CARLETON UNIVERSITY

SCHOOL OF
MATHEMATICS AND STATISTICS

HONOURS PROJECT

TITLE: Integer and Constraint Programming Revisted for Mutually Orthogonal Latin Squares.

AUTHOR: Noah D. Rubin

SUPERVISORS: Dr. Brett Stevens and Dr. Curtis Bright

DATE: Monday January 10, 2022

Many thanks to my supervisors Brett Stevens, Curtis Bright and Kevin Cheung for their support and guidance throughout this project.

Abstract

In this project we use the mathematical programming paradigms of integer and constraint programming to solve the Mutually Orthogonal Latin Squares problems for sets of two and three squares. We implement two of the solution methods used by Appa *et al.* in their paper "*Searching for mutually orthogonal Latin squares via integer and constraint programming*" and further improve upon both their models introduced and techniques of reducing symmetries in the solution spaces. Using our new models and symmetry breaking methods we significantly decrease the amount of time taken for our chosen solvers to find solutions to the aforementioned problems or prove their nonexistence. We also attempt to extrapolate the time it would take to find a solution for a higher order instance of the problem – the existence of which is currently an open question in design theory.

1. Introduction

A *Latin square* X of order $n \in \mathbb{N}$ is an $n \times n$ array of symbols, where the element in the i^{th} row and j^{th} column is denoted $X_{ij} \in \{0,1,\dots,n-1\}$. Latin squares have the restriction that each symbol $\{0,1,\dots,n-1\}$ must appear exactly once in every row and column of X, like a sudoku puzzle. This restriction is referred to as the *Latin property* of X. We wish to study sets of Latin squares with an additional property. Two Latin squares X, Y are *orthogonal* if the system of equations $X_{ij} = a, Y_{ij} = b$ has exactly n^2 unique solutions. In other words, X and Y are orthogonal if for every i, j the pair (X_{ij}, Y_{ij}) is unique. We call a set $\{X^1, \dots, X^k\}$ k *Mutually Orthogonal Latin Squares of Order* n or k MOLS(n) if X^i is orthogonal to X^j for $i \neq j$.

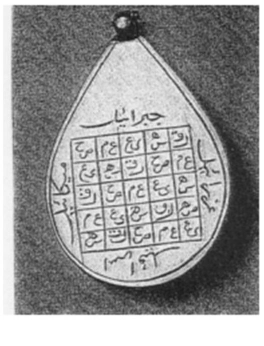

Fig 1 – A silver amulet from Damascus depicting a Latin square of order 5

Latin Squares have been found inscribed on Arabic amulets dating to as early as the 10th century [1]. As with much of early mathematics, the highly structured form of these squares gave them a mystical quality. Methods to construct Latin Squares were known for centuries, and Arabic mathematicians used them in their studies, often relating to the construction of Magic Squares. In 1700 the Korean Mathematician Choi Soek-jeong presented 2MOLS(9) in relation to magic squares, but he was unable to find a pair for order 10 [1]. The first major advancement in the study of Latin Squares and their properties was made by Euler in the late 18th century, in his attempt to answer the 36 officers problem[1] – which is equivalent to a set of 2MOLS(6). In his paper, Euler uses the first 6 symbols of the Latin alphabet as the possible values of the first square, giving the name "Latin Squares" to the objects. Trying to answer the 36 officers problem led Euler to the concept of orthogonality of Latin Squares and prompted the question of which Latin Squares have orthogonal mates. Euler conjectured that no such pair of squares existed for even orders not divisible by 4 i.e. $n \equiv 2 \bmod 4$ (including 2 and 6). The first proof of the nonexistence of 2MOLS(6) was published by Gaston Tarry in 1901, using exhaustive methods to disprove Euler's claim. Tarry's proof came well over a century after Euler's time, and there are historical records that claim a proof of nonexistence from almost 50 years prior. In a letter to Gauss in August 1842, Heinrich Schumacher stated that Thomas Clausen had divided all Latin Squares of order 6 into 17 families and exhaustively showed that none of them had orthogonal mates. Clausen's division of the square into 17 families is an example of the concept of "symmetry breaking" – recognizing the equivalence between different sets of Latin Squares to reduce the number of such squares we need to consider. Clausen's proof has been lost to time, but it is believed that his work was genuine given his sound method and academic talent for combinatorics. Euler's conjecture regarding the order of squares which could not be part of a set of MOLS was disproven almost 150 years after its proposal, when R.C. Bose and S.S. Shrikhande constructed two mutually orthogonal Latin squares of order 22 [2]. Bose and Shrikhande along with E.T. Parker later proved that $n = 2{,}6$ are the only orders of n for which a pair of 2MOLS(n) do not exist [3].

