Orthogonal Latin Squares of Order Ten with Two Relations: A SAT Investigation

Curtis Bright[1,2,3][0000−0002−0462−625X], Amadou Keita[2][0009−0001−5861−4617], and Brett Stevens[3][0000−0003−4336−1773]

[1] School of Computer Science, University of Waterloo, Canada
[2] Department of Mathematics and Statistics, University of Windsor, Canada
[3] School of Mathematics and Statistics, Carleton University, Canada
cbright@uwaterloo.ca, keitaa@uwindsor.ca, brett@math.carleton.ca

Abstract. A k-net(n) is a combinatorial design equivalent to $k-2$ mutually orthogonal Latin squares of order n. A relation in a net is a linear dependency over $\mathbb{F}_2$ in the incidence matrix of the net. A computational enumeration of all orthogonal pairs of Latin squares of order 10 whose corresponding nets have at least two nontrivial relations was achieved by Delisle in 2010 and verified by an independent search of Myrvold. In this paper, we confirm the correctness of their exhaustive enumerations with a satisfiability (SAT) solver approach instead of using custom-written backtracking code. Performing the enumeration using a SAT solver has at least three advantages. First, it reduces the amount of trust necessary, as SAT solvers produce independently-verifiable certificates that their enumerations are complete. These certificates can be checked by formal proof verifiers that are relatively simple pieces of software, and therefore easier to trust. Second, it is typically more straightforward and less error-prone to use a SAT solver over writing search code. Third, it can be more efficient to use a SAT-based approach, as SAT solvers are highly optimized pieces of software incorporating backtracking-with-learning for improving the efficiency of the backtracking search. For example, the SAT solver completely enumerates all orthogonal pairs of Latin squares of order ten with two nontrivial relations in under 2 hours on a desktop machine, while Delisle's 2010 search used 11,700 CPU hours. Although computer hardware was slower in 2010, this alone cannot explain the improvement in the efficiency of our SAT-based search.

Keywords: Latin square · orthogonal Latin square · net · satisfiability solving.

1 Introduction

A k-net(n) is a set of n^2 points and kn lines with the following properties:

1. Every line contains n points and every point lies on k lines.
2. There are k parallel classes of lines, with each parallel class containing n lines that do not intersect each other.

3. Every pair of lines from different parallel classes intersect exactly once.

An *incidence matrix* of a k-net(n) is an $n^2 \times kn$ matrix over $\mathbb{F}_2 = \{0, 1\}$ whose (i, j)th entry is 1 exactly when the ith point lies on the jth line. In this paper without loss of generality we assume that the lines in a net are always ordered by parallel class, i.e., the first parallel class consists of the first n lines in the net, the second parallel class consists of the next n lines in the net, etc. Under this ordering, the axioms of a net imply that if A is the incidence matrix of a k-net(n), then

$$A^T A = \begin{bmatrix} nI & J & \cdots & J \\ J & nI & \cdots & J \\ \vdots & \vdots & \ddots & \vdots \\ J & J & \cdots & nI \end{bmatrix}$$

where here A and A^T are considered as matrices over $\mathbb{Z}$, I is the identity matrix of order n, and J is the all-ones matrix of order n. This follows because the (i, j)th entry of $A^T A$ counts the number of points occurring on both the ith and jth line, and this count is 1 when the lines are in different parallel classes, 0 when the lines are distinct and in the same parallel class, and n when the lines are identical (i.e., $i = j$).

It is well-known that a k-net(n) is equivalent to $k - 2$ mutually orthogonal Latin squares of order n, denoted $(k - 2)\,\mathrm{MOLS}(n)$, and it is straightforward to convert a k-net to a collection of $k - 2$ Latin squares and vice versa. Roughly speaking, the net's first parallel class corresponds to Latin square rows, the net's second parallel class corresponds to Latin square columns, and for $\ell > 2$ the net's ℓth parallel class corresponds to symbols of the $(\ell - 2)$th Latin square. When (i, j)th entry of the $(\ell - 2)$th Latin square contains symbol s, its corresponding net will contain a point that lies on the ith line of the first parallel class, the jth line of the second parallel class, and the sth line of the ℓth parallel class.

A *relation* in a k-net(n) is a linear dependency in the columns of A over $\mathbb{F}_2$. The *rank* of k-net(n) is the rank of its incidence matrix. The rank of a k-net(n) is at most $kn - k + 1$ (c.f. [13]) since there are $k - 1$ trivial relations formed by the lines in the first parallel class and the ith parallel class for $2 \leq i \leq k$. Howard [14] considered nets of order $n \equiv 2 \pmod 4$ of which the case $n = 10$ is of particular interest. She showed that a 6-net(10) has rank ≤ 53 and therefore contains at least two nontrivial relations, and also that a 4-net(10) has rank ≥ 33 and therefore contains at most four nontrivial relations.

Delisle, under the supervision of Myrvold, computationally enumerated all 4-nets(10) with at least two nontrivial relations [10]. They found that there exist no 4-nets(10) with four nontrivial relations, up to isomorphism there are six 4-nets(10) with exactly three nontrivial relations, and up to isomorphism there are 85 4-nets(10) with exactly two nontrivial relations. More recently, Gill and Wanless [12] computationally enumerated all 4-nets(10) with at least one nontrivial relation and found that up to isomorphism there are exactly 18,526,229 4-nets(10) with exactly one nontrivial relation.

A 4-net(10) is equivalent to an orthogonal pair of Latin squares of order 10. Latin squares of order 10 are of particular interest because 10 is the first order for which the largest collection of mutually orthogonal Latin squares is unknown. It is known that 9 mutually orthogonal Latin squares of order 10 would be equivalent to a projective plane of order ten, but this was ruled out by exhaustive computer search [17]. Combined with a result of Bruck [9], this implies that 7 MOLS(10) do not exist. Thus, the maximum number of mutually orthogonal Latin squares of order 10 is between 2 and 6. As a consequence of the searches of Delisle [10] and Gill and Wanless [12], all 2 MOLS(10) with nontrivial relations are known and none of them are part of a 3 MOLS(10).

In this paper, we verify the searches of Delisle [10] by exhaustively enumerating all 2 MOLS(10) with two nontrivial relations. Our approach differs from Delisle's original search and the independent verification of Gill and Wanless [12] because our search uses a Boolean satisfiability (SAT) solver. We reduce the existence of a 2 MOLS(10) with two relations into a problem of Boolean logic and then use a SAT solver to find all solutions of the logic problem (and correspondingly all 2 MOLS(10) with two relations). SAT solvers can be surprisingly effective at solving various problems in mathematics [6] and recently they have been used with increasing frequency to solve problems in combinatorics and design theory [23]. For example, in 2021 they were used to verify Lam et al.'s result that order ten projective planes do not exist [4].

Traditionally, backtracking algorithms are used to computationally search for combinatorial designs [15]. SAT solvers offer an alternative approach to the backtracking paradigm. Although SAT solvers also perform a form of backtracking, they also use a powerful learning process known as conflict-driven clause-learning [19]. This technique along with many other optimizations and heuristics that have been fine-tuned over decades enables SAT solvers to outperform traditional backtracking search in many problems of interest.

