2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

Towards a Latin-Square Search Engine

Wenxiu Fang, Rebecca J. Stones, Trent G. Marbach, Gang Wang, Xiaoguang Liu
College of Computer Science, Nankai University, Tianjin, China
Emails: {fangwx, becky, trent.marbach, wgzwp, liuxg}@nbjl.nankai.edu.cn

***Abstract*—Latin squares are combinatorial matrices that are widely used in diverse areas of research such as codes and cryptography, software testing, mathematical research, and experimental designs. All of these fields would benefit from a search engine for Latin squares.**

One major obstacle to developing a Latin-square search engine is that any Latin square has a large number of equivalent Latin squares, which are contained in multiple equivalence classes, and thus we need an efficient online method for canonical labelling Latin squares. Canonical labelling usually proceeds via the Nauty graph isomorphism software, but this incurs conversion costs. Moreover, the canonical labels are practically random members of their equivalence classes. A second obstacle is how large amounts of searchable Latin-square data may be stored efficiently.

In this paper, we design data structures and algorithms suitable for a Latin-square search engine. We use a tree-based data structure for storing large numbers of Latin squares that also enables efficient search capabilities. We design an efficient canonical labelling algorithm (via partial Latin squares, PLSs) which does not require graph conversion, facilitates compression, and the labels are more humanly meaningful. We implement and experiment with a skeletal prototype of the Latin-square search engine. Experimental results confirm that the PLS method is faster than Nauty, and has reduced space requirements.

***Index Terms*—Latin square, partial Latin square, search engine, storage, information retrieval, mathematical knowledge management**

I. Introduction

The curation of large datasets of mathematical information [1], [2] is generally understudied [3]. In this paper, we focus on *Latin squares*, which are $n \times n$ matrices containing n symbols so that every symbol occurs exactly once in each row and each column. Latin squares are used in multiple domains of science (Section III). However, Latin-square data is only available as bulk downloads via academic webpages, e.g., by Prof. Brendan McKay[1] or Prof. Ian Wanless[2] (prominent Latin-square researchers). While valuable, utilizing this data usually requires human time to write purpose-built code to process the datasets.

In contrast, we have the Online Encyclopedia of Integer Sequences (OEIS; http://oeis.org/). Unlike academic webpages, anyone can update the OEIS to add new integer sequences, references, formulas, and so on. Researchers often use the OEIS to "advertise" their papers by adding references to the relevant sequences, which facilitates researchers discovering relevant references. (For graph data there is the House of Graphs [4] and the Online Graph Atlas [5].)

Motivated by the success of the OEIS, we consider a comparable search engine for storing and retrieving Latin squares along with relevant mathematical information pertaining to each Latin square. However, the number of Latin squares grows super-exponentially, so we require efficiency in a Latin-square search engine for it to be useful and practical.

To create an effective search engine for such a large number of objects, we must optimize how the objects are stored. If we attempted to store each Latin square of a given size n, we could only perform this storage for $n \leq 6$, which is both useless and redundant: many Latin squares are equivalent in a mathematical sense. To alleviate this problem to a certain extent, we only store inequivalent Latin squares. The broadest commonly studied equivalence is paratopism (formalized in Section II). We list the number of unique Latin squares vs. those up to paratopism in Table I (also see [6]); by only storing one representative for each paratopism class of Latin squares, we could conceivably represent all $n \times n$ Latin squares in storage for $n \leq 9$.

The major contributions of this paper are as follows:

- We design data structures for storing Latin-square data, and a matching search-engine architecture.
- To search among inequivalent Latin squares, we develop a novel Latin-square canonical labelling method, involving partial Latin squares.
- We experimentally compare the canonical labelling method with the normal method of using Nauty in terms of size and run times.

The rest of this paper is structured as follows. We describe the mathematical background and how Latin squares are used throughout science in Section II and Section III, respectively. We describe the architecture and data structure of the proposed Latin-square search engine, the canonical labelling method based on partial Latin squares (PLSs), and compression in Section IV. We present the results of run time and compression experiments in Section V. We conclude this paper with some ideas for future work in Section VI.

II. Mathematical background

An $n \times n$ Latin square is said to have *order* n. We can define an equivalence relation among Latin squares via various

Supported by NSF of China (61602266, 61872201), the Science and Technology Development Plan of Tianjin (17JCYBJC15300, 16JCYBJC41900), and the Fundamental Research Funds for the Central Universities and SAFEA: Overseas Young Talents in Cultural and Educational Sector. Stones is also supported by the Thousand Youth Talents Plan in Tianjin.

[1]https://users.cecs.anu.edu.au/~bdm/data/latin.html

[2]http://users.monash.edu.au/~iwanless/data/index.html

978-1-7281-4328-6/19/$31.00 ©2019 IEEE
DOI 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00110

TABLE I
THE NUMBER OF LATIN SQUARES VS. THE NUMBER OF PARATOPISM CLASSES OF ORDER n FOR $n \leq 11$.

n	No. Latin squares of order n	Reference	No. paratopism classes of order n	Reference
1	1		1	
2	2		1	
3	12		1	
4	576		2	
5	161280		2	
6	812851200		12	
7	61479419904000		147	
8	108776032459082956800		283657	
9	5524751496156892842531225600		19270853541	
10	9982437658213039871725064756920320000	McKay and Rogoyski [7]	34817397894749939	McKay et al. [8]
11	776966836171770144107444346734230682311065600000	McKay and Wanless [9]	2036029552582883134196099	Hulpke et al. [10]

transformations; we focus on two important and commonly used equivalences. Throughout this paper, we use $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ as the set of symbols, and use $\mathbb{Z}_n$ to also index the rows and columns of a Latin square.

Isotopism For a Latin square of order n, an *isotopism* is any triple of permutations of $\mathbb{Z}_n$. So, for example,

$$\theta = ((01)(2), (0)(1)(2), (012)) \tag{1}$$

is an isotopism where $n = 3$. We use an isotopism to act on a Latin square L: if $\theta = (\alpha, \beta, \gamma)$, we:

- permute the rows of L according to α,
- permute the columns of L according to β, and
- permute the symbols of L according to γ.

Equivalence classes under isotopisms are called *isotopism classes*, and two Latin squares belonging to the same isotopism class are called *isotopic*.