Latin Squares, sets of Mutually Orthogonal Latin Squares and Magic Squares have many applications in a diverse set of fields, from experimental design to recreational math [1]. The high dimensionality and number of possible squares makes the kMOLS(n) problem one which is effectively solved in a systematic way, usually by a computer

[1] For more info see https://mathworld.wolfram.com/36OfficerProblem.html

program – indeed much more progress has been made studying these objects in higher orders since the beginning of the era of modern computers, including a computer-aided proof of the nonexistence of 9MOLS(10) [4] [5]. Currently the question of whether there exists a solution to 3MOLS(10) is an open problem in design theory, and one which we seek to illuminate. Our project began as an attempt to replicate the work of Appa *et al.* in their paper *Searching for Mutually Orthogonal Latin Squares via integer and constraint programming.* Appa *et al.* show promising results solving the kMOLS(n) problem by combining the paradigms of integer and constraint programs into hybrid algorithms [6]. The efficacy of such algorithms was of interest to us. However, we realized that the scope of the research project was too small to attempt to replicate Appa *et al.*'s work fully. We attempted to replicate and then improve upon Appa *et al.*'s methods for the pure integer and constraint programs. We were able to make improvements through two methods and are hopeful that the results we observed can be extended to higher order squares for a potential solution to the 3MOLS(10) problem.

2. Integer and Constraint Programming

Integer Programming is a special case of the mathematical programming paradigm Linear Programming in which all variables of the problem are subject to taking on integer values. In general, a linear program is a problem in which we are asked to optimize a linear "objective" function f over a set of variables $x_1, x_2, \dots, x_n$ subject to a set of linear constraints which impose that the domain of f is bounded by a set of hyperplanes. We express a linear program as

$$\begin{aligned}
&\text{maximize}(c_{01}x_1 + c_{02}x_2 + \cdots + c_{0n}x_n)\\
&\text{subject to}\\
&\qquad c_{11}x_1 + c_{12}x_2 + \cdots + c_{1n}x_n \leq a_1\\
&\qquad c_{21}x_1 + c_{22}x_2 + \cdots + c_{2n}x_n \leq a_2\\
&\qquad\qquad\qquad \vdots\\
&\qquad c_{k1}x_1 + c_{k2}x_2 + \cdots + c_{kn}x_n \leq a_k\\
&\qquad x_1, x_2, \dots, x_n \geq 0
\end{aligned}$$

An integer program also enforces the constraint $x_1, x_2, \dots, x_n \in \mathbb{Z}$ or equivalently $x_1, x_2, \dots, x_n$ are integer.

The above information constitutes what is known as a *model.* Linear programs are solved with algebraic methods such as the *simplex* algorithm, which attempts to jump across extreme points in the convex hull generated by the constraints of the program until a solution is found. When the variables of a linear program are constrained to be integral the problem becomes NP-hard and requires a more brute-force method of solution. A common approach is the branch and cut method, in which the solver forms an enumeration tree and uses a myriad of algebraic methods, preprocessing[2] and simple linear programming to prune large portions of the tree. The tree itself constitutes many sub-problems called *nodes* which are generated by fixing a variable to a particular value and appending the resulting sub-problem's enumeration tree[3]. The relaxations of these sub-problems are found by ignoring the integral constraints on the variables. These relaxed sub-problems are then solved and used to generate new inequalities which eliminate possible values the variables can take (called *cutting planes*). When no cutting planes can be generated the solver will choose another variable to branch on by some heuristic. The heuristics used to branch will ultimately define the shape of the enumeration tree, and so effectively choosing the variables is extremely important. Popular commercial grade optimizers such as Gurobi (our chosen solver) and ILOG-CPLEX employ many heuristics to increase the speed at which solutions are found.

Complementing the algebraic approach of Integer Programming is the mathematical programming paradigm of Constraint Programming (CP). CP problems are defined by a set of variables $\{x_1, x_2, \ldots, x_n\}$, their domains $\{D_1, D_2, \ldots, D_n\}$ and constraints $\{C_1, C_2, \ldots, C_k\}$. The CP Model is then defined as

$$\begin{aligned}
&\text{Determine a value of } x_i \qquad \forall 1 \le i \le n \\
&\text{subject to} \\
&\qquad x_i \in D_i \qquad\qquad\qquad\qquad \forall 1 \le i \le n \\
&\qquad x_1, x_2, \ldots, x_n \text{ satisfy } C_j \qquad \forall 1 \le j \le k
\end{aligned}$$

CP allows for much more flexibility in both the definition of variables and the constraints which define feasible solutions. Of particular importance is the all_different$(x_1, x_2, \ldots, x_k)$ constraint, which enforces that $x_i \neq x_j$. CP solvers are purely search-based, unlike IP solvers which are also numerical. CP solvers benefit from their

[2] Preprocessing allows the solver to reduce the problem size before the solve begins.

[3] This is referred to as *branching* in a search tree.

simplicity, in that most of the work performed while solving the problem is done in the propagation of variable values through the tree[4].