Moreover, even if SAT solvers were not as efficient as custom backtracking approaches, they have the advantage of making the search process less error-prone because using a SAT solver does not require writing any code for performing a search. Unfortunately, it is a reality of software development that almost all computer programs have bugs, and writing efficient search code is an inherently error-prone process [16]. This is a particularly important consideration when a computer program purports to perform an exhaustive classification of a mathematical object. How can we trust that a bug in the program did not cause some objects to be missed in the search? It is typically impossible for even professional programmers to guarantee their code has no bugs due to the inherent difficulties in writing computer code. One way of decreasing the chance that a bug results in a missed object is to perform the same search with two implementations written independently. While this does reduce the chance of something being missed, it still relies on trusting code that cannot be certified to be correct. Indeed, it has happened that mathematical designs have been missed by a search with an independent verification, for example, in Lam's problem [8] and the enumeration of good matrices of order 27 [5].

Using a SAT solver sidesteps the need for writing search code. Instead, one generates a "SAT encoding" specifying the properties that the object in question must satisfy, and then the SAT solver searches for the object. Moreover, the SAT solver *does not need to be trusted itself*, because during the search it produces a certificate that can be checked for correctness by a proof verifier—a simpler piece of software that can be written independently from the SAT solver. This approach does rely on the SAT encoding being correct, but this typically requires less trust than would be required of a search procedure. We describe the SAT encoding we use in Section 3, a crucial component of which is a SAT encoding of the symmetry breaking used in Delisle's original search (described in detail along with other background in Section 2).

Although SAT solvers do not always perform well on mathematical problems, they performed remarkably well in the enumeration of orthogonal pairs of Latin squares of order ten with two relations, and this only required minor modifications to the SAT solver and proof verifier (as described in Section 4). We enumerated all 91 pairs of Latin squares of order ten with two relations in less than 1.75 hours on a desktop computer (see Section 5). Conversely, Delisle's original backtracking search, written in the programming language C and optimized for speed, used around 488 CPU days in 2010. While some of the improvements in speed of our SAT-based search is undoubtedly from the improvements in computer hardware, this alone cannot account for an over 6000× speedup in CPU time, demonstrating the computational efficiency of SAT solvers on problems of this type.

2 Background

The *type* of a relation is a list of the number of lines each parallel class contributes to the relation. For example, a relation in a 4-net(n) that has 4 lines from the first two parallel classes and 6 lines from the last two parallel classes has type $[4, 4, 6, 6]$. Because the union of two parallel classes form a trivial relation, without loss of generality any relation can be "complemented" and written in a form where at most one entry in the type is larger than $n/2$. In her PhD thesis, Howard [14] studies relations in nets of order $n \equiv 2 \pmod 4$. In particular, she proved the following proposition.

Proposition 1 ([14, Prop. 5.4]). *Every nonempty relation in a 4-net of order $n \equiv 2 \pmod 4$ with at most $n/2$ lines in three classes must be of the type $[k, k, k, k]$ where k is an even integer with $n/3 \leq k < n/2$.*

For example, Proposition 1 implies that every nontrivial relation in a 4-net(10) is complementable to one of type $[4, 4, 4, 4]$, because every nontrivial relation in a 4-net(10) is complementable to a nonempty relation with at most 5 lines in three classes. Now consider the case of a 4-net(10) with two linearly independent nontrivial relations. Note that the sum of two nontrivial relations will produce a third relation. Although this relation will not be linearly independent, it will be nontrivial, and therefore by Proposition 1 complementable to a relation of the type $[4, 4, 4, 4]$. The following proposition constrains the manner in which two linearly independent relations R_1 and R_2 will intersect.

Proposition 2 (cf. [10]). *Suppose R_1 and R_2 are two linearly independent relations of type $[4,4,4,4]$ in a 4-net(10). Then each parallel class contains exactly one or two lines from both R_1 and R_2.*

Proof. Suppose x is the number of lines in the first parallel class and both R_1 and R_2. Since the sum of R_1 and R_2 is a third relation that is complementable to one of type $[4,4,4,4]$, we have that $(4-x)+(4-x)$, the number of lines in the third relation and the first parallel class, is either 4 or $10-4=6$. In the former case $x=2$ and in the latter case $x=1$. The same argument applies to every parallel class. □

Following Delisle, we will suppose the lines appearing in both R_1 and R_2 are ordered to appear first in each parallel class, followed by the lines in R_1 and not R_2, and then the lines in R_2 and not R_1. The remaining lines, those not in either R_1 and R_2, appear last. Say that there are x_i lines in parallel class i and $R_1 \cap R_2$, there are y_i lines in parallel class i and $R_1 \setminus R_2$, and there are z_i lines in parallel class i and $R_2 \setminus R_1$. We use the notation $[[x_1,y_1,z_1],[x_2,y_2,z_2],[x_3,y_3,z_3],[x_4,y_4,z_4]]$ to denote the form of R_1 and R_2. By Proposition 2, without loss of generality the possible forms can be taken to be one of the five cases

$$[[1,3,3],[1,3,3],[1,3,3],[1,3,3]],$$
$$[[1,3,3],[1,3,3],[1,3,3],[2,2,2]],$$
$$[[1,3,3],[1,3,3],[2,2,2],[2,2,2]],$$
$$[[1,3,3],[2,2,2],[2,2,2],[2,2,2]],$$
$$[[2,2,2],[2,2,2],[2,2,2],[2,2,2]].$$

Furthermore, Delisle used a counting argument to rule out cases 2 and 4 [10, pg. 17]. Thus, Delisle's search focused on cases 1, 3, and 5. Interestingly, using our approach a SAT solver rules out cases 2, 3, and 4 in a few seconds each. Given case 3 was ruled out by Delisle using 23 days of compute time, the SAT solver shows its strength at uncovering contradictions by ruling out case 3 relatively quickly. At the end of Section 5 we present a less computational argument ruling out case 3 that was found independently from the SAT computation.

2.1 Delisle's symmetry breaking

In order to perform an exhaustive enumeration up to isomorphism it is advantageous to restrict the search space as much as possible without losing any solutions up to isomorphism—this process is known as *symmetry breaking*. In this section we recount the symmetry breaking used in Delisle's thesis [10] for 4-nets(10) with two relations (stated in terms of 2 MOLS(10)). Say (A,B) is a 2 MOLS(10) whose corresponding net has 2 nontrivial relations. Delisle's method for adding symmetry breaking constraints on (A,B) is based on adding additional constraints on the entries in the first column and row of A and B.

Suppose (A,B) is a pair of orthogonal Latin squares with two relations of the form

$$[[x_1,y_1,z_1],[x_2,y_2,z_2],[x_3,y_3,z_3],[x_4,y_4,z_4]].$$

The form of the relations define equivalence classes on the rows (from parallel class 1), columns (from parallel class 2), symbols of A (from parallel class 3), and symbols of B (from parallel class 4). For example, the row equivalence classes are determined by the values of x_1, y_1, and z_1: explicitly, the equivalence classes of rows will be defined by the index sets $[0, x_1)$, $[x_1, x_1 + y_1)$, $[x_1 + y_1, x_1 + y_1 + z_1)$, and $[x_1 + y_1 + z_1, 10)$. Note that the following six equivalence operations on pairs of Latin squares (A, B) preserve the orthogonality of the squares and the form of the relations.

