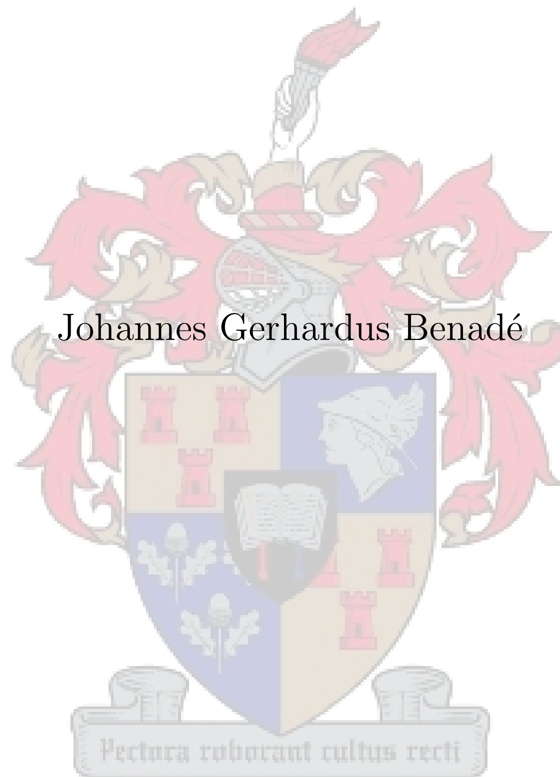
# A distributed system for enumerating main classes of sets of mutually orthogonal Latin squares




Johannes Gerhardus Benadé

Thesis presented in partial fulfilment of the requirements for the degree of
**Master of Science**
in the Faculty of Science at Stellenbosch University

Supervisor: Prof JH van Vuuren
Co-supervisor: Dr AP Burger

December 2014

# Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 1, 2014

Copyright © 2014 Stellenbosch University

All rights reserved

# Abstract

A *Latin square* is an $n \times n$ array containing $n$ copies of each of $n$ distinct symbols in such a way that no symbol is repeated in any row or column. Two Latin squares are *orthogonal* if, when superimposed, the ordered pairs in the $n^2$ cells are all distinct. This notion of orthogonality extends naturally to sets of $k > 2$ *mutually orthogonal Latin squares* (abbreviated in the literature as $k$-MOLS), which find application in scheduling problems and coding theory.

In these instances it is important to differentiate between structurally different $k$-MOLS. It is thus useful to classify Latin squares and $k$-MOLS into equivalence classes according to their structural properties — this thesis is concerned specifically with *main classes* of $k$-MOLS, one of the largest equivalence classes of sets of Latin squares.

The number of main classes of $k$-MOLS of orders $3 \leq n \leq 8$ have been enumerated in the literature by recursive backtracking algorithms. All enumeration attempts for $k$-MOLS of order $n > 8$ have, however, encountered a computational barrier using current computing technology in traditional computing paradigms. In this thesis, the feasibility of these enumerations of order $n > 8$ is analysed and a potential way of overcoming this computational barrier is proposed.

A backtracking enumeration algorithm from the literature is implemented and validated, after which novel estimates of the sizes of the enumeration search trees for $k$-MOLS of orders $n > 8$ produced by this backtracking algorithm are presented.

It is also advocated that the above-mentioned computational barrier may be overcome by volunteer computing, a computing paradigm in which large computations are distributed over thousands or even millions of volunteered computing devices, such as desktop computers and Android cellphones. A volunteer computing project is designed for the distributed enumeration of main classes of $k$-MOLS. Initial test results obtained from this volunteer computing project have called for a novel work unit issuing policy which allows the participating host resources to be utilised effectively during enumerations of main classes of $k$-MOLS of arbitrary orders.

A local pilot study involving the enumeration of main classes of 3-MOLS of order 8 has confirmed the feasibility of adopting the volunteer computing project as an avenue of approach towards the enumeration of $k$-MOLS of orders $n > 8$ and preliminary results of an ongoing enumeration attempt for the main classes of 7-MOLS of order 9 are presented.

# Uittreksel

'n *Latynse vierkant* is 'n $n \times n$ skikking wat $n$ kopieë van elk van $n$ verskillende simbole bevat sodat geen simbool in enige ry of kolom daarvan herhaal word nie. Indien twee Latynse vierkante op mekaar gesuperponeer word, en die geordende pare simbole wat sodoende in die $n^2$ selle gevorm word, almal verskillend is, word die vierkante *ortogonaal* genoem. Die begrip van ortogonaliteit veralgemeen op 'n natuurlike wyse na $k > 2$ *onderling ortogonale Latynse vierkante* (wat in die internasionale literatuur as $k$-MOLS afgekort word) en vind toepassing in skeduleringsprobleme en kodeerteorie.

In hierdie toepassings is dit belangrik om 'n onderskeid te tref tussen struktureel verskillende $k$-MOLS. Dit is gevolglik nuttig om Latynse vierkante en $k$-MOLS in ekwivalensieklasse volgens hul strukturele eienskappe te klassifiseer. In hierdie verhandeling word daar gefokus op *hoofklasse* van $k$-MOLS, een van die grootste ekwivalensieklasse van versamelings Latynse vierkante.

Die getal hoofklasse van $k$-MOLS van ordes $3 \leq n \leq 8$ is in die literatuur deur middel van rekursiewe algoritmes met terugkering getel. Geen poging om hoofklasse van $k$-MOLS van ordes $n > 8$ te tel, kon egter daarin slaag om 'n berekeningstruikelblok te oorkom wat as gevolg van huidige rekentegnologiese beperkings bestaan nie. In hierdie verhandeling word die haalbaarheid van sulke telpogings vir orde $n > 8$ ondersoek en word 'n metode voorgestel waarmee hierdie berekeningstruikelblok moontlik oorkom kan word.

'n Bestaande telalgoritme met terugkering word geïmplementeer en gevalideer, waarna nuwe afskattings van die groottes van die soekbome vir hoofklasse van $k$-MOLS van ordes $n > 8$ wat deur hierdie algoritme deurstap moet word, daargestel word.

Daar word geargumenteer dat die bogenoemde berekeningstruikelblok moontlik oorkom kan word deur gebruik te maak van 'n grootskaalse parallelle rekenparadigma waarin groot berekeninge oor duisende of selfs miljoene rekentoestelle, soos tafelrekenaars of Android sellulêre telefone wat vrywillig deur gebruikers vir hierdie doel beskikbaar gemaak word. So 'n verspreide berekeningsprojek word vir hoofklasse van $k$-MOLS ontwerp. Aanvanklike resultate wat uit hierdie projek voortgespruit het, het 'n nuwe beleid genoodsaak waarvolgens werkeenhede aan deelnemende rekentoestelle op só 'n wyse uitgedeel word dat die projek doeltreffend van hulpbronne gebruik maak, selfs wanneer hoofklasse van $k$-MOLS van arbitrêre ordes bepaal word.

'n Lokale proefstudie word geloods waartydens bekende telresultate vir hoofklasse van $k$-MOLS van orde 8 bevestig word. Die haalbaarheid van 'n verspreide berekeningsbenadering, waaraan baie vrywilligers kan deelneem om hoofklasse van $k$-MOLS van orde $n > 8$ te tel, word ondersoek en die resultate van 'n huidige verspreide berekeningspoging om hoofklasse van 7-MOLS van orde 9 te tel, word gerapporteer.

# Acknowledgements

The author wishes to acknowledge the following people for their various contributions towards the completion of this work:

- My supervisor Prof JH van Vuuren, for his excellent guidance, enthusiasm towards this project and dedication to his students.
- My co-supervisor Dr Alewyn Burger, for his accessibility, friendliness and invaluable assistance whenever required.
- The National Research Foundation and MIH Media Lab for their financial assistance.
- My fellow students, who made procrastinating so much more enjoyable.
- My family and girlfriend, for their love, understanding and support.

# Table of Contents


**List of Figures** xi

**List of Tables** xiii

**1 Introduction** 1

1.1 Historical background . . . 1

1.2 Problem statement . . . 4

1.3 Scope and objectives . . . 5

1.4 Thesis organisation . . . 6

**2 Mathematical preliminaries** 9

2.1 Combinatorics . . . 9

2.2 Group theory . . . 12

2.3 Latin squares . . . 13

2.3.1 Basic definitions . . . 13

2.3.2 Orthogonal Latin squares . . . 15

2.3.3 Operations on Latin squares . . . 16

2.4 Chapter summary . . . 19

**3 The enumeration of MOLS** 21

3.1 The classification of Latin squares . . . 21

3.2 A historical overview of the enumeration of Latin squares . . . 24

3.3 The enumeration methodology adopted in this thesis . . . 27

3.4 On the enumerability of larger-order search spaces . . . 36

3.5 Chapter summary . . . 41

**4 Volunteer computing** 43

4.1 A historical overview of public-resource computing . . . 43

4.2 The Berkeley Open Infrastructure for Network Computing . . . 46

4.2.1 Basic workflow and concepts of volunteer computing . . . 46
4.2.2 Grid-enabling a simple BOINC project . . . 47
4.2.3 Special types of applications . . . 52
4.2.4 Setting up a server and project maintenance . . . 53
4.2.5 Security concerns . . . 54
4.2.6 Challenges facing volunteer computing . . . 54
4.3 Chapter summary . . . 55

**5 A distributed volunteer project for the enumeration of $k$-MOLS 57**
5.1 A volunteer project for counting 3-MOLS of order 8 . . . 57
5.1.1 Server architecture . . . 58
5.1.2 Grid-enabling the exhaustive enumeration algorithm . . . 58
5.1.3 Deamons . . . 59
5.1.4 First enumeration results . . . 59
5.2 Generalising to the enumeration of $k$-MOLS of order $n$ . . . 60
5.2.1 Limiting work unit sizes . . . 61
5.2.2 Dynamic splitting of work units . . . 63
5.2.3 Implementing and validating the generalisation . . . 64
5.3 Enumeration results emanating from an implementation . . . 66
5.4 Chapter summary . . . 68

**6 Conclusion 69**
6.1 Overview of the work contained in this thesis . . . 69
6.2 An appraisal of the contributions of this thesis . . . 71
6.3 Future work . . . 71

**References 73**

# List of Figures


1.1 A four-by-four arrangement of cards as solution to a 1624 puzzle . . . . . . . . . 1
1.2 The 2-MOLS of order 10 constructed by Parker to disprove Euler's conjecture . . 4

3.1 The relationships between the transformations applicable to Latin squares . . . . 22
3.2 Three Latin squares of order 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.3 The backtracking enumeration search tree for 2-MOLS of order 5 . . . . . . . . . 33
3.4 Estimating the size of a tree by performing random dives . . . . . . . . . . . . . 37
3.5 The number of feasible candidate universals passing the `isOrthogonal` test . . . 38

4.1 The basic workflow on the client and BOINC project server . . . . . . . . . . . . 47
4.2 The interaction between the BOINC server, the database, clients and daemons . 51
4.3 Examples of graphical applications used by SETI@Home and WCG . . . . . . . 52

5.1 A graphical representation of the checkpointing strategy . . . . . . . . . . . . . . 58
5.2 A hypothetical volunteer computing project with four hosts . . . . . . . . . . . . 62
5.3 The effect of recycling work units in the hypothetical volunteer project . . . . . . 63
5.4 The effect of splitting recycled work units in the hypothetical volunteer project . 63

# List of Tables


1.1 Scheduling an experiment involving combinations of grapes and yeast . . . . . . 3

2.1 The cyclical nature of permutations . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 The Cayley table of the group $(\mathbb{Z}_4, +)$. . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.1 A summary of transformations applicable to the classification of Latin squares . 24
3.2 The number of main classes of $k$-MOLS of order $n \in \{3, 4, \ldots, 10\}$ . . . . . . . . 26
3.3 The active branches of the search tree for 3-MOLS of order $n \leq 8$ . . . . . . . . . 34
3.4 The active nodes and the enumeration time for 3-MOLS of order 8 . . . . . . . . 35
3.5 Validating the implementation of Algorithm 3.1 . . . . . . . . . . . . . . . . . . . . 35
3.6 Validating the implementation of Algorithm 3.1 with normalised runtimes . . . . 35
3.7 Estimated tree sizes after the `isOrthogonal` test for orders 8, 9 and 10 . . . . . . 38
3.8 The average proportion of nodes passing the `isSmallest` test . . . . . . . . . . . 39
3.9 The estimated enumeration tree size and runtime for 3-MOLS of orders 9, 10 . . 39
3.10 Nodes on level 0 for main classes of $k$-MOLS of orders $n \in \{3, 4, \ldots, 10\}$ . . . . . 40
3.11 The nodes on level 0 per section for 3-MOLS of order 9 . . . . . . . . . . . . . . 40

4.1 The core of the BOINC C/C++ API [92]. . . . . . . . . . . . . . . . . . . . . . . . . 48
4.2 Parameters that may be specified in the input template of a BOINC project . . . 50
4.3 Parameters that may be specified in the output template of a BOINC project . . 51

5.1 Results of an initial distributed enumeration attempt for 3-MOLS of order 8 . . . 60
5.2 Host contribution during an initial enumeration for 3-MOLS of order 8 . . . . . . 60
5.3 The results issued in each section of the tree for 3-MOLS of order 8 . . . . . . . . 61
5.4 The file format of a starting position, checkpoint and result . . . . . . . . . . . . 64
5.5 Validating the work unit management policy at node 9 for 3-MOLS of order 8 . . 65
5.6 Results issued under a new work management policy . . . . . . . . . . . . . . . . 66
5.7 Host contribution under the new work unit management policy . . . . . . . . . . 66
5.8 The distribution of the nodes on level 0 for 7-MOLS of order 9 . . . . . . . . . . 67

5.9 Partial results for two sections of the enumeration for 7-MOLS of order 9 . . . . 67

# CHAPTER 1

# Introduction

Contents


1.1 Historical background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Problem statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Scope and objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Thesis organisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6


## 1.1 Historical background

In 1624 Claude Gaspard de Bachet published a book of mathematical puzzles entitled "Problèmes plaisants & délectables: qui se font par les nombres." One of these puzzles asked in how many ways it is possible to arrange the sixteen court cards from a standard deck of playing cards in a four-by-four grid such that every row and column of the grid contains exactly one card of each of the four ranks and one card of each of the four suits [7]. An example of such an arrangement may be found in Figure 1.1. Bachet mistakenly claimed that there are 72 such designs if rotations and reflections of a design are not considered to be different designs. The correct number of such designs is, however, 144 [43].

FIGURE 1.1: *A four-by-four arrangement of the court cards from a deck from cards which is a solution to the puzzle posed by Claude Gaspard de Bachet.*

Approximately 150 years later, the Swiss mathematician Leonhard Euler started a 1782 paper with a reference to a similar puzzle occupying his thoughts:

> "A very curious question that has taxed the brains of many inspired me to undertake the following research that has seemed to open a new path in Analysis and in particular in the area of combinatorics. This question concerns a group of thirty-six officers of six different ranks, taken from six different regiments, and arranged in a square in a way such that in each row and column there are six officers, each of a different rank and regiment." [33]

Euler used the first six letters of the Latin and Greek alphabets to denote respectively the six regiments and ranks in his attempts at constructing such a design, leading to the contemporary term "Greaco-Latin square of side 6" when referring to such a design. The problem may be restated by asking for a six-by-six arrangement of pairs of symbols, one specifying a soldier's rank and the other his regiment, in such a way that no rank or regiment is repeated in any row or column. Implicit in this definition is the fact that every pair of symbols is unique (in other words, no two soldiers of the same rank also hail form the same regiment).

Euler was unable to find such an arrangement of soldiers and conjectured not only that no such arrangement exists for six-by-six grids, but also that none exists for $(4k + 2) \times (4k + 2)$ grids, where $k \in \mathbb{Z},\ k \geq 1$. Approximately 120 years later, Gaston Tarry [90] proved (in 1900) that Euler's "36-Officers problem," as it had then become known, indeed has no solutions. Another sixty years later, however, constructions for Greaco-Latin squares of sides 10 [78] and 22 [14] were found, thereby disproving Euler's conjecture for these cases. Shortly afterwards, general constructions were established for Greaco-Latin squares of side $4k + 2$ for all $k \in \mathbb{Z},\ k > 1$ [13].

Today, these designs are no longer referred to as *Greaco-Latin squares of side $n$*, but rather as pairs of *mutually orthogonal Latin squares of order $n$* (which may, of course, be superimposed to form a Greaco-Latin square). The term *orthogonal* here means that the pairs of superimposed symbols are unique as ordered pairs (in other words, no two soldiers are both of the same rank and the same regiment). Orthogonality may be generalised to a collection of $k > 2$ Latin squares which are mutually orthogonal in pairs, called a *set of $k$ mutually orthogonal Latin squares of order $n$*, and abbreviated in this thesis as *$k$-MOLS of order $n$*.

These early investigations described above summarise the three main concerns involving Latin squares. First there is the question of the *existence* of a Latin square or a $k$-MOLS of a certain order. Secondly, attempts are made to find *constructions* for Latin squares or $k$-MOLS of certain order, possibly satisfying additional properties. It was seen in Bachets' algorithm that solutions which are rotations or reflections of other designs were not counted as distinct designs. This introduces the notion that certain distinct Latin squares and $k$-MOLS share fundamental structural properties which remain invariant under certain symmetry and other operations. Latin squares and $k$-MOLS may be partitioned into equivalence classes according to these properties. One of the most general such classes, and one that is particularly important in this thesis, is called a *main class*. The final question commonly asked in relation to combinatorial designs, such as Latin squares or $k$-MOLS, concerns the *enumeration* of all structurally different designs that exist of a given order.

Despite the fact that Latin squares were seen as mathematical curiosities for a long time, they, and especially $k$-MOLS, have interesting applications. Perhaps the best-known application occurs in the design of experiments, as described by Fischer and Yates [39]. Consider a study involving objects of $n$ different types treated in $n$ different ways. Suppose every type of object receives all the possible types of treatment and a subset of $n$ objects are to be repeatedly sampled in such a way that every type of object and every type of treatment is included in each

Table 1.1: *An experiment testing combinations of grapes and yeast for a vineyard. The entries in the table represent the number of weeks after which the combination should be drank to ensure a balanced sampling.*

| | Merlot | Pinot noir | Malbec | Garnay |
|---|---|---|---|---|
| Saccharomyces | 1 | 2 | 3 | 4 |
| Candida | 4 | 1 | 2 | 3 |
| Kloeckera | 3 | 4 | 1 | 2 |
| Zygosaccharomyces | 2 | 3 | 4 | 1 |

sample, but that no two samples of the same period may coincide in either the variety of the grape or the type of the yeast used. A Latin square of order $n$ provides a feasible schedule for the sampling of such an experiment over up to $n$ time periods. For example, if a vintner is interested in experimenting with four different varieties of grape in combination with four different strains of yeast, and sampling takes place at the end of every week for a month, then the schedule in Table 1.1, which is a Latin square of order 4, ensures that every variety of grape and yeast is included in each of the weekly samples, but that no two of the wines tasted in any week contain the same variety of grape or yeast. Furthermore, if it is, for example, believed that the nature (*e.g.* the type of wood, the way it was treated or the age) of the casket in which the wine matures affects the wine's taste, a second Latin square of order 4, orthogonal to the Latin square in Table 1.1, may be used to ensure that a wine from every type of casket is sampled every week.

In addition to assisting in experimental design, Latin squares and $k$-MOLS have a number of applications in coding theory, which deals with techniques for ensuring that a transmitted signal/message is interpreted correctly. A simple application in this field involves the use of the tuples $(i, j, \boldsymbol{L}(i, j))$[1] from a Latin square $\boldsymbol{L}$ of order $n$ to represent $n^2$ code words. This code has the property that any single error during transmission will not only be detected, but may also be corrected. This schema of using Latin squares in error correcting codes may be extended to multiple-error corrections by employing $k$-MOLS[2] instead of single Latin squares. Other notable applications of Latin squares and MOLS to scheduling problems include computer memory access schemes [66] and sports scheduling [50, 52, 83]. In these applications, every structurally different Latin square or $k$-MOLS represents an additional solution and therefore allows the scheduler more freedom to consider additional constraints that are external to the basic problem description.

A considerable amount of research has been done since 1782 on partitioning Latin squares and $k$-MOLS into equivalence classes based on their structural properties and attempting to count these equivalence classes. This is not an easy enumeration problem in view of the fact that a single row of a Latin square may take $n!$ different forms. Almost all studies attempting to enumerate these equivalence classes have, in fact, encountered a computational barrier in the form of what Erdös called a "combinatorial explosion" [31] — even for relatively small orders of Latin squares and $k$-MOLS. For example, the main classes of $k$-MOLS constitute one of the larger types of equivalence classes and have only been enumerated for $k$-MOLS of orders not exceeding $n = 8$ [53].

The difficulties associated with these types of enumeration problems may perhaps best be ap-

[1]Here the notation $\boldsymbol{L}(i, j)$ denotes the entry in row $i$ and column $j$ of a Latin square $\boldsymbol{L}$.