Often it is easier to think of Latin squares as sets of entries; an *entry* of a Latin square $L = (l_{ij})$ is a triple (i, j, l_{ij}) where $i, j \in \mathbb{Z}_n$. For example, the Latin square

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2}$$

has four entries tabulated below:

row index	column index	symbol index	entry
0	0	0	$(0,0,0)$
0	1	1	$(0,1,1)$
1	0	1	$(1,0,1)$
1	1	0	$(1,1,0)$

.

So the Latin square in (2) is equivalent to the entry set

$$\{(0,0,0), (0,1,1), (1,0,1), (1,1,0)\}.$$

In this way, if we apply an isotopism $\theta = (\alpha, \beta, \gamma)$ to a Latin square $L = (l_{ij})$ the result is precisely the Latin square with the entry set

$$\{(\alpha(i), \beta(j), \gamma(l_{ij})) \colon i, j \in \mathbb{Z}_n\}.$$

For example, we apply (1) to a Latin square below:

$$\begin{bmatrix} 0 & 1 & 2 \\ 2 & 0 & 1 \\ 1 & 2 & 0 \end{bmatrix} \xrightarrow{\theta} \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix}.$$

Paratopism *Paratopism* generalizes the notion of isotopism: we write $\theta = (\alpha, \beta, \gamma; \delta)$ where δ, which is called the *parastrophe*, is a permutation of $\mathbb{Z}_3$. We analogously define the terms *paratopism class* and *paratopic*. We first apply the isotopism (α, β, γ), then apply the parastrophe δ. To apply a parastrophe, every entry (e_0, e_1, e_2) in the Latin square is transformed into the entry $(e_{\delta(0)}, e_{\delta(1)}, e_{\delta(2)})$. An example to show how to apply a parastrophe is given below:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{array}{c} \text{entries} \\ \hline (0,0,0) \\ (0,1,1) \\ (1,0,1) \\ (1,1,0) \end{array} \quad \xrightarrow{\delta=(021)} \quad \begin{array}{c} \text{entries} \\ \hline (0,0,0) \\ (1,1,0) \\ (0,1,1) \\ (1,0,1) \end{array} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

We emphasize that the above example only illustrates the parastrophe component δ, and not a full paratopism. We reiterate that when applying the paratopism $\theta = (\alpha, \beta, \gamma; \delta)$, we first apply the isotopism (α, β, γ), then the parastrophe δ; we obtain a different result if we do it in the other order.

There are multiple possible paratopic forms for any given Latin square, and it is unlikely that a user's input is an exact match in a database of Latin squares. To facilitate the search, we propose storing a *canonical label* (i.e., a unique representative) from each class, and when searching, we begin by searching for the paratopism-class canonical label.

III. HOW LATIN SQUARES ARE USED

To get an idea of how a Latin-square search engine would be useful and to understand the target audience of such a system, we give a broad survey of how Latin squares themselves are used (i.e., their major applications).

A. Experimental designs

The most common application of Latin squares is in experimental design. To illustrate how a Latin-square experimental design might be used, suppose we are testing the performance of software where we have four distinct hard disks A, B, C, and D, and we want to investigate the impact of "cache size" and "preprocessing time". We might test these factors together by using a Latin square design, such as

		preprocessing time			
		5 sec	10 sec	20 sec	40 sec
cache size	1GB	A	B	C	D
	2GB	C	D	A	B
	4GB	B	C	D	A
	8GB	D	A	B	C

Instead of implementing an experiment for each of the 64 combinations, we only implement 16 experiments. After which, we use analysis of variance (ANOVA) techniques to measure statistical significance; see e.g., [11, Ch. 9].

Experimental designs involving Latin squares are not limited to extremely small Latin squares. Latin squares of order 8 were used in studying how to best assist dyslexia patients [12], order 10 in experiments for optimizing fly traps [13], order 12 in an experiment analyzing the negative impact of social media on television news [13], and order 14 in the "Human Genome U95" dataset. However, most experimental designs use Latin squares of orders between 3 and 9, so the focus of a Latin-square search engine should be on smaller Latin squares.

Special types of Latin squares are sometimes specifically chosen for experimental designs. We can sample Latin squares randomly [14], but this does not give a mechanism for generating Latin squares with specific desirable properties, as would be possible in a Latin-square search engine.

B. Latin squares in mathematical research

Mathematicians study Latin squares both in their own right, and for their applications in a range of areas in mathematics. Latin squares can be interpreted as adjacency matrices of edge-colored graphs, and thus are useful throughout graph theory, especially in regards to graph-decomposition problems and block designs. Latin squares also give rise to Latin square graphs, and thus we can think about the graph-theoretic properties of Latin squares, such as its chromatic number [15].

C. Software testing

The seminal paper in applying combinatorial designs to software testing was by Mandl [16], which specialized in applying orthogonal Latin squares for compiler testing. The entry set of a Latin square gives rise to an experimental test suite, e.g.:

entries		test suite	
$(0,0,0)$		test 1:	red, thin, dotted
$(0,1,1)$	$\longleftrightarrow$	test 2:	red, thick, dashed
$(1,0,1)$		test 3:	blue, thin, dashed
$(1,1,0)$		test 4:	blue, thick, dotted

In this toy example, we have three properties: color (red or blue), thickness (thin or thick), and pattern (dotted or dashed). Instead of testing all $2^3 = 8$ possible triples, we only test 4 triples and we inspect every possible 2-way interaction (e.g., "red and thick" or "thin and dashed").

D. Latin squares in codes and cryptography

Special Latin squares are used in designing efficient erasure codes. The underlying structure of many array codes (XOR-based erasure codes) is essentially a Latin square: e.g., X-code [17], RDP codes [18], and K-PLEX codes [19], [20].

There is also much research interest into orthogonal Latin square codes, a kind of multiple-error correcting code [21]. In such codes, we convert a set of orthogonal Latin squares to a parity check matrix. Much research into orthogonal Latin square codes is about adapting and applying them to various research problems; e.g., generating burst-error correction codes [22], extending the data block size in orthogonal Latin square codes [23], and as a double-error correction code [24].