1. Permutation of rows of A and B preserving the row equivalence classes.
2. Permutation of columns of A and B preserving the column equivalence classes.
3. Permutation of the symbols of A preserving the A-symbol equivalence classes.
4. Permutation of the symbols of B preserving the B-symbol equivalence classes.
5. Taking the transpose of A and B (when the row and column equivalence classes match, i.e., cases 1–3 and 5).
6. Swapping A and B (when the A-symbol and B-symbol equivalence classes match, i.e., cases 1 and 3–5).

Fix an ordering of the entries of a Latin square in the following way: the entries of the first column (from top to bottom) are first, and then the entries of the first row (from left to right) are next. The remaining entries can be ordered arbitrarily. In this way, if A and A' are two distinct Latin squares we say $A < A'$ if on the first entry in which A and A' differ, say at index (i, j), we have $A_{ij} < A'_{ij}$. Similarly, pairs of Latin squares can be ordered after providing an ordering on pairs of symbols. For this, a lexicographic ordering is used: say that $(a, b) < (a', b')$ when either $a < a'$ or $a = a'$ and $b < b'$. Then, if (A, B) and (A', B') are two distinct pairs of Latin squares, we say $(A, B) < (A', B')$ if on the first entry in which (A, B) and (A', B') differ, say at index (i, j), we have $(A_{ij}, B_{ij}) < (A'_{ij}, B'_{ij})$.

A pair of orthogonal Latin squares (A, B) is said to be a *minimal pair* if (A, B) cannot be decreased under the ordering described above by using the equivalence operations described above. Delisle's symmetry breaking method is based on the following six propositions. In each, (A, B) is a pair of orthogonal Latin squares and each proposition gives a necessary condition for (A, B) to be a minimal pair.

Proposition 3. *If (A, B) is a minimal pair then $(A_{1,0}, B_{1,0}) < (A_{0,1}, B_{0,1})$ (except possibly in case 4).*

Proof. Suppose (A, B) is a minimal pair with $(A_{1,0}, B_{1,0}) \geq (A_{0,1}, B_{0,1})$. Since A and B are orthogonal, $(A_{1,0}, B_{1,0})$ and $(A_{0,1}, B_{0,1})$ are distinct, and thus $(A_{1,0}, B_{1,0}) > (A_{0,1}, B_{0,1})$. Applying the transpose operation to A and B does not affect $A_{0,0}$ and $B_{0,0}$, but replaces $(A_{1,0}, B_{1,0})$ with $(A_{0,1}, B_{0,1})$, so $(A^T, B^T) < (A, B)$ in contradiction to the fact that (A, B) is minimal. (This argument does not work in case 4, as the transpose operation is not an equivalence operation in case 4.) □

Proposition 4. *If (A, B) is a minimal pair then $A < B$ (except possibly in case 2).*

Proof. Suppose (A, B) is a minimal pair with $A \geq B$. Since A and B are orthogonal, $A \neq B$, and thus $A > B$ and there is some entry on which A and B do not match. After swapping A and B the first entry on which the mismatch occurs will still be in the same place, so $(B, A) < (A, B)$ in contradiction to (A, B) being minimal. (This argument does not work in case 2, as swapping A and B is not an equivalence operation in case 2.) □

Proposition 5. *If (A, B) is a minimal pair then the symbols in the first column of A appear in sorted order within the rows of each row equivalence class.*

Proof. Suppose (A, B) is a minimal pair with the symbols in the first column of A not in sorted order within the rows of each row equivalence class. Using row permutations, permute the rows of (A, B) to form (A', B') such that the rows of the first column of A' are now sorted within the rows of each row equivalence class. Consider the first entry $A'_{i,0}$ of A' that has changed after applying these permutations. (This will also be the first entry of B' that has changed, since the same permutations are applied to A and B.) Because $A'_{i,0} < A_{i,0}$ and $(A'_{j,0}, B'_{j,0}) = (A_{j,0}, B_{j,0})$ for all $j < i$ we have $(A', B') < (A, B)$ in contradiction to (A, B) being minimal. □

Proposition 6. *If (A, B) is a minimal pair then the symbols in the first row of A appear in sorted order within the columns of each column equivalence class (except possibly in case 4; in that case $A_{0,0}$ and $A_{0,1}$ may appear out of order).*

Proof. Suppose (A, B) is a minimal pair with the symbols in the first row of A not in sorted order within the columns of each column equivalence class. First consider cases 1–3, in which case the first column equivalence class consists of only the first column. In this case, the proposition is vacuous for the first column equivalence class, as the list $[A_{0,0}]$ has length 1 and is vacuously sorted. Using column permutations, permute the columns of (A, B) to form (A', B') such that the columns of the first row of A' are now sorted within the columns of the remaining three column equivalence classes. Note that the first column of (A', B') matches the first column of (A, B) since the first column was not permuted. Consider the first entry $A'_{0,i}$ that has changed after applying these permutations. Because $A'_{0,i} < A_{0,i}$ and $(0, i)$ is the first entry in which (A', B') differs from (A, B), it follows that $(A', B') < (A, B)$, in contradiction to (A, B) being minimal.

In cases 4 and 5, the above argument works to sort the entries in the first row of A in each of the last three column equivalence classes, but not the first, so $A_{0,0}$ and $A_{0,1}$ may be out of order. However, in case 5, since (A, B) is a minimal pair whose first two rows are in the same row equivalence class, by Proposition 5 we have $A_{0,0} < A_{1,0}$. By Proposition 3, we also have $A_{1,0} \leq A_{0,1}$. Thus in case 5 the entries of $A_{0,0}$ and $A_{0,1}$ will also appear in sorted order. □

Proposition 7. *If (A, B) is a minimal pair then for each A-symbol equivalence class, the symbols of that equivalence class in the first column of A appear in sorted order.*

Proof. Suppose (A, B) is a minimal pair where the symbols in the same A-symbol equivalence class in the first column of A do not appear in sorted order. Permute the symbols of A to form A' so that for each A-symbol equivalence class the symbols in that equivalence class in the first column of A' appear in sorted order. Consider the first entry $A'_{i,0}$ that differs from $A_{i,0}$. Since $A'_{i,0} < A_{i,0}$ and $A'_{j,0} = A_{j,0}$ for all $j < i$, we have $(A', B) < (A, B)$ in contradiction to (A, B) being minimal. □

Proposition 8. *If (A, B) is a minimal pair then for each B-symbol equivalence class, the symbols of that equivalence class in the first column of B appear in sorted order.*

Proof. Suppose (A, B) is a minimal pair where the symbols in the same B-symbol equivalence class in the first column of B do not appear in sorted order. Permute the symbols of B to form B' so that for each B-symbol equivalence class the symbols in that equivalence class in the first column of B' appear in sorted order. Consider the first entry $B'_{i,0}$ that differs from $B_{i,0}$. Since $B'_{i,0} < B_{i,0}$ and $B'_{j,0} = B_{j,0}$ for all $j < i$, we have $(A, B') < (A, B)$ in contradiction to (A, B) being minimal. □

Finally, we prove another property of minimal pairs that we exploit in our encoding.