[2]The application of error-correcting codes is not considered any further in this thesis. The interested reader is, however, referred to Golomb and Poner [**Golomb** ], Bossen *et al.* [48] and Elspas *et al.* [65] for descriptions of the applications of Latin squares in coding theory.

preciated by considering the case of a pair of orthogonal Latin squares of order 10, which was a significant stepping stone in the long process of disproving Euler's conjecture. Prior to Parker's 1959 construction of the pair of orthogonal Latin squares of order 10, shown in Figure 1.2, researchers doubted the existence of such a design for approximately 170 years. Today, however, it has been heuristically argued [62] that there are approximately $10^{15}$ such distinct pairs, all of which somehow managed to elude the combinatorial research community for nearly two centuries. Today, the question of the existence of a 3-MOLS of order 10 is one of the most celebrated open questions in design theory and it is very possible that, if such designs exist, there are multiple instances of these designs that have thus far evaded detection.

$$\left\{ \begin{bmatrix} 1&2&3&4&5&6&7&8&9&0\\ 7&4&2&0&6&5&8&9&3&1\\ 5&1&4&6&0&8&9&2&7&3\\ 0&7&1&3&8&9&4&5&1&6\\ 3&5&7&8&9&1&0&4&6&2\\ 2&0&5&9&8&3&1&6&5&8\\ 4&3&0&5&2&7&6&1&8&9\\ 8&9&6&2&3&0&5&7&1&4\\ 6&8&9&7&1&4&2&3&0&5\\ 9&6&8&1&4&2&3&0&5&7 \end{bmatrix}, \begin{bmatrix} 2&3&1&6&9&4&8&7&5&0\\ 4&2&7&9&1&8&5&0&3&6\\ 1&4&5&7&8&0&3&2&6&9\\ 7&1&0&8&3&2&4&6&2&5\\ 5&7&3&2&4&1&6&9&0&8\\ 0&5&2&1&7&6&9&3&8&4\\ 3&0&4&5&6&9&2&8&1&7\\ 9&8&6&4&2&3&0&5&7&1\\ 8&6&9&0&5&7&1&4&2&3\\ 6&9&8&3&0&5&7&1&4&2 \end{bmatrix} \right\}$$

FIGURE 1.2: *The 2-MOLS of order 10 constructed by Parker [78] approximately 170 years after Euler first questioned their existence. Today it is estimated that there may be up to $10^{15}$ such distinct designs [62].*

Although the enumeration of main classes of MOLS is undoubtedly a very challenging problem from a computational point of view, it is usually attacked within the traditional framework of scientific computing, consisting of the use of a desktop computer or a high-performance computing cluster. The rapid rise in popularity of personal computers and mobile devices over the last decade has, however, created a world in which an estimated ten billion devices are connected through the internet [4, 34], only a very small portion of which is actually harnessed by researchers. Two very common examples of wasted computing resources may perhaps briefly be considered. In the six months after its release in 2013, forty million Samsung Galaxy S4 smartphones were sold. Every Galaxy S4 boasts a quad-core processor, a quad-core graphical processing unit and 2Gb of random access memory, but spends the majority of its lifespan in standby mode. Similarly, Stellenbosch University owns approximately 4 000 computers, scattered among various computer user areas, offices and administrative buildings, but the vast majority of these multi-core machines are idle more than eight hours per day.

Changing technology demands that scientists adapt their tools! Indeed, it is conceivable that many computational barriers may be shifted dramatically and that many open research questions may be resolved if only a fraction of the computing power available today is used efficiently. Might the existence question of 3-MOLS of order 10 perhaps be resolved as a result of such a barrier shift?

## 1.2 Problem statement

The enumeration of main classes of $k$-MOLS of order $n > 8$ has been found to be computationally too challenging for the current computing technology if conducted in a traditional scientific

computing paradigm. The feasibility of establishing a distributed computing project that makes use of volunteers' idle computing resources is considered in this thesis as a potential way of overcoming the computational barrier currently experienced in the enumeration of main classes of $k$-MOLS of order $n > 8$.

A number of middleware systems exist for establishing collaborative grid computers. Of these, the *Berkeley open infrastructure for network computing* (BOINC) is the most widely used in the constext of public volunteer computing. A BOINC project is designed in this thesis for the exhaustive enumeration of main classes of $k$-MOLS of order $n$ and the feasibility of this enumeration approach is confirmed in the form of a pilot study for $n = 8$ and $n = 9$.

## 1.3 Scope and objectives

The following objectives are pursued in this thesis:

I To *survey* the literature related to the theory of Latin squares and $k$-MOLS, as well as the practice of distributed computing.

II To *review* a variety of popular equivalence classes of Latin squares and $k$-MOLS and to *document* previous attempts at enumerating these classes of combinatorial objects.

III To *design* an effective algorithm for the enumeration of main classes of $k$-MOLS.

IV To *implement* this algorithm and to *verify* its correctness by comparing its results to known enumeration results for $k$-MOLS of order $n \leq 8$.

V To *estimate* the sizes of the enumeration search trees for $k$-MOLS of orders $n = 9$ and $n = 10$, which are currently computationally too expensive to traverse serially.

VI To *design* a distributed computing project for the enumeration of main classes of $k$-MOLS.

VII To *launch* a local pilot volunteer computing project for the enumeration of main classes of $k$-MOLS of orders 8 and 9 by means of volunteer computing.

VIII To *establish* the feasibility of using public volunteer computing for the enumeration of main classes of MOLS of orders 9 and 10, including the potential of the contribution of such an enumeration approach to towards settling the infamous existence question of 3-MOLS of order 10.

Combinatorial designs other than Latin squares are largely considered to be beyond the scope of this thesis, as are geometric and algebraic representations of Latin squares and $k$-MOLS. Although the enumeration of Latin squares and $k$-MOLS is considered in this thesis, the question of the existence of these objects is not considered explicitly, except to the extent in which an enumeration attempt may imply the (non-)existence of a design. More specifically, only main classes of $k$-MOLS are considered; other equivalence classes are reviewed merely to provide a context for the current study of Latin square main classes.

Finally, this study is restricted to volunteer computing, specifically employing BOINC as middleware, as this is the predominant volunteer computing middleware in use today. Alternative desktop grid architectures are not considered, and neither are alternative computing paradigms, such as cloud computing.

## 1.4 Thesis organisation

In the second chapter of this thesis various mathematical prerequisites are reviewed that are vital for an understanding of the work in the remainder of the thesis. The notion of a permutation is introduced and it is shown how a binary composition operator may act on permutations. Groups, quasigroups and Latin squares are defined, and it is described how a Latin square is the Cayley table of a quasigroup. The notion of a universal, which plays an important role in the enumeration process documented in later chapters, is also introduced. The notion of orthogonality between pairs of Latin squares is formally introduced and generalised to sets of $k$ mutually orthogonal Latin squares. Finally, consideration is given to a number of permutations which may act on the row, column or symbol indexing sets of a Latin square without changing its underlying structural properties, as well as a the set of conjugate operations which may act on a Latin square.

The third chapter is devoted to types of transformations applicable to Latin squares and $k$-MOLS, each consisting of specific allowable operations, and the way in which these transformations generate equivalence classes. A historical overview of previous work on the enumeration of equivalence classes of Latin squares and $k$-MOLS follows in the second section of the chapter. It is shown that an ordering may be imposed on a set of Latin squares or $k$-MOLS, facilitating the design of an exhaustive backtracking enumeration algorithm for finding the lexicographically smallest $k$-MOLS, called the class representative, in every main class. Numerical results obtained by this enumeration algorithm are also presented in order to validate the algorithm and to demonstrate the effectiveness of the enumeration approach for main classes of $k$-MOLS of order $n \leq 8$. Because this enumeration process becomes computationally very expensive for $k$-MOLS of order $n > 8$, Knuth's [54] and Purdom's [79] well-known techniques for estimating the size of a rooted tree by a series of random dives down from the root are slightly modified and applied to the enumeration search trees for $k$-MOLS of order $n \leq 8$ in order to elucidate the structures of these trees. The sizes of the enumeration search trees for main classes of $k$-MOLS are estimated for orders $n \leq 10$.

The concept of volunteer computing is explored in Chapter 4. Volunteer computing offers the general public the opportunity to participate in scientific research and, in turn, provides scientists with access to volunteers' idle computing resources. A number of large organisations, such as IBM, Oxford University and CERN, currently manage volunteer computing projects related to some of their research. In §4.2, the focus falls on the ubiquitous middleware system responsible for handling interaction between project scientists and engineers, called BOINC. Various aspects of establishing and maintaining a volunteer computing project, such as the workflow, the steps required to grid-enable an application, the set-up of a server, and the security concerns involved, are explored.

The fifth chapter is devoted to the fundamental question of whether it is possible, and practical, to grid-enable the enumeration algorithm presented in Chapter 3 for main classes of $k$-MOLS in the hope that volunteer computing may provide access to sufficient computing power for overcoming the computational barrier currently encountered in the numeration of main classes of $k$-MOLS of order $n > 8$. In §5.1, the design and components of such a volunteer computing project are outlined, the application is modified to make use of the BOINC application programming interface and a proof of concept is demonstrated by enumerating 3-MOLS of order 8 on five hosts. This enumeration reveals serious concerns about the way in which work units are generated that render larger enumeration attempts all but impossible. The second section of the chapter is therefore concerned with resolving these problems through the introduction of an improved work unit management policy. A local pilot project is launched to test the effectiveness

of this policy, the results of which are encouraging and presented in §5.3. The chapter closes with a summary of partial enumeration results for main classes of 7-MOLS of order 9.

The thesis closes with in Chapter 6 with a summary of the work presented, an appraisal of the contributions of this thesis and a discussion on potential avenues for further work.

# CHAPTER 2

# Mathematical preliminaries

Contents


2.1 Combinatorics . . . . . . . . 9
2.2 Group theory . . . . . . . . 12
2.3 Latin squares . . . . . . . . 13
*2.3.1 Basic definitions* . . . . . . . . *13*
*2.3.2 Orthogonal Latin squares* . . . . . . . . *15*
*2.3.3 Operations on Latin squares* . . . . . . . . *16*
2.4 Chapter summary . . . . . . . . 19


The basic prerequisite mathematical knowledge for the study of Latin squares is presented in this chapter. Some combinatorial concepts, centred around the notion of a permutation, are reviewed in §2.1. Groups, quasigroups and loops are defined in §2.2, while §2.3 is an introduction to the theory of Latin squares, starting with basic concepts and exploring the relationship between Latin squares and quasigroups in §2.3.1. The notions of universals and transversals are also considered in §2.3.1 and feature prominently in the following section on the orthogonality of Latin squares and sets of Latin squares. In §2.3.3 consideration is given to the ways in which Latin squares and sets of mutually orthogonal Latin squares may be transformed without changing their fundamental structural properties.

## 2.1 Combinatorics

The two fundamental principles underlying the enumeration of combinatorial objects of certain types, which may be found in most introductory textbooks in combinatorics, are called the *addition principle* and the *multiplication principle*. According to Wallis and George [96], the addition principle states that the total number of possible outcomes of an experiment, if drawn from mutually exclusive pools, is simply the sum of the number of outcomes of the experiment in each of the pools. The multiplication principle, on the other hand, claims that, when building an arrangement of objects in stages in such a way that the choices at each stage do not depend on the choices at the other stages, the total number of possible arrangements is the product of the number of choices at every stage. By the addition principle, for example, a man buying a car and deciding between three different Audis and four different BMWs has $4 + 3 = 7$ choices in total. If, once he has picked a car, he is offered the choice of six different colours, two interior designs

and three engine sizes, he has a total of $6 \times 2 \times 3 = 36$ possible choices by the multiplication principle.

When selecting an ordered subset of $k$ objects from a set of $n$ distinct objects, where $k$ and $n$ are integers with $k \leq n$, there are clearly $n$ possible objects which may be selected first. The second object is chosen from $n-1$ distinct objects, as some object has already been selected. By repeated application of this observation, the $k$-th object will be selected from $n+1-k$ distinct objects. By the multiplication principle, the number of ways of selecting such a subset is $n \cdot (n-1) \cdots (n+1-k)$, or $n!/(n-k)!$, where the notation $n!$ denotes the product $n \times (n-1) \times \ldots \times 1$ of the first $n$ natural numbers. Such an ordered subset is called a *permutation* of order $k$ [96]. In the remainder of this thesis the term permutation will be restricted to refer specifically to permutations of order $n$, in other words, the $n!$ rearrangements of a set $n$ of objects. It is clear that any set of $n$ objects may be labelled by the integers $0, 1, \ldots, n-1$ (the elements of the set of integers modulo $n$, which is commonly denoted $\mathbb{Z}_n$) and referred to by these labels, so that any permutation may be considered without specific reference to its underlying objects.

A permutation $p$ may thus be seen as an ordering of the elements of the set $\mathbb{Z}_n$, and may be represented, in what Boná [11, p.73] calls *two-line notation*, as

$$p = \begin{pmatrix} 0 & 1 & \ldots & n-1 \\ p(0) & p(1) & \ldots & p(n-1) \end{pmatrix},$$

where $p(i) \in \mathbb{Z}_n$ is the image of $i \in \mathbb{Z}_n$ under the permutation $p$ . This notation emphasizes that a permutation $p$ is a function $p : \mathbb{Z}_n \mapsto \mathbb{Z}_n$. As long as $p(i)$ appears under $i$ in this representation, the elements $i \in \mathbb{Z}_n$ may appear in any order. Alternatively, if the elements $i \in \mathbb{Z}_n$ are fixed to appear in natural order, the top row may be omitted and the permutation simply expressed as $(p(0), p(1), \ldots, p(n-1))$. The integers $i \in \mathbb{Z}_n$ for which $i = p(i)$ remain invariant under the permutation $p$ and are called *fixed points*. The permutation of order $n$ with $n$ fixed points, in other words for which $i = p(i)$ for all $i \in \mathbb{Z}_n$, leaves the order of the elements of $\mathbb{Z}_n$ invariant and is thus called the *identity permutation*, denoted by the symbol $e$.

A permutation $p$ is *lexicographically smaller* than a permutation $q$, denoted by $p < q$, if $p(j) < q(j)$ for some $j \in \mathbb{Z}_n$ and $p(i) = q(i)$ for all $i < j \in \mathbb{Z}_n$. For example, $(0, 2, 1, 3, 4) < (0, 2, 1, 4, 3)$, and it is clear that the identity permutation is the lexicographically smallest permutation. Any set of permutations may be ordered lexicographically.

The product, or composition, of two permutations $p$ and $q$ of the same order is defined as $(q \circ p)(i) = q(p(i))$ for $i \in \mathbb{Z}_n$ [96]. Applying the composition $q \circ p$ to the identity permutation is clearly the same as first applying $p$ and then applying $q$ to the resulting permutation. As an example, note that

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 1 & 5 & 0 & 4 \end{pmatrix} \circ \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 0 & 2 & 3 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 & 5 & 0 \end{pmatrix}.$$

Repeated application of a permutation $p$ to itself reveals an interesting property of permutations. The effect of the permutation $p = (3, 5, 1, 0, 4, 2)$ repeatedly acting on itself may be seen in Table 2.1. It is clear that the position of the element 4 remains invariant, while the elements 0 and 3 are permuted among themselves, and the elements $1, 2$ and 5 among themselves. No matter how often $p$ is applied, it will always be the case that 0 is mapped to either 0 or 3, 1 is mapped to $1, 2$ or 5, *etc.* It is therefore said that *p cyclically* permutes 0 and 3, and similarly for 1, 2 and 5. This concept allows the permutation $p$ to be expressed in so-called *cycle notation* as $p = (4)(03)(152)$, where every integer is mapped to the one on its right, except for the last integer of every cycle, which is mapped to the first. The *length* of a cycle is the number of elements permuted by the cycle. Notice that, for cycles of length three or more, the order

Table 2.1: *The cyclical nature of permutations is revealed by repeated compositions of a permutation with itself.*

| Composition | Product |
|---|---|
| $p$ | $(3, 5, 1, 0, 4, 2)$ |
| $p \circ p$ | $(0, 2, 5, 3, 4, 1)$ |
| $p \circ p \circ p$ | $(3, 1, 2, 0, 4, 5)$ |
| $p \circ p \circ p \circ p$ | $(0, 5, 1, 3, 4, 2)$ |

in which the elements are noted matter, since the cycles (152) and (125), for example, define different actions. A cycle may, however, be *rotated*; no distinction is made between the cycles (152), (521) and (215). The number of equivalent ways of expressing a cycle equals the length of the cycle. Since the same permutation may be expressed in different forms in cycle notation, such as $p = (4)(03)(152) = (30)(4)(215)$, a unique way of writing permutations in cycle notation is required. In *canonical cycle notation*, the largest element of a cycle is written first and cycles are ordered in increasing order of these front elements. The canonical representation of the cycle $p$ above is $(30)(4)(521)$.

In cycle notation, fixed points are equivalent to 1-cycles (cycles of length 1), and may generally be omitted. The permutation (03) therefore fixes every element of a permutation of order $n \geq 4$, except for the integers 0 and 3, which are interchanged.

The *type* of a permutation $p$ of order $n$ is denoted by an $n$-tuple $(a_1, a_2, \ldots, a_n)$ which summarizes the lenghts of the cycles of $p$ in such a way that $a_i$ is the number of cycles of length $i$ for $i \in \mathbb{Z}_n$. The *cycle structure* of a permutation $p$ is denoted $z_1^{a_1} z_2^{a_2} \cdots z_n^{a_n}$, where $z_i$ is a placeholder which facilitates easier reading. A factor of the form $z_i^0$ is usually omitted in this notation for any $i \in \mathbb{Z}_n$, while a factor of the form $z_i^1$ is merely written as $z_i$. The permutation $p = (3, 5, 1, 0, 4, 2)$ considered earlier is of type $(1, 1, 1, 0, 0, 0)$ and has a cycle structure $z_1 z_2 z_3$.

A lexicographical ordering may also be imposed on cycle structures of the same order. The cycle structure $z_1^{a_1} z_2^{a_2} \cdots z_n^{a_n}$ is lexicographically smaller than the cycle structure $z_1^{b_1} z_2^{b_2} \cdots z_n^{b_n}$ if $a_j > b_j$ for some $j \in \mathbb{Z}_n$ and $a_i = b_i$ for all $i < j \in \mathbb{Z}_n$. The cycle structure $z_1 z_2^3$ is therefore lexicographically smaller than $z_1 z_6$.

The lexicographically smallest permutation with a given cycle structure is called the *cycle structure representative* and is found by arranging the cycles in order of increasing lengths and inserting the elements of $\mathbb{Z}_n$ in natural order from left to right. The cycle structure representative of $z_1 z_2 z_3$, for example, is $(0)(12)(345)$, or the permutation $(0, 2, 1, 4, 5, 3)$. It should be noted that arranging cycle structures lexicographically also arranges the respective cycle structure representatives lexicographically.

As mentioned above, that the composition $p \circ q$ of two permutations $p$ and $q$ of the same order maps any $i \in \mathbb{Z}_n$ to $q(p(i)) \in \mathbb{Z}_n$. If the permutation $q$ is defined to be the specific permutation for which $q(p(i)) = i$, then $q$ maps the permutation $p$ to the identity permutation $e$, so $q \circ p = e$. In this case $q$ is called the *inverse* of $p$, which may be denoted by $p^{-1}$, and has the property that $q(j) = p^{-1}(j) = i$ if $p(i) = j$ for $i, j \in \mathbb{Z}_n$. It may be shown that $(p \circ q)^{-1} = (q^{-1} \circ p^{-1})$, since the operation that was applied most recently must be inverted first.

Finally, note that the set of all permutations may be partitioned into equivalence classes. A permutation $p$ is a *conjugate permutation* of a permutation $q$ if there exists a third permutation $r$ such that $q = r \circ p \circ r^{-1}$, in which case $p$ and $q$ are in the same conjugacy class. Conjugate permutations share many basic properties. It may, for example, be shown that two permutations are in the same conjugacy class if and only if they are of the same type [11, Lemma 3.13].

## 2.2 Group theory

Much of what is known about Latin squares is inextricably tied to group theory. A number of basic notions from group theory are therefore reviewed in this section, so as to facilitate easier understanding of the material in the remainder of the thesis.

Consider a set of elements $G$. A *binary operation* acting on this set is a mapping $\circ : G \times G \mapsto G$, In other words, a binary operation maps an ordered pair of elements of $G$ to some other element in $G$ [2, Definition 2.7.1]. The *composition* or *product* of two elements $g_1, g_2 \in G$ is denoted by $g_1 \circ g_2 = g_1 g_2$. The set $G$ is said to be *closed* under the binary operation $\circ$ if the product $g_1 g_2$ is an element of $G$ for all $g_1, g_2 \in G$. An element $e \in G$ with the property that $e \circ g = g \circ e = g$ for all $g \in G$ is called the *identity element* of $G$ and is usually denoted by the symbol $e$. If, for some pair $g_1, g_2 \in G$, $g_1 \circ g_2 = g_2 \circ g_1 = e$, then $g_1$ is the *inverse* of $g_2$ in $G$ and is denoted by $g_2^{-1}$ (similarly, $g_2$ is the inverse of $g_1$ and is denoted by $g_1^{-1}$). These notions may be used to state the axiomatic conditions for the existence of a group, which may be found in most textbooks on group theory and is presented here following the approach of Allenby [2].

A *group* of cardinality $n$ is an ordered pair $(G, \circ)$, where $G$ is a non-empty set of cardinality $n$ and $\circ$ is a binary relation satisfying the following axioms:

- G1 *Associativity:* $g_1 \circ (g_2 \circ g_3) = (g_1 \circ g_2) \circ g_3$ for all $g_1, g_2, g_3 \in G$;
- G2 *The existence of an identity element:* there exists an $e \in G$ such that $g \circ e = e \circ g = g$ for any $g \in G$; and
- G3 *The existence of inverses:* for every $g \in G$, there exists a unique element of $G$, denoted by $g^{-1}$, such that $g \circ g^{-1} = g^{-1} \circ g = e$.

It may, for example, be verified that the set $\mathbb{Z}_n$, for integer values of $n$, together with the binary operation of addition modulo $n$, form a group. Addition modulo $n$ is associative since regular addition is associative and the element 0 has the property that $g + 0 = 0 + g = g$ for all $g \in \mathbb{Z}_n$. Finally, for any $g \in \mathbb{Z}_n$, the element $b = n - g$ has the property that $b + g = g + b = e$. $(\mathbb{Z}_n, +)$ is therefore an example of a group of cardinality $n$, for all values of $n \in \mathbb{N}$. It is also easy to verify that the set of all permutations of order $n$, together with the composition operation, as defined in the context of permutations in §2.1, fulfil all the requirements of a group. This group is called the *symmetric group of order* $n$ and is denoted by $S_n$.