In cryptography, Shannon, in his fundamental paper [25], describes how Latin squares model a "perfect system" with equal numbers of cryptograms, messages, and keys. The two-dimension nature of Latin squares makes them suitable for image encryption. For example, Xu and Tian [26] used self-orthogonal Latin squares as a kind of two-dimensional permutation. Latin squares are also useful for edge detection in images [27]. Quasigroups (algebraically equivalent to Latin squares) are used in cryptography, such as in extended Feistel ciphers [28], [29] and other block and stream ciphers [30]–[32]. They are noted for their strength and simplicity [33].

E. Other applications of Latin squares

There are also sporadic applications of Latin squares, such as cryptographic hash functions [34], geographical data browsing and retrieval [35], message routing [36], and in biochemistry [37, for example].

IV. A prototype Latin-square search engine

A. Architecture and work flow

The envisaged Latin-square search engine architecture is depicted in Fig. 1. It has two indexes, which we call the *LS index* and the *property index*. It also has an *LS database* containing rooted trees, called *LS-trees*, where each node of an LS-tree is a Latin square belonging to the same paratopism class, and the root node is the paratopism-class canonical label. We envisage two search types:

- *Latin-square query.* The user inputs a Latin square, and we retrieve it or an equivalent Latin square.
- *Latin-square property query.* The user inputs some Latin square properties, such as requesting "Latin squares of order 7 with no subsquares", and we retrieve all Latin squares in the database with all of the inputted properties.

When a user inputs a query, the query is analyzed to determine its query type. For a Latin-square query, we search the B+Tree of the LS index to find the LS-tree containing the paratopism-class canonical label of the input Latin square as its root node. If found, it searches within the LS-tree (see Section IV-B). We return the hits and pre-computed data (or properties) concerning the Latin square. For a Latin-square property query, we search the property index and we return the Latin squares which have these properties.

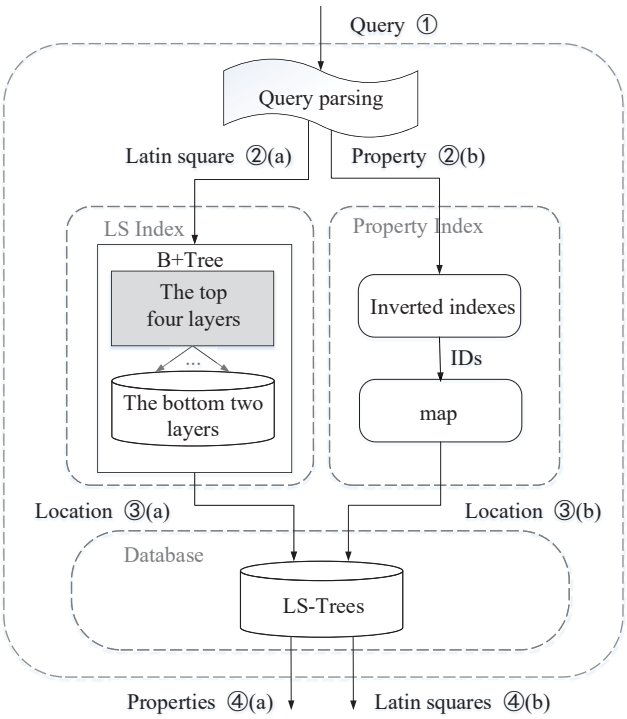

Fig. 1. Architecture and work flow of a Latin-square search engine. If the query is a Latin square, it follows the path numbered 1, 2(a), 3(a), and 4(a), otherwise it follows the path numbered 1, 2(b), 3(b), and 4(b).

B. Data structures

The proposed search engine uses two index structures for the two types of searches. Considering the characteristics of Latin squares, we propose a tree-based data structure to store canonical labels along with their properties.

Each B+Tree node corresponds to the MD5 hash of the paratopism-class canonical label and is stored as a 4KB block. Fig. 2 depicts the structure of this block, where h_i denotes a hash value and p_i denotes a pointer. The head of each node contains some metadata e.g., the node type, the number of children of this node, and the locations of itself and its parent node, and so on. This is followed by the MD5 hashes of 101 paratopism-class canonical labels (up to order 9) along with 102 corresponding pointers. A B+Tree with 6 layers can store the hashes of all paratopism-class canonical labels up to order 9. To improve the query response time, we load the first four layers of the B+Tree into memory.

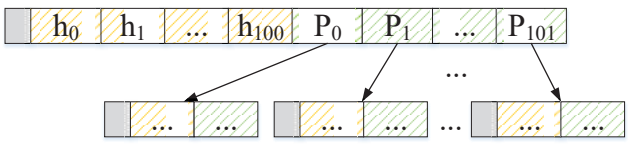

Fig. 2. Data structure of the nodes in the first two layers of the 6-layer B+Tree.

We propose using a tree-based data structure, LS-Tree, for the database. The database contains one LS-Tree for each paratopism class, and is used for storing four types of internal Latin squares:

1) a paratopism-class canonical label,
2) isotopism-class canonical labels,
3) the raw user input, and
4) normalized Latin squares (i.e., the first row is in order).

Fig. 3 depicts the LS-tree storage layers. All of the LS-Trees are stored on disk. Further, each Latin square that is an equivalence-class canonical label is compressed (compression is described in Section IV-E). Since paratopism-class equivalence is the broadest form of equivalence, we perform a 2-step search, i.e., given a Latin square query for L:

- The B+Tree LS index is searched to find the canonical label that is paratopic to L.
- If there is a hit, it searches within the associated LS-Tree for remaining relevant data.

The leaf nodes of the non-clustered B+Tree (see e.g., [38]) store the location of each root node of the LS-Tree, and so is used in the search for paratopism-class canonical labels. The LS-tree data structure, however, is used for searching within paratopism classes.

Many computational results are invariant up to some form of equivalence, e.g., the number of transversals is the same for paratopic Latin squares. We give examples of properties which are stored at each level in Fig. 3. By storing Latin-square data at the appropriate level, we reduce storage costs.