Appa *et al.*'s first hybrid algorithm used an IP solver to resolve sub-problems within the CP enumeration tree. At any node of the tree some subset of the variables are set to values within their domains, and each additional setting decreases the size of the problem. After a certain number of variables are set to values the hybrid algorithm passes all information to the IP solver which solves the relaxation of the current node. More variables being set makes it easier to solve the problem but means that the depth of the node is increased significantly. The effectiveness of this method comes from the fact that the IP solver has the potential to determine that a sub-problem is infeasible at the root node. The CP enumeration tree may thus be pruned significantly at these specific nodes, and the overall number of branches to check is cut down. The second hybrid algorithm uses the CP preprocessor to propagate variable fixings throughout the IP search tree. This algorithm allows the IP solver to benefit from a reduction in the size of the sub-problems it encounters within the search tree.

3. Modelling kMOLS(n)

Our work is primarily focused on the case of 2MOLS(n). However the models we use are easily extended to higher orders at the cost of massively increasing the number of variables and constraints. We thus propose different methods of computing 3MOLS(n) which are discussed in Section 1. Let X and Y be two Latin Squares of order n.

3.1 IP Model

The IP model for expressing the kMOLS(n) problem is a binary linear program in n^{k+2} variables. In the case of $k = 2$ we define

$$x_{ijlm} := \begin{cases} 1, & X_{ij} = l, Y_{ij} = m \\ 0, & \text{otherwise} \end{cases}$$

The following sets of constraints define the structure of X, Y and their orthogonality:

$$\textstyle\sum_{0 \le l,m < n} x_{ijlm} = 1 \quad \forall i,j \quad \text{each cell contains 1 value} \tag{1}$$

[4] In the subtree generated by setting a variable to a specific value, the value of that variable replaces the variable itself in all subsequent nodes, and consequently the domains of all variables are updated. This is called *propagation*.

$$\textstyle\sum_{0\le j,m<n} x_{ijlm} = 1 \quad \forall i,l \qquad \text{Latin property in rows of } X \tag{2}$$

$$\textstyle\sum_{0\le j,l<n} x_{ijlm} = 1 \quad \forall i,m \qquad \text{Latin property in rows of } Y \tag{3}$$

$$\textstyle\sum_{0\le i,m<n} x_{ijlm} = 1 \quad \forall j,l \qquad \text{Latin property in columns of } X \tag{4}$$

$$\textstyle\sum_{0\le i,l<n} x_{ijlm} = 1 \quad \forall j,m \qquad \text{Latin property in columns of } Y \tag{5}$$

$$\textstyle\sum_{0\le i,j<n} x_{ijlm} = 1 \quad \forall l,m \qquad \text{orthogonality of } X \text{ and } Y \tag{6}$$

This model is extended to $k = 3$ and squares X, Y, Z by using

$$x_{ijlmo} := \begin{cases} 1, & X_{ij} = l, Y_{ij} = m, Z_{ij} = o \\ 0, & \text{otherwise} \end{cases}$$

and subsequent values of k require even more subscripts. We impose the additional constraints of the Latin property in Z and the pairwise orthogonality between all squares. This method extends to any k squares, but in general takes $\binom{2+k}{2} n^2$ constraints, which for all but $k = 2$ quickly exceed our ability to solve. For the $k = 3$ case we may encode the orthogonality based on the assumption that Z is already known. This allows for some flexibility and uses far fewer variables. To extend the 2MOLS(n) model to account for Z we use two additional sets of constraints. These sets of constraints allow for us to model 3MOLS(n) without adding any additional variables to the problem:

$$\sum_{\substack{Z_{ij}=z \\ 0\le m<n}} x_{ijlm} = 1 \quad \forall l,z \qquad \text{orthogonality of } X \text{ and } Z \tag{7}$$

$$\sum_{\substack{Z_{ij}=z \\ 0\le l<n}} x_{ijlm} = 1 \quad \forall m,z \qquad \text{orthogonality of } Y \text{ and } Z \tag{8}$$

We may then conduct the search by first fixing Z and then attempting to solve the 2MOLS(n) model with (7) and (8) imposed.

3.2 CP Model

Constraint programming allows us to formulate kMOLS(n) much more naturally than the IP version. For $k = 2$ we use $2n^2$ variables defined as

$$X_{ij} := \text{value of cell } (i,j) \text{ in } X$$

$$Y_{ij} := \text{value of cell } (i,j) \text{ in } Y$$

$$i, j, X_{ij}, Y_{ij} \in \{0,1,\ldots,n-1\}$$

The Latin properties of X and Y are enforced by all_different constraints on the rows and columns of the squares:

$\text{all\_different}(X_{ij}\ \forall j)\ \forall i$	Latin property in rows of X
$\text{all\_different}(Y_{ij}\ \forall j)\ \forall i$	Latin property in rows of Y
$\text{all\_different}(X_{ij}\ \forall i)\ \forall j$	Latin property in columns of X
$\text{all\_different}(Y_{ij}\ \forall i)\ \forall j$	Latin property in columns of Y