Proposition 9. *If (A, B) is a minimal pair then $A_{0,0} = B_{0,0}$ (except possibly in case 2).*

Proof. By Proposition 7 if (A, B) is a minimal pair then $A_{0,0}$ must be in $\{0, 1, 4, 7\}$ in case 1 and in $\{0, 2, 4, 6\}$ in cases 3–5. Similarly, by Proposition 8 if (A, B) is a minimal pair then $B_{0,0}$ must be in $\{0, 1, 4, 7\}$ in case 1 and in $\{0, 2, 4, 6\}$ in cases 3–5. Delisle [10, pg. 18] gives the possibilities for the values of $(A_{0,0}, B_{0,0})$ in case 1, and the only ones which are in $\{0, 1, 4, 7\} \times \{0, 1, 4, 7\}$ are $(0, 0)$, $(1, 1)$, $(4, 4)$, and $(7, 7)$. Similarly, [10, pg. 20] gives the possibilities for the values of $(A_{0,0}, B_{0,0})$ in case 3 (and these are identical in cases 4 and 5), and the only ones which are in $\{0, 2, 4, 6\} \times \{0, 2, 4, 6\}$ are $(0, 0)$, $(2, 2)$, $(4, 4)$, and $(6, 6)$. □

3 SAT Encoding

In this section we describe our SAT encoding for the problem of enumerating 2 MOLS(10) with two nontrivial relations. In order to encode a pair of orthogonal Latin squares (A, B) of order n we use the $2n^3$ Boolean variables A_{ijk} and B_{ijk} for $0 \leq i < n$. The variable A_{ijk} will be true exactly when the (i, j)th entry of square A contains the symbol k, and similarly for the variables B_{ijk} and the entries of the square B.

Modern SAT solvers require the input formulae to be in a format known as conjunctive normal form. An expression in Boolean logic is in *conjunctive normal form* when it is a conjunction of clauses, a *clause* being a disjunction of variables or negated variables. For example, $\neg x \vee y \vee z$ is a clause. We may use the implication operator to express clauses with the meaning that $(x_1 \wedge \cdots \wedge x_n) \to (y_1 \vee \cdots \vee y_m)$ is shorthand for the clause $\neg x_1 \vee \cdots \vee \neg x_n \vee y_1 \vee \cdots \vee y_m$.

Our SAT encoding contains four kinds of constraints: constraints asserting that A and B are Latin squares (see Section 3.1), constraints asserting that A and B are orthogonal (see Section 3.2), constraints asserting that A and B have two nontrivial relations and are in one of the forms specified by cases 1–5 (see Section 3.3), and finally symmetry breaking constraints asserting that A and B satisfy Propositions 3 to 8 (see Section 3.4).

3.1 Latin square encoding

Considering the Boolean variables A_{ijk} as integers (0 for false and 1 for true), in order to specify that they encode a Latin square of order 10 we need to enforce the following three constraints.

1. $\sum_{k=0}^{9} A_{ijk} = 1$ for all $0 \le i, j \le 9$ (every cell has exactly one symbol).
2. $\sum_{j=0}^{9} A_{ijk} = 1$ for all $0 \le i, k \le 9$ (every row contains every symbol exactly once).
3. $\sum_{i=0}^{9} A_{ijk} = 1$ for all $0 \le j, k \le 9$ (every column contains every symbol exactly once).

The most straightforward way of representing the constraint $\sum_{i=1}^{n} x_i = 1$ in Boolean logic is to encode $\sum_{i=1}^{n} x_i \le 1$ via $\bigwedge_{i<j}(\neg x_i \vee \neg x_j)$ and $\sum_{i=1}^{n} x_i \ge 1$ via $\bigvee_{i=1}^{n} x_i$. This encoding performed well in our experiments, but we observed slightly better performance using Sinz's sequential counter encoding [20].

To encode $\sum_{i=1}^{n} x_i \le 1$ in the sequential counter encoding, first the new variables $s_1, \ldots, s_n$ are introduced (s_i representing that at least one of $x_1, \ldots, x_i$ are true) using the $2n - 1$ clauses

$$x_i \to s_i \quad \text{and} \quad s_{i-1} \to s_i$$

for $1 \le i \le n$ (when $i = 1$ the clause $s_0 \to s_1$ is skipped). Once the s_i variables have been introduced, $\sum_{i=1}^{n} x_i \le 1$ is encoded using the $n - 1$ additional clauses $\neg x_i \vee \neg s_{i-1}$ for $2 \le i \le n$, the idea being that x_i and s_{i-1} can never both be true, because that would imply at least two variables in $x_1, \ldots, x_n$ are true. Moreover, $\sum_{i=1}^{n} x_i \ge 1$ can be encoded by setting s_n to true and adding the clauses $s_i \to (s_{i-1} \vee x_i)$ for $1 \le i \le n$ (when $i = 1$ the literal s_0 is left out). Setting s_n to true causes two clauses to be trivially satisfied, so they can be removed. Altogether, we encode $\sum_{i=1}^{n} x_i = 1$ using $4n - 4$ clauses.

3.2 Orthogonality encoding

The orthogonality of two Latin squares A and B of order n can be specified by the logical constraints

$$(A_{ij} = k \wedge A_{i'j'} = k \wedge B_{ij} = l \wedge B_{i'j'} = l) \rightarrow (i = i')$$

for $0 \leq i, j, i', j', k, l < n$ (cf. Zhang [22, Lemma 1]). Taking the contrapositive and writing this using the variables A_{ijk} and B_{ijk}, this is equivalent to the clauses $\neg A_{ijk} \vee \neg A_{i'j'k} \vee \neg B_{ijl} \vee \neg B_{i'j'l}$ where $0 \leq i, j, i', j', k, l < n$ with $i \neq i'$. This encoding of orthogonality uses $O(n^6)$ clauses of length 4. An alternative encoding of orthogonality that performs better in practice [22, Lemma 2] uses $O(n^4)$ clauses of length 3 and n^3 new auxiliary variables. To describe this orthogonality encoding, we follow the derivation of Bright, Keita, and Stevens [7] based on a composition square.

Consider the rows of a Latin square X of order n as a collection of n permutations of the symbols $\{0, \ldots, n-1\}$. The row inverse square X^{-1} is defined to be the Latin square whose rows are formed by the inverses of the rows of X, and the composition square XY is defined to be the square whose ith row is the ith row of X composed with the ith row of Y (in a right-to-left way). Note that the square XY is *not* a Latin square in general. We now provide a theorem that the orthogonality encoding we use relies on.