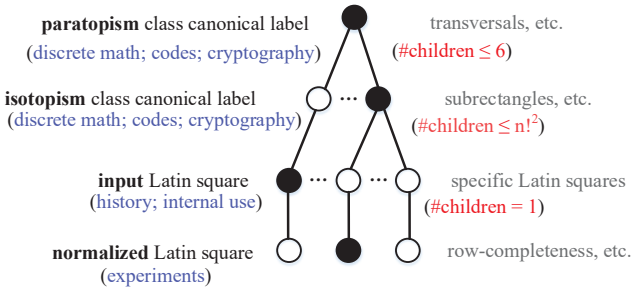

Fig. 3. A LS-tree for storing Latin squares belonging to a single paratopism class. Black nodes are those where we store the Latin square, whereas for white nodes, the Latin square happens to be equal to the Latin square at a higher level, so we do not re-store those Latin squares. In blue we list some areas we envisage this data level to have applications. On the right, we also include an example of what metadata we might store at each level.

For a property, e.g., "has exactly k transverals", we generate a posting list for that property,

$$\ell_k = \langle \mathrm{id}_0, \mathrm{id}_1, \ldots \rangle ,$$

where id_i is the index of the i-th Latin square with exactly k transversals. When a user inputs a Latin-square property query, we intersect the relevant posting lists. For example, the user might input something equivalent to

$$\text{num\_subsquares}_{2\times 2} = 0 \quad \text{AND} \quad \text{num\_transversals} = 1$$

to search for Latin squares with no 2×2 subsquares and exactly one transversal. The proposed search engine fetches and intersects the two corresponding posting lists.

To facilitate the search, we add a *map* between the inverted indexes and the LS database (also depicted in Fig. 1). Each pair in this map has the structure $\langle l_{\text{id}}, [f_{\text{id}}, \sigma] \rangle$, where l_{id} is the index of a Latin square, the array $[f_{\text{id}}, \sigma]$ indicates the index of a storage file in an LS-tree and an offset in this file, respectively, e.g., $\langle 0, [0, 4] \rangle$ implies that the Latin square with index 0 starts at the 4-th byte of the 0-th storage file (we also use some bytes to store version information, and so on, at the beginning of each file). Thus, the search engine can rapidly access the Latin squares according to the map.

C. Partial-Latin-square canonical labels

When a user inputs a Latin-square query, the search engine first converts it to its canonical form, and uses its canonical form to search in the Latin square database. Currently, the way to perform canonical labels of Latin squares is through Nauty [39]. However there are drawbacks to using Nauty: Nauty's canonical labels are not humanly interpretable, and using Nauty incurs computational overhead in converting Latin squares to and from graphs. Moreover, compression would benefit from canonical labels which are similar to one another.

If we take Nauty's original canonical labels, we reduce the resulting Latin square by the following process:

1) we relabel the symbols, so the first row becomes $(0, 1, \ldots, n-1)$, then
2) we reorder the rows $1, \ldots, n-1$ (leaving row 0 unchanged) so the first column becomes $(0, 1, \ldots, n-1)^T$.

Two example paratopism-class canonical labels arising this way are the following:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 4 & 2 & 5 & 3 \\ 2 & 3 & 5 & 4 & 1 & 0 \\ 3 & 5 & 0 & 1 & 2 & 4 \\ 4 & 2 & 3 & 5 & 0 & 1 \\ 5 & 4 & 1 & 0 & 3 & 2 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 3 & 5 & 0 & 2 \\ 2 & 0 & 5 & 4 & 3 & 1 \\ 3 & 2 & 0 & 1 & 5 & 4 \\ 4 & 5 & 1 & 0 & 2 & 3 \\ 5 & 3 & 4 & 2 & 1 & 0 \end{bmatrix}. \tag{3}$$

We highlight where the two Latin squares agree, and observe that there is not much agreement. Thus, we design a new method which results in greater agreement between different canonical labels.

We decompose any Latin square of order n into $n-1$ partial Latin squares: for each $N \in \{1, 2, \ldots, n-1\}$, we only keep the entries $(i, j, L[i, j])$ which satisfy $i \leq N$, and $j \leq N$, and $L[i, j] \leq N$. For cells $(i, j, L[i, j])$ with $i \leq N$ and $j \leq N$ and $L[i, j] > N$, we replace $L[i, j]$ with ∞. We decompose two Latin squares in this way in Fig. 4. All the infinities are the same, i.e., $\infty = \infty$; we also have $\infty > s$ for all finite s.

We compare the PLS sequences lexicographically: for the smallest pair of PLSs which disagree, the first is lexicographically smaller, thus we consider the first Latin square to be "less than" the second. In this way, we define the *PLS total ordering*. We define the *isotopism-class PLS canonical label* as the minimum Latin square in an isotopism class under the PLS total ordering. We define the *paratopism-class PLS canonical label* of a Latin square L as lexicographically minimum isotopism-class PLS canonical label of the six (not necessarily distinct) parastrophes of L.

The following Latin squares are the paratopism-class PLS canonical labels of the Latin squares in (3):

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 2 & 5 & 4 \\ 2 & 4 & 0 & 5 & 3 & 1 \\ 3 & 2 & 5 & 4 & 1 & 0 \\ 4 & 5 & 1 & 0 & 2 & 3 \\ 5 & 3 & 4 & 1 & 0 & 2 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 2 & 5 & 4 \\ 2 & 4 & 0 & 5 & 3 & 1 \\ 3 & 5 & 4 & 0 & 1 & 2 \\ 4 & 2 & 5 & 1 & 0 & 3 \\ 5 & 3 & 1 & 4 & 2 & 0 \end{bmatrix}$$

These canonical labels agree more than Nauty's labels (highlighted in green). Moreover, these canonical labels are also more humanly understandable.

D. Computing PLS canonical labels

The algorithm for computing PLS canonical labels is based on the following lemma.