Appa *et al.* encode the orthogonality between X and Y by using linear constraints. Define the additional variables

$$Z_{ij} := X_{ij} + nY_{ij}$$

$$Z_{ij} \in \{0,1,\ldots,n^2-1\}$$

Since each ordered pair of values (X_{ij}, Y_{ij}) yields a unique value of Z_{ij}, if all Z_{ij} are distinct then X and Y are orthogonal. Hence, we use

$$\text{all\_different}(Z_{ij}\ \forall i, j)$$

to encode orthogonality. We call this model "CP-linear". Appa *et al.* built their own constraint programming solver from the ground up, allowing them to have a much greater level of control over how their solver found solutions. Given the scope of this research project we were unable to construct such a solver, nor were we able to obtain a copy of Appa *et al.*'s source code. We thus opted to use a "black box" solver which seemed to handle linear constraints poorly. Because of this we use a different method of encoding orthogonality between two squares, leveraging what is known as "indexing" constraints. These allow us to impose constraints on array indices based on the values of other variables, i.e., $A[b] = c$ encodes the condition that the b^{th} element of the array A is equal to c, for an array of values A and singular values b and c.

Definition The *composition* of two Latin Squares X, Y, denoted XY, is the square whose i^{th} row is the composition as permutations[5] of the i^{th} row of X with that of Y. This means $(XY)_{ij} = Y_i(X_i(j))$.

[5] This composition is left to right i.e. $fg(x) = g(f(x))$.

Definition The *inverse* of a Latin Square X is denoted X^{-1}, where the i^{th} row of X^{-1} is the inverse permutation of the i^{th} row of X. This means $(X^{-1})_{ij} = X_i^{-1}(j)$.

Theorem[6] Two Latin Squares X, Y are orthogonal if and only if there exists a Latin Square W such that $XW = Y$.

Corollary Two Latin Squares X, Y are orthogonal if and only if there exists a Latin Square W such that $W = X^{-1}Y$.

Using this corollary, we define

$$W_{ij} \coloneqq \text{value of cell } (i,j) \text{ in } W = X^{-1}Y$$

$$W_{ij} \in \{0,1,\dots,n-1\}$$

The condition $XW = Y$ is encoded as $Y_{ij} = W_{i,X_{ij}}$. Expressed as an indexing constraint this is $W_i[X_{ij}] = Y_{ij}$, where W_i is the i^{th} row of W. We thus only need to impose the Latin property on W to force X and Y to be orthogonal. We call this model "CP-index". Either of the two CP models presented may be extended to higher orders by defining $Z_{ij}^{X^aX^b}$ or $W_{ij}^{X^aX^b}$ to be the variables encoding orthogonality between X^a and X^b in a system of kMOLS(n). We thus use $\binom{k}{2}$ such sets of variables to represent the mutual orthogonality between all k squares.

4. Symmetry Breaking

The search space for the kMOLS(n) problem is filled with many solutions which are functionally identical, or isomorphic. These sets of isomorphic solutions constitute what are called *symmetries* of the problem, and elimination of such symmetries reduces the size of the search space[7]. In a system of kMOLS(n) with squares $(X^1, X^2, \dots, X^k)$ we may perform any of the following operations while maintaining the solution

i) Permute $X^1, X^2, \dots, X^k$
ii) Permute the rows of $X^1, X^2, \dots, X^k$ simultaneously
iii) Permute the columns of $X^1, X^2, \dots, X^k$ simultaneously
iv) Permute the symbols $\{0,1,\dots,n-1\}$ independently in each square

[6] Numerous proofs of this theorem seem to exist, see https://www.jstor.org/stable/2235844 for a proof published in 1942, the earliest we were able to find.

[7] The search space is the set of all possible ways of fixing the variables.

This set of symmetries constitutes up to $k!\,(n!)^{k+2}$ isomorphic solutions for any single solution to kMOLS(n). There are more symmetries which exist [7] however for the scope of our project we only actively eliminate some of those mentioned above. We will focus on only three mutually orthogonal Latin Squares X, Y, Z. Appa *et al.* fix the first row of each square in the system is fixed to be in lexicographic order or $(0,1,\ldots,n-1)$ – this prevents permutations of the columns and symbols. They also fix the first column of X to be in lexicographic order which prevents permutations of the rows. With these fixed entries, we see $Y_{i0} \neq i$. Appa *et al.* further reduce the symmetries of the system by constraining the domains of the first column of Y significantly. This is accomplished by showing that a set of kMOLS(n) is isomorphic to one where $Y_{10} = 2, Y_{i0} \neq i$ and $Y_{i0} \leq i+1$. Imposing the above method of symmetry breaking reduces the number of ways of fixing the first column of Y to the $(n-2)^{\text{th}}$ Fibonacci number [8].