The *Cayley table*, or *multiplication table*, of a group $(G, \circ)$ of cardinality $n$ contains a succinct representation of the way in which the binary operation $\circ$ acts on $G$ in the form of an $n \times n$ grid, bordered by the elements of $G$, in which the cell in row $g_1$ and column $g_2$ contains the value $g_1 \circ g_2$. The Cayley table of $(\mathbb{Z}_4, +)$ is, for example, given in Table 2.2.

TABLE 2.2: *The Cayley table of the group* $(\mathbb{Z}_4, +)$.

| + | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 3 | 0 |
| 2 | 2 | 3 | 0 | 1 |
| 3 | 3 | 0 | 1 | 2 |

A *quasigroup* is a set of elements $S$, together with a binary operation $\circ$, such that the equations $s_1 \circ x = s_2$ and $y \circ s_1 = s_2$, each have exactly one solution for any $s_1, s_2 \in S$. A *loop* is a

quasigroup with an identity element. The chief difference between groups and quasigroups, or loops, is that a quasigroup does not need to be associative. These notions will have particular relevance to the study of Latin squares in this thesis.

A *complete mapping* of a group, quasigroup or loop is a one-to-one mapping $\theta : G \mapsto G$ such that the mapping $\mu : G \mapsto G$ defined by $\mu(x) = x \circ \theta(x)$ is again a one-to-one mapping of $G$.

## 2.3 Latin squares

Concepts from the preceding two sections will be of assistance when introducing the notion of a Latin square, the combinatorial object which is central to this thesis.

### 2.3.1 Basic definitions

A *Latin square* of order $n$ is commonly defined (see, amongst others, Colbourn and Dinitz [24, Definition 1.1]) to be an $n \times n$ array in which every cell contains a single symbol from an $n$-set $S$, such that each symbol of $S$ occurs exactly once in each row and column.

If, for example, $S$ contains the four suits of playing cards, in other words $S = \{\blacklozenge, \blacktriangledown, \clubsuit, \spadesuit\}$, then the $4 \times 4$ array

$$\begin{bmatrix} \blacklozenge & \heartsuit & \clubsuit & \spadesuit \\ \spadesuit & \blacklozenge & \heartsuit & \clubsuit \\ \clubsuit & \spadesuit & \blacklozenge & \heartsuit \\ \heartsuit & \clubsuit & \spadesuit & \blacklozenge \end{bmatrix}$$

is an example of a Latin square of order 4.

Let $S(\boldsymbol{L})$ denote the symbol set of a Latin square $\boldsymbol{L}$ and let $R(\boldsymbol{L})$ and $C(\boldsymbol{L})$ denote its row and column indexing sets, respectively. For any $i \in R(\boldsymbol{L})$ and $j \in C(\boldsymbol{L})$, define $\boldsymbol{L}(i,j) \in S(\boldsymbol{L})$ as the element in the $i$-th row and the $j$-th column of $\boldsymbol{L}$. In the remainder of this thesis it is assumed that $R(\boldsymbol{L}) = C(\boldsymbol{L}) = S(\boldsymbol{L}) = \mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ for a Latin square $\boldsymbol{L}$ of order $n$, without any subsequent loss of generality.

The *transpose* of $\boldsymbol{L}$, denoted by $\boldsymbol{L}^T$, is the Latin square for which $\boldsymbol{L}^T(j,i) = \boldsymbol{L}(i,j)$ for all $i \in R(\boldsymbol{L})$ and $j \in C(\boldsymbol{L})$. The *k-th diagonal* of $\boldsymbol{L}$ is the set of entries $\{((k+i) \mod n, i) \mid i \in \mathbb{Z}_n\}$ for some $k \in \mathbb{Z}_n$ and the 0-th diagonal of $\boldsymbol{L}$ is simply referred to as the *main diagonal* of $\boldsymbol{L}$. Any row or column in which all of the entries of $S(\boldsymbol{L})$ appear in numerical order, *i.e.* $0, 1, \ldots, n-1$, is said to be in *natural order*.

A Latin square may also be defined as an $n \times n$ array with the additional property that every row and column is a permutation of the elements of $S(\boldsymbol{L})$. Let $\boldsymbol{L}(i)$ and $\boldsymbol{L}^T(j)$ denote the $i$-th row and the $j$-th column of the Latin square $\boldsymbol{L}$, respectively (note that the $j$-th column of $\boldsymbol{L}$ is, by definition, also the $j$-th row of $\boldsymbol{L}^T$). Then $\boldsymbol{L}(i)$ may be expressed as the permutation

$$\boldsymbol{L}(i) = \begin{pmatrix} 0 & 1 & \ldots & n-1 \\ \boldsymbol{L}(i,0) & \boldsymbol{L}(i,1) & \ldots & \boldsymbol{L}(i,n-1) \end{pmatrix}.$$

It is clear that every element $k \in \mathbb{Z}_n$ is mapped to a distinct element $\boldsymbol{L}(i,k) \in S(\boldsymbol{L})$ by every permutation in the set of row permutations $\{\boldsymbol{L}(i) \mid i \in \mathbb{Z}_n\}$ in order to prevent the repetition of symbols in column $k$. A similar observation holds for the set of column permutations, $\{\boldsymbol{L}^T(j) \mid j \in \mathbb{Z}_n\}$.

Although Latin squares were studied by Leonhard Euler as early as 1782, the British mathematician Arthur Cayley was first to notice, nearly a century later, that the multiplication table (or *Cayley table*) of a group is an appropriately bordered Latin square. When the abstract concept of a group was generalised to *quasigroups* and *loops* during the 1930s, Latin squares again emerged as the corresponding Cayley tables, as is evident from the following result which may be found in Dénes and Keedwell [29, Theorem 1.1.1].

**Theorem 2.1** (**[29]**). *The Cayley table of a quasigroup is a Latin square.*

For any Latin square $\boldsymbol{L}$, the *underlying quasigroup* of $\boldsymbol{L}$ is the group $(G, \circ)$ where $a \circ b = c$ if $\boldsymbol{L}(a,b) = c$. In the case where the first row and column of $\boldsymbol{L}$ both appear in natural order, $\boldsymbol{L}$ is said to be a *reduced Latin square* or *in standardised form* [29, p.105]. The element 0 in the underlying quasigroup of a reduced Latin square $\boldsymbol{L}$ is therefore the identity element of the quasigroup $(G, \circ)$ so that $(G, \circ)$ may be referred to as the *underlying loop* of $\boldsymbol{L}$. Quasigroups and loops are examples of a more primitive mathematical structure called a groupoid, in which every ordered pair of elements uniquely determines a product. The Cayley table of the group $(\mathbb{Z}_n, +)$ provides such a reduced form Latin square for any $n \in \mathbb{Z}$. For example, the reduced Latin square

$$\boldsymbol{L}_{2.1} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 0 \\ 2 & 3 & 4 & 5 & 0 & 1 \\ 3 & 4 & 5 & 0 & 1 & 2 \\ 4 & 5 & 0 & 1 & 2 & 3 \\ 5 & 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

of order 6 is the Cayley table of $(\mathbb{Z}_6, +)$. The Cayley table of the group $(\mathbb{Z}_n, +)$ is also an example of a *symmetric* Latin square, that is, a Latin square such that $\boldsymbol{L}(i,j) = \boldsymbol{L}(j,i)$ for all $i \in R(\boldsymbol{L}), j \in C(\boldsymbol{L})$.

In addition to symmetry, a Latin square $\boldsymbol{L}$ may also exhibit various other structural properties. It may, for example, contain an $s \times s$ subarray that is itself also a Latin square, called a *subsquare* of side $s$. If $R' \subset R(\boldsymbol{L})$ and $C' \subset C(\boldsymbol{L})$ are subsets of the row and column indexing sets, both of cardinality $s$, then a subsquare is formally defined as the set of entries $\{(i,j) \mid i \in R', j \in C'\}$ in $\boldsymbol{L}$. It is easy to see that, as a subsquare is embedded in a Latin square, a necessary and sufficient condition for the existence of an subsquare of side $s$ is that it contains exactly $s$ different symbols. For example, the Latin square

$$\boldsymbol{L}_{2.2} = \begin{bmatrix} \mathbf{0} & \mathbf{3} & 6 & \mathbf{1} & 5 & 4 & 2 \\ \mathbf{3} & \mathbf{1} & 4 & \mathbf{0} & 2 & 6 & 5 \\ 6 & 4 & 2 & 5 & 1 & 3 & 0 \\ \mathbf{1} & \mathbf{0} & 5 & \mathbf{3} & 6 & 2 & 4 \\ 5 & \underline{2} & 1 & \underline{6} & 4 & 0 & 3 \\ 4 & \underline{6} & 3 & \underline{2} & 0 & 5 & 1 \\ 2 & 5 & 0 & 4 & 3 & 1 & 6 \end{bmatrix}$$

contains at least two disjoint subsquares, a subsquare of side 3 (shown in boldface), defined by $R' = \{0,1,3\}$ and $C' = \{0,1,3\}$, and a subsquare of side 2 (underlined), defined by $R'' = \{4,5\}$ and $C'' = \{1,3\}$. A subsquare of side 2 of a Latin square $\boldsymbol{L}$ is also sometimes called an *intercalate* of $\boldsymbol{L}$ [71].

The relationship between the Cayley tables of quasigroups and Latin squares extend naturally to subquasigroups and subsquares. More specifically, the Cayley table of a subquasigroup $(G', \circ)$ of a quasigroup $(G, \circ)$ is a subsquare of the Latin square formed by the Cayley table of $(G, \circ)$.

Conversely, an appropriately bordered subsquare of the Latin square formed by the Cayley table of $(G, \circ)$ is the Cayley table of a subquasigroup.

Interestingly, the largest possible subsquare of an $n \times n$ Latin square has sides $s \leq \lfloor n/2 + 1 \rfloor$ due to a group theoretic result by Mann and McWorter [59].

Two important notions involving Latin squares are those of transversals and universals. Euler [33] introduced the notion of a *transversal* of a Latin square under the name *formule directrix* and it has also merely been called a *directrix*, notably by Norton [70]. A transversal $V$ of a Latin square $\boldsymbol{L}$ of order $n$, is a set of $n$ distinct, ordered pairs $(i, j)$, one from each row and column of $\boldsymbol{L}$, containing all of the $n$ symbols of $\boldsymbol{L}$ exactly once [24, Definition 1.27]. Transversals are important in many constructions of Latin squares and have close ties to complete mappings in quasigroups, as described in §2.2 and highlighted by the following result, which may be found in [24, p. 345].

**Theorem 2.2 ([24]).** *There is a one-to-one correspondence between the transversals of a Latin square $\boldsymbol{L}$ and the complete mappings of a quasigroup $(G, \circ)$ with $\boldsymbol{L}$ as Cayley table.*

A *universal* $U$ of a Latin square $\boldsymbol{L}$, on the other hand, is a set of $n$ distinct, ordered pairs $(i, j)$, one from each row and column, containing only one symbol of $\boldsymbol{L}$. A universal of $\boldsymbol{L}$ is therefore the set of all the entries containing a single symbol in $\boldsymbol{L}$, a particularly useful concept introduced by Kidd *et al.* [53] in 2012 to facilitate the enumeration of specific classes of Latin squares.

Both transversals and universals may be expressed in permutation form. In a *transversal permutation* $v$, it holds that $v(i) = j$ if $(i, j) \in V$, while in the *universal permutation of (the symbol)* $k$, it holds that $u_k(i) = j$ if $\boldsymbol{L}(i, j) = k$. In the Latin square $\boldsymbol{L}_{2.2}$, for example, the main diagonal $V = \{(0,0), (1,1), \ldots, (5,5)\}$ is clearly a transversal, while the universal of 0 is given by $U_0 = \{(0,0), (1,3), (2,6), (3,1), (4,5), (5,4), (6,2)\}$. The corresponding permutations are $v = \binom{0\ 1\ 2\ 3\ 4\ 5\ 6}{0\ 1\ 2\ 3\ 4\ 5\ 6}$ and $u_0 = \binom{0\ 1\ 2\ 3\ 4\ 5\ 6}{0\ 3\ 6\ 1\ 5\ 4\ 2}$, respectively.

A Latin square containing a transversal in natural order on its main diagonal, like $\boldsymbol{L}_{2.2}$, is said to be *idempotent*. Formally, an idempotent Latin square of order $n$ has $\boldsymbol{L}(i, i) = i$ for all $i \in \mathbb{Z}_n$. A Latin square with a universal on the main diagonal is said to be *unipotent*.

### 2.3.2 Orthogonal Latin squares

According to Colbourn and Dinitz [24, Definition 3.1], two Latin squares of order $n$, $\boldsymbol{L}$ and $\boldsymbol{L}'$, are *orthogonal* if $\boldsymbol{L}(i, j) = \boldsymbol{L}(k, \ell)$ and $\boldsymbol{L}'(i, j) = \boldsymbol{L}'(k, \ell)$ implies that $i = k$ and $j = \ell$. Equivalently, orthogonality implies that every element of $\mathbb{Z}_n \times \mathbb{Z}_n$ appears exactly once among the ordered pairs $(\boldsymbol{L}(i, j), \boldsymbol{L}'(i, j))$ for $i, j \in \mathbb{Z}_n$.

Latin squares were first formally studied by Euler when he considered the so-called "36-Officers problem," asking whether it is possible to arrange thirty-six soldiers of six different ranks and from six different regiments in a square platoon so that every row and column of the platoon contains exactly one soldier of every rank, and one soldier from every regiment [32]. Labelling the ranks and regiments from the symbol set $\mathbb{Z}_n$, it is clear that Euler was attempting to find a pair of orthogonal Latin squares of order 6, where the entry in $\boldsymbol{L}(i, j)$ would indicate the rank of the soldier in position $(i, j)$ and $\boldsymbol{L}'(i, j)$ his regiment. Euler was unable to find such an arrangement of soldiers and continued to propose what has become known as *Euler's Conjecture*, that no pair of orthogonal Latin squares order $n$ exists when $n = 4m + 2$ for integer values of $m$ [32].

Euler's expectation was lent some credence more than a century later when amateur French mathematician Gaston Tarry proved in two papers that a solution to the "36-Officers problem"

(and hence to the special case of Euler's Conjecture where $n = 6$) does, indeed, not exist [90]. Sixty years later, however, pairs of orthogonal Latin squares of order 10 [78] and order 22 [14] were constructed, thereby disproving Euler's Conjecture in general, before Bose *et al.* [13] showed that it is possible to construct such pairs for all cases of Euler's Conjecture, except when $n = 6$.

It should be noted that orthogonality may also be expressed in terms of transversals and universals. Specifically, if $\boldsymbol{L}$ and $\boldsymbol{L}'$ are two orthogonal Latin squares, it is necessary that the entries in every transversal of $\boldsymbol{L}$ correspond to a universal of $\boldsymbol{L}'$ [96, p. 183]. It follows that a Latin square $\boldsymbol{L}$ has an orthogonal mate $\boldsymbol{L}'$ if and only if $\boldsymbol{L}$ has $n$ disjoint universals [29, Theorem 5.1.1], as each of these transversals corresponds to a universal in $\boldsymbol{L}'$. The Latin square $\boldsymbol{L}$ of order $2k$ with $\boldsymbol{L}(i, j) = i + j \pmod{2k}$, which is the Cayley table of the the group $(\mathbb{Z}_{2k}, +)$, is an example of a Latin square without any transversals and therefore has no orthogonal mate. It may, for example, be confirmed that the Latin square $\boldsymbol{L}_{2.1}$ corresponding to the Cayley table of the group $(\mathbb{Z}_6, +)$ contains no transversals.

The notion of orthogonality generalises to sets of Latin squares $\boldsymbol{L}_1, \boldsymbol{L}_2, \ldots, \boldsymbol{L}_k$. Such a set is called a *k-set of mutually orthogonal Latin squares*, abbreviated to $k$-MOLS, if $\boldsymbol{L}_i$ and $\boldsymbol{L}_j$ are orthogonal for all $1 \leq i < j \leq k$. The set of Latin squares

$$\mathcal{M}_{2.1} = \left\{ \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 2 & 3 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \right\},$$

for example, is a 3-MOLS of order 4.

MOLS have been shown to have important applications to coding theory [56], various subfields of statistics including experimental design (notably by RA Fisher in [37] and [38]) and the scheduling of sports tournaments (see, amongst many others, Keedwell [50], Kidd [52] and Robinson [83]).

It is natural to consider the number of Latin squares in the largest possible MOLS of order $n$, denoted by $N(n)$. It is possible to establish an upper bound on $N(n)$ by considering a MOLS with the property that every Latin square has been relabelled so that the first row appears in natural order. There are clearly exactly $n-1$ possible symbols for the first element in the second row of the Latin square and, therefore, at most $n-1$ Latin squares in the MOLS. This informal argument may be formalised (see, for example, Dénes and Keedwell [29, Theorem 5.1.5]) to establish the well-known result that $N(n) \leq n - 1$ for all natural numbers $n > 1$. The example above shows that such an $(n-1)$-MOLS of order $n$, or *complete MOLS*, exists for $n = 4$ and in general, complete MOLS of order $n$ may be constructed whenever $n$ is a prime power[1]. Bruck and Ryser [17] showed that there is also an infinite subset of orders $n \in \mathbb{N}$ for which $N(n) < n-1$.

### 2.3.3 Operations on Latin squares

A topic very close to the central theme of this thesis is the notion of equivalence classes of Latin squares and MOLS. Two Latin squares $\boldsymbol{L}$ and $\boldsymbol{L}'$ of order $n$ are *equal* if $\boldsymbol{L}(i, j) = \boldsymbol{L}'(i, j)$ for all $i, j \in \mathbb{Z}_n$; otherwise they are *distinct*. The appearance of a Latin square may, however, be changed in a number of very natural ways without changing any of its underlying structural

[1] A proof of this result was initially proposed by EH Moore, but is often attributed to RC Bose due to his finding that a complete MOLS of order $n$ exists if and only if there exist a finite projective plane of order $n$ [12]. Finite projective planes, however, fall outside the scope of this study. See Mann [59], Dénes and Keedwell [29, Chapter 8] for further information on the equivalence of finite projective planes and complete MOLS, and Lam [55] for a proof of the non-existence of a finite projective plane of order 10.

properties. Specifically, any of the $n!$ permutations of the elements of $\mathbb{Z}_n$ may be applied to the column, row and symbol indexing sets of a Latin square to generate another, possibly distinct, Latin square. Applying a permutation $p$ to the column indexing set of a Latin square $\boldsymbol{L}$ in which the element $(i, j)$ is mapped to $k$ produces another Latin square $\boldsymbol{L}'$ for which $(i, p(j))$ is mapped to $k$, in other words, $\boldsymbol{L}(i, j) = \boldsymbol{L}'(i, p(j))$. Similarly, when applying a permutation to the row and symbol sets of $\boldsymbol{L}$ to form $\boldsymbol{L}''$ and $\boldsymbol{L}'''$, respectively, it holds that $\boldsymbol{L}''(p(i), j) = \boldsymbol{L}(i, j)$ and $p(\boldsymbol{L}'''(i, j)) = \boldsymbol{L}(i, j)$. For example, applying the permutation $p = \binom{0\ 1\ 2\ 3}{0\ 1\ 3\ 2}$ to the row, column and symbol indexing sets of the Latin square

$$\boldsymbol{L}_{2.3} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 3 & 0 & 1 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

produces the Latin squares

$$\boldsymbol{L}_{2.4} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 0 & 1 & 2 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{L}_{2.5} = \begin{bmatrix} 0 & 1 & 3 & 2 \\ 3 & 0 & 2 & 1 \\ 2 & 3 & 1 & 0 \\ 1 & 2 & 0 & 3 \end{bmatrix} \quad \boldsymbol{L}_{2.6} = \begin{bmatrix} 0 & 1 & 3 & 2 \\ 1 & 0 & 2 & 3 \\ 3 & 2 & 0 & 1 \\ 2 & 3 & 1 & 0 \end{bmatrix},$$

respectively. Combinations of permutations may also be applied to a Latin square. For example, applying the permutation $p$ to the rows of $\boldsymbol{L}_{2.3}$ and $p_c = \binom{0\ 1\ 2\ 3}{3\ 2\ 1\ 0}$ and $p_s = \binom{0\ 1\ 2\ 3}{0\ 2\ 3\ 1}$ to the columns and symbols, respectively, results in the Latin square

$$\boldsymbol{L}_{2.7} = \begin{bmatrix} 1 & 3 & 2 & 0 \\ 3 & 2 & 0 & 1 \\ 0 & 1 & 3 & 2 \\ 2 & 0 & 1 & 3 \end{bmatrix},$$

in which each of the triples $(i, j, \boldsymbol{L}(i, j))$ is replaced by the triple $(p(i), p_c(j), p_s(\boldsymbol{L}(i, j)))$. Permutations applied to the rows, columns and symbols may thus be applied in any order. Multiple permutations may also be applied consecutively to, say, the rows of a Latin square, in which case the resulting operations are equivalent to applying the composition of the permutations. For example, applying a permutation $p$, followed by a permutation $q$, to the rows of a Latin square $\boldsymbol{L}$ has the effect of moving row $i$ first to position $p(i)$ and finally to position $q(p(i))$, which is equivalent to simply applying $q \circ p$ to the rows of $\boldsymbol{L}$. If one supposes that $q = p^{-1}$ it is clear that transformations of Latin squares may be reversed by applying the appropriate inverse permutations. These properties also hold for the columns and symbols of a Latin square. Any Latin square may be transformed to standard form by applying a series of permutations to its rows and columns. For example, $\boldsymbol{L}_{2.3}$ may be transformed to standard form by applying the permutation $\binom{0\ 1\ 2\ 3}{0\ 3\ 2\ 1}$ to its rows.