Lemma 1. *If $\theta = (\alpha, \beta, \gamma)$ is an isotopism for which $\theta(L)$ is reduced, then θ is determined from α and the column y for which $\beta(y) = 0$.*

Proof. The column y in L maps to the column 0 in $\theta(L)$, and since $\theta(L)$ is reduced, the first column of $\theta(L)$ is $(0, 1, \ldots, n-1)^T$. Writing this mathematically, we have

$$\begin{aligned} &\{(\alpha(i), \beta(y), \gamma(L[i, y])) \colon i \in \mathbb{Z}_n\} \\ &= \{(\alpha(i), 0, \gamma(L[i, y])) \colon i \in \mathbb{Z}_n\} \\ &= \{(i, 0, i) \colon i \in \mathbb{Z}_n\}. \end{aligned}$$

Thus $\gamma(L[i, y]) = \alpha(i)$ for all $i \in \mathbb{Z}_n$, which implies that α and y determine γ. Let $x \in \mathbb{Z}_n$ satisfy $\alpha(x) = 0$. Hence

$$\begin{aligned} &\{(\alpha(x), \beta(j), \gamma(L[x, j])) \colon j \in \mathbb{Z}_n\} \\ &= \{(0, \beta(j), \gamma(L[x, j])) \colon j \in \mathbb{Z}_n\} \\ &= \{(0, j, j) \colon j \in \mathbb{Z}_n\}. \end{aligned}$$

Thus $\beta(j) = \gamma(L[x, j])$ for all $j \in \mathbb{Z}_n$, which implies that γ and x (which are determined from α and y) determines β. □

We turn Lemma 1 into an algorithm for generating all reduced Latin squares in an isotopism class, which is presented in Algorithm 1. Algorithm 1 can be used in conjunction with the PLS-based canonical labels, which allows early termination when using a backtracking algorithm for generating the permutations α (Line 3 in Algorithm 1).

Algorithm 2 is the backtracking algorithm used for iterating through the permutations α in $\theta = (\alpha, \beta, \gamma)$. If we omit CHECKCANONICAL in Line 10 then Algorithm 1 iterates through all $n\,n!$ reduced Latin squares isotopic to L. We include CHECKCANONICAL (as defined by Algorithm 3) for early termination: in the p-th iteration, we compare the $N = p$ PLS for $\theta(L)$ vs. $\theta_{\text{best}}(L)$, and do not follow the branch if $\theta(L)$ is worse than $\theta_{\text{best}}(L)$. Importantly, θ is incompletely defined, and thus $\theta(L)$ is incompletely defined; but the PLS is completely defined. When CHECKCANONICAL returns `FALSE`,

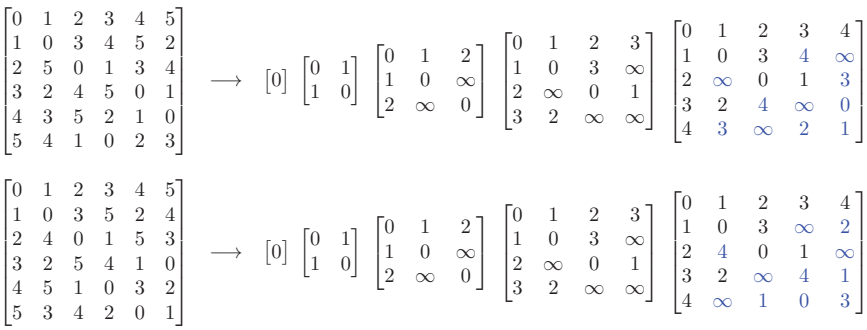

Fig. 4. Two Latin squares of order 6 decomposing into partial Latin squares. We highlight disagreement (in blue) in the decomposition.

Algorithm 1 Iterating through all reduced Latin squares in an isotopism class

Require: A Latin square L of order n
Ensure: The set of reduced Latin squares isotopic to L
1: **for** x from 0 to $n-1$ **do** ▷ $\alpha(x) = 0$
2: **for** y from 0 to $n-1$ **do** ▷ $\beta(y) = 0$
3: **for all** permutations α for which $\alpha(x) = 0$ **do**
4: Compute β and γ according to Lemma 1 (from α and y)
5: Compute the reduced Latin square $\theta(L)$, where $\theta = (\alpha, \beta, \gamma)$

every completion of θ results in $\theta(L)$ being worse than $\theta_{\text{best}}(L)$, so we skip the whole branch. (This is why we choose the non-obvious PLS definition for a canonical label.)

Algorithm 2 Backtracking algorithm for the loop in Line 3 in Algorithm 1

Require: A Latin square L of order n, a cell (x, y), the current position p (where $p \geq 1$)
1: **function** AlphaBacktracking(p, x, y)
2: Set σ_x as the permutation defined by row x ▷ $\sigma_x(j) = L[x, j]$ for all $j \in \mathbb{Z}_n$
3: Set σ_y as the permutation defined by column y ▷ $\sigma_y(i) = L[i, y]$ for all $i \in \mathbb{Z}_n$
4: **for** i from 0 to $n-1$ **do**
5: **if** $\alpha(i)$ is already defined **then**
6: **continue** ▷ clash: i already maps to something else
7: $\alpha(i) \leftarrow p$ ▷ determines $\alpha(*) = p$
8: $\gamma(\sigma_y(i)) \leftarrow p$ ▷ determines $\gamma(*) = p$
9: $\beta(\sigma_x^{-1}(\sigma_y(i))) \leftarrow p$ ▷ determines $\beta(*) = p$
10: **if** CheckCanonical(p) returns TRUE **then** ▷ terminate branches containing no canonical label
11: **if** $p = n-1$ **then**
12: $\theta_{\text{best}} \leftarrow \theta$ where $\theta = (\alpha, \beta, \gamma)$
13: **else**
14: AlphaBacktracking($p+1, x, y$)

E. Compression

Although the number of paratopism classes is significantly less than the number of Latin squares of each order, it is still challenging to store them all. For example, there are 19270853541 paratopism classes of 9×9 Latin squares. Each entry in each Latin square can be stored in around 4 bits with a suitable encoding. So the storage cost we expect is around

$$4\,\text{bits} \times 81\,\text{entries} \times 19270853541\,\text{Latin squares} \simeq 780\text{GB}.$$

Making this data practically searchable is a long-term goal of this research, extending beyond the scope of this paper. As a first step, we consider a method for compressing the data.

The rapidly increasing number of equivalence classes makes it impractical to store all the uncompressed canonical labels. Fortunately, benefiting from the PLS method, we are able to compress Latin squares according to their similarity. Consider the following four example paratopism-class PLS canonical labels of order 8 (each written in a single line; omitting the first row which is always $(0, 1, \ldots, 7)$):

10325476230167453210765445670123 5476	10326745230176543210
10325476230167453210765445670123 5476	10326745231076543201
10325476230167453210765445670123 5476	23016745321076541032
10325476230167453210765445670123 5476	23106754103276453201

We shade some recurring 4-tuples of symbols, and we use a vertical line to indicate the common prefix.