Theorem 1 (Mann [18]). *Two Latin squares A and B are orthogonal if and only if AB^{-1} is a Latin square.*

Let Z denote the composition square AB^{-1}, and let the Boolean variables Z_{ijk} be true exactly when the (i, j) entry of Z contains the symbol k (where $0 \leq i, j, k < n$). The square Z can be specified to be a Latin square using the encoding from Section 3.1. In order to encode $Z = AB^{-1}$, we need to enforce that the (i, B_{ij})th entry of Z contains the symbol A_{ij}. This is done using the clauses

$$(A_{ijk} \wedge B_{ijl}) \rightarrow Z_{ilk}$$

for $0 \leq i, j, k, l < n$. In fact, from $A = ZB$ we also derive the similar clause $(Z_{ilk} \wedge B_{ijl}) \rightarrow A_{ijk}$, and from $B = Z^{-1}A$ we derive $(Z_{ilk} \wedge A_{ijk}) \rightarrow B_{ijl}$. These last two types of clauses are technically logically redundant, but in practice they improve the performance of the SAT solver as they allow the solver to make additional useful propagations.

3.3 Relation encoding

Let R_1 and denote the indices of the lines in the first relation, and let R_2 denote the indices of the lines in the second relation. Delisle [10] defines an equivalence class on (row, column) Latin square index pairs using a labelling function called RC_CLASS that we describe below. In the following, i represents a Latin square row index, and j represents a Latin square column index, so $0 \leq i, j \leq 9$. Recall that we order the lines of a 4-net(10) so that lines 0 to 9 correspond to row

indices of Latin squares while lines 10 to 19 correspond to column indices of Latin squares. Under such an ordering, note that line $j+10$ corresponds to the jth column of the Latin squares. In what follows the notation $x \leftrightarrow y$ denotes x and y have the same truth value (i.e., x and y are both true or both false), while $x \not\leftrightarrow y$ denotes x and y take opposite truth values. Delisle's RC_CLASS labelling function is now defined by

$$\text{RC\_CLASS}(i,j) = \begin{cases} 0 & \text{if } (i \in R_1 \leftrightarrow j+10 \in R_1) \text{ and } (i \in R_2 \leftrightarrow j+10 \in R_2), \\ 1 & \text{if } (i \in R_1 \leftrightarrow j+10 \in R_1) \text{ and } (i \in R_2 \not\leftrightarrow j+10 \in R_2), \\ 2 & \text{if } (i \in R_1 \not\leftrightarrow j+10 \in R_1) \text{ and } (i \in R_2 \leftrightarrow j+10 \in R_2), \\ 3 & \text{if } (i \in R_1 \not\leftrightarrow j+10 \in R_1) \text{ and } (i \in R_2 \not\leftrightarrow j+10 \in R_2). \end{cases}$$

Similarly, Delisle defines an equivalence class on symbol pairs (s,t) where s is a symbol of the first Latin square A and t is a symbol of the second Latin square B using a labelling function ST_CLASS. Representing s and t as integers in $\{0,\ldots,9\}$, note that line $s+20$ of the net corresponds to symbol s in the first Latin square, and line $t+30$ of the net corresponds to symbol t in the second Latin square. Concretely, $\text{ST\_CLASS}(s,t)$ is defined to be

$$\begin{cases} 0 & \text{if } (s+20 \in R_1 \leftrightarrow t+30 \in R_1) \text{ and } (s+20 \in R_2 \leftrightarrow t+30 \in R_2), \\ 1 & \text{if } (s+20 \in R_1 \leftrightarrow t+30 \in R_1) \text{ and } (s+20 \in R_2 \not\leftrightarrow t+30 \in R_2), \\ 2 & \text{if } (s+20 \in R_1 \not\leftrightarrow t+30 \in R_1) \text{ and } (s+20 \in R_2 \leftrightarrow t+30 \in R_2), \\ 3 & \text{if } (s+20 \in R_1 \not\leftrightarrow t+30 \in R_1) \text{ and } (s+20 \in R_2 \not\leftrightarrow t+30 \in R_2). \end{cases}$$

Delisle then notes that the condition that relations R_1 and R_2 exist in the 4-net(10) corresponding to the orthogonal Latin pair (A,B) is equivalent to the condition

$$\text{RC\_CLASS}(i,j) = \text{ST\_CLASS}(A_{ij}, B_{ij}) \text{ for all } 0 \le i,j \le 9.$$

That is, if $(A_{ij}, B_{ij}) = (s,t)$ then $\text{RC\_CLASS}(i,j) = \text{ST\_CLASS}(s,t)$. We encode this condition directly into our SAT encoding. As we encode each case 1–5 separately, we can assume the forms of R_1 and R_2 are known, and therefore the values of $\text{RC\_CLASS}(i,j)$ and $\text{ST\_CLASS}(s,t)$ are known in advance for all $0 \le i,j,s,t \le 9$. We encode the relation constraint in contrapositive form: for all i, j, s, t with $\text{ST\_CLASS}(s,t) \neq \text{RC\_CLASS}(i,j)$ we enforce that $(A_{ij}, B_{ij}) \neq (s,t)$, i.e., $A_{ij} \neq s$ or $B_{ij} \neq t$. In Boolean logic, this becomes the clauses

$$\neg A_{ijs} \vee \neg B_{ijt} \text{ for all } 0 \le i,j,s,t \le 9 \text{ with } \text{RC\_CLASS}(i,j) \neq \text{ST\_CLASS}(s,t).$$

3.4 Symmetry breaking

In this section we describe how we encode Delisle's symmetry breaking propositions into conjunctive normal form. In particular, we encode properties of minimal pairs of orthogonal Latin squares with two relations described in Section 2.1, as

such constraints do not remove any solutions up to isomorphism. Thus, in this section we assume that (A, B) is a minimal pair of orthogonal Latin squares.

First, consider Proposition 3, which applies in all cases except case 4. It says that $(A_{1,0}, B_{1,0}) < (A_{0,1}, B_{0,1})$. First, we encode the weaker constraint that $A_{1,0} \leq A_{0,1}$ in conjunctive normal form via

$$\bigwedge_{\substack{0 \leq k,l \leq 9 \\ k > l}} (A_{1,0,k} \to \neg A_{0,1,l}).$$

Next, we encode that when $A_{1,0} = A_{0,1}$ we have $B_{1,0} < B_{0,1}$. This is done via

$$\bigwedge_{\substack{0 \leq k,l,m \leq 9 \\ k \geq l}} \big((A_{1,0,m} \wedge A_{0,1,m} \wedge B_{1,0,k}) \to \neg B_{0,1,l}\big).$$

Now consider Proposition 5, which says that the symbols in the first column of A appear in sorted order within the rows in the same row equivalence class. Let R denote a row equivalence class, and let $R' \coloneqq R \setminus \{\max(R)\}$. For example, in cases 1–4, the possible nonempty values for R' are $\{1, 2\}$, $\{4, 5\}$, and $\{7, 8\}$. In case 5, the possible values for R' are $\{0\}$, $\{2\}$, $\{4\}$, and $\{6, 7, 8\}$. In order to ensure that the symbols in the first column of A whose rows are in R we use the constraints

$$\bigwedge_{\substack{i \in R' \\ 0 \leq l < k \leq 9}} (A_{i,0,k} \to \neg A_{i+1,0,l}).$$

Proposition 6 says that the symbols in the first row of A appear in sorted order within the columns in the same column equivalence class and can be handled similarly. Let C denote a column equivalence class and let $C' \coloneqq C \setminus \{\max(C)\}$. We ensure the symbols in the first row of A whose columns are in C using the constraints

$$\bigwedge_{\substack{j \in C' \\ 0 \leq l < k \leq 9}} (A_{0,j,k} \to \neg A_{0,j+1,l}).$$

(In case 4, we skip the clauses from this constraint with $C' = \{0\}$, since Proposition 6 does not apply to column 0 in case 4.)