The six *conjugates* of a Latin square $\boldsymbol{L}$ may be found by applying a permutation uniformly to the set of triples $(i, j, \boldsymbol{L}(i, j))$ for all $i, j \in \mathbb{Z}_n$. Thus applying the permutation $\binom{0\ 1\ 2}{0\ 1\ 2}$ clearly leaves a Latin square invariant, while applying $\binom{0\ 1\ 2}{1\ 0\ 2}$ yields the transpose $\boldsymbol{L}^T$ of $\boldsymbol{L}$. The transformations $\binom{0\ 1\ 2}{0\ 2\ 1}$ and $\binom{0\ 1\ 2}{2\ 1\ 0}$ yield the row and column inverses of $\boldsymbol{L}$, denoted by $\boldsymbol{L}^{-1}$ and ${}^{-1}\boldsymbol{L}$, respectively, while the transformations $\binom{0\ 1\ 2}{1\ 2\ 0}$ and $\binom{0\ 1\ 2}{2\ 0\ 1}$ yield their respective transposes, $(\boldsymbol{L}^{-1})^T$ and $({}^{-1}\boldsymbol{L})^T$. Let $\iota, \tau$ and $\rho$ denote the conjugate operation which leaves a Latin square invariant, replaces a Latin square with its transpose and replaces each row of a Latin square with its inverse, respectively. The composition $\gamma = \tau \circ \rho \circ \tau$ denotes the conjugate operation which replaces every column of a Latin square with its inverse to form ${}^{-1}\boldsymbol{L}$, while $\tau \circ \rho = \rho \circ \gamma$

maps $\boldsymbol{L}^{-1}$ to its transpose $(\boldsymbol{L}^{-1})^T$. Finally, the composition $\tau \circ \gamma = \rho \circ \tau$ is the operation which maps ${}^{-1}\boldsymbol{L}$ to its transpose $({}^{-1}\boldsymbol{L})^T$.

Permutations may also be applied to the row, column and symbol sets of a $k$-MOLS. However, any permutation applied to the row or column indexing sets must be applied to each of the $k$ squares to maintain orthogonality. The symbol set of any Latin square in the $k$-MOLS may be permuted independently of the others without affecting orthogonality. Applying the permutation $p = \binom{0\ 1\ 2\ 3}{0\ 1\ 3\ 2}$ to the rows of $\mathcal{M}_{2.1}$ and the permutation $q = \binom{0\ 1\ 2\ 3}{3\ 1\ 2\ 0}$ to the symbols of $\boldsymbol{L}_0$, for example, yields the 3-MOLS

$$\mathcal{M}_{2.2} = \left\{ \begin{bmatrix} 3 & 1 & 2 & 0 \\ 0 & 2 & 1 & 3 \\ 2 & 0 & 3 & 1 \\ 1 & 3 & 0 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 2 & 3 \\ 2 & 3 & 0 & 1 \\ 1 & 0 & 3 & 2 \\ 3 & 2 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 3 & 2 & 1 & 0 \\ 2 & 3 & 0 & 1 \end{bmatrix} \right\},$$

A set of mutually orthogonal Latin squares in which the first row of every Latin square is in natural order and in which the first column of exactly one of the Latin squares are in natural order is called a *standardized set* [29, p. 159]. The set $\mathcal{M}_{2.1}$ is already a standardised set, while $\mathcal{M}_{2.2}$ may be transformed into a standardised set by applying, for example, the suitable inverses of the operations applied previously to the rows and the symbols of $\boldsymbol{L}_0$.

Analogously to the way in which the conjugates of a single Latin square are found, the $(k+2)!$ conjugates of a $k$-MOLS may be generated by applying a permutation uniformly to the $(k+2)$–tuples $(i, j, \boldsymbol{L}_0(i,j), \ldots, \boldsymbol{L}_{k-1}(i,j))$. Indeed, the conjugates of a single Latin square is the special case of the conjugates of a $k$-MOLS, where $k = 1$. These $(k+2)$-tuples are the columns of what is known as an *orthogonal array of degree* $k+2$ *and order* $n$, denoted $OA(n, k+2)$, which traditionally takes the form

$$\mathcal{OA} = \begin{bmatrix} 0 & 0 & \cdots & (n-1) & (n-1) \\ 0 & 1 & \cdots & (n-2) & (n-1) \\ \boldsymbol{L}_0(0,0) & \boldsymbol{L}_0(0,1) & \cdots & \boldsymbol{L}_0(n-1,n-2) & \boldsymbol{L}_0(n-1,n-1) \\ \vdots & \vdots & & \vdots & \vdots \\ \boldsymbol{L}_{k-1}(0,0) & \boldsymbol{L}_{k-1}(0,1) & \cdots & \boldsymbol{L}_{k-1}(n-1,n-2) & \boldsymbol{L}_{k-1}(n-1,n-1) \end{bmatrix}$$

and has the property that no $2 \times n^2$ subarray contains a repeating column, as this would mean that the corresponding Latin squares are not pairwise orthogonal. The orthogonal array corresponding to $\mathcal{M}_{2.2}$, for example, is

$$\mathcal{OA}_{2.1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \\ 3 & 1 & 2 & 0 & 0 & 2 & 1 & 3 & 2 & 0 & 3 & 1 & 1 & 3 & 0 & 2 \\ 0 & 1 & 2 & 3 & 2 & 3 & 0 & 1 & 1 & 0 & 3 & 2 & 3 & 2 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 & 3 & 2 & 3 & 2 & 1 & 0 & 2 & 3 & 0 & 1 \end{bmatrix}.$$

Uniformly applying a permutation to the tuple $(i, j, \boldsymbol{L}_0(i,j), \ldots, \boldsymbol{L}_{k-1}(i,j))$ is equivalent to reordering the rows of the orthogonal array of a $k$-MOLS. For a 2-MOLS there are 24 potential conjugates, a few of which are of sufficient interest to discuss briefly. The permutation which interchanges the first two elements of the tuple $(i, j, \boldsymbol{L}_0(i,j), \ldots, \boldsymbol{L}_{k-1}(i,j))$ yields the transposes of each of the Latin squares. A permutation which fixes the first two elements while reordering the remaining elements has the effect of reordering the bottom rows of the orthogonal array and therefore changes the order of the Latin squares in the corresponding MOLS.

Orthogonal arrays are also useful because they provide a way of constructing sets of mutually orthogonal Latin squares of specific new orders from existing MOLS. For example, it may be shown that if there exists an $OA(n_1, k)$ and an $OA(n_2, k)$, it is possible to construct an $OA(n_1n_2, k)$ [29, Theorem 11.1.2]. This implies that if sets of $k$-MOLS of orders $n_1$ and $n_2$ exist, it is possible to construct a $k$-MOLS of order $n_1n_2$. This ability to construct orthogonal arrays and MOLS from existing MOLS is often used in proving the existence of sets of mutually orthogonal Latin squares of specific orders. Indeed, it was by finding constructions of 2-MOLS of orders 10 and 22 (and later in general for all orders $4n+2$ where $n \geq 2$) that Bose *et al.* [14] disproved Euler's Conjecture. A number of different methods of constructing MOLS exist, but fall outside the scope of this thesis. The interested reader is referred to Dénes and Keedwell [28, 29] for an introduction to the recursive construction of MOLS.

## 2.4 Chapter summary

The notion of a permutation was defined in §2.1. It was shown that permutations may be ordered lexicographically and that the composition of permutations reveal their cyclical nature. It was illustrated how the cycle structure of a permutation defines its type, and it was mentioned that permutations are in the same conjugacy class if and only if they are of the same type.

The well-known group axioms were stated in §2.2 and the notions of a Cayley table and of a quasigroup were reviewed very briefly.

A concise introduction to the theory of Latin squares was presented in §2.3. The notion of a Latin square was defined in §2.3.1 and mention was made of the link between Latin squares and quasigroups before the notions of transversals and universals were introduced. Orthogonality between Latin squares, and its generalisation to sets of $k$ mutually orthogonal Latin squares, were discussed in §2.3.2. In §2.3.3 the focus fell on the effect of allowing permutations to act on the row, column and symbol indexing set of a Latin square or MOLS without changing its underlying structural properties. The $(k+2)!$ conjugate operations of a $k$-MOLS of order $n$ were also reviewed.

A variety of operations on Latin squares and $k$-MOLS which partition the set of all Latin squares or MOLS into equivalence classes will be considered in the following chapter, and an algorithm for enumerating these equivalence classes will be presented.

# CHAPTER 3

# The enumeration of MOLS

Contents


3.1 The classification of Latin squares . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.2 A historical overview of the enumeration of Latin squares . . . . . . . . . . . . 24
3.3 The enumeration methodology adopted in this thesis . . . . . . . . . . . . . . . . 27
3.4 On the enumerability of larger-order search spaces . . . . . . . . . . . . . . . . . 36
3.5 Chapter summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41


The processes of enumerating equivalence classes of Latin squares and sets of mutually orthogonal Latin squares are the topic of this chapter. A number of transformations, together with their respective transformation classes, are considered in §3.1, followed by a historical review of attempts at enumerating these classes in §3.2. An exhaustive backtracking algorithm for the enumeration of main classes of $k$-MOLS is presented in §3.3. Enumerating these main classes is, however, computationally very expensive for $k$-MOLS of larger orders. The chapter therefore closes with a discussion on techniques for estimating the sizes of these larger enumeration search trees in §3.4.

## 3.1 The classification of Latin squares

As was mentioned in §2.3.1, a Latin square is the Cayley table of a quasigroup; it is therefore an example of a groupoid. In order to classify Latin squares into equivalence classes it is first necessary to consider the operations that may act on them and on groupoids in general. The summary of operations on Latin squares in this section largely follows the description of Dénes and Keedwell [29], except where mentioned otherwise.

An *isotopism*, in the notation of Dénes and Keedwell [29, §1.3], is an operation consisting of an ordered triple of three permutations $(\theta, \varphi, \psi)$ applied to a groupoid of order $n$. There are $(n!)^3$ different isotopisms that may be applied to a groupoid of order $n$, and it may be shown that the set of all isotopisms forms a group if the product of two isotopisms $(\theta_1, \varphi_1, \psi_1)$ and $(\theta_2, \varphi_2, \psi_2)$ is defined as the ordered triple $(\theta_1\theta_2, \varphi_1\varphi_2, \psi_1\psi_2)$ of permutations [29, p.122]. This group is denoted by $I_n$ and it may be shown that $I_n$ is isomorphic to $S_n \times S_n \times S_n$, since each of the permutations $\theta, \varphi$ and $\psi$ is selected from the symmetric group[1] of order $n$. An isotopism

[1]The symmetric group of order $n$, denoted $S_n$ is the set of all permutations of $\mathbb{Z}_n$. The reader is referred to §2.2 for a brief introduction to basic group theoretic concepts.

in which $\psi$ is the identity permutation, denoted by $\iota$, is called a *principal isotopism*. It is clear that the group of all principal isotopisms is contained in $I_n$ and has a cardinality $(n!)^2$.

An *autotopism* is an isotopism which leaves the groupoid on which it acts invariant. A trivial example of an autotopism is the triple of identity permutations $(\iota, \iota, \iota)$. The set of all autotopisms forms a subgroup of $I_n$. Any two of the operations of an autotopism determines the third, and the number of autotopisms may therefore not exceed $(n!)^2$. A *principal autotopism* is an autotopism for which $\psi = \iota$. Because a principal autotopism is determined by knowledge of either of the non-identity elements, a groupoid of order $n$ admits at most $n!$ principal isotopisms. Furthermore, the set of all principal autotopisms is contained in the set of all principal isotopisms as well as the set of all autotopisms.

Recall that an isomorphism is an isotopism in which $\theta = \varphi = \psi$. It is therefore clear that the set of all *isomorphisms* is contained in the set of all isotopisms. An *automorphism* is an isomorphism which leaves the groupoid on which it acts invariant. Note that the set of all automorphisms is contained in both the set of all isomorphisms and the set of all autotopisms. The only isotopism which is both an autotopism and an automorphism is $(\iota, \iota, \iota)$.

The notion of the conjugates of a groupoid, as they apply specifically to a Latin square, was discussed in §2.3.3 and may be used to form a larger set of transformations which contains all the elements of $I_n$ as subset. Specifically, a *paratopism* is an ordered set of four operations $(\theta, \varphi, \psi, \epsilon)$, where $\theta, \varphi, \psi \in S_n$ and $\epsilon$ is one of the conjugate operations described in §2.3.3. A paratopism may be viewed as an isotopism which additionally allows a conjugate operation or, alternatively, an isotopism is a paratopism for which the conjugate operation is the identity (*i.e.* $\epsilon = \iota$). Some variations on the notion of a paratopism for which any two of $\theta, \varphi$ and $\psi$ are equal have also been studied [53].

The relationships between the sets of transformations described above are summarised graphically in Figure 3.1.

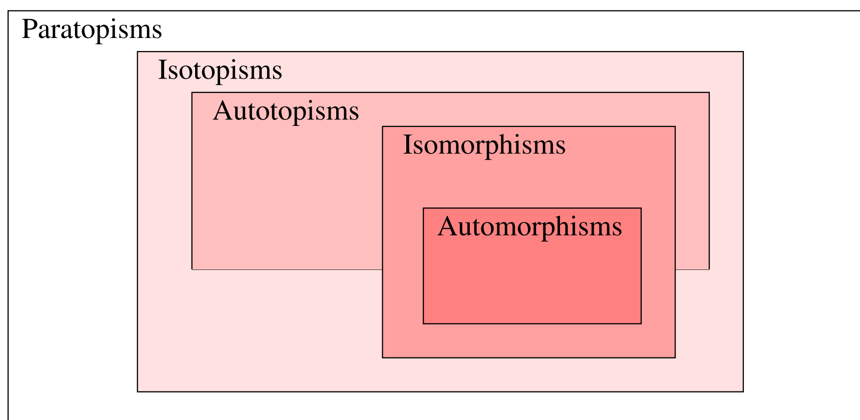



FIGURE 3.1: *The relationships between the transformations of paratopisms, isotopisms, autotopisms, isomorphisms and automorphisms of Latin squares.*

Before proceeding to consider these transformations acting on a Latin square $\boldsymbol{L}$ in further detail, a number of concepts have to be clarified. In what follows, an *operation* is considered to be either a permutation acting on $R(\boldsymbol{L})$, $C(\boldsymbol{L})$ or $S(\boldsymbol{L})$, or a conjugate operation, in other words a permutation acting on the elements of $T(\boldsymbol{L})$. A combination of such operations is henceforth called a *transformation*. If a transformation acting on the set of all distinct Latin squares of order $n$, denoted by $\Omega_n$, forms an equivalence class, this class is called a *transformation class*. A transformation class containing a specific Latin square $\boldsymbol{L}$ is referred to as the *transformation*

*class generated by $\boldsymbol{L}$.*

Furthermore, note that all the transformations discussed above may be defined by a set of four operations $(\theta, \varphi, \psi, \epsilon)$, where $\theta, \varphi, \psi \in S_n$ and $\epsilon$ is a conjugate operation. In the case of a Latin square $\boldsymbol{L}$, $\theta$ may be defined to be the operation acting on the rows of $\boldsymbol{L}$, while $\varphi$ and $\psi$ are operations acting on the columns and symbols of $\boldsymbol{L}$, respectively. Alternatively, following the notation of Kidd [53], the *type* of a transformation applied to a Latin square $\boldsymbol{L}$ may be specified somewhat more flexibly as a tuple $(\pi_{t_1}, \ldots, \pi_{t_k}, \epsilon)$, where $t_i \in \{r, c, s, rc, rs, cs, rcs\}, 1 \leq k \leq 3$ and $\epsilon$ is a conjugate operation. Here $\pi$ represents a specific permutation of the same order as $\boldsymbol{L}$ and the subscript indicates whether it is applied to the rows, columns or symbols of $\boldsymbol{L}$, or to some combination of them. A paratopism, for example, is a transformation of the type $(\pi_r, \pi_c, \pi_s, \epsilon)$ since there are no restrictions on the operations applied to $R(\boldsymbol{L}), C(\boldsymbol{L})$ or $S(\boldsymbol{L})$, or on the conjugate $\epsilon$. An isomorphism acting on $\boldsymbol{L}$, on the other hand, is of the type $(\pi_{rcs}, \iota)$ since the same permutation is applied to the rows, columns and symbols of $\boldsymbol{L}$ and the conjugate is restricted to the identity operation. In this case, and for all isotopisms, the conjugate may be omitted from the definition of a type because it is restricted to the identity operation, as may any of the subscripts $r, c$ or $s$.

If there exists a paratopism transforming a Latin square $\boldsymbol{L}$ to another Latin square $\boldsymbol{L}'$ of the same order, then $\boldsymbol{L}$ is *paratopic* to $\boldsymbol{L}'$, denoted by $\boldsymbol{L} \sim \boldsymbol{L}'$. Furthermore, $\boldsymbol{L}$ and $\boldsymbol{L}'$ are said to be in the same *main class*. Since any Latin square $\boldsymbol{L}$ is a member of exactly one main class, the set of all Latin squares is partitioned into main classes. Three Latin squares of order 4 may be seen in Figure 3.2. In this example, $\boldsymbol{L}_{3.2}$ is paratopic to $\boldsymbol{L}_{3.1}$ by the $(\pi_{rcs})$-transformation $\binom{0\ 1\ 2\ 3}{0\ 1\ 3\ 2}$, but $\boldsymbol{L}_{3.2}$ and $\boldsymbol{L}_{3.3}$ are not paratopic and reside in separate main classes.

$$\boldsymbol{L}_{3.1} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{bmatrix} \qquad \boldsymbol{L}_{3.2} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 3 & 0 & 2 \\ 2 & 0 & 3 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \qquad \boldsymbol{L}_{3.3} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

FIGURE 3.2: *Three Latin squares of order 4.*

Kidd [53, Definition 4.1.4] introduced the notion of a CS-paratopism, which is a $(\pi_r, \pi_{cs}, \rho)$-transformation (recall from §2.3.3 that $\rho$ denotes the permutation $\binom{0\ 1\ 2}{0\ 2\ 1}$ applied to $T(\boldsymbol{L})$). The $(\pi_r, \pi_{cs}, \rho)$-transformation class is called the *CS-paratopy class*, and two Latin squares in the same CS-paratopy class are *CS-paratopic*. Similar transformations may be found by replacing the prefixes "CS" and "$(\pi_r, \pi_{cs}, \rho)$" by "RC" and "$(\pi_s, \pi_{rc}, \tau)$", or "RS" and "$(\pi_c, \pi_{rs}, \gamma)$," where $\tau$ and $\gamma$ are the respective conjugate operations from §2.3.3, but these transformations have been shown to be equivalent to RC-paratopisms [53, Proposition 4.1.1].

If there exists an isotopism (isomorphism) transforming a Latin square $\boldsymbol{L}$ to another Latin square $\boldsymbol{L}'$ of the same order, then $\boldsymbol{L}$ is *isotopic* (*isomorphic*) to $\boldsymbol{L}'$, denoted by $\boldsymbol{L} \simeq \boldsymbol{L}'$ ($\boldsymbol{L} \cong \boldsymbol{L}'$). Clearly, if two Latin squares are isotopic or isomorphic, then their underlying quasigroups share the same equivalence relation. The Latin square $\boldsymbol{L}$, together with all its isotopes, form an isotopy class. Isotopy classes are disjoint and each isotopy class is fully contained in a main class generated by the same square. An *isomorphism class* consists of a Latin square $\boldsymbol{L}$ and all its isomorphic Latin squares, and these classes are, in turn, fully contained in isotopy classes. The Latin square

$$\boldsymbol{L}_{3.4} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 1 & 0 \\ 3 & 2 & 0 & 1 \end{bmatrix},$$

for example, is isomorphic to $\boldsymbol{L}_{3.1}$ by the operation $\binom{0\ 1\ 2\ 3}{0\ 2\ 1\ 3}$ and therefore not isotopic to $\boldsymbol{L}_{3.3}$.

TABLE 3.1: *The chief transformations in the classification of Latin squares, together with their types and transformation classes.*

| Transformation | Tuple | Type | Transformation class |
|---|---|---|---|
| Paratopism | $(\theta, \varphi, \psi, \epsilon)$ | $(\pi_r, \pi_c, \pi_s, \epsilon)$ | Main |
| RC-paratopism | $(\theta, \theta, \psi, \epsilon)$ | $(\pi_{rc}, \pi_s, \epsilon)$ | RC-paratopism |
| Isotopism | $(\theta, \varphi, \psi, \iota)$ | $(\pi_r, \pi_c, \pi_s)$ | Isotopy |
| Isomorphism | $(\pi_r, \pi_c, \pi_s, \epsilon)$ | $(\pi_{rcs})$ | Isomorphism |

The types of transformations and their respective transformation classes described above are summarised in Table 3.1.