We observe that lines share a substantial similarity, which can be utilized to improve compression. Not only do they have a common prefix but also a lot of symbol combinations in common, e.g., the symbol combination 1032 exists in all four Latin squares. In addition, some symbol combinations appear more than once within a single Latin square. By trial-and-error in the even case, we found that when we cut each row of the Latin squares in the middle, the symbol combinations of each half row typically have a high similarity.

When the order of the Latin squares is odd, we omit the first column in order to split into half-rows. We do not lose any information in doing so, as the first column does not require storage due to the fact that the canonical labels generated by the PLS method are reduced (i.e., the first row is $(0, 1, \ldots, n-1)$ and the first column is $(0, 1, \ldots, n-1)^T$). We enumerate half-row symbol combinations and compress-build a Huffman code for PLS canonical labels of the same order.

Algorithm 3 We compare the PLSs of orders $1, \ldots, p_{\max}$ for $\theta(L)$ vs. $\theta_{\text{best}}(L)$

Require: A Latin square L of order n, a cell (x, y), the current partial isotopism $\theta = (\alpha, \beta, \gamma)$, current best isotopism $\theta_{\text{best}} = (\alpha_{\text{best}}, \beta_{\text{best}}, \gamma_{\text{best}})$

```
1:  function CHECKCANONICAL(p_max)
2:      for p from 0 to p_max do
3:          for i from 0 to p do
4:              for j from 0 to p do
5:                  s ← γ_best(L[α_best^{-1}(i), β_best^{-1}(j)])      ▷ θ_best(L) has symbol s in cell (i, j)
6:                  k ← L[α^{-1}(i), β^{-1}(j)]   ▷ θ maps symbol k to a symbol in cell (i, j); at this stage, we may have not
                                                    yet decided what γ(k) is, but we have defined α^{-1}(i) and β^{-1}(j) since
                                                    i ≤ p and j ≤ p
7:                  if s > p then                  ▷ θ_best(L) has ∞ in cell (i, j) in its order-p PLS
8:                      if γ(k) is defined and γ(k) ≤ p then
9:                          return TRUE            ▷ θ(L) (for any completion of θ) is a better canonical label than θ_best(L)
                                                    (case: finite < ∞)
10:                     continue                   ▷ θ(L) also has ∞ in cell (i, j) in its PLS (case: ∞ = ∞)
                                                   ▷ now we have s ≤ p, so s < ∞
11:                 if γ(k) is undefined then
12:                     return FALSE               ▷ θ(L) (for any completion of θ) is a worse canonical label than θ_best(L)
                                                    (case: ∞ > finite)
                                                   ▷ now γ(k) is defined
13:                 if γ(k) < s then
14:                     return TRUE                ▷ θ(L) (for any completion of θ) is a better canonical label than θ_best(L)
                                                    (case: finite < finite)
15:                 if γ(k) > s then
16:                     return FALSE               ▷ θ(L) (for any completion of θ) is a worse canonical label than θ_best(L)
                                                    (case: finite > finite)
17:                 continue                       ▷ θ(L) also has s in cell (i, j) in its PLS (case: finite = finite)
18:     return TRUE                                ▷ θ(L) and θ_best(L) have the same PLSs
```

We expect compression could be improved by utilizing prefixes of PLS canonical labels through a prefix tree, but a prefix tree needs additional metadata such as pointers. This results in a trade-off between search speed and storage space, so improving storage via prefixes remains to be explored.

V. Experimental results

We carry out the experiments on a PC server which has two CPUs (Intel Xeon E5-2650 v4) and 512GBs of memory, running CentOS 7.4.1708. Each CPU has 12 physical cores, and is clocked at 1200MHz. The L1 data cache and instruction cache is 32KB, L2 and L3 is 256KB and 30,720KB, respectively. The run time and compression experiments are implemented in C++ and are compiled with g++ version 7.2.0, with optimization flag -O3.

We use some random Latin squares and Latin square representatives on which we perform the experiments. Details of the dataset are provided in Table II.

TABLE II
Experimental dataset summary.

experiment	type	order n	quantity
run time	random LSs	$7 \le n \le 13$	10^6 for each n
compress.	paratopism class	$n = 8$	283657
	isotopism class	$n = 8$	1676267
	random LSs	$9 \le n \le 13$	10^6 for each n
	random LSs	$9 \le n \le 13$	10^7 for each n

A. Run time

Nauty is a highly optimized program for computing automorphism groups of graphs and the most common method to produce canonical labels. In this section, we compare the run times of generating canonical labels using the proposed PLS method vs. Nauty as the baseline. We generate one million random Latin squares of each order n with $7 \le n \le 13$ and compute equivalence-class canonical labels using these two methods for each of these Latin squares.

Fig. 5 plots the experimental run times using the PLS method vs. Nauty to compute isotopism-class canonical labels, in which the data is presented as a box plot and the median is circled. We observe that the run times are concentrated within a small box, with some particularly slow outliers for larger orders (12 and 13). The PLS method outperforms Nauty, having $178\times$ median speedup over Nauty for order 8. This is unsurprising, given the overhead in converting the Latin square to Nauty's internal format, and converting the canonical

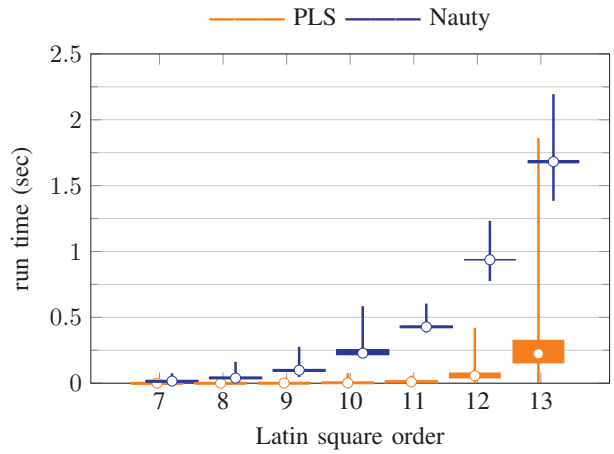

Fig. 5. Experimental run times using PLS vs. Nauty to compute isotopism-class canonical labels.

label back from Nauty's internal format. The run time for smaller orders is negligible for the PLS method. For very large orders, Nauty typically runs faster than the PLS method, since there are $n\,n!$ (not necessarily distinct) reduced Latin squares isotopic to a Latin square of order n. However, we focus on small Latin squares in the context of a search engine.