Proposition 7 says that the symbols in the same A-symbol equivalence class are sorted in the first column of A. Let S denote an A-symbol equivalence class and let $S' \coloneqq S \setminus \{\max(S)\}$. We ensure the symbols of S appear in sorted order in the first column of A using the constraints

$$\bigwedge_{\substack{s \in S' \\ 0 \leq i' < i \leq 9}} (A_{i,0,s} \to \neg A_{i',0,s+1}).$$

Proposition 8 is handled in the same way. Letting T denote a B-symbol equivalence class and $T' \coloneqq T \setminus \{\max(T)\}$, we use the constraints

$$\bigwedge_{\substack{t \in T' \\ 0 \leq i' < i \leq 9}} (B_{i,0,t} \to \neg B_{i',0,t+1}).$$

Finally, we discuss Proposition 4, which applies in all cases except case 2 and says that $A < B$. Since we are not in case 2, we have $A_{0,0} = B_{0,0}$ by Proposition 9. As a result, we start by considering the $(1, 0)$th entries of A and B, noting that $A < B$ implies that $A_{1,0} \leq B_{1,0}$. We encode $A_{1,0} \leq B_{1,0}$ into Boolean logic with the constraints

$$\bigwedge_{\substack{0 \leq k,l \leq 9 \\ k > l}} (A_{1,0,k} \rightarrow \neg B_{1,0,l}).$$

Next, we consider the case when $A_{1,0} = B_{1,0}$. In this case, $A < B$ implies that $A_{2,0} \leq B_{2,0}$, and we encode $A_{1,0} = B_{1,0} \rightarrow A_{2,0} \leq B_{2,0}$ into Boolean logic with the constraints

$$\bigwedge_{\substack{0 \leq k,l,m \leq 9 \\ k > l}} \big((A_{1,0,m} \wedge B_{1,0,m} \wedge A_{2,0,k}) \rightarrow \neg B_{2,0,l}\big).$$

We could continue in this fashion and also encode constraints corresponding to $(A_{1,0} = B_{1,0} \wedge A_{2,0} = B_{2,0}) \rightarrow A_{3,0} \leq B_{3,0}$, etc. However, in practice it was sufficient to only consider the entries to the $(2, 0)$th entry. In other words, we did not encode the full constraint $A < B$ in our SAT instances but the strictly weaker constraint $[A_{0,0}, A_{1,0}, A_{2,0}] \leq [B_{0,0}, B_{1,0}, B_{2,0}]$.

4 Exhaustive Enumeration and Proof Generation

This section explains the process by which we use a SAT solver to find all solutions of a SAT instance (see Section 4.1) and the process by which we generate and check proof certificates that the enumeration was performed correctly, with no missing solutions (see Section 4.2).

4.1 Exhaustive enumeration

Typical modern SAT solvers stop as soon as a satisfying assignment is found and do not support exhaustively enumerating all solutions of a SAT instance. However, the IPASIR-UP interface [11], as supported by the SAT solver CaDiCaL [1], enables us to find all solutions. IPASIR-UP is an interface that can be used to inject custom code into a SAT solver in order to change its behaviour. One function supported by IPASIR-UP is `cb_check_found_model`, a function that is called when the SAT solver has found a new solution. Inside `cb_check_found_model` users are able to add code that modifies a SAT instance whenever a solution is found.

In our case, we add a "blocking clause" into the SAT instance every time a solution is found that prevents the solution from occurring again. Once the blocking clause has been added, the solver continues looking for a new solution. Eventually, once all solutions have been found, the solver reports that the updated instance (i.e., the instance augmented with all blocking clauses) is *unsatisfiable*—it has no solutions.

The blocking clause that we inject into the SAT instance must only block the single solution that was found and no others. Suppose (S, T) is the pair of orthogonal Latin squares that the solver found. We want to add the constraint

$$\neg\Big(\bigwedge_{0\leq i,j\leq 9}(A_{i,j,S_{i,j}} \wedge B_{i,j,T_{i,j}})\Big),$$

which is logically equivalent to $\bigvee_{0\leq i,j\leq 9}(\neg A_{i,j,S_{i,j}} \vee \neg B_{i,j,T_{i,j}})$. Note that this clause contains $2 \cdot 10^2 = 200$ literals. An observation that allows us to shorten this clause is to note that it is sufficient to block only the upper-left 9×9 entries in (S, T), because once those entries have been fixed the remaining entries are forced by the Latin square constraints. This allows us to replace the bound $0 \leq i, j \leq 9$ in the blocking clause with the bound $0 \leq i, j \leq 8$, thereby shrinking the clause to $2 \cdot 9^2 = 162$ literals.

4.2 Proof generation and verification

All our calls to a SAT solver will eventually finish with an unsatisfiable result (i.e., no solutions) as a result of the blocking clauses that we inject into the SAT instance in order to perform an exhaustive search. Modern SAT solvers such as CaDiCaL support generating a "proof certificate" of unsatisfiability. The proof certificate contains a log of the deductions that the solver made in order to determine that the SAT instance has no solutions. The certificate can then be checked by a proof verifier, a separate program that verifies each deduction in the proof logically follows from the previous deductions.

The certificates we generate are based on the DRAT proof format [21]. A DRAT certificate consists of the list of clauses deduced by the solver in the order in which they were deduced. The final clause in an unsatisfiability certificate will be the empty clause. The empty clause being a logical consequence of the original SAT instance proves that the original instance was unsatisfiable, since no truth assignments satisfy the empty clause.

Every step in a DRAT proof is classified as either an *addition*—a clause that can be deduced from previously deduced clauses or clauses in the original SAT instance—or a *deletion*, a clause that was previously added but is no longer needed and should be removed. Our work uses a simple extension of the DRAT format first proposed by Bright et al. [3]. In this extension a third kind of step is supported, a *trusted addition*—a clause that will be added into the list of clauses in the proof even though it cannot necessarily be deduced from the previous clauses in the proof.

We require trusted clauses in our proofs because the blocking clauses we used to perform exhaustive enumeration were added through the IPASIR-UP interface, not deduced by the solver, and therefore cannot be derived through the typical logical deduction process. Thus, in our DRAT proofs when a blocking clause is generated a trusted addition is written into the DRAT proof.