The extension of these transformations and transformation-classes to sets of mutually orthogonal Latin squares is of particular interest in this thesis. The type of transformation for a $k$-MOLS of order $n$ includes the symbols $\pi_{t_i}^{(j)}$ or $\epsilon^{(j)}$ if the operation $\pi_{t_i}$ or $\epsilon$ is to be applied to $\boldsymbol{L}_j$, respectively, for all $0 \leq j \leq k-1$. In addition to conjugate operations on individual squares within the $k$-MOLS, the symbol $\delta$ is used to indicate that the transformation may contain any of the $(k+2)!$ conjugate operations which act on the entire set of $k$ mutually orthogonal Latin squares. A paratopism of a $k$-MOLS is therefore a $(\pi_r^{(0,1,\ldots,k-1)}, \pi_c^{(0,1,\ldots,k-1)}, \pi_s^{(0)}, \ldots, \pi_s^{(k-1)}, \delta)$-transformation, and the main class of $k$-MOLS is a $(\pi_r^{(0,1,\ldots,k-1)}, \pi_c^{(0,1,\ldots,k-1)}, \pi_s^{(0)}, \ldots, \pi_s^{(k-1)}, \delta)$-transformation class. The set of three mutually orthogonal Latin squares of order 4,

$$\mathcal{M}_{3.1} = \left\{ \begin{bmatrix} 1 & 3 & 0 & 2 \\ 0 & 2 & 1 & 2 \\ 3 & 1 & 2 & 0 \\ 2 & 0 & 3 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 3 & 0 & 2 \\ 3 & 1 & 2 & 0 \\ 2 & 1 & 3 & 0 \\ 0 & 2 & 1 & 3 \end{bmatrix}, \begin{bmatrix} 3 & 2 & 0 & 1 \\ 1 & 0 & 2 & 3 \\ 0 & 1 & 3 & 2 \\ 2 & 3 & 1 & 0 \end{bmatrix} \right\},$$

is, for example, paratopic to, and resides in the same main class as, the 3-MOLS $\mathcal{M}_{2.1}$ of order 4 in §2.3.2. This may be demonstrated, for example, by a $(\pi_r, \pi_s^{(1,2)}, \tau)$-transformation, where the permutation $\binom{0\ 1\ 2\ 3}{2\ 1\ 0\ 3}$ is applied to the row set of $\mathcal{M}_{3.1}$ and where $\binom{0\ 1\ 2\ 3}{0\ 2\ 3\ 1}$ is applied to the symbol sets of $\boldsymbol{L}_1$ and $\boldsymbol{L}_2$.

The following section contains an historical overview of previous enumeration attempts for certain transformation classes of Latin squares and sets of orthogonal Latin squares.

## 3.2 A historical overview of the enumeration of Latin squares

In addition to posing his now-famous question about the existence of a set of two orthogonal Latin squares of order 6, Leonhard Euler was the first to consider the enumeration of Latin squares and showed that there is one reduced Latin square of order 2, one of order 3, four distinct reduced squares of order 4 and fifty six of order 5 [33].

These values were subsequently confirmed by Cayley [21] in 1890, who also provided a formula for determining the number of possible second rows of a Latin square if the first row is assumed to be in natural order. Frolov [42] proceeded, in that same year, to enumerate the 9 408 reduced Latin squares of order 6 and claimed that there are 221 276 160 reduced Latin squares of order 7, although this value was later found to be incorrect. He additionally proposed recurrence relations for the number of reduced Latin squares of a given order, as well as the total number of Latin squares of order $n$ in terms of the reduced Latin squares of order $n$ and the number of Latin squares generated by the cyclic group of order $n$.

The next development in the enumeration of Latin squares occurred when MacMahon [58] published a full algebraic solution for Latin squares of finite order in 1898. His algebraic expression contains terms for each of the ways in which the Latin square can conceivably be completed and those terms which are in conflict with the requirements of completing a Latin square are eventually cancelled out. The manipulation of this algebraic expression is considered to be more difficult than the direct enumeration by exhaustive searches [29, p. 141], and the majority of subsequent work has therefore focused on exhaustive searches.

In 1900, Tarry [90] partitioned the reduced Latin squares of order 6 into seventeen families (twelve of these families were isotopy classes and five families were the unions of pairs of isotopy classes which are equivalent when taking the transpose) and successfully showed that none of these families admits orthogonal mates, thereby proving the special case of Euler's conjecture for Latin squares of order 6. There is some evidence that the non-existence of a 2-MOLS of order 6 was established approximately sixty years earlier by an assistant to a German astronomer [45], but none of his works remain in existence.

In 1930, Schönhardt [86] correctly counted two isotopy classes of reduced Latin squares order 5 and twenty two isotopy classes of order 6. He was also among the first to investigate isomorphism classes of Latin squares and showed that there are six isomorphism classes of order 5 and 109 of order 6.

Later papers by Fischer and Yates [39] and by Norton [71] contained attempts at classifying Latin squares into equivalence classes based on the nature of their main diagonals. These authors introduced the notion of an intercalate (see §2.3.1) and called the isotopy classes and main classes of Latin squares *transformation sets* and *species*, respectively. In 1934, Fisher and Yates found 9 048 reduced Latin squares of order 6 which may be arranged into twelve main classes and twenty two isotopy classes, ten of which may be paired up since they are equivalent after applying the transpose operation, as previously found by Tarry and Schönhardt. Norton [71] classified Latin squares of order 7 according to the numbers of intercalates that they admit and, in 1939, found 146 main classes of Latin squares of order 7 and 16 927 968 distinct reduced Latin squares of order 7. The number of intercalates of a Latin square is of a class of properties of particular interest in the enumeration of Latin squares which is preserved under paratopisms (recall, from §2.3.1, that these properties are called *main class invariates*).

In 1948, Sade [85] used a different method to enumerate the reduced Latin squares of order 7 and found 16 942 020 such squares. The discrepancy with Norton's above-mentioned count was later shown to be the result of Norton miscounting a single main class — there are, in fact, 147. Sade exhaustively constructed Latin squares row-by-row and his method is of interest in the development of an enumeration algorithm adopted in this thesis. His ideas were also adapted for computer use by Wells [97] in 1968, who confirmed his number of reduced Latin squares of order 7, counted 535 281 401 856 reduced Latin squares of order 8 and estimated that they may be partitioned into at least 250 000 main classes.

Brown [16] completed the enumeration of isotopy classes of Latin squares of orders $n \leq 8$ in 1968 by finding 1 676 257 isotopy classes of Latin squares of order 8.

A number of authors have also attempted the enumeration of sets of orthogonal Latin squares. Initial work was prompted by Euler's Conjecture and dealt with the existence of orthogonal sets of Latin squares. Much later, complete sets of orthogonal Latin squares were considered (recall from §2.3.2 that a set of orthogonal Latin squares of order $n$ is complete if it contains $n-1$ Latin squares). Since every complete set of orthogonal Latin squares is equivalent to exactly one finite projective plane of order $n$ (see Bose [12] for further details), many researchers have focussed their enumeration efforts for Latin squares on finite projective planes. It is well-known

that finite projective planes exist whenever $n$ is a prime power. Bruck and Ryser [17] have also shown that there are infinitely many orders for which no finite projective plane exists. A number of authors[2] have contributed to the current knowledge that there is, up to isomorphism, only one finite projective plane of each of the orders $2 \leq n \leq 8$; correspondingly, the $(n-1)$-MOLS of order $n$ for $2 \leq n \leq 8$, $n \neq 6$ each consists of a single isotopy class. There are also four isomorphism classes of projective planes of order 9 [36], and none of order 10 [55].

Owens and Preece [76] found in 1995 that, if one were to extend the definition of an isotopy class to a set of $k$ mutually orthogonal Latin squares, the 8-MOLS of order 9 may be partitioned into nineteen isotopy classes. Burger *et al.* [53] enumerated the main classes of $k$-MOLS of orders $3 \leq n \leq 8$ for all $k < n$ in 2011, as well as those of self-orthogonal Latin squares[3] up to order 10 [20]. The main class representatives of 2-MOLS of orders $3 \leq n \leq 8$ are also available on McKay's website [61]. Finally, McKay *et al.* [62] have argued heuristically that there are approximately $10^{15}$ distinct 2-MOLS of order 10, although the number of main classes remains unknown. The numbers of main classes $k$-MOLS for orders $n \in \{3, 4, \ldots, 10\}$ are summarised in Table 3.2.

Table 3.2: *The number of main classes of $k$-MOLS of order $n \in \{3, 4, \ldots, 10\}$*

| | $k$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $n$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 1 | | | | | | | |
| 4 | 1 | 1 | | | | | | |
| 5 | 1 | 1 | 1 | | | | | |
| 6 | 0 | 0 | 0 | 0 | | | | |
| 7 | 7 | 1 | 1 | 1 | 1 | | | |
| 8 | 2 165 | 39 | 1 | 1 | 1 | 1 | | |
| 9 | $\geq 1$ | $\geq 1$ | $\geq 1$ | $\geq 1$ | $\geq 1$ | $\geq 1$ | 7 | |
| 10 | $\geq 1$ | ? | ? | ? | ? | 0 | 0 | 0 |

It is perhaps fitting to conclude this section on the history of the enumeration of Latin squares by mentioning a topic which has led to many recent investigations into orthogonal Latin squares, namely the size of the largest set of orthogonal Latin squares of order $n$, denoted by $N(n)$ [62]. It is clear that $N(n)$ has been determined for all $n \leq 9$ and that $2 \leq N(10) \leq 6$. Determining whether or not there exists a 3-MOLS of order 10 (and thereby deciding whether $N(10) = 2$ or $N(n) \geq 3$) is a celebrated open problem in design theory.

It is natural to ask, given the number of main classes of $k$-MOLS of order $n$, whether this information is of assistance in the enumeration of $(k+1)$- or $(k-1)$-MOLS of order $n$.

After perhaps the most comprehensive search for 3-MOLS of order 10 to date, McKay *et al.* [62] concluded that it is computationally infeasible, using current computing technology, to construct 3-MOLS of order 10 by attempting to add an additional Latin square to pairs of orthogonal Latin squares of order 10. It is expected in general that knowledge of the main classes of $(k-1)$-MOLS of order $n$ will be of little assistance in the enumeration of main classes of $k$-MOLS of order $n$. Indeed, the time required to enumerate main classes of mutually orthogonal Latin squares of a specific order $n$ decreases as $k$ increases. Therefore, even if it were somehow possible use class representatives of $(k-1)$-MOLS for the generation of the main classes of $k$-MOLS, this approach

[2]The interested reader is referred to Veblen and Wedderburn [95] for results on orders 2, 3 and 4, MacInnes [57] for order 5, Bose and Nair [15] for order 7 and Hall *et al.* [46] for order 8.

[3]A Latin square $\boldsymbol{L}$ is *self-orthogonal* if it is orthogonal to its transpose $\boldsymbol{L}^T$.

would still require that the most computationally intensive of the enumerations be completed first. Furthermore, although the existence of a $k$-MOLS of order $n$ implies the existence of all smaller sets of mutually orthogonal Latin squares, it is not possible to determine the number of main classes of $\ell$-MOLS of order $n$ for $\ell < k$ from class representatives of $k$-MOLS of order $n$. Independent enumerations are therefore required for 2-MOLS of order $n$, for 3-MOLS of order $n$, and so on.

One of the objectives in this thesis is to sow the seeds of a potential, eventual contribution towards the celebrated question on the existence of 3-MOLS of order 10 by designing a distributed algorithm for enumerating main classes of MOLS. The nature of such enumerations forms the focus of discussion in the following section.

## 3.3 The enumeration methodology adopted in this thesis

As mentioned in the preceding section, Sade [85] was among the first researchers to enumerate classes of Latin squares by exhaustively constructing Latin squares row-by-row. This was done by successively constructing the $k \times 7$ Latin rectangles for $k = 1, 2, \ldots, 7$ which are in separate isotopy classes. The $(k+1) \times 7$ Latin rectangles were formed by adding rows to the respective $k \times 7$ Latin rectangles and eliminating those that have equivalent mates under permutations to the row, column and symbol sets. The total number of squares was found by summing over the numbers of equivalent squares and their resulting completions. This method may be illustrated by considering the enumeration of reduced Latin squares of order 4. In this case, the first row must consist of the elements of $\mathbb{Z}_n$ in natural order, after which there are three possible second rows, specifically

$$\boldsymbol{L}_{3.5} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}, \quad \boldsymbol{L}_{3.6} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 3 & 0 & 2 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \text{ and } \boldsymbol{L}_{3.7} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix},$$

where the symbol "·" is used as a placeholder representing an as yet unspecified Latin square entry. However, applying the permutation $\binom{0\ 1\ 2\ 3}{0\ 1\ 3\ 2}$ to the rows and symbols of $\boldsymbol{L}_{3.6}$ shows that it is isotopic to $\boldsymbol{L}_{3.5}$, and so there are two isotopy classes of $2 \times 4$ Latin rectangles. There is only one possible third row for $\boldsymbol{L}_{3.5}$, [2 3 0 1], and therefore only one possible completion to a Latin square. $\boldsymbol{L}_{3.7}$ has two possible third rows, [2 3 0 1] and [2 3 1 0], which may be extended into two possible completions. The feasible completion of $\boldsymbol{L}_{3.5}$ is

$$\boldsymbol{L}_{3.8} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{bmatrix}, \text{ while } \boldsymbol{L}_{3.9} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \text{ and } \boldsymbol{L}_{3.10} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 1 & 0 \\ 3 & 2 & 0 & 1 \end{bmatrix}$$

are the possible completions of $\boldsymbol{L}_{3.7}$. In this case there are four reduced Latin squares of order four in total, since there are two $2 \times 4$ Latin rectangles in the first equivalence class, each of which completes to a single reduced Latin square, and a single Latin rectangle in the second equivalence class which completes to two further reduced Latin squares.

This procedure may be expressed formally as a *backtracking algorithm*, in which a solution is built up one step at a time, and is an example of an *exhaustive search*, since all feasible solutions are pursued. Generally, in backtracking algorithms, *pruning* methods are used to avoid considering

unnecessary options. In Sade's algorithm above, for example, $\boldsymbol{L}_{3.6}$ was pruned away because it is equivalent to $\boldsymbol{L}_{3.5}$, and would thus have the same number of completions as $\boldsymbol{L}_{3.5}$.

A number of distinct combinatorial enumeration problems may, in fact, be solved by the method of backtracking. Generally, in these problems, the solution may be expressed as a list of selected variables $X = [x_0, x_1, \ldots, x_{n-1}]$, where the nature of $x_0, x_1, \ldots$ depends on the problem specifications. These variables may, for example, be binary decision variables representing the selection of a vertex or an edge of a graph. The value of $x_i$ is, however, always selected from a *possibility set* $P_i$ for all $0 \leq i \leq n-1$. Elements are selected successively and for a partial solution $X = [x_0, x_1, \ldots, x_{\ell-1}]$, all members of $P_0 \times P_1 \times \ldots \times P_\ell$ are considered, either explicitly or implicitly, for all $0 \leq \ell \leq n-1$. The search may be visualised as a rooted search tree in which every node represents a partial solution and branching takes place on the elements of the possibility set. The length of the solution list is the depth of the node representing that solution. Typically, the problem constraints mean that only a subset of the elements of $P_i$ is feasible at each level of the search tree — this selection is called the *choice set*, denoted by $C_i$. The partial solution $X = [x_0, x_1, \ldots, x_{\ell-1}, y]$, where $y \in P_i$ but $y \notin C_i$, is not considered, as it, and all further nodes with $X = [x_0, x_1, \ldots, x_{\ell-1}, y]$ as root, are infeasible. The subtree rooted at the partial solution $X = [x_0, x_1, \ldots, x_{\ell-1}, y]$ is said to be *pruned* away and becomes inactive. An active node of the search tree is defined as a partial solution which has feasible children.

A pseudo-code listing of a general backtracking algorithm is presented in Algorithm 3.1 as it is found in Kreher and Stinson [89, §4.2]. Processing a partial solution, as in Step 3, may take any number of forms, such as outputting it to the screen, saving it to memory or comparing it to the current best known solution. It is clear that, if the run-time or complexity of a backtracking algorithm is to be improved, the bulk of such improvements must take place in either the way that the possibility set is computed or in the computation of the choice set.

**Algorithm 3.1**: Backtrack($\ell$)

**input** : An index $\ell$
**global** : A partial solution $X = [x_0, x_1, \ldots]$

```
1 begin
2   if [x_0, x_1, ..., x_{ℓ-1}] is a feasible partial solution then
3     process it
4   Compute C_ℓ
5   for every x ∈ C_ℓ do
6     x_ℓ ← x
7     Backtrack(ℓ + 1)
8 end
```

It is perhaps worthwhile to consider an example of a simple backtracking algorithm before turning to the design of an algorithm taylor-made for the enumeration of sets of mutually orthogonal Latin squares. In a classical combinatorial optimization problem, called the *knapsack problem*, $n$ objects, labelled $o_0, o_1, \ldots o_{n-1}$, are considered, each of which has a weight or cost associated with it, denoted by $w_0, w_1, \ldots w_{n-1}$, as well as a profit or benefit, denoted by $p_0, p_1, \ldots p_{n-1}$. The knapsack problem requires a selection of objects to pack into a knapsack, so as to maximize the total profit of the items in the knapsack while constraining the total weight of the items selected to be at most the capacity of the knapsack, denoted by $M$ [89]. One possible solution method entails associating a binary decision variable $x_i$ with object $o_i$, so that $x_i = 1$ if the object $o_i$ is selected in the final solution, or $x_i = 0$ otherwise, for all $i = 0, \ldots, n-1$. Clearly, the

objective is to maximise $\sum_{i=0}^{n-1} x_i p_i$, subject to $\sum_{i=0}^{n-1} x_i w_i \leq M$. A naive solution may iterate over all possible $n$-tuples $[x_0, x_1, \ldots x_{n-1}]$, test every solution for feasibility and in the process update the best solution found thus far. Algorithm 3.1 provides a way of constructing the $2^n$ possible $n$-tuples if $C_\ell = \{0, 1\}$, in other words, if the choice set consists of the binary variables. Of course, it is likely that not all of the $n$-tuples are feasible. Indeed, it is often possible to recognise early on during the construction of the search tree that a $k$-tuple will be infeasible for some $k < n$. The algorithm may therefore be improved by introducing a pruning rule aimed at eliminating infeasible partial solutions.

Returning to the enumeration of Latin squares, it is easy to see that Sade's algorithm may be cast as a backtracking algorithm in which a solution consists of a list of rows which make up a reduced Latin square and in which pruning takes place on the basis of equivalence classes. This is, however, not the only way to design a backtracking algorithm for the enumeration of mutually orthogonal Latin squares, and Burger *et al.* [20] proposed an alternative approach in which the partial solution list is the ordered list of universals $[u_0^{(0)}, u_0^{(1)}, \ldots, u_{n-1}^{(k-2)}, u_{n-1}^{(k-1)}]$ and in which branching takes place on the inclusion of the next universal, instead of on the inclusion of rows. As Kidd [53, §5.1] describes, the chief difference between the two approaches lies in the fact that, in the first case, rows are inserted subject to the restriction that every partial universal formed in this way must intersect with the partial universals in the other squares at most once, while the second approach is based on the insertion of universals and subject to the constraint that every universal must intersect exactly once with all the universals that have are already been inserted in the other squares. The first approach is thus less restrictive, leading Kidd [53] to believe that the second approach may be an improvement on the first. Indeed, this has been observed to be the case — the insertion of universals has led to faster enumerations and enumeration trees with fewer branches in every one of the enumerations for $k$-MOLS of order $n$ where $n \leq 6$ and $k < n$.

A *partially completed Latin square* of order $n$ is an $n \times n$ array with at most one symbol from the set $\mathbb{Z}_n$ in every position and in which no symbol is repeated in any row or column. A partially completed Latin square of order $n$ may, for example, be obtained by inserting a subset of $\ell \leq n$ universals in which no pair of universals intersects. Note that the Latin rectangles found in the intermediate steps of Sade's enumeration algorithm are also special types of partial Latin squares by this definition, as are completed Latin squares. A partial $k$-MOLS of order $n$ is a set of $k$ pairwise orthogonal partial Latin squares.

For the purposes of this discussion, a partial $k$-MOLS $\mathcal{M}$ of order $n$ consisting of the universals $[u_0^{(0)}, u_0^{(1)}, \ldots, u_i^{(\ell)}]$ is said to be on level $i \cdot \ell$ of the enumeration search tree that will be constructed. The level $i \cdot (k-1)$ is simply referred to as level $i$, since every symbol up to and including $i$ has been inserted into all of the Latin squares of $\mathcal{M}$.

In the enumeration of transformation classes of Latin squares and mutually orthogonal Latin squares, it is sufficient to count a single element of every transformation class. An *ordering* may be imposed on the Latin squares of order $n$ to ensure that there is a single, unambiguous element representing every transformation class. Without loss of generality, this element is selected to be the lexicographically smallest element and is called the *class representative* of a transformation class. The counting of transformation classes is therefore equivalent to the counting of class representatives. More specifically, a Latin square $\boldsymbol{L}$ is defined to be *lexicographically smaller* than a Latin square $\boldsymbol{L}'$, denoted by $\boldsymbol{L} < \boldsymbol{L}'$, if $u_k < u'_k$, where $u_k \in U(\boldsymbol{L})$, $u'_k \in U(\boldsymbol{L}')$ and $u_i = u'_i$ for all $u_i \in U(\boldsymbol{L})$, $u'_i \in U(\boldsymbol{L}')$ and $i < k$. In other words, $\boldsymbol{L}$ is lexicographically smaller than $\boldsymbol{L}'$ if, for some $k \in \mathbb{Z}_n$, the first $k$ universals of $\boldsymbol{L}$ and $\boldsymbol{L}'$ are equal, and $\boldsymbol{L}$ contains a lexicographically smaller universal in the $(k+1)$-th position.