For paratopism-class canonical labels, we compute the 6 conjugates of each isotopism-class canonical label generated by PLS and take the lexicographical minimum of these 6 conjugates as the paratopism-class canonical label. The PLS method still has an advantage over Nauty for each case within our experimental orders. We envisage in a practical Latin-square search engine, these 6 computations would be performed in parallel, and thus there would be no practical difference between isotopism-class and paratopism-class canonical-label run times.

B. Compression

We compare the storage requirements of canonical labels generated by PLS and Nauty. We download the 283657 paratopism-class representatives and the 1676267 isotopism-class representatives of order 8 from Prof. McKay's website and compute canonical labels using both the PLS method and Nauty. Table III lists the compressed size of these two collections; we also include the downloadable gzip file size for comparison (actually using gzip is unrealistic). We observe 15% to 17% reduction in size for the PLS method vs. Nauty, and 6% to 14% reduction in size vs. gzip.

We also generate random Latin squares of orders 9 through 13 to mimic users adding Latin squares to the database. For compression, we count the duplicates of symbol combinations in canonical labels generated by these two methods and create a Huffman code for them. We store the Huffman code in ASCII format. Fig. 6 plots the compressed size of the isotopism-class canonical labels. It turns out both PLS and Nauty are space-efficient, although PLS uses around 10% less space than Nauty

TABLE III
COMPRESSED SIZE (MB) OF ALL PARATOPISM-CLASS AND ISOTOPISM-CLASS CANONICAL LABELS OF ORDER 8.

method	paratopism	isotopism
Nauty	5.3	31.5
PLS	4.4	26.8
gzip	5.1	28.5

for each experimental order, which remains consistent as we increase the number of random Latin squares.

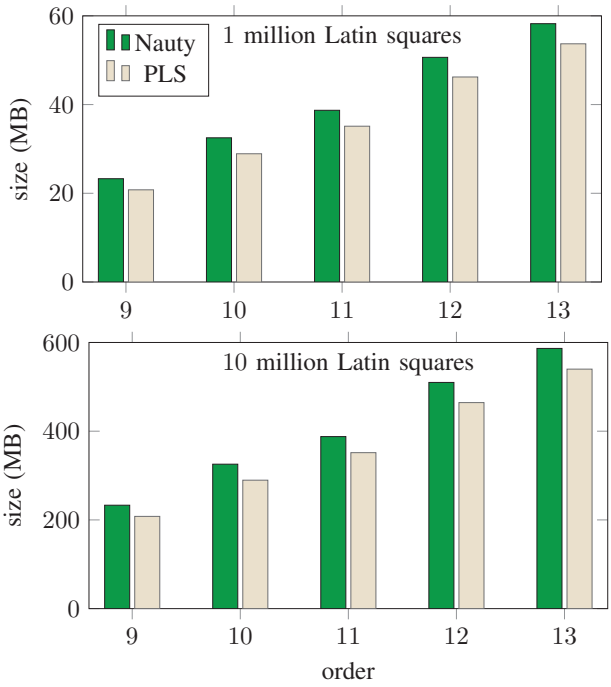

Fig. 6. Compressed size of the isotopism-class canonical labels generated by PLS and Nauty for 1 million (top) or 10 million (bottom) random Latin squares per order.

VI. CONCLUDING REMARKS

In this paper, we take the first step towards a functional Latin-square search engine where we envisage users searching for either a Latin square or property values. We introduce partial Latin square (PLS) canonical labels that are more humanly understandable and somewhat aid compression. We also give an algorithm for computing PLS canonical labels, which is efficient for small orders, and small orders are the most relevant in the context of a Latin-square search engine. We also propose a LS-Tree data structure to store canonical labels and Latin-square properties to avoid duplicate storage.

There are several ways in which this work may be built upon towards a Latin-square search engine.

- Canonical labelling could benefit from the use of invariants (e.g. the "train" invariant [40]). Invariants would likely improve the speed of computing canonical labels for both the PLS method and Nauty (provided we use quick-to-compute invariants). However, invariants would change the canonical labels thereby hindering compression, and result in less "humanly interpretable" canonical labels, thus giving a trade-off. Invariants which are useful for Latin squares are not necessarily useful for partial Latin squares [41], so there is no obvious "best" choice of invariant; computing PLS invariants is a current research topic [42]. We thus consider invariants beyond the scope of this paper, but a worthwhile topic for future research.
- We have not exhausted all means of data compression possible in a fully-fledged Latin-square search engine: it would also be possible to compress consecutive PLS canonical labels according to their common prefixes.
- Caching frequently searched ("hot") Latin squares and using stronger compression on infrequently searched Latin squares may also be worthwhile in practice.
- Another useful feature would be to add text search, similar to the OEIS, containing e.g., relevant references, source code, weblinks, historical remarks, and so on.

In this paper, we have restricted our attention to smaller orders since most real-world applications of Latin squares are for small orders. Incorporating larger orders (say up to order 100) would either require a different canonical labelling method than the PLS method, or the usage of an invariant.

REFERENCES

[1] S. Matthias, Reichenbach, A. Agarwal, and R. Zanibbi, "Rendering expressions to improve accuracy of relevance assessment for math search," in *Proc. SIGIR*, 2014, pp. 851–854.

[2] K. D. V. Wangari, R. Zanibbi, and A. Agarwal, "Discovering real-world use cases for a multimodal math search interface," in *Proc. SIGIR*, 2014, pp. 947–950.

[3] M. Schubotz, A. Youssef, V. Markl, and H. S. Cohl, "Challenges of mathematical information retrievalin the NTCIR-11 math Wikipedia task," in *Proc. SIGIR*, 2015, pp. 951–954.

[4] G. Brinkmann, K. Coolsaet, J. Goedgebeur, and H. Mélot, "House of Graphs: a database of interesting graphs," *Discrete Appl. Math.*, vol. 161, pp. 311–314, 2013, http://hog.grinvin.org.

[5] S. Swain, "Graph parameters: theory, generation and dissemination," Ph.D. dissertation, Monash University, 2019.