We improve upon Appa *et al.*'s method of symmetry breaking by further reducing the number of such first columns of Y to sequence value $n-1$ of entry A002865 in the Online Encyclopedia of Integer Sequences (OEIS). Let X and Y be two mutually orthogonal Latin squares where the first rows of X and Y and the first column of X are fixed in lexicographic order.

Definition Let $M = (X^1, X^2, \ldots, X^k)$ be k MOLS of order n. M is said to be in *standard form* if the first row of X^i and the first column of X^1 is in lexicographic order for $i = 1, \ldots, k$.

Definition The *first column permutation* of a Latin Square Y is the permutation on the symbols $\{0,1,\ldots,n-1\}$ defined by $\rho(i) = Y_{i0}$. The *ordered cycle* of a Latin Square Y is the representation of the first column permutation of Y as a product of disjoint cycles:

$$(a_{0,0}, \ldots, a_{0,l_0-1})(a_{1,0}, \ldots, a_{1,l_1-1}) \ldots (a_{c,0}, \ldots, a_{c,l_c-1})$$

where each cycle is the lexicographic least cyclic shift[8], and the cycles are in lexicographic order. The *ordered cycle type* of Y is $(l_0, l_1, \ldots, l_c)$

Theorem Let X and Y be a pair of orthogonal Latin squares of order n in standard form. Then X and Y are isomorphic to a pair of orthogonal Latin squares X' and Y' in standard form, where the ordered cycle type of Y' is non-decreasing.

[8] The lexicographic least cyclic shift of a cycle is the equivalent representation of that cycle with minimal ordering i.e., for cycle (3,4,2) the least cyclic shift would be (2,3,4).

Proof Denote by ρ the first column permutation of Y and suppose

$$\rho = (0)(a_{0,0}, \dots, a_{0,l_0-1})(a_{1,0}, \dots, a_{1,l_1-1}) \dots (a_{c,0}, \dots, a_{c,l_c-1}),$$

where $l_0, l_1, \dots, l_c$ are in non-decreasing order. Let r be the permutation whose values $r(0), r(1), \dots, r(n-1)$ are given by

$$0, a_{0,0}, a_{0,1}, \dots, a_{0,l_0-1}, a_{1,0}, a_{1,1}, \dots, a_{1,l_1-1}, \dots, a_{c,0}, a_{c,1}, \dots, a_{c,l_c-1}.$$

We apply r to the symbols of X and Y and denote by X', Y' the resulting squares. This gives $X'_{i0} = r(i)$ and $Y'_{i0} = r(Y_{i0})$. We then permute the columns of X' and Y' so that the first rows of each square are in lexicographic order. Applying r^{-1} to the rows of X' and Y' gives $X'_{i0} = r\left(r^{-1}(i)\right) = i$ and $Y'_{i0} = r(Y_{r^{-1}(i)0}) = r\rho r^{-1}(i)$ which is the conjugate permutation of ρ by r. X', Y' are now in standard form. Conjugation preserves the cycle structure of a permutation and replaces each element $x \in \rho$ with $r(x)$. Thus $r\rho r^{-1} = (1,2, \dots, l_0 - 1)(l_0, \dots, l_1 - 1) \dots (l_{c-1}, \dots, l_c - 1)$ and the ordered cycle type $(l_0, l_1, \dots, l_c)$ is non-decreasing. This method extends equally well to any set of kMOLS(n). ■

The implication of this theorem is that we may fix the first column of Y to be one of several "canonical representatives". We choose these representatives so that their ordered cycle types are non-decreasing, only ensuring that each has a different cycle type and that they obey Appa *et al.*'s domain reduction method. This allows us to conduct our search for solutions in two possible ways. The first way is by fixing the first column of Y to be each of the canonical representatives and running each column fixing in parallel, allowing us to fix all values of the first column of Y rather than imposing constraints on its domain. Both the IP and CP solvers we used seem to heavily benefit from the fixing of variables, but only the CP solver responded well to Appa *et al.*'s domain reduction method. This is because the IP model does not permit individual values to be set, we may only specify the tuple $(X_{ij}, Y_{ij}) = (a, b)$. We thus impose domain reduction by telling the IP solver which tuples are not allowed, which does not reduce the number of constraints in the problem. The running time for solving any model using this method of parallelization is thus the time it takes for the first solution to be found across any of the instances. The second way to conduct the search is to implement so called "blocking constraints", which enforce that the first column of Y be canonical. In this method we choose a_n tuples – each having a different cycle type – and use constraints to enforce that the first column of Y be fixed to one of these tuples. This way we let the solver decide how to fix the first column, having the number of possibilities severely reduced. Let $a(n)$ be the $(n-2)^{\text{th}}$ Fibonacci number and $b(n)$ be the $(n-1)^{\text{th}}$ value of A002865 in the

