection as Minion is performing complete symmetry-breaking – is not apparent before search starts.

Remark 3. Since GAP is the master process, the GAP times include all book-keeping work such as generating Minion files, storing solutions, reporting statistics etc. This pads out the GAP times, but emphasises the speed of Minion.

The value for “small” used in practice was determined by analysis of results from (anti-)isomorph rejection. Table 3 lists several diagonals of order 9. Diagonal 1 is an obvious candidate for (anti-)isomorph rejection, since the stabiliser is large but only 60,169 solutions need to be tested for minimality. Diagonals 2 and 3 are a problem for both methods: the stabilisers are large and so are the numbers of solutions returned without SB constraints. This type of diagonal will make calculation for $n = 10$ much more difficult. However, diagonals such as 4 & 5 are better suited to SB constraints: we need only post 48 constraints to rule out over 1.5 million solutions. The remaining pathological diagonals require even fewer constraints. Hence, for $n = 9$ we first used SB constraints for stabilisers of size 48 and below. It transpires that – for this class of problems – using SB constraints for stabilisers of size 240 and below is a computational win, improving the balance between complete and incomplete symmetry-breaking. However, this does not appear to give an insight for the optimal choice of stabiliser size cutoff for higher order problems, since we need to know the size of Y_D and the size of the stabiliser before deciding upon an optimal strategy for obtaining the non-equivalent solutions \hat{Y}_D .

The numbers of monoids unique up to (anti-)isomorphism is given in Table 3. The values for $n = 8$ and $n = 9$ are new.

Table 4: Numbers of monoids up to (anti-)isomorphism

n	$ H_{n-1} $	$ \hat{Y}_n $	$ M_n $
2	1	1	2
3	4	2	6
4	18	9	27
5	126	30	156
6	1,160	213	1,373
7	15,973	1,757	17,730
8	836,021	22,956	858,977
9	1,843,120,128	955,569	1,844,075,697

Legend: $|H_{n-1}|$ is the number of non-equivalent semigroups of order $n - 1$; $|\hat{Y}_n|$ is the number of non-equivalent monoids with more than one 1 in their table; $|M_n|$ is the number of non-equivalent monoids, and is the sum of $|\hat{Y}_n|$ and $|\hat{Y}_n|$.

4 Discussion

We have analysed properties of monoids to obtain algorithms that permit the efficient enumeration and storage of monoids. Our implementation involves symbolic algebraic computation both as a pre-process and as a post-process. Our AI backtrack search tool, Minion, is fast – as claimed by its developers – but does not scale well on problems that grow with the factorial of the instance dimension. This is not surprising: there are no known solutions to the problem of combinatorial explosion.

As well as demonstrating the efficacy of combining symbolic computation with AI search, our results provide new numbers in algebraic combinatorics. We have stored each solution, and provide a library of monoids (and semigroups) [7] that can be accessed and analysed by the research community. We are confident in the accuracy of our results: we have used two different approaches to the same problem, and obtained identical answers in each case. In addition we have used similar algorithms to verify the enumeration of non-equivalent semigroups of order up to 8.

Future avenues of research involve solving similar problems having, as yet, unknown answers but being small enough to be tractable by our methods. In order to solve the monoid problem for $n = 10$, it is clear that our methods will work well for many diagonals. However, we have shown that pathological diagonals exist, for which neither complete symmetry breaking nor *post hoc* (anti-)isomorph rejection will be efficient. Moreover, the number of semigroups of order 9 is as yet unknown.

Acknowledgements

Our work is supported by EPSRC grant EP/CS23229/1. We thank Ian Gent, Steve Linton, Victor Maltcev, James Mitchell and Nik Ruškuc for their help and comments.

References

1. Sloane, N.J.A.: The on-line encyclopedia of integer sequences. <http://www.research.att.com/~njas/sequences/Seis.html> (2008)
2. : GAP – Groups, Algorithms, and Programming, Version 4.4.10. <http://www.gap-system.org> (2007)
3. Linton, S.: Finding the smallest image of a set. In: ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, ACM (2004) 229–234
4. Rossi, F., van Beek, P., Walsh, T., eds.: The Handbook of Constraint Programming. Elsevier (2006)
5. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: ECAI, IOS Press (2006) 98–102
6. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: KR. (1996) 148–159
7. Distler, A., Mitchell, J.D.: smallsemi – a GAP package. <http://turnbull.mcs.st-and.ac.uk/~jamesm/smallsemi/> (2008)
8. Plemmons, R.J.: There are 15973 semigroups of order 6. *Math. Algorithms* **2** (1967) 2–17
9. Satoh, S., Yama, K., Tokizawa, M.: Semigroups of order 8. *Semigroup Forum* **49** (1994) 7–29
10. Harary, F., Palmer, E.M.: Graphical enumeration. Academic Press, New York (1973)

5 Revised Results & Conclusions

Since submitting this paper, we have re-run our calculations with a modified version of complete SB. When posting lex-leader constraints one has to choose a fixed order for the literals. It is known that this ordering plays a crucial role for the performance of the computation – although it is of no theoretical importance. Generally speaking, the ordering of the literals should agree with the search order of the constraint solver. This is to ensure that the violation of a SB constraint is discovered as early as possible during search. The implementation that provided the timings shown in Table 2 neglected this important fact. The timings for complete SB after correction of this mistake are given in Table 5. For these results we added a further optimisation, removing those literals from lex-ordering tests that cannot differ in a solution.

Table 5: Solutions & Revised Timings

Method	n :	5	6	7	8	9
	Diagonals	27	81	242	699	2,026
Revised Complete SB	Solutions	30	213	1,757	22,951	955,569
	GAP cpu (s)	2	13	91	686	10,557
	Minion cpu (s)	0	1	7	72	3,667
	Total cpu (s)	2	148	98	758	14,224

We can now revise our conclusions, based on these improved results. Minion can handle more symmetries than we envisaged without incurring a large computational penalty. The number of non-equivalent monoids of order less than 10 can be obtained using complete SB in a reasonable time. Our isomorph rejection method, as described in Section 2.5, is no longer crucial even for order 9. However, it remains an improvement for pathological diagonal no. 1, i. e. $[1, 1, 3, 4, 5, 6, 7, 8, 9]$, which has a stabiliser of size 10,080 in $S_9 \times C_2$.

Moreover, we are certain that the GAP times can be improved by implementing specialised stabiliser calculations for the action on tables and diagonals. This being the case, the problem of enumerating the monoids of order 10 is likely to be tractable using our methods.