[6] D. S. Stones, "The many formulae for the number of Latin rectangles," *Electron. J. Combin.*, vol. 17, 2010, a1.

[7] B. D. McKay and E. Rogoyski, "Latin squares of order 10," *J. Combinatorics*, vol. 2.3, pp. 1–4, 1995.

[8] B. D. McKay, A. Meynert, and W. Myrvold, "Small Latin squares, quasigroups, and loops," *J. Combin. Des.*, vol. 15, pp. 98–119, 2007.

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

[10] A. Hulpke, P. Kaski, and P. R. J. Östergård, "The number of Latin squares of order 11," *Math. Comp.*, vol. 80, pp. 1197–1219, 2011.

[11] P. Armitage, G. Berry, and J. N. S. Matthews, *Statistical Methods in Medical Research, Fourth Edition*. Blackwell Science, 2008.

[12] M. H. Schneps, J. M. Thomson, G. Sonnert, M. Pomplun, C. Chen, and A. Heffner-Wong, "Shorter lines facilitate reading in those who struggle," *PLoS One*, 2013.

[13] J. Kätsyri, T. Kinnunen, K. Kusumoto, P. Oittinen, and N. Ravaja, "Negativity bias in media multitasking: The effects of negative social media messages on attention to television news broadcasts," *PLoS One*, 2016.

[14] M. Jacobson and P. Matthews, "Generating uniformly distributed Latin squares," *J. Combin. Des.*, vol. 4, no. 6, pp. 405–437, 1996.

[15] N. Besharati, L. Goddyn, E. S. Mahmoodian, and M. Mortezaeefar, "On the chromatic number of Latin square graphs," *Discrete Math.*, vol. 339, no. 11, pp. 2613–2619, 2016.

[16] R. Mandl, "Orthogonal Latin squares: an application of experiment design to compiler testing," *Commun. ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[17] L. Xu and J. Bruck, "X-code: MDS array codes with optimal encoding," *IEEE Trans. Information Theory*, vol. 45, pp. 272–276, 1999.

[18] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proc. FAST*, 2004.

[19] L. Yi, R. J. Stones, and G. Wang, "Two-erasure codes from 3-plexes," in *Proc. NPC*, 2019.

[20] R. J. Stones, "K-plex 2-erasure codes and Blackburn partial Latin squares," submitted for publication.

[21] M. Y. Hsiao, D. C. Bossen, and R. T. Chien, "Orthogonal Latin square codes," *IBM J. Res. Develop.*, vol. 14, pp. 390–394, 1970.

[22] R. Datta and N. A. Touba, "Generating burst-error correcting codes from orthogonal Latin square codes—a graph theoretic approach," in *Proc. IEEE DFT*, 2011.

[23] P. Reviriego, S. Pontarelli, A. Sánchez-Macián, and J. A. Maestro, "A method to extend orthogonal Latin square codes," *IEEE Trans. VLSI Systems*, vol. 22, no. 7, pp. 1635–1639, 2014.

[24] M. Demirci, P. Reviriego, and J. A. Maestro, "Implementing double error correction orthogonal Latin squares codes in SRAM-based FPGAs," *Microelectronics Reliability*, vol. 56, pp. 221–227, 2016.

[25] C. Shannon, "Communication theory of secrecy systems," *Bell System Technical Journal*, vol. 28, pp. 656–715, 1949.

[26] M. Xu and Z. Tian, "A novel image encryption algorithm based on self-orthogonal Latin squares," *Optik*, vol. 171, pp. 891–903, 2018.

[27] I. Kadar and L. Kurz, "A class of robust edge detectors based on Latin squares," *Pattern Recognition*, vol. 11, pp. 329–339, 1979.

[28] V. Čanda and T. V. Trung, "Scalable block ciphers based on Feistel-like structure," *Tatra Mountains Mathematical Pub*, vol. 25, pp. 39–66, 2002.

[29] A. Mileva and S. Markovski, "Quasigroup representation of some lightweight block ciphers," *Quasigroups Related Syst.*, vol. 22, pp. 267–276, 2014.

[30] Y. Xu, "Stream cipher based on post-commutative quasigroups," in *Proc. International Conference on Information Science and Engineering*, 2010.

[31] M. Satti and S. Kak, "Multilevel indexed quasigroup encryption for data and speech," *IEEE Trans. Broadcasting*, vol. 55, 2009.

[32] M. Battey and A. Parakh, "Efficient quasigroup block cipher for sensor networks," in *Proc. ICCCN*, 2012.

[33] S. K. Pal, S. Kapoor, A. Arora, R. Chaudhary, and J. Khurana, "Design of strong cryptographic schemes based on Latin squares," *J. Discr. Mathe. Sciences Crypt.*, vol. 25, pp. 233–256, 2013.

[34] R. Ghosh, S. Verma, R. Kumar, S. Kumar, and S. Ram, "Design of hash algorithm using Latin square," *Procedia Computer Science*, pp. 759–765, 2015.

[35] R. Anstee and A. Sali, "Latin squares and low discrepancy allocation of two-dimensional data," *European J. Combin.*, vol. 28, pp. 2115–2124, 2007.

[36] J. Shen, T. Zhou, X. Liu, and Y.-C. Chang, "A novel Latin-square-based secret sharing for M2M communications," *IEEE Trans. Industrial Informatics*, vol. 14, no. 2, pp. 3659–3668, 2018.

[37] D. S. Paul and N. Gautham, "MOLS 2.0: software package for peptide modeling and protein-ligand docking," *J. Molecular Modeling*, vol. 22:239, 2016.

[38] C. Qi, "On index-based query in SQL server database," in *Proc. CCC*, 2016.

[39] B. D. McKay, "Nauty graph isomorphic software," http://cs.anu.edu.au/~bdm/nauty.

[40] I. M. Wanless, "Atomic Latin squares based on cyclotomic orthomorphisms," *Electron. J. Combin.*, vol. 12, 2005, r22, 23 pp.

[41] R. J. Stones, R. M. Falcón, D. Kotlar, and T. G. Marbach, "Computing autotopism groups of partial Latin rectangles: a pilot study," submitted for publication.

[42] E. Danan, R. M. Fálcon, D. Kotlar, T. G. Marbach, and R. J. Stones, "Refining invariants for computing autotopism groups of partial Latin rectangles," *Discrete Math.*, to appear.