This notion of a lexicographical ordering of members of a transformation class of Latin squares may be extended to sets of $k$ mutually orthogonal Latin squares, where the relation is denoted by the symbol $\prec$. Two $k$-MOLS, $\mathcal{M} = [u_0^{(0)}, u_0^{(1)}, \ldots, u_{n-1}^{(k-2)}, u_{n-1}^{(k-1)}]$ and $\mathcal{M}' = [u_0'^{(0)}, u_0'^{(1)}, \ldots, u_{n-1}'^{(k-2)}, u_{n-1}'^{(k-1)}]$, may be lexicographically ordered by comparing the corresponding universals $u_i^{(j)}$ and $u_i'^{(j)}$ for $i \in \mathbb{Z}_n$ and $j \in \mathbb{Z}_k$ in the order presented until the universals differ, in which case the one $k$-MOLS is lexicographically smaller than the other, or until all universals have been compared, in which case $\mathcal{M}$ and $\mathcal{M}'$ are lexicographically equal and therefore the same $k$-MOLS. The universals of a $k$-MOLS of order $n$ are thus compared symbol-by-symbol so that all the 0-universals in the respective Latin squares are considered before continuing with the 1-universal in the first square, *etc.* Note that for both partial Latin squares and partial MOLS there may be empty universals which have not yet been assigned. If the relationship between two Latin squares or MOLS has not yet been established by the time that the first such empty universal is encountered, the comparison is inconclusive and all further universals are ignored. A lexicographical comparison of the two Latin squares

$$\boldsymbol{L}_{3.11} = \begin{bmatrix} 0 & 1 & \cdot & \cdot \\ \cdot & 0 & 1 & \cdot \\ \cdot & \cdot & 0 & 1 \\ 1 & \cdot & \cdot & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{L}_{3.12} = \begin{bmatrix} 0 & \cdot & 2 & \cdot \\ \cdot & 0 & \cdot & 2 \\ 2 & \cdot & 0 & \cdot \\ \cdot & 2 & \cdot & 0 \end{bmatrix},$$

for example, is inconclusive since the zero universals are equal, and $L_{3.12}$ does not contain a universal corresponding to the 1 symbol. However, if the two 2-MOLS

$$\mathcal{M}_{3.2} = \left\{ \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 2 & 3 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \end{bmatrix} \right\} \text{ and } \mathcal{M}_{3.3} = \left\{ \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 3 & 1 & 2 \\ 1 & 2 & 0 & 3 \\ 2 & 1 & 3 & 0 \\ 3 & 0 & 2 & 1 \end{bmatrix} \right\}$$

of order 4 are compared, the first difference occurs at $u_1^{(1)}$, the respective 1-universals in the second Latin square, implying that $\mathcal{M}_{3.2} \prec \mathcal{M}_{3.3}$. This lexicographical ordering allows for the orderly generation of class representatives by a recursive backtracking algorithm, and the following notions assist in streamlining the computation of the possibility and choice sets.

In §3.2 it was mentioned that invariants play an important role in enumeration algorithms since they may remain unaltered by transformations. In the enumeration of mutually orthogonal Latin squares a very useful invariant is the *relative cycle structure* of two permutations, or in this specific case, universals. For a $k$-MOLS of order $n$, the relative cycle structure of two universals $u_i^{(j)}$ and $u_\ell^{(m)}$ is defined as the cycle structure of

$$u_\ell^{(m)} \circ \left(u_i^{(j)}\right)^{-1}.$$

For a proof that the relative cycle structure is invariant under permutations performed on the rows and columns of a $k$-MOLS or the symbols of any of the individual Latin squares, and also under the conjugate operations which transpose every Latin square in the $k$-MOLS, or changes in their order, the interested reader is referred to [53, p.72–73]. It may further be noted that, due to the orthogonality of the Latin squares in a $k$-MOLS, every relative cycle structure has exactly one fixed point. The relative cycle structure of two universals $\binom{0\,1\,2\,3\,4\,5}{0\,2\,1\,4\,5\,3}$ and $\binom{0\,1\,2\,3\,4\,5}{2\,0\,5\,1\,4\,3}$ is therefore the cycle structure of $\binom{0\,1\,2\,3\,4\,5}{2\,0\,5\,1\,4\,3} \circ \binom{0\,1\,2\,3\,4\,5}{0\,2\,1\,5\,3\,4} = \binom{0\,1\,2\,3\,4\,5}{1\,0\,4\,2\,3\,5}$, which is $z_1 z_2 z_3$.

A further useful notion is that of a *row-reduced* $k$-MOLS of order $n$, or a $k$-MOLS in which the first row of every Latin square is in natural order. Any $k$-MOLS may be transformed to a

row-reduced $k$-MOLS by applying a relevant symbol permutation to each of the Latin squares, and so it is only necessary to consider row-reduced Latin squares when enumerating main classes of MOLS. A row-reduced Latin square also has the property that the universals of every Latin square are lexicographically ordered. Kidd [53, p.73–74] proved the following lemma with respect to relative cycle structures of row-reduced Latin squares.

**Lemma 3.1.** *[53, Lemma 4.3.3] For any $i, \ell \in \mathbb{Z}_n$ and $j, m \in \mathbb{Z}_k$ the universal permutations $u_i^{(j)} \in U(\mathcal{M})$ and $u_\ell^{(m)} \in U(\mathcal{M})$ in a row-reduced $k$-MOLS $\mathcal{M} = \{\boldsymbol{L}_1, \boldsymbol{L}_1, \ldots, \boldsymbol{L}_{k-1}\}$ of order $n$ may be mapped to the universal permutations $v_0^{(0)} \in U(\mathcal{M}')$ and $v_0^{(1)} \in U(\mathcal{M}')$ of a new row-reduced $k$-MOLS $\mathcal{M}' = \{\boldsymbol{L}_1, \boldsymbol{L}_1, \ldots, \boldsymbol{L}_{k-1}\}$ of order $n$, respectively, using a paratopism in such a way that $v_0^{(0)}$ is the identity permutation and $v_0^{(1)}$ is a cycle structure representative.*

Using the facts that the relative cycle structure of any pair of universals of $k$-MOLS of order $n$ is invariant under the operations described above, and that pairs of universals may be mapped to the zero universals in the first two Latin squares by Lemma 3.1, which are exactly those universals considered first when lexicographically ordering MOLS, Kidd derived the following sufficient conditions for identifying when a $k$-MOLS of order $n$ is *not* a class representative.

**Theorem 3.2.** *[53, Theorem 4.3.2] If $\mathcal{M} = \{\boldsymbol{L}_0, \ldots, \boldsymbol{L}_{k-1}\}$ is the lexicographically smallest row-reduced $k$-MOLS of order $n$ in its main class, and if $u_i^{(j)}$ is the universal permutation of $i \in \mathbb{Z}_n$ in $\boldsymbol{L}_j$, then*
*(a) $u_0^{(0)}$ is the identity permutation,*
*(b) $u_0^{(1)}$ is a cycle structure representative, and*
*(c) the relative cycle structure of two universal permutations $u_i^{(j)}, u_\ell^{(m)}$ is not lexicographically smaller than the cycle structure of $u_0^{(1)} \in U(\mathcal{M})$ for all $i, j \in \mathbb{Z}_n$ and $j, m \in \mathbb{Z}_n$.*

It is important to note that the characteristics of a $k$-MOLS in Theorem 3.2 are shared by main class representatives, but that they are not sufficient conditions for identifying main class representatives. Theorem 3.2 is therefore most useful in identifying those row-reduced $k$-MOLS of order $n$ which do *not* have the potential to be completed to a main class representative, thereby providing the first criterion by which partial MOLS may be safely pruned away from an enumeration search tree. Note that any partial $k$-MOLS of order $n$ which passes this first search criterion must have $u_0^{(0)} \in U(\mathcal{M})$ as the identity permutation and $u_0^{(1)} \in U(\mathcal{M})$ as a cycle structure representative.

The second criterion by which a partial MOLS may be pruned away from an enumeration search tree is the existence of a lexicographically smaller paratopic MOLS, because the defining property of a main class representative is that it is the lexicographically smallest MOLS in its main class. By Lemma 3.1, any pair of universals may be mapped to the zero universals of the first two squares and it is clear that the mapping is only necessary for those pairs of universals $(u_i^{(j)}, u_\ell^{(m)}) \in U(\mathcal{M}) \times U(\mathcal{M})$ for $j \neq m$ with the same relative cycle structure as $u_i^{(j)}$ and $u_\ell^{(m)}$ (which is simply the cycle structure of $u_\ell^{(m)}$). After mapping $(u_i^{(j)}, u_\ell^{(m)}) \in U(\mathcal{M}) \times U(\mathcal{M})$ to $(v_0^{(0)}, v_0^{(1)}) \in U(\mathcal{M}') \times U(\mathcal{M}')$, where $\mathcal{M}'$ is the $k$-MOLS resulting from this transformation, it has to be verified that $\mathcal{M}'$ has no paratopes that are lexicographically smaller than $\mathcal{M}$.

Testing for the existence of such a lexicographically smaller MOLS is, however, computationally very expensive since there are potentially $n!$ operations that may be applied to the rows and columns of a MOLS, $k \cdot n!$ potential symbol operations that may be applied to the squares individually, and a further $(k+2)!$ conjugate operations. There are clearly far too many transformations to apply all of them, except for the very smallest values of $n$ and $k$. Fortunately,

because any row-reduced MOLS which passes the first search criterion must have the identity as $u_0^{(0)}$ and a cycle structure representative as $u_0^{(1)}$, only a small subset of these transformations actually have to be considered.

When determining whether a partial $k$-MOLS $\mathcal{M}$ of order $n$ is in the same main class as some lexicographically smaller paratopic partial $k$-MOLS $\mathcal{M}'$ of order $n$, it is clear that $v_0^{(0)} \in U(\mathcal{M}')$ must also be the identity permutation. Similarly, if the conjugate operation is restricted to transposes and the reordering of squares, then $v_0^{(1)} \in U(\mathcal{M}')$ must be the same cycle structure representative as $u_0^{(1)} \in U(\mathcal{M})$. From the fact that $v_0^{(0)}$ is the identity permutation, it may be deduced that the same permutation, say $p$, is to be applied to the rows and columns of $\mathcal{M}$. In order to preserve the cycle structure of $v_0^{(1)}$, the permutation $p$ must have the same cycle structure as $u_0^{(1)}$. More specifically, the relevant operations are those which map the cycles of $u_0^{(1)}$, and all their rotations and reorderings, to the cycle structure representative $u_0^{(1)}$. If the cycle structure of $u_0^{(1)}$ is $z_1 z_2^{n_2} \dots z_p^{n_p}$, for example, then $\prod_{i=1}^{i \leq p} \cdot i^{n_i} n_i!$ operations need to be considered in total. All $(p_{rc}, \pi_0^{(1)}, \dots, \pi_0^{(k-1)}, \epsilon)$-transformations, where $p$ is one of the operations described above, $\pi$ is any permutation of order $n$ and $\epsilon$ is the conjugate which either transposes the Latin squares of $\mathcal{M}$ or reorders them, are therefore investigated to ascertain whether they result in a lexicographically smaller row-reduced $k$-MOLS.

If no such lexicographically smaller paratopic MOLS is found, then $\mathcal{M}$ may not be pruned away from the search tree. The algorithm continues by generating a list of candidate universals to insert into the next Latin square in $\mathcal{M}$ and attempts, in turn, to insert each of the universals into $\mathcal{M}$ while repeatedly testing whether the resulting partial MOLS may be pruned away from the search tree. Whenever all the universals in the candidate list have been considered, the search backtracks by moving one level up in the search tree, removing the previous universal and considering its remaining alternatives. Whenever a completed $k$-MOLS of order $n$ is found, it is compared to the relevant paratopes of its $(k+2)!$ conjugates (recall that, for transformations on partial MOLS, only two of the conjugates are allowable) to test whether it is a main class representative.

Theorem 3.2 stresses the importance of the universal $u_0^{(1)} \in U(\mathcal{M})$ in classifying portions of the enumeration search tree with specific properties. Specifically, two partial $k$-MOLS are in the same *section* of the enumeration search tree if their respective $u_0^{(1)}$ universals are the same cycle structure representative. A psuedo-code description of the enumeration process is given in Algorithm 3.2 and the process is exemplified by the enumeration of 2-MOLS of order 5 in Figure 3.3 [8, 53].

According to Theorem 3.2, $u_0^{(0)}$ must be the identity permutation and $u_0^{(1)}$ a cycle structure representative, of which there are two possibilities for order 5, namely $z_1 z_2^2$ and $z_1 z_4$ (recall that there must be exactly one fixed point to ensure orthogonality with the identity permutation). Where branches become inactive it is indicated in Figure 3.3 that either (a) no candidate universals preserve orthogonality, (b) a lexicographically smaller partial MOLS has been found in the same main class, or (c) a class representative had indeed been discovered. The enumeration search tree consists of two sections and a completed 2-MOLS of order 5 is found in both these sections. However, the one in the second section may be shown to be in the same main class as, and lexicographically larger than, the one in the first section.

Algorithmic implementations should, of course, be tested for accuracy, but this is rarely a simple task. For any given test case, a number of blocks of code, or combinations thereof, may never be visited, often raising doubts about the overall correctness of the algorithms. This problem is largely avoided here due to the fact that the numbers of branches on every level of the

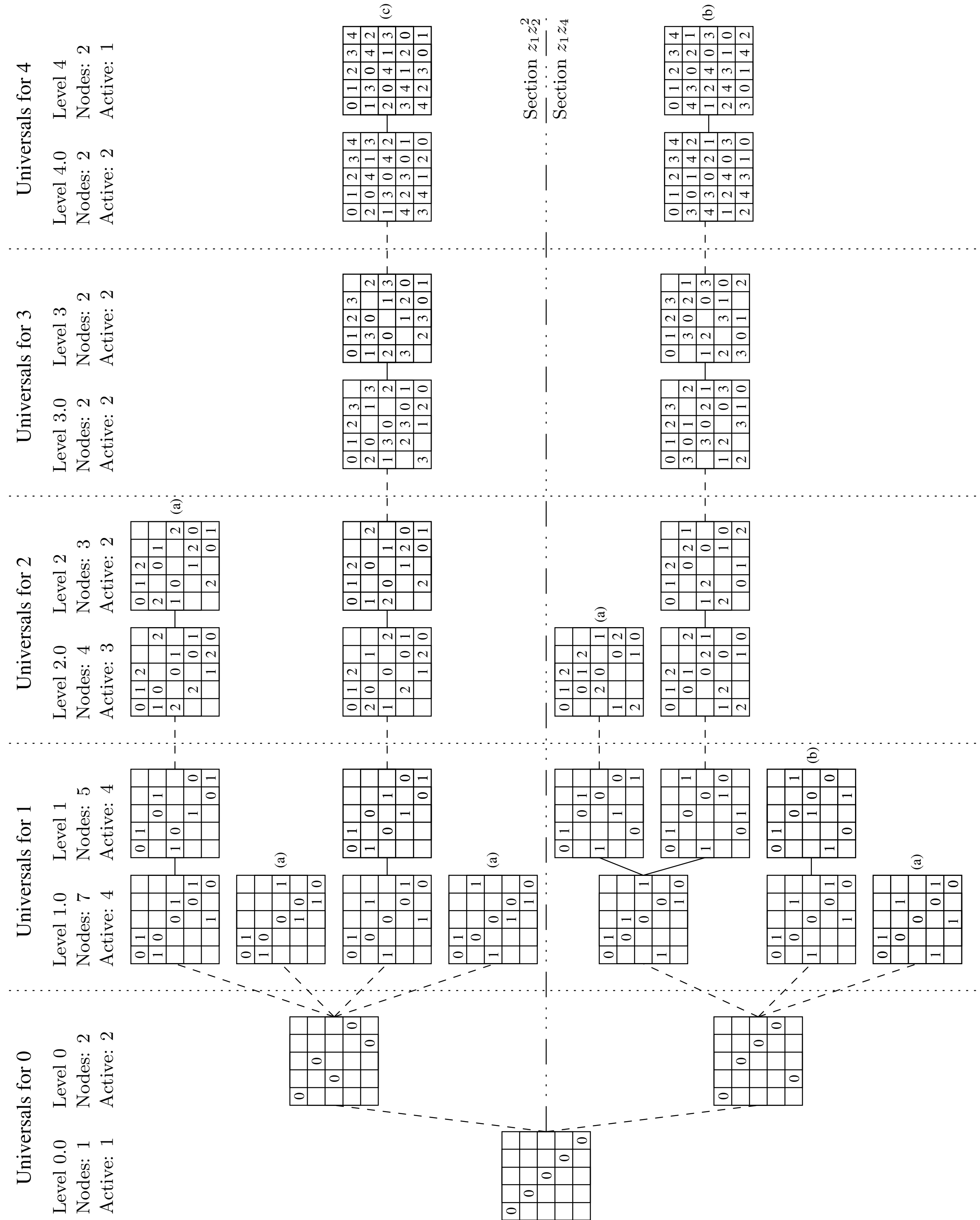



Figure 3.3: *The backtracking enumeration search tree for 2-MOLS of order 5. At every leaf it is either indicated that (a) no candidate universals preserve orthogonality, or that (b) a lexicographically smaller partial MOLS has been found in the same main class, or that (c) a class representative has been found.*

**Algorithm 3.2**: enumerateMOLS($\mathcal{P}$) [8]

**input** : A partial $k$-MOLS $\mathcal{P}$
**output**: All completed class representatives in the subtree of the enumeration search tree rooted at $\mathcal{P}$

```
1  begin
2      if P is complete then
3          if none of the conjugates of P has smaller isotopes then
4              output P as class representative
5              return
6          else
7              return
8      for every candidate universal c do
9          if c preserves orthogonality and P ∪ c is valid by Theorem 3.2(c) then
10             if P ∪ c has no smaller isotope k-MOLS then
11                 enumerateMOLS(P ∪ c)
12 end
```

enumeration search trees for main classes of $k$-MOLS of order $n$ have already been enumerated and published by Kidd [53] for all $n \leq 8$ and all $2 \leq k < n$. The current implementation of Algorithm 3.1 was validated by comparing the active branches on every level of the search tree for main classes of $k$-MOLS of order $n$ (for the above-mentioned values for $n$ and $k$) to those in the literature under the assumption that any significant error in the implementation of Algorithm 3.1 would lead to an invalid traversal of the search tree and an incorrect count.

The number of active nodes encountered on every level of the search trees for main classes of 3-MOLS of order $n \leq 8$ may be seen in Table 3.3 and the values are identical to those found by Kidd [53, Table 5.21]. A more complete summary of the enumeration results for 3-MOLS of order 8, which groups the nodes according to sections of the search tree, may be found in Table 3.4.

The most important aspect of the implementation, apart from its correctness which has already been established, is perhaps its efficiency, or the time that an enumeration takes. The computation time required to fully enumerate the main classes of $k$-MOLS of order $n$ is directly related to the sizes of the corresponding search trees. The enumeration times required for main

TABLE 3.3: *The number of active branches on every level of the enumeration search tree for main classes of 3-MOLS of order $n \leq 8$ produced by Algorithm 3.1.*

| | Nodes on level | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 3 | 1 | 1 | 1 | | | | | |
| 4 | 1 | 1 | 1 | 1 | | | | |
| 5 | 2 | 4 | 2 | 2 | 1 | | | |
| 6 | 3 | 20 | 0 | 0 | 0 | 0 | | |
| 7 | 14 | 10 529 | 3 800 | 3 | 3 | 3 | 1 | |
| 8 | 45 | 15 948 763 | 1 546 241 258 | 18 877 734 | 216 | 168 | 159 | 39 |

Table 3.4: *The number of active nodes in every section and on every level of the enumeration search tree for main classes of 3-MOLS of order 8 produced by Algorithm 3.1, together with the total time that the enumeration of that section required on an Intel i5 processor with 8Gb of RAM.*

| | Level | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Section | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Time ($s$) |
| $z_1 z_2^2 z_3$ | 17 | 12 501 028 | 1 484 518 094 | 18 814 494 | 55 | 23 | 22 | 20 | 775 321 |
| $z_1 z_2^1 z_5$ | 14 | 3 358 273 | 61 708 802 | 63 157 | 97 | 92 | 84 | 17 | 60 011 |
| $z_1 z_3 z_4$ | 5 | 52 059 | 5 283 | 1 | 0 | 0 | 0 | 0 | 93 |
| $z_1 z_7$ | 9 | 37 403 | 9 079 | 82 | 64 | 53 | 53 | 2 | 112 |
| Total | 45 | 15 948 763 | 1 546 241 258 | 318 877 734 | 216 | 168 | 159 | 39 | 835 537 |

classes of $k$-MOLS of orders 7 and 8 may be compared to those of Kidd [53, p. 111–115], as provided in Table 3.5. It is clear that the current implementation was significantly faster than the implementation of Kidd [53], but this may simply be the result of using different hardware systems.

If the effect of the hardware is assumed to remain constant, it may be removed by normalising the data. In this case the computation times are normalised by dividing the computation time required to enumerate the main classes of $k$-MOLS of order $n$ by the computation time required for, say, the enumeration of main classes of 3-MOLS of order $n$. The normalised computation times for enumerating these main classes for orders 7 and 8 may be found in Table 3.6.

In both implementations the enumeration of main classes took the longest for $k = 2$ and $k = 3$. It is noticeable that the current implementation of Algorithm 3.1 performs significantly better

Table 3.5: *A comparison of the computation times (in seconds) required to enumerate main classes of k-MOLS of orders 7 and 8 to those achieved by Kidd [53].*

| | | $k$ | | | | | |
|---|---|---|---|---|---|---|---|
| | $n$ | 2 | 3 | 4 | 5 | 6 | 7 |
| Kidd | 7 | 14 | 12 | 4 | 5 | 32 | |
| | 8 | 3 255 981 | 3 141 695 | 393 412 | 88 815 | 8 481 | 20 394 |
| Current implementation | 7 | 10 | 6 | 2 | 1 | 1 | |
| | 8 | 309 791 | 835 537 | 123 538 | 18 199 | 343 | 113 |

Table 3.6: *A comparison of the normalised computation times required to enumerate main classes of k-MOLS of orders 7 and 8, expressed as a fraction of the computation time required for 2-MOLS of order n, to those achieved by Kidd [53].*

| | | $k$ | | | | | |
|---|---|---|---|---|---|---|---|
| | $n$ | 2 | 3 | 4 | 5 | 6 | 7 |
| Kidd | 7 | 1.16 | 1 | 0.33 | 0.42 | 2.67 | |
| | 8 | 1.03 | 1 | 0.12 | 0.03 | $3 \times 10^{-3}$ | $6 \times 10^{-3}$ |
| Current implementation | 7 | 1.67 | 1 | 0.33 | 0.17 | 0.17 | |
| | 8 | 0.37 | 1 | 0.15 | 0.02 | $4 \times 10^{-4}$ | $1 \times 10^{-4}$ |

for 2-MOLS of order 8 and in general for large values of $k$. Although the enumerations of main classes for larger values of $k$ are relatively easy for $k$-MOLS of order 8 this seems to suggest that, if a higher-order enumeration is to be performed, the implementation may perhaps be most profitably used to pursue the enumeration of main classes of the 7-MOLS or 8-MOLS of order 9.