OEIS. The following table shows the growth rate of $a(n)$ compared to that of $b(n)$, and demonstrates how many fewer column possibilities we consider using the cycle type method.

n	3	4	5	6	7	8	9	10	11	12
$a(n)$	1	1	2	3	5	8	13	21	34	55
$b(n)$	1	1	2	2	4	4	7	8	12	14

Table 1 – Growth rate of $a(n)$ vs. $b(n)$

5. Implementation

The programs to solve systems of 2MOLS(n) and 3MOLS(n) were written in Microsoft Visual C++ and ran on an Intel Core i9 processor with 32GB of RAM. Each process was allotted 1 core (approx. 10% of the CPU power at any given point) to ensure comparability of results based on raw processing power. To solve the IP model, we used Gurobi version 9 [9], which is a popular commercial linear optimizer with free academic licenses. For the CP models we use Google OR-Tools version 7 [10], which is an open-source constraint satisfaction solver. Both solvers are extremely efficient. However they provide little to no information on their internal mechanics. For solving the 3MOLS(n) problem we are unable to use either the IP model or the CP model directly[9] (without additional information or symmetry breaking) for all but the smallest values of n. We instead perform the search for 3MOLS(n) by generating a candidate square for Z and attempting to find X and Y such that X, Y, Z are mutually orthogonal using one of our models. The candidate square is found using the CP solver, and for all orders we tested took negligible time to find. We first generate constraint sets (7) and (8) from the particular Z and attempt to solve the model. If no solution exists or is found within our timeout of 60,000 seconds, we choose a new Z, remove the previous constraints and repeat. Using the CP model extended to 3MOLS(n) we simply set the values of Z and run the model normally. The source code for all our programs as well as the logs from various time trials are available online at https://github.com/noahrubin333/CP-IP.

6. Results

6.1 2MOLS Timings

In every instance we ran, the solver either timed out after 60,000s, found a solution, or reported that no such solution exists. Our control timings were those which use no

[9] For $n \leq 5$ the pure models did return solutions.

symmetry breaking and let the solvers run in their default states. We also ensure that the flag in Gurobi used to control symmetry breaking is set to its "off" state. This makes Gurobi not perform its own internal symmetry breaking, which would have significantly skewed the observed results. Out of the three models tested, the CP-index model performed the best in the control state. It is important to see how the solvers perform without any assistance, as this allows us to understand how much of an impact our symmetry breaking has.

Model	5	6	7	8	9	10
IP	0.1	Timeout	3.2	6.4	344.5	3,046.4
CP-linear	0.0	Timeout	8.0	1,967.1	58,637.8	Timeout
CP-index	0.0	Timeout	7.8	36.3	378.7	214.8

Table 2 – Time in seconds for each model to solve 2MOLS(n) *with no symmetry breaking.*

We next present the timings observed when applying different forms of symmetry breaking to the IP and CP models. When applying symmetry breaking methods to the models, we indicate whether the strategy used was Appa *et al.*'s domain reduction[10], or our cycle type method. In the latter case we report the median running time across three trials, each given a random first column fixing which has the specified cycle type. The column fixings are categorized by a Python script, which assigns an ordered cycle type to a valid tuple representing the first column of Y.

n	Sym. Breaking	IP	CP-linear	CP-index
5	Domain Red.	0	0	0
5	(1,2,2)	0	0	0
5	(1,4)	0	0	0
6	Domain Red.	2	1	0
6	(1,5)	0	0	0
6	(1,2,3)	0	0	0
7	Domain Red.	6	1	0
7	(1,2,2,2)	0	1	0
7	(1,6)	0	3	0
7	(1,2,4)	0	4	0
7	(1,3,3)	0	1	0
8	Domain Red.	23,729	759	6
8	(1,3,4)	2	1,282	5
8	(1,2,2,3)	4	293	7
8	(1,2,5)	4	164	13
8	(1,7)	2	368	10
9	Domain Red.	Timeout	Timeout	290

[10] The domain reduction method imposes $Y_{10} = 2, Y_{i0} \neq i$ and $Y_{i0} \leq i + 1$.