A *proof verifier* takes as input the SAT instance and a DRAT proof of unsatisfiability and verifies every addition step in the proof logically follows from

the current set of derived clauses in conjunction with the clauses in the original SAT instance. Deletion steps remove clauses in the current set of derived clauses when they are no longer needed in order to improve the efficiency of the proof verifier. Trusted addition steps add a clause into the current set of derived clauses without verifying its deducibility.

5 Results

We now discuss our computational results enumerating all 4-nets(10) with at least two nontrivial relations. Our results were run on an Intel i7 CPU running at 2.8 GHz and using the SAT solver CaDiCaL 1.9.4 using up to 250 MiB of memory. Our scripts are freely available at `https://github.com/curtisbright/Delisle-MOLS`.

We use a Python script to generate SAT instances in each of the five possible forms (cases 1–5) following the encoding described in Section 3. In order to mitigate the effect of randomness in the search, each case was independently solved 45 times, each time with a different random seed. The differing random seeds ensure that each instance of CaDiCaL will make different choices during the solving process. A tabular summary of the results of these trials is provided in Table 1, and a box plot of the running times is provided in Figure 1.

The SAT solver determined that cases 2, 3, and 4 all had no solutions and these cases were always solvable in a few seconds. It is interesting to note that Delisle [10] ruled out cases 2 and 4 theoretically using a counting argument, but despite the fact we did not explicitly use this fact in the SAT encoding, the SAT solver quickly proved unsatisfiability on its own. The fact that the solver also quickly ruled out case 3 suggested to us that case 3 was also resolvable using a counting argument, and we were successful in finding one (included at the end of this section). Unfortunately, the proofs produced by the SAT solver, while logically correct, are not intended to be human-readable and the contents of the proofs did not provide us with any mathematical insight.

In case 1 the SAT solver found 3,904 solutions, and in case 5 the SAT solver found 22,320 solutions. The latter count agrees with the count reported by Delisle, but the former count is exactly half of Delisle's count. We contacted Delisle and Myrvold (who ran independent searches for 4-nets(10) with two relations) and

Table 1. A summary of the running times (in seconds) of the 45 instances run in each of the five cases. The minimum DRAT proof size is also provided as well as the number of solutions found in each case.

case	mean	median	minimum	maximum	proof size	solutions
1	5971.2	6116.8	4467.2	7307.3	3.6 GiB	3904
2	0.9	0.9	0.7	1.4	2.1 MiB	0
3	2.1	2.1	1.7	2.6	4.1 MiB	0
4	2.7	2.5	2.0	4.9	5.3 MiB	0
5	1981.2	1965.4	1775.2	2267.8	1.6 GiB	22320

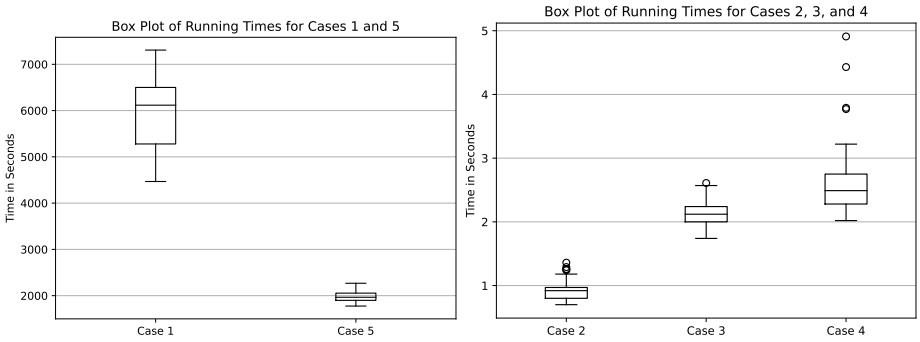

Fig. 1. A box plot visualization of the running times of the 45 instances solved in each case. The left plot shows the results for cases 1 and 5. The right plot shows the results for cases 2, 3, and 4.

their complete enumeration in cases 1 and 5 matched ours exactly, so the count reported by Delisle in case 1 was simply a misprint. We also verified that up to main class equivalence there are exactly 91 solutions (7 in case 1 and 84 in case 5) and that 6 of these (all in case 5) are of rank 34 while the other 85 are of rank 35.

As mentioned in Section 4.2, CaDiCaL was configured to generate DRAT proofs and each time a blocking clause was generated a "trusted addition" clause was added to the proof. Since our proofs in cases 1 and 5 use trusted additions for the blocking clauses, they cannot be verified using a standard DRAT proof checker like DRAT-trim [21]. However, DRAT-trim-t (a fork of DRAT-trim) supports trusted additions, so we verified all our proofs using DRAT-trim-t [2]. The proof size of the shortest proof produced in each case is given in Table 1. The proofs for cases 2–4 were all verified in under a second, while the proof in case 1 was verified in 3.2 hours and the proof in case 5 was verified in 0.9 hours.

The fact that the SAT solver was able to quickly rule out case 3 inspired us to look for a counting argument that could rule out this case. We were successful using an approach similar to the arguments used by Gill and Wanless to count the number of points of certain type in a net [12, Thm. 3.1].

Theorem 2. *There exist no 4-nets(10) with two relations in cases 2–4.*

Proof. Say R_1 and R_2 are two relations in a 4-net(10). In what follows we use the relational code '0' to denote $R_1 \cap R_2$, '1' to denote $R_1 \setminus R_2$, '2' to denote $R_2 \setminus R_1$, and '3' to denote $\overline{R_1} \cap \overline{R_2}$. The *type* of a point is a 4-character $\{0,1,2,3\}$-string denoting the point's relational codes from each parallel class. For example, a point of type 1122 is in R_1 but not R_2 in the first two classes, and is in R_2 but not R_1 in the last two classes. Let t_{ijkl} denote the number of points in the net of type $ijkl$. Note that for R_2 to be a relation it must be the case that $t_{ijkl} = 0$ when $i + j + k + l \not\equiv 0 \pmod 2$, and similarly for R_1 to be a relation it must be the case that $t_{ijkl} = 0$ when $\lfloor i/2 \rfloor + \lfloor j/2 \rfloor + \lfloor k/2 \rfloor + \lfloor l/2 \rfloor \not\equiv 0 \pmod 2$, so there are $4^4/4 = 64$ nonzero t_{ijkl} variables.

Let $[[x_1, y_1, z_1], [x_2, y_2, z_2], [x_3, y_3, z_3], [x_4, y_4, z_4]]$ denote the form of R_1 and R_2 as defined in Section 2. Now count the number of points of type $00{*}{*}$ where the symbols '$*$' are arbitrary. Note there are exactly x_1x_2 points which lie on both a line with index in $[0, x_1)$ and a line with index in $[10, 10 + x_2)$, so

$$\sum_{0 \le k,l \le 3} t_{00kl} = x_1x_2.$$

Similar equations can be derived by counting points of other types. For example, there are exactly x_1x_3 points of type $0{*}0{*}$, exactly y_1y_4 points of type $1{*}{*}1$, and exactly z_1x_2 points of type $20{*}{*}$, resulting in the equations

$$\sum_{0 \le j,l \le 3} t_{0j0l} = x_1x_3, \qquad \sum_{0 \le j,k \le 3} t_{1jk1} = y_1y_4, \qquad \sum_{0 \le k,l \le 3} t_{20kl} = z_1x_2.$$

In case 3, the linear system corresponding to the $3^2\binom{4}{2} = 54$ ways of fixing two entries in the point type to values in $\{0, 1, 2\}$ when converted into reduced row echelon form has a row corresponding to $t_{0123} - t_{3210} = -1/2$ which has no integer solutions.