A technique for estimating the sizes and characteristics of higher-order enumeration search trees in the case where it is impractical to visit all, or even a significant portion of, the nodes in the search tree beforehand is suggested in the next section.

## 3.4 On the enumerability of larger-order search spaces

Consider a rooted search tree $T$ with $\ell$ levels in which the root is the only node on level 0 and the number of nodes on level $i$ is denoted by $n_i$ for all $i = 0, 1, \ldots, \ell - 1$. It is clear that in the special case of a complete $k$-ary tree[4], the number of nodes on level $i$ is $k^i$ and the total number of nodes in $T$, denoted $n(T)$, is simply the sum of the nodes on every level, over all levels. In the slightly more general case where every node on level $i$ has $\gamma_i$ children, it holds that $n_{i+1} = n_i \cdot \gamma_i$ and that the total number of nodes in $T$ is

$$n(T) = 1 + \gamma_0 + \gamma_0\gamma_1 + \gamma_0\gamma_1\gamma_2 + \ldots = 1 + \sum_{i=0}^{\ell-1} \prod_{j=0}^{i} \gamma_j.$$

In 1975, Knuth [54] proposed a technique for estimating $n(T)$ by approximating the average number $\gamma_i$ of children of a node of the tree $T$ on level $i$, by a value $\gamma_i^*$, based on the average findings of a number of trial traversals of random paths downwards in the search tree. Such a downwards traversal, which may be likened to a random "dive" down into the search tree, starts at the root and finds a path by randomly selecting the next node from the children of the current node according to a uniform distribution. The path ends at a node with no children. Knuth also considered selecting the successor of a node with non-uniform probability and showed that the expected value of both techniques is $n(T)$, the actual total number of nodes in the backtracking tree. The tree in Figure 3.4, for example, has a total of 25 nodes and $\overline{\gamma} = [4, 2, 1.5, 0]$. If the four random paths culminating in the nodes labelled $P_1, P_2, P_3$ and $P_4$ in Figure 3.4 are used to approximate $n(T)$, it is estimated that $\overline{\gamma^*} = [4, 2.33, 2.5, 0]$ and that there are approximately 37 nodes in the enumeration search tree — an overestimation. By selecting the four different paths $P_3, P_4, P_5, P_6$, however, the number of nodes in the search tree is underestimated at 17.

Knuth expected this approach to lead to underestimates in cases where the lower levels of the search tree are only visited with a very low probability, in other words, when the majority of the potential paths from the root do not reach the lower levels of the tree. In the specific context of the enumeration search trees encountered in this thesis, however, this underestimate is not likely to be significant as only a very small proportion of the nodes reside in the lower levels of the enumeration search tree. The number of nodes on the respective levels of the search tree for main classes of 3-MOLS of order 8 in Table 3.4 provides an example of this phenomenon. Knuth also noted that this estimation method is likely to yield estimates with a very high variance, especially in cases where a relatively small number of these dives are averaged. Some evidence of this expectation was seen in the example above. Purdom [79] refined Knuth's algorithm and attempted to decrease this variance by investigating more than one child of every node along the path. It is, however, clear that there is a delicate balance between visiting enough nodes

[4]Every node in a $k$-ary tree, except for the leaves, has exactly $k$ children.

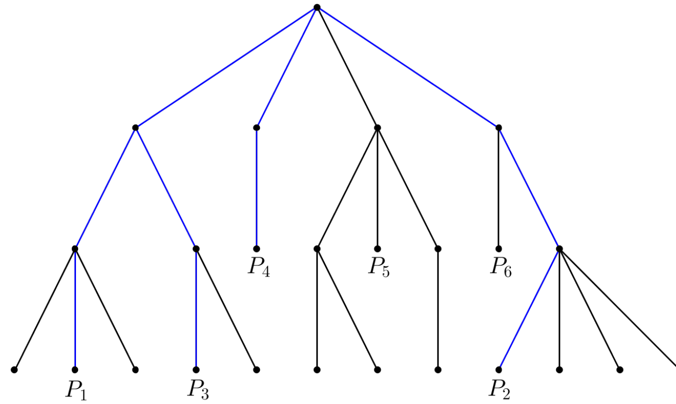



FIGURE 3.4: *The four random paths ending in the nodes labelled $P_1, P_2, P_3$ and $P_4$ may be used to estimate the number of nodes in the tree as 25. An alternative estimate may be calculated by using any subset of paths from the root, such as the set of paths ending at $P_3, P_4, P_5$ and $P_6$, which yields an estimate of 17 nodes. The tree actually contains 25 nodes.*

of the search tree to decrease the variance of the estimator sufficiently, and visiting few enough nodes so that the estimate remains computationally relatively inexpensive.

Two adaptations are proposed to tailor these classical estimators to estimating the sizes of the backtracking search trees produced by Algorithm 3.1.

In the first adaptation proposed, the pruning of candidate universals which takes place in Algorithm 3.1 is decomposed into two stages: during the first stage pruning takes place based whether a candidate universal preserves orthogonality and is valid by Theorem 3.2 (Step 9 of Algorithm 3.1), while during the second stage universals inserted into partial MOLS are pruned away if the resulting partial MOLS have lexicographically smaller isotopes (Step 10 of Algorithm 3.1). It is expected that these two steps of the algorithm, when considered individually, may exhibit properties of the enumeration search trees that assist in obtaining accurate estimates of their sizes. For convenience, these two tests that candidate universals must pass in order to be feasible are referred to as `isOrthogonal` and `isSmallest` in the remainder of this discussion.

It was found that that the average number of candidate universals which pass the `isOrthogonal` test depends sensitively on the cycle structure of $u_0^{(1)}$, but remains largely constant within a given section of the tree. Evidence of this phenomenon may be seen for the 45 active nodes on level 0 of the enumeration search tree for 3-MOLS of order 8 in Figure 3.5 for the two sets of universals $u_1^{(j)}$ and $u_2^{(j)}$ with $j \in \mathbb{Z}_k$. Notice in the figure, that the average number of children of nodes in the same section is very similar in the absence of the `isSmallest` test. Furthermore, the average number of candidate universals which pass the `isOrthogonal` test decreases with every additional universal inserted into the partial MOLS as it becomes progressively harder to preserve orthogonality. This regularity in the number of children of a node of the search tree, as well as its sensitive dependence on the cycle structure of $u_1^{(0)}$ was also observed in the search trees for 3-MOLS of orders 7, 9 and 10.

These properties make it possible to estimate $\gamma_i$ in any given section accurately by only examining a very small random selection of partial MOLS that are in the same section of the tree on level $i$, since the average of this sample is highly likely to be close to the actual value of $\gamma_i$ for that section. In order to further improve the estimate, and to exploit the fact that the variance in the respective estimates $\gamma_i^*$ is likely to be small, the notion of *polling* some of the children of a partial

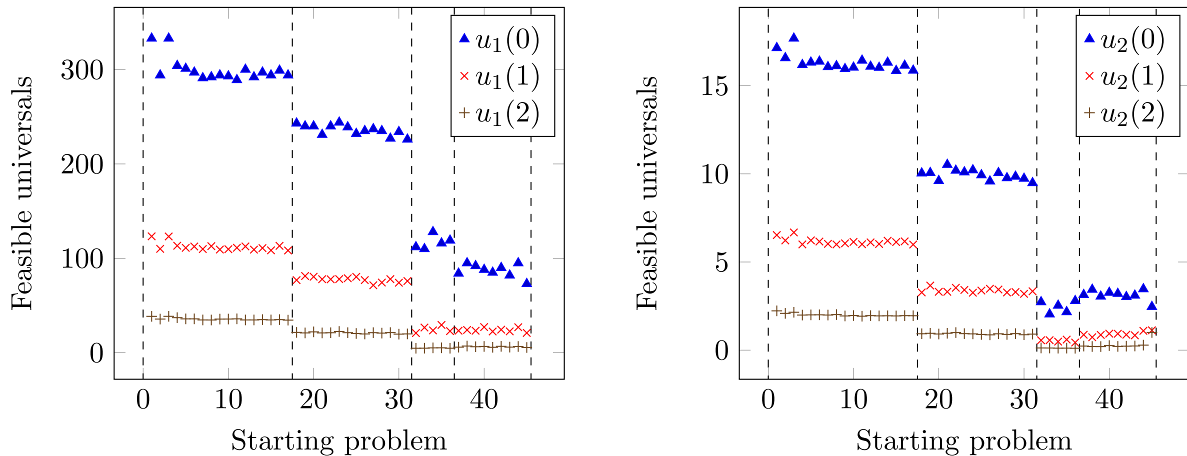



FIGURE 3.5: *The average number of feasible candidate universals $u_i(j)$ found for $i = 1, 2$ and $j \in \mathbb{Z}_k$ in the enumeration of main classes of 3-MOLS of order 8 for each of the 45 partial 3-MOLS which pass the* `isSmallest` *test on level 0 of the search tree. The dashed lines indicate in which section the starting position resides,* i.e. *whether the permutation $u_0(1)$ in the initial partial 3-MOLS has the cycle structure $z_1z_2^2z_3$, $z_1z_2z_5$, $z_1z_3z_4$ or $z_1z_7$, in that order.*

MOLS is proposed. In addition to letting the number of children of a node which are visited be a function of the total number of children of that node, a number of children are also *polled*, in other words, they are probed to ascertain the number of children that they themselves have. Although these polled nodes are not fully visited in the sense that none of their children are visited, they affect the estimation $\gamma_i^*$ as though they were, in fact, visited. This approach seems to be a sensible compromise between managing the increase in accuracy and the decrease in variance expected when applying Purdom's algorithm and the much shorter enumeration times which may be expected if fewer nodes of the search tree are visited, as in Knuth's version of the algorithm. It is believed that this polling process will provide much of the same benefit as visiting multiple branches of the search at every stage of the dive without drastically affecting the computation time, since the total number of nodes visited remains largely the same.

The estimated numbers of nodes on levels 1, 2 and 3 of the enumeration search trees for main classes of 3-MOLS of orders 8, 9 and 10 may be seen in Table 3.7. These estimates have proved to be fairly accurate for order 8. Notice that the actual number of nodes on level 1 of the search tree for main classes of 3-MOLS of order 8, for example, differs from the number given in Table 3.4. This is because the nodes in Table 3.7 were counted in the absence of the `isSmallest` test.

TABLE 3.7: *A comparison of the actual and estimated total numbers of nodes on levels $0, 1, 2$ and 3 of the enumeration search trees for main classes of 3-MOLS of order 8 in the absence of the* `isSmallest` *test, together with similar estimates for orders 9 and 10.*

| | Order 8 | | Order 9 | Order 10 |
|---|---|---|---|---|
| | Actual | Estimated | Estimated | Estimated |
| Level 1 | $2.61 \times 10^7$ | $2.60 \times 10^7$ | $5.79 \times 10^{10}$ | $2.41 \times 10^{14}$ |
| Level 2 | $4.34 \times 10^9$ | $3.74 \times 10^9$ | $3.39 \times 10^{15}$ | $9.67 \times 10^{21}$ |
| Level 3 | $9.96 \times 10^8$ | $9.31 \times 10^8$ | $2.15 \times 10^{16}$ | |

In order to find estimates for the total numbers of nodes on these levels of the respective

search trees, it remains to apply the pruning effect of the `isSmallest` test. Unfortunately, the `isSmallest` test does not readily exhibit the same regularity as the `isOrthogonal` test in terms of the number of candidate universals that pass the test. In order to estimate the number of active nodes on levels 1 and 2 of the enumeration search trees, the pruning effect of the `isSmallest` test must be applied to these estimated total numbers of nodes on all levels of the trees. Let $p_i$ denote the percentage of partial 3-MOLS which pass the `isSmallest` test on level $i$ after having passed the `isOrthogonal` test. The values of $p_1$ and $p_2$ for orders $6, 7$ and 8 may be seen in Table 3.8. Notice that approximately 50% of the nodes on levels 1 and 2 pass the `isSmallest` test. Based on this evidence, the numbers of active nodes on levels 1 and 2 of the enumeration search trees for orders 9 and 10 were estimated for three values of $p = p_1 = p_2$, specifically $p = 0.5$, together with expected over and under estimate values of $p = 0.4$ and $p = 0.6$, respectively. Note that this pruning propagates downwards through the tree. The total numbers of estimated nodes on levels 1, 2 and 3 of the search trees for main classes of 3-MOLS of orders $8, 9$ and 10 for $p = 0.5$ may be seen in Table 3.9, the numbers for different values of $p$ are calculated similarly.

TABLE 3.8: *The average proportions of nodes which pass the `isSmallest` test on levels 1 and 2 during the enumeration of main classes of 3-MOLS of orders 6, 7 and 8.*

| $n$ | 6 | 7 | 8 |
|---|---|---|---|
| Level 1 | 0.55 | 0.483 | 0.573 |
| Level 2 | 0 | 0.538 | 0.511 |

TABLE 3.9: *The estimated total number of active nodes on different levels of the enumeration search trees for main classes of 3-MOLS of orders 9 and 10, as well as the estimated time that the enumeration would take.*

| | Order 8 | Order 9 | Order 10 |
|---|---|---|---|
| $p$ | Actual | 0.5 | 0.5 |
| Level 1 | 15 948 763 | $2.89 \times 10^{10}$ | $1.21 \times 10^{14}$ |
| Level 2 | 1 546 241 258 | $8.48 \times 10^{14}$ | $2.42 \times 10^{21}$ |
| Level 3 | 18 877 734 | $2.68 \times 10^{15}$ | — |
| GHz-days | 32 | $5.64 \times 10^{8}$ | $1.42 \times 10^{18}$ |

Note that, once the number of nodes on any level of the search tree is known (or has been estimated), the time required for the enumeration may be estimated by randomly selecting a representative sample of the nodes on that level, traversing their subtrees and calculating the time required for the traversals of all the subtrees rooted on that level. To enable a later comparison between computing systems of different speeds, the estimated time to completion is expressed in GHz-days — the number of days that a single 1Ghz processor would take to complete the computations. It is expected that a complete enumeration of 3-MOLS of order 9 would take approximately $5.64 \times 10^8$ GHz-days, while for order 10 this is expected to take approximately $1.42 \times 10^{18}$ GHz-days (these estimates may also be found in Table 3.9).

It is already known that certain sections of the search tree are significantly denser and require more computations to traverse than others (see, for example, Table 3.4). In addition to knowledge about the size of an enumeration search tree, knowledge about the shape and characteristics of the tree may lead to a more effective approach. The way in which the shape of the enumeration search trees change for different values of $k$ may also provide further insight into the way that the number of Latin squares in the MOLS and the number of universals already inserted into the partial MOLS influence the pruning process.

The numbers of nodes on level 0 of the enumeration search trees for main classes of $k$-MOLS of orders $n$ were computed for $n \leq 10,\ k < n$ and appear in Table 3.10. The way in which these nodes are distributed over the different sections of the enumeration trees for main classes of order 9 may be seen in Table 3.11. The numbers of nodes on level 0 of the corresponding trees for main classes of $5, 6, 7, 8$ and 9-MOLS of order 10 could not be determined after 3 months of continuous computation.

As may be seen in Table 3.11, the number of nodes per section of the search tree increases gradually before reaching a peak and then dropping down to zero. This may be explained by considering the interplay between pruning away partially completed $k$-MOLS because no candidate universal is orthogonal to the current partial $k$-MOLS, and pruning them away because a partially completed $k$-MOLS may no longer be completed to be a class representative. In the lexicographically smaller sections of the search tree there are very few potential lexicographically smaller isotopes while in the later, lexicographically larger sections there are many permutations that may potentially lead to lexicographically smaller isotopes.

This peak shifts towards earlier sections of the tree for larger values of $k$, in other words, enumeration search trees for large values of $k$ are relatively broader in the first few levels of the tree. This is because the orthogonality constraint becomes restrictive much more quickly when there are many Latin squares to consider. This broader top for large values of $k$ may also explain the apparent contradiction between the fact that the number of nodes on level 0 increases with

TABLE 3.10: *The numbers of nodes on level 0 of the enumeration search trees for main classes of $k$-MOLS of order $n \in \{3, 4, \ldots, 10\}$.*

| | $k$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $n$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 1 | | | | | | | |
| 4 | 1 | 1 | | | | | | |
| 5 | 2 | 2 | 2 | | | | | |
| 6 | 2 | 3 | 3 | 2 | | | | |
| 7 | 2 | 14 | 44 | 33 | 17 | | | |
| 8 | 4 | 45 | 808 | 3 712 | 1 895 | 324 | | |
| 9 | 7 | 259 | 48 285 | 2 379 263 | 14 610 901 | 6 670 346 | 842 227 | |
| 10 | 8 | 1 700 | 796 067 067 | — | — | — | — | — |

TABLE 3.11: *The distribution of nodes on level 0 over the different sections of the enumeration search tree for main classes of $k$-MOLS of order 9.*

| | $k$ | | | | | | |
|---|---|---|---|---|---|---|---|
| Section | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $z_1z_2^4$ | 1 | 20 | 2 458 | 166 245 | 1 479 282 | 926 486 | 152 090 |
| $z_1z_2^2z_4$ | 1 | 53 | 18 502 | 1 268 778 | 9 401 217 | 4 727 973 | 617 476 |
| $z_1z_2z_3^2$ | 1 | 39 | 10 444 | 508 454 | 2 493 403 | 776 991 | 59 447 |
| $z_1z_2z_6$ | 1 | 77 | 14 038 | 413 831 | 1 222 225 | 238 243 | 12 206 |
| $z_1z_3z_5$ | 1 | 47 | 2 600 | 21 630 | 14 710 | 647 | 7 |
| $z_1z_4^2$ | 1 | 23 | 253 | 325 | 63 | 6 | 0 |
| $z_1z_8$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 7 | 259 | 48 285 | 2 379 263 | 14 610 901 | 6 670 346 | 842 227 |

$k$ up to a certain peak, after which it decreases, while the enumeration times strictly decrease as $k$ increases. For larger values of $k$ the search trees are shallower and can therefore be traversed faster because of the increased restrictiveness of the orthogonality test.

Based on the shape and characteristics of the enumeration search trees considered here, the enumeration of main classes of $k$-MOLS of order $n$ is likely to be quickest for large values of $k$, since the search tree is very broad initially, but shallower overall.

## 3.5 Chapter summary

The focus of this chapter was on the problem of enumerating equivalence classes of Latin squares and MOLS. In §3.1, various transformations were presented that partition the set of all Latin squares into equivalence classes. The notion of a paratopism which, in turn, gives rise to the notion of main classes of Latin squares, was of particular interest and was extended to sets of $k$ mutually orthogonal Latin squares. The theoretical and algorithmic developments with respect to enumerating these equivalence classes since Euler first considered the problem in 1782 were described in §3.2.

A backtracking algorithm for the enumeration of main classes of MOLS, which may be seen as a variation on Sade's algorithm for counting reduced Latin squares, was presented in §3.3. The algorithm constructs all the main class representatives and was used successfully to enumerate main classes of $k$-MOLS of order $n$ for $n \leq 8$ and $k < n$. The algorithm has, however, proven to be too computationally expensive for $k$-MOLS of larger orders. Techniques for estimating the sizes of the enumeration search trees for main classes of MOLS of orders $n > 8$ were considered in §3.4 in order to gain some insight into the likely computation time required for these larger enumeration attempts, which include the 3-MOLS of order 10. It was also shown that as the value of $k$ increases the enumeration search tree becomes both broader at the top and shallower, making these enumerations potentially easier.

In the next chapter, a computing paradigm is introduced which may provide a way of overcoming the computational barrier which has, up to now, prohibited the enumeration of main classes of $k$-MOLS of orders $n > 8$.

# CHAPTER 4

# Volunteer computing

Contents


4.1 A historical overview of public-resource computing . . . . . . . . . . . . . . . . 43
4.2 The Berkeley Open Infrastructure for Network Computing . . . . . . . . . . . . 46
*4.2.1 Basic workflow and concepts of volunteer computing . . . . . . . . . . . 46*
*4.2.2 Grid-enabling a simple BOINC project . . . . . . . . . . . . . . . . . . . 47*
*4.2.3 Special types of applications . . . . . . . . . . . . . . . . . . . . . . . . . 52*
*4.2.4 Setting up a server and project maintenance . . . . . . . . . . . . . . . 53*
*4.2.5 Security concerns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54*
*4.2.6 Challenges facing volunteer computing . . . . . . . . . . . . . . . . . . . 54*
4.3 Chapter summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55


The basic concepts of distributed and volunteer computing are reviewed in this chapter. A brief historical overview of some popular distributed and volunteer projects is provided in §4.1. *Berkeley Open Infrastructure for Network Computing* (BOINC), in particular, is examined in some detail in §4.2, starting with basic concepts and the steps required to create a volunteer computing project using this predominant middleware system. Some consideration is also given to special types of applications and the requirements of setting up a BOINC server, as well as common security concerns and challenges facing distributed and volunteer project administrators.