9	(1,3,5)	808	Timeout	485
9	(1,2,2,4)	612	Timeout	212
9	(1,8)	303	Timeout	1,200
9	(1,2,6)	1,082	Timeout	61
9	(1,2,2,2,2)	1,832	Timeout	784
9	(1,4,4)	1,315	Timeout	917
9	(1,2,3,3)	65	Timeout	223
10	Domain Red.	Timeout	Timeout	1,379
10	(1,4,5)	52,329	Timeout	3,892
10	(1,2,2,5)	29,387	Timeout	4,383
10	(1,2,2,2,3)	6,128	Timeout	1,326
10	(1,2,3,4)	15,534	Timeout	835
10	(1,3,6)	31,005	Timeout	3,244
10	(1,2,7)	Timeout	Timeout	1,413
10	(1,3,3,3)	36,959	Timeout	3,886
10	(1,9)	Timeout	Timeout	1,319
11	Domain Red.	Timeout	Timeout	29,925
11	(1,3,7)	Timeout	Timeout	32,815
11	(1,2,2,3,3)	Timeout	Timeout	41,665
11	(1,3,3,4)	Timeout	Timeout	Timeout
11	(1,2,8)	Timeout	Timeout	11,524
11	(1,2,2,6)	Timeout	Timeout	8,982
11	(1,2,2,2,4)	Timeout	Timeout	8,299
11	(1,2,2,2,2,2)	Timeout	Timeout	10,206
11	(1,2,4,4)	Timeout	Timeout	19,571
11	(1,10)	Timeout	Timeout	13,210
11	(1,4,6)	Timeout	Timeout	40,077
11	(1,2,3,5)	Timeout	Timeout	Timeout
11	(1,5,5)	Timeout	Timeout	7,359

Table 3 – Median time in seconds (across 3 trials) for each model to solve 2MOLS(n) *with specified symmetry breaking.*

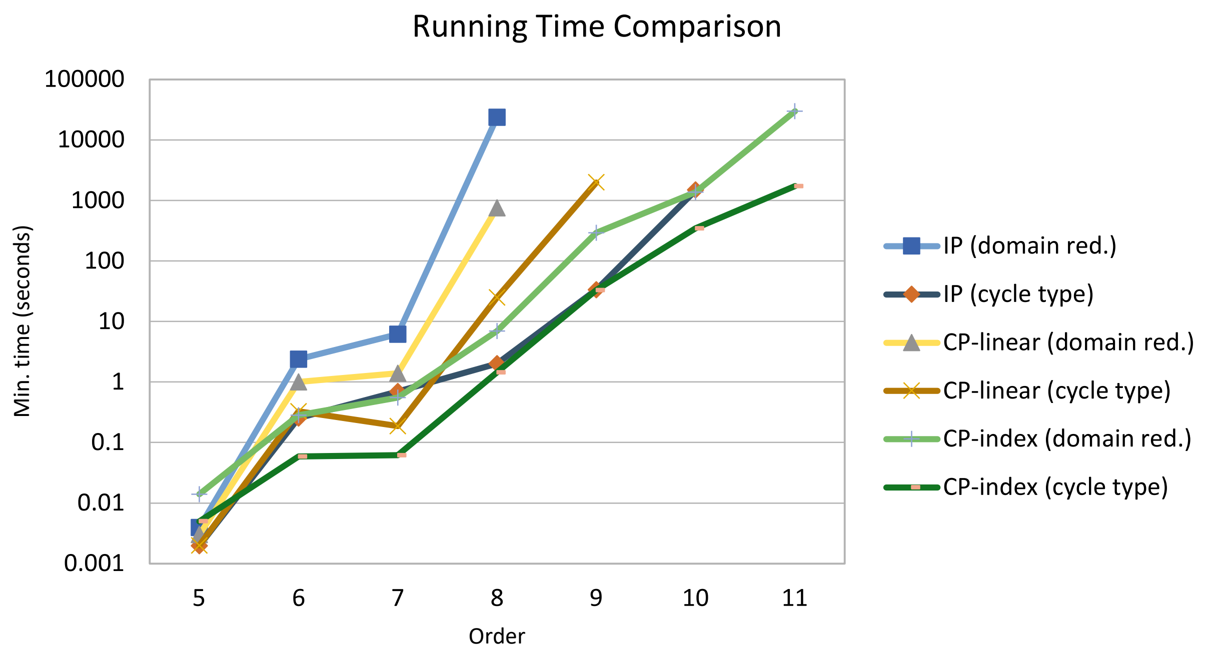

Fig 2 – Plot of minimum running time in seconds across all models.

Analyzing the collected data, we see that the IP solver performs quite poorly when we impose the Domain Reduction symmetry breaking. The IP solver also sees the most benefit from imposing the Cycle Type symmetry breaking. The reason for the poor performance of the domain reduction method is hypothesized to be due to the internal mechanics of Gurobi. Unfortunately, due to the proprietary nature of Gurobi we cannot know for sure how the solver is affected by our symmetry breaking.

Across all symmetry breaking methods we see that the CP solver always performs much better when using indexing constraints instead of linear. Within each CP model we see that using the Cycle Type method often resulted in better running times, but there were some outlier cases in which the solver was slowed down by the choice of column fixing. This behavior does not seem to be related to the cycle type of the column fixing, but rather simply how the fixing itself is handled by the solver, and the level of randomness present in the search[11,12]. Overall, we were very encouraged by the running times we observed in the 2MOLS(n) cases. In some instances, the Cycle Type method allowed us to obtain a solution to the problem an order of magnitude faster. In the future

[11] Likely due to the internal heuristics being employed by the solvers – unfortunately it is extremely difficult to determine how different fixings will affect runtime before they are tried.