In cases 2 and 4, the linear systems corresponding to the $4^2\binom{4}{2} = 96$ ways of fixing two entries in the point type to values in $\{0, 1, 2, 3\}$ are both inconsistent over the reals. □

The counting argument in the proof of Theorem 2 provides some theoretical conditions that were speculated on by Delisle in their original work:

> *Interestingly, no pairs of MOLS are completable for case 3. Some theoretical conditions possibly exist to explain this, but are not known at this time.* [10]

6 Conclusion

In this paper we recreated Delisle's 2010 enumeration of all 4-nets(10) with two nontrivial relations. In contrast to Delisle's original search that used a custom-written backtracking program, we use a SAT solver and found that the SAT solver could complete the search over 6000 times faster (when run on modern hardware) than the original backtracking code. This is in part due to improvements in processing power, but it is also due to the powerful search-with-learning algorithms used in modern SAT solvers that can be effective at solving problems in design theory. For example, the author of the SAT solver SATO, H. Zhang, observed SAT solvers are particularly effective at solving Latin square problems:

> *In the earlier stage of our study of Latin square problems, the author wrote two special-purpose programs. After observing that these two programs could not do better than SATO, the author has not written any special-purpose search programs since then.* [23]

Moreover, our results are more trustworthy in the sense that they do not require trusting the implementation of a search algorithm. Instead, we generate certificates that our search was exhaustive, and our results only require trusting the reduction of the problem into Boolean logic (as described in Section 3) and the proof verifier that we use (as described in Section 4.2).

For future work, it would be interesting to use a SAT solver to investigate the results of Gill and Wanless [12] who enumerated all 4-nets(10) with a single nontrivial relation and ruled out the existence of a relation of type $[2, 2, 2, 4, 6]$ in a 5-net(10).

References

1. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: CaDiCaL 2.0, p. 133–152. Springer Nature Switzerland (2024). https://doi.org/10.1007/978-3-031-65627-9_7
2. Bright, C.: The DRAT-trim-t extension of the DRAT-trim checker. https://github.com/curtisbright/drat-trim-t (2025)
3. Bright, C., Cheung, K.K.H., Stevens, B., Kotsireas, I., Ganesh, V.: Nonexistence Certificates for Ovals in a Projective Plane of Order Ten, p. 97–111. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-48966-3_8
4. Bright, C., Cheung, K.K.H., Stevens, B., Kotsireas, I., Ganesh, V.: A SAT-based resolution of Lam's problem. Proceedings of the AAAI Conference on Artificial Intelligence **35**(5), 3669–3676 (May 2021). https://doi.org/10.1609/aaai.v35i5.16483
5. Bright, C., Đoković, D.Ž., Kotsireas, I., Ganesh, V.: A SAT+CAS approach to finding good matrices: New examples and counterexamples. Proceedings of the AAAI Conference on Artificial Intelligence **33**(01), 1435–1442 (Jul 2019). https://doi.org/10.1609/aaai.v33i01.33011435
6. Bright, C., Gerhard, J., Kotsireas, I., Ganesh, V.: Effective Problem Solving Using SAT Solvers, p. 205–219. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-41258-6_15
7. Bright, C., Keita, A., Stevens, B.: Myrvold's results on orthogonal triples of 10×10 Latin squares: A SAT investigation (2025). https://doi.org/10.48550/ARXIV.2503.10504, arXiv:2503.10504
8. Bright, C., Kotsireas, I., Ganesh, V.: When satisfiability solving meets symbolic computation. Communications of the ACM **65**(7), 64–72 (Jun 2022). https://doi.org/10.1145/3500921
9. Bruck, R.H.: Finite nets. II. Uniqueness and imbedding. Pacific Journal of Mathematics **13**(2), 421–457 (Jun 1963). https://doi.org/10.2140/pjm.1963.13.421
10. Delisle, E.: The Search for a Triple of Mutually Orthogonal Latin Squares of Order Ten: Looking Through Pairs of Dimension Thirty-Five and Less. Master's thesis, University of Victoria (2010), http://hdl.handle.net/1828/2964
11. Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., Biere, A.: Satisfiability modulo user propagators. Journal of Artificial Intelligence Research **81**, 989–1017 (Dec 2024). https://doi.org/10.1613/jair.1.16163
12. Gill, M.J., Wanless, I.M.: Pairs of MOLS of order ten satisfying non-trivial relations. Designs, Codes and Cryptography **91**(4), 1293–1313 (Apr 2023). https://doi.org/10.1007/s10623-022-01149-6

13. Howard, L., Myrvold, W.: A counterexample to Moorhouse's conjecture on the rank of nets. Bull. Inst. Combin. Appl **60**, 101–105 (2010)
14. Howard, L.: Nets of Order $4m + 2$: Linear Dependence and Dimensions of Codes. Ph.D. thesis, University of Victoria (2009), http://hdl.handle.net/1828/1566
15. Kaski, P., Östergård, P.R.: Classification Algorithms for Codes and Designs. Springer-Verlag (2006). https://doi.org/10.1007/3-540-28991-7
16. Lam, C.W.H.: Opinion: How reliable is a computer-based proof? The Mathematical Intelligencer **12**(1), 8–12 (Dec 1990). https://doi.org/10.1007/bf03023977
17. Lam, C.W.H., Thiel, L., Swiercz, S.: The non-existence of finite projective planes of order 10. Canadian Journal of Mathematics **41**(6), 1117–1123 (Dec 1989). https://doi.org/10.4153/cjm-1989-049-4
18. Mann, H.B.: The construction of orthogonal Latin squares. The Annals of Mathematical Statistics **13**(4), 418–423 (1942). https://doi.org/10.1214/aoms/1177731539
19. Marques Silva, J., Sakallah, K.: GRASP—A new search algorithm for satisfiability. In: Proceedings of International Conference on Computer Aided Design. p. 220–227. ICCAD-96, IEEE Comput. Soc. Press (1996). https://doi.org/10.1109/iccad.1996.569607
20. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints, p. 827–831. Springer Berlin Heidelberg (2005). https://doi.org/10.1007/11564751_73
21. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs, p. 422–429. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-09284-3_31
22. Zhang, H.: Specifying Latin square problems in propositional logic, pp. 115–146. MIT Press, Cambridge, Massachusetts (1997), https://dl.acm.org/doi/10.5555/271101.271124
23. Zhang, H.: Combinatorial designs by SAT solvers. In: Handbook of Satisfiability. pp. 819–858. IOS Press (Feb 2021). https://doi.org/10.3233/faia201005