## 4.1 A historical overview of public-resource computing

In John Brunner's 1975 science-fiction novel, *The Shockwave Rider* [18], the main character creates a "tapeworm" program which is able to propagate itself through a network of computers, replicating when needed and consuming resources wherever available. This type of program, capable of harnessing resources on any number of machines, piqued the interest of a group of researchers at Xerox's Palo Alto Research Center (famously the home of the first graphical user interface) which set about creating a number of "worms" for controlling multi-machine performance measurements of their pioneering first Ethernet network [88]. Distributed computing systems were mainly confined to networks within organisations or academic departments for the following two decades. An early example of such a distributed computing system is *HTCondor* [93, 91]. This system was developed in 1988, was originally known as *Condor*, and is still active

today. The recent widespread public adoption of personal computers and the internet has, however, made it possible to involve the public in distributed computing, also known as *volunteer computing.*

Anderson *et al.* [92] define volunteer computing as a form of computing where both organizations and members of the public can donate unused computing resources to computing projects, which are usually applications of a scientific or academic nature. Volunteer computing is built largely on trust and mutual goodwill as it is necessary that the volunteers trust the projects not to perform unapproved calculations using their resources and to guard their personal account details carefully. Similarly, although IP and email addresses may be linked to individual volunteers, volunteers remain largely anonymous and no disciplinary steps are typically available to curb volunteers who wilfully corrupt computations. For project administrators the main advantages of volunteer computing are, firstly, that it provides access to at least part of the approximately ten billion devices linked to the internet [4, 34] and, secondly, that it raises public awareness of scientific endeavours and provides a mechanism through which a project of large popular appeal, but little funding, may flourish [92]. Volunteers, on the other hand, are able to contribute to scientific projects which interest them and are rewarded in credits, which become somewhat of a status symbol.

Volunteer computing is somewhat similar to two other forms of distributed computing, namely *grid computing* and *peer-to-peer* networks. Grid computing is generally considered a paradigm through which computing resources are shared within and between organizations in a mutually beneficial way. An example of such a system is a desktop grid consisting of all the desktop computers within an organization. Grid computing has many aspects in common with volunteer computing, but differs from it in that the "volunteers" are usually more reliable because of a lack of anonymity and the accompanying existence of disciplinary measures [40, 92]. The principles of peer-to-peer computing, on the other hand, are perhaps best seen in file-sharing services such as *Napster* [68] or *BitTorrent* [9] — data transfer takes place between computers without any form of coordination by central servers [75, 87]. Volunteer computing, on the other hand, relies on central servers hosting projects and it is not possible for two clients to communicate directly in such a computing paradigm without coordination by the central servers.

There are, however, many challenges associated with publicly distributed computing. For example, the platforms on which applications must be able to execute accurately have become rather more heterogeneous and a distributed computing network is likely to be much more unreliable than a computer network within a corporation or laboratory. Despite these limitations, scientists have become increasingly interested in unlocking the massive computing power which lies dormant in the idle computer cycles of the public. Two early volunteer computing projects, the *Great Internet Mersenne Prime Search* (GIMPS), which searches for very large Mersenne primes [63], and *distributed.net* [30], which facilitates brute-force decryption, were established in 1996 and 1997, respectively.

GIMPS is still active today and currently runs on more than 730 000 CPUs around the world, with an average annual throughput close to 130 TeraFLOPs[1] [63]. Mersenne primes are primes of the form $2^n - 1$ and account for most of the very large prime numbers known today. The first Mersenne prime discovered by GIMPS, namely $2^{1\,398\,269} - 1$, was found in November 1996 [47], while the forty-eighth Mersenne prime, and the currently largest known prime number, namely $2^{57\,885\,161} - 1$, was found on 25 January 2013 [63]. The network *distributed.net* also remains active and concerns itself with finding optimal Golomb rulers (combinatorial curiosities with application to the placement of radio antennas in astronomy) and deciphering encoded

[1]One TeraFLOPS of computing power is equivalent to $10^{12}$ floating point operations per second, or approximately 500 GHz-days.

messages [30], a trend which, according to Hayes [47], started when the company RSA Data Security issued a number of challenges in 1997, hoping to test how easily their encryption systems were breakable and to demonstrate the inefficiencies of rival schemes. Notable early successes were the breaking of RSA-129, which involved the factorisation of a 129-digit number and took 600 volunteers eight months to perform, and the decryption of a message encrypted using the *Data Encryption Standard* (DES), which was developed under US government sponsorship during the 1970s [30]. Another early project, which has grown into one of the most influential volunteer computing projects, is *SETI@Home* (an abbreviation for *Search for Extraterrestrial Intelligence*), launched in 1999 by a group of scientists from the University of California at Berkeley to examine radio waves detected by a telescope operated by Cornell University and the National Science Foundation in Arecibo, Puerto Rico [6]. The project was very well received by the public and attracted millions of volunteers. By 2004 the average sustained processing power of the SETI@Home project was more than 70 TeraFLOPs — more than double the 35 TeraFLOPs of the most powerful supercomputer at that stage, the NEC Earth Simulator [3].

According to Anderson *et al.* [6], the encouraging success of these early projects led to wider support for frameworks which could be used for public-resource or large-scale distributed computing. In 1999, the *Global Grid Forum* was formed as an umbrella corporation for a number of distributed projects, collectively called *The Grid* [40]. The mandate of this corporation was to coordinate resource sharing amongst research organizations, while many private organizations, such as *Platform Computing* and *United Devices*, were developing corporate systems for distributed storage and computing. A number of middleware systems were launched to facilitate distributed computing grids. Examples of such systems include *Sun Grid Engine* in 2001 [74] (now called the *Oracle Grid Engine*), *Advanced Resource Connector* in 2002 [69] and the *Globus* toolkit in 2004 [25]. BOINC was also launched in 2004 [3], spearheaded by David Anderson, who was the Chief Science Officer at United Devices at the time and also co-founder of the SETI@Home volunteer computing project.

More than 80% of the currently active public-resource computing projects make use of BOINC and, as such, it is the focus in the remainder of this chapter as an example of a middleware framework for distributed computing. BOINC, as well as other middleware, provides simple interfaces for both volunteers and scientists, and handles the required network connections automatically, thereby considerably decreasing the difficulty of establishing a new volunteer project. It also provides scientists with a comprehensive *Application Programming Interface* (API) with which to interact with volunteers and to enable volunteers to use the same client to connect to multiple volunteer computing projects. The first project to make use of BOINC was, unsurprisingly, SETI@Home. A large number of scientific projects followed suit with applications ranging from testing Einstein's theory of general relativity (*Einstein@Home*) [1] and finding new arrangements into which proteins may fold themselves (*Folding@Home*) [77] to the distributed rendering of animated films (*BURP*) [82]. A number of large organizations are also engaged in volunteer computing through BOINC, such as IBM through their *World Community Grid* (WCG) initiative which serves the community by computing vaccines for malaria and modelling the earth's fresh-water supply [49], CERN, which uses the *LHC@Home* project to process vast quantities of data obtained from experiments in their Large Hadron Collider [22], and Oxford University, which runs a number of different models estimating the effects of climate change on *Climateprediction.net* [23].

BOINC remains under active development by a group of software developers and project administrators. A BOINC client for Google's open-source mobile operating system *Android* has recently been released, thereby enabling the approximately 470 million Android smartphones in the market [67], many of which have two, four or even eight processors, to take part in volunteer

computing [60].

## 4.2 The Berkeley Open Infrastructure for Network Computing

The main aim of BOINC is to simplify setting up a large-scale volunteer computing project, to minimize the time required for administrating the day-to-day running of the project and to enable a single computer, after a few weeks of work, to perform the role of a server for a project involving thousands of volunteers. Reducing the entry cost to volunteer computing has enabled many scientists with widely varying backgrounds and only moderate programming skills to take advantage of public resources for their computing endeavours — something which would not have been possible had it been necessary for them to design the entire system themselves. According to Anderson [3], secondary design goals included support for a diverse set of applications and programming languages, allowing for the sharing of resources between autonomous projects by letting volunteers simultaneously take part in a number of projects by assigning a priority to each project and, finally, rewarding the volunteers in some way by measuring their contributed resources.

As enticing as the advantages of public resource computing and BOINC, in particular, may seem, it is important to note that not all applications work equally well within this model. For a project to be successful in utilizing public resources, especially in the developing world where high-speed internet is still a rarity, it is firstly very important that the application should have a large computation to data ratio (so that small data transfers to volunteers can lead to multiple hours of computations) and, secondly, that the execution of the application should be independently parallelisable. Some classes of applications which seem particularly well suited to this type of distribution are described in [5]. An example of such an application is the simulation of physical systems in which every simulation is independent of the others or a physical model in which complex parameter spaces which must be explored. Random and genetic algorithms may also benefit from volunteer computing, as every independent trail can be performed by a different volunteer in the case of random algorithms, while for genetic algorithms every volunteer may receive an initial population of solutions to evolve from independently. A final category of projects, which has perhaps proven most successful in practice, involves the analysis of large amounts of data, such as from radio telescopes (as in the case of SETI@Home) or from the Large Hadron Collider (as in the case of LHC@Home) [5].

### 4.2.1 Basic workflow and concepts of volunteer computing

According to [92], a BOINC *project* is a self-contained entity consisting of a database, website, *applications*, *work units* and *results*.

An application consists of a number of different executable programs, usually one for every *platform* on which the project is available. Standard platforms are pre-defined in BOINC and are combinations of CPU architectures and operating systems (*e.g.* a 32-bit Intel processor running Windows or a 64-bit Intel processor running Linux). Knowing on which platform a computation is performed is important for ensuring correct results, as different platforms handle floating-point operations in different ways and this may lead to discrepancies in results if ignored.

A work unit is a computation task that has to be performed. It is associated with a specific application and has various attributes, including the names of its input files, the estimated execution time, and the resources required to complete it. As was mentioned in §4.1, the results returned by volunteers may not necessarily be trustworthy. Volunteer computing projects usually

mitigate this risk through *redundant computing*, which means that every work unit is computed multiple times by different volunteers and the results are compared for correctness. A *result* may therefore be seen as a copy of a work unit which is sent to a volunteer for computation. Once a certain number, called a *quorum*, of results of a specific work unit have been returned, they are compared to find the definitive or *canonical* result for that work unit.

All work units and results are stored in a database on the server, along with additional information related to applications, work units, volunteers and hosts. A distinction is made between a volunteer, who creates an account with an email address and password when joining a project, and a host, which is the actual machine performing the computation. A single account may thus have a number of hosts associated with it, each running a version of the BOINC client which allows the host to attach to scientific projects. Some measure of credit is associated with an account, so that all hosts running under the same account earn credit in the same place.

Once a volunteer host has connected to a specific project through the client it may request work from the project's servers. A so-called *deamon* (a small, periodically executed task-specific program) running on the server, called the *scheduler*, takes the host's resources into account and dispatches suitable results, if any are available, after which the computation takes place on the host. Once the computation has been completed the client on the host reports the results to the server along with a request for further work. The results thus received are stored on the server until their respective quorums are met, after which the results are tested for correctness, a process called *validation*, credit is assigned to correct results and the results are finally processed or *assimilated.* In the meantime, the host would have received new results for processing and the process, as illustrated in Figure 4.1, may be repeated as long as there are uncompleted results available.

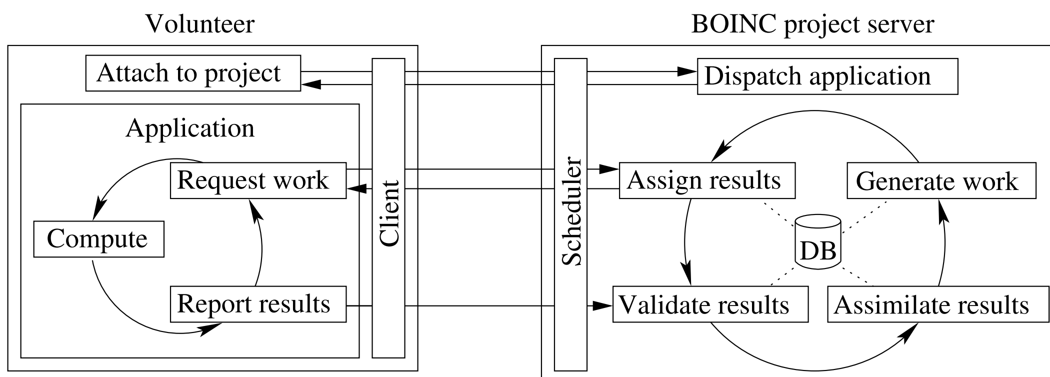



FIGURE 4.1: *The basic workflow on the client and BOINC project server*

### 4.2.2 Grid-enabling a simple BOINC project

A scientist attempting to turn an existing application into a volunteer computing project must determine how the application will be parallelised, modify the existing application to make use of the BOINC API and finally provide the server-side deamons responsible for scheduling, generating new work units and validating and assimilating incoming results. The first of these tasks, parallelising the computation, is highly application-specific and has little to do with the structure and components of a BOINC project, but it may be insightful to give further consideration to the BOINC API and the various deamons running on the server.

### The BOINC API

The BOINC API provides a set of functions allowing the client and the application to communicate and improving portability of the application between different platforms. An application must notify the client when it initialises through a call to `boinc_init` and return a value when terminating so that the client may track whether the computation was successful or not. The API also handles input and output, resolving logical names to actual file names as specified in the work unit and providing wrappers for functions which execute differently on different platforms. An example of such a wrapper occurs in the `boinc_fopen` function which replaces the standard `fopen` call for opening functions so that on Windows hosts, where files may become temporarily locked, the function attempts to open the file multiple times within short succession while on Unix, where `fopen` occasionally fails with a specific error code, it tests for this error code and retries accordingly.

*Checkpointing*, or saving the current state of the computation in such a way that the computation may be resumed from that point, is very important in volunteer computing as there is no way of knowing when a host may be turned off or fail. An application may choose its own minimum time between checkpoints. This decision is usually based on how long it takes to checkpoint, and may repeatedly call the function `boinc_time_to_checkpoint` to determine when to checkpoint. Checkpointing is also an example of a *critical section* of an application, which may not be interrupted by the client.

TABLE 4.1: *The core of the BOINC C/C++ API [92].*

| API method | Description |
|---|---|
| `int boinc_init()` | The call notifying the client that a computation has begun. |
| `boinc_cpu_time()` | The CPU time that the current computation has taken. |
| `int boinc_finish(int status)` | The call notifying the client that a computation has been completed, along with its exit status — 0 if successful or some integer error code. |
| `int boinc_resolve_filename(char *logical_name, char *physical_name)` | Converts logical file names, such as 'in,' used in the application to physical file names. |
| `bool boinc_time_to_checkpoint()` | Tests whether it is time for the application to checkpoint. |
| `void boinc_checkpoint_completed()` | Notifies the client that a checkpoint has been completed, that the countdown timer for checkpointing must be reset and that the program is exiting a critical section. |
| `boinc_fraction_done(double fraction_done)` | Reports the current progress to the client. |
| `void boinc_child_start()` | Notifies the client that the main program has started a subsidiary thread. |
| `void boinc_child_done(double total_cpu)` | Notifies the client that a subsidiary thread has been completed. |

BOINC allows communication between the application and server through *trickle messages*,

which are small messages sent up from the client or down from the server and which are assigned a very high priority. A trickle down message may, for example, be sent to notify an application that the result it is computing is no longer needed and should be aborted. Similarly, a trickle-up message may be sent to the server to notify it that, although a result has exceeded its deadline, it is still being computed successfully, in which case the original deadline should be extended. The remainder of the API mainly focusses on providing methods for allowing the client to register the time that a computation takes and monitor its progress. The full API, along with a brief summary of every function as described in [92], may be found in Table 4.1.

#### Deamons

The main task of the work generator is inserting new work units or jobs into the project database. This may happen once, at the start of the project, in a number of discrete batches or continuously as results are returned by volunteers. It is also possible to develop a web interface which allows for the remote submission of jobs. Before implementing the work generator, however, the project manager must decide on templates for the work units and results. These templates are *XML* (extensible markup language) files which provide the application with the necessary information for performing a work unit and are typically re-used for millions of jobs. An input template may, for example, specify the name of the actual file to which the logical file name must be resolved, possible command-line arguments along with work unit attributes such as an estimate of its execution time, the resources required for the computation and the deadline by which the server expects to receive the result before re-issuing the work unit to a different host. Similarly, the output template defines the number of files returned to the server and where they may be located or generated. A selection of the most important options that may be specified in the input and output templates may be found in Tables 4.2 and 4.3, respectively. The work generator ensures that the required files and templates are available before entering the work unit into the database, either through a command-line tool or a C/C++ method which is part of the BOINC API.

It is highly likely that a new project will, at least initially, make use of the default scheduler provided along with the BOINC source code. The scheduler is mainly responsible for dispatching new versions of applications, accepting requests for work from hosts and assigning work to hosts based on a combination of their available resources, the estimated execution time of the available work units and the historical accuracy of a host. Larger work units are thus preferably sent to hosts who have historically returned fewer erroneous results and who have more resources available in order to minimise the expected computation time that will be wasted if a host returns an error after an extended period of work. Secondary objectives which may be included in the scheduler are completing a specific batch of work units as soon as possible and limiting the number of work units received by a host on a single day (so as to prevent hosts from repeatedly returning the same results and claiming credit for them).

The main task of the validator is to compare results for correctness. Care should be taken when comparing results from different platforms — for example, the end-of-line character is different in Windows and Linux, making a character-by-character comparison of the output files impossible. If an application performs many floating-point operations, projects may elect to enforce a measure called *homogeneous redundancy* to ensure that results from the same work unit are assigned to identical platforms. Alternatively, “fuzzy” comparisons may be made by comparing floating-point numbers with a degree of tolerance [92]. A measure called *adaptive replication* may also be used, which dynamically determines the number of replications of a specific work unit based on the historical error rate of the host to whom the first result is

TABLE 4.2: *A selection of the parameters that may be specified in the input template of BOINC [92].*

| Work unit attributes | Description |
|---|---|
| `open_name` | The logical name of the file used in the application, *e.g. 'in'.* |
| `command_line` | The command-line arguments to be passed to the main application. |
| `rsc_fpops_est` | An estimate of the number of floating point operations required to complete a work unit, used to estimate how long the work unit will take on a given host. |
| `rsc_fpops_bound` | An upper bound on the number of floating point operations required to complete a work unit. If this bound is exceeded, the result will be aborted. |
| `rsc_memory_bound` | An estimate of a work unit's largest working set size. A result of this work unit is only sent to hosts with at least this much available RAM. If this bound is exceeded, the result is aborted. |
| `rsc_disk_bound` | A bound on the maximum disk space used by a work unit, including all input, temporary, and output files. The result of this work unit is only sent to hosts with at least this much available disk space. If this bound is exceeded, the result is aborted. |
| `rsc_bandwidth_bound` | If non-zero, a result is sent only to hosts with at least this much download bandwidth. Mainly used for work units with very large input files. |
| `delay_bound` | An upper bound on the time (in seconds) between sending a result to a client and receiving a reply. If the client does not respond within this interval, the server 'gives up' on the result and generates a new result, to be assigned to another client. |
| `min_quorum` | The minimum size of a quorum. The validator is run when there are this many successful results. If a strict majority agrees, the result is considered correct. This value is set to two or more if redundant computing is required. |
| `target_nresults` | The number of results created initially. This must be at least `min_quorum`. It may be more in order to reflect the ratio of result loss, or to get a quorum more quickly. |
| `max_error_results` | If the number of client error results exceeds this value, the work unit is declared to contain an error; no further results are issued, and the assimilator is triggered. This safeguards against work units that cause the application to crash. |
| `max_total_results` | If the total number of results for a work unit exceeds this value, the work unit is declared to be in error. This safeguards against work units that are never reported (*e.g.* because they crash the core client). |
| `max_success_results` | If the number of successful results for a work unit exceeds this value, and a consensus has not been reached, the work unit is declared to be in error. This safeguards against work units that produce non-deterministic results. |
| `priority` | Higher-priority work units are dispatched first. |
| `size_class` | Used to define work units of different sizes, for example in the case where the GPU version of an application is orders of magnitudes faster than the CPU version. |

TABLE 4.3: *A selection of the parameters that may be specified in the output template of BOINC [92] .*

| Output attributes | Description |
|---|---|
| `name` | The physical file name of the output file. |
| `open_name` | The "logical name" by which the application will reference the file. |
| `max_nbytes` | Maximum file size. If the actual size exceeds this, the file will not be uploaded, and the job will be marked as an error. |
| `url` | The URL of the file upload handler. |
| `no_delete` | If present, the file will not be deleted on the server even after the job is finished. |
| `report_immediately` | If present, clients will report this job immediately after the output files are uploaded, otherwise they may wait up to a day. |

assigned— if the host usually returns accurate results, the probability of replication taking place for that work unit will be small. Validation takes place by majority once a quorum of results has been received and marks a result, along with its associated output files, as the canonical result against which any other results of the same work unit are compared in order to determine whether it was computed successfully and deserves credit. Two sample validators are provided with the BOINC source code: the first is mainly used for testing purposes and simply marks every received result as successful, while the second performs a bitwise comparison of output files.

Once results have been verified, they should be marked accordingly in the database so that they are not assigned to any further hosts. The input files of verified results may be deleted from the location where they were accessible to hosts and the canonical output files may be moved to a specific location. The sample assimilator does exactly this and it is up to project administrators to decide which other post-processing tasks to perform on validated results.

Projects that use trickle messages must also provide a deamon to handle these messages. Trickle messages may, for example, be used by the application to send its current computation state to the server periodically so that the server may determine whether to assign partial credit for the work done up to the current point or to abort the computation based on some internal logic.

The most important interactions in a BOINC project are illustrated in Figure 4.2. Note that the transitioner and file deleter are responsible for generating new results if invalid or erroneous results are received and deleting the input files of fully completed work units, respectively, but have not been discussed since the default versions suffice.