[12] We pass a random seed to the solver each trial to introduce variability into the searches.

we would also like to test how the solvers respond to the alternative method of Cycle Type symmetry breaking – where we impose that the first column of Y be one of the canonical column fixings, but not a specific one. As an example of this, consider the following set of canonical column fixings for $n = 8$:

(0,2,1,4,5,6,7,3) with cycle type (1,2,5)

(0,2,3,4,5,6,7,1) with cycle type (1,7)

(0,2,3,1,5,6,7,4) with cycle type (1,3,4)

(0,2,1,4,3,6,7,5) with cycle type (1,2,2,3)

Our first strategy would use 4 parallel runs, where each run has the first column of Y set to one of these canonical fixings. In this new strategy we instead impose a minimal set of constraints which impose that of the possible ways to fix the first column of Y, only those which are in the above set of canonical fixings may be used. This amounts to the additional constraints $Y_{50} = 6$ and $Y_{40} \neq 1$, which are added to the existing constraints from Appa *et al.*'s domain reduction method.

6.2 3MOLS Timings

As mentioned before, the models used for 3MOLS(n) are extensions of those used for 2MOLS(n) where the third square, Z, is predetermined. We use OR-Tools to enumerate all Latin Squares of order n sequentially, and then use Gurobi to construct X and Y. Extending the symmetry breaking to Z we fix the first row of Z in lexicographic order and ensure $Z_{i0} \neq i$. This helps to avoid generating Latin Squares which would trivially conflict with the already fixed values in X and Y, thus reducing the overall number of possible Z to search. The times reported are the average time taken by the IP solver to report either a solution or the lack thereof. This is calculated as our timeout (60,000s) divided by the number of squares the solver could process before this timeout.

n	Time per Z
7	0.02
8	3.58
9	397.35
10	4,615.38
11	Timeout

Table 4 – Time in seconds to solve an instance of 3MOLS(n) *with Z fixed using IP.*

7. Conclusion

In this project we use the mathematical programming paradigms of Integer and Constraint programming to solve the 2MOLS(n) and later 3MOLS(n) problems. We attempt to replicate the work of Appa *et al.* and improve upon their CP model and symmetry breaking methods. Using these new methods, we reduce our observed running times by a significant factor. The Cycle Type method of symmetry breaking has potential for the parallelization of a search for higher orders of k and n – and an interesting direction this project may eventually take would involve testing the efficacy of the Cycle Type method on a large-scale computational cluster (such as Microsoft Azure or Compute Canada supercomputers).

Although the initial scope of this project was as a research training award with a timeline of 16 weeks, I have found myself continuing the work over a year later. I am extremely grateful for the opportunity afforded to me by my supervisors – and am thrilled by the fact that our work has been published in the ICTAI 2021 conference proceedings [11]. The time I have spent working in the realm of combinatorics and operations research has forever changed the trajectory of my academic career, and I look forward to continuing this research.

Bibliography

[1] C. J. Colbourn and J. H. Dinitz, Handbook of combinatorial designs, CRC press, 2006.

[2] R. C. Bose and S. S. Shrikhande, "On the falsity of Euler's conjecture about the non-existence of two orthogonal Latin squares of order 4t+2," *Proc. Nat. Acad. Sci. U.S.A.,* vol. 45, p. 734–737, 1959.

[3] R. C. Bose, S. S. Shrikhande and E. T. Parker, "Further results on the construction of mutually orthogonal Latin squares and the falsity of Euler's conjecture," *Canadian J. Math.,* vol. 12, p. 189–203, 1960.

[4] C. Bright, K. Cheung, B. Stevens, D. Roy, I. Kotsireas and V. Ganesh, "A Nonexistence Certificate for Projective Planes of Order Ten with Weight 15

Codewords," *Applicable Algebra in Engineering, Communication and Computing,* vol. 31, p. 195–213, 2020.

[5] C. Bright, K. Cheung, B. Stevens, I. Kotsireas and V. Ganesh, "A SAT-based Resolution of Lam's Problem," in *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, 2021.

[6] G. Appa, D. Magos and I. Mourtos, "Searching for mutually orthogonal Latin squares via integer and constraint programming," *European journal of operational research,* vol. 173, p. 519–530, 2006.

[7] B. D. McKay and I. M. Wanless, "On the number of Latin squares," *Ann. Comb.,* vol. 9, p. 335–344, 2005.

[8] N. Rubin, C. Bright, K. K. H. Cheung and B. Stevens, "Integer and Constraint Programming Revisited for Mutually Orthogonal Latin Squares," *arXiv:2103.11018,* 2021.

[9] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual,* 2020.

[10] L. Perron and V. Furnon, *OR-Tools,* 2019.

[11] N. Rubin, C. Bright, K. K. H. Cheung, B. Stevens, "Improving Integer and Constraint Programming for Graeco-Latin Squares," in *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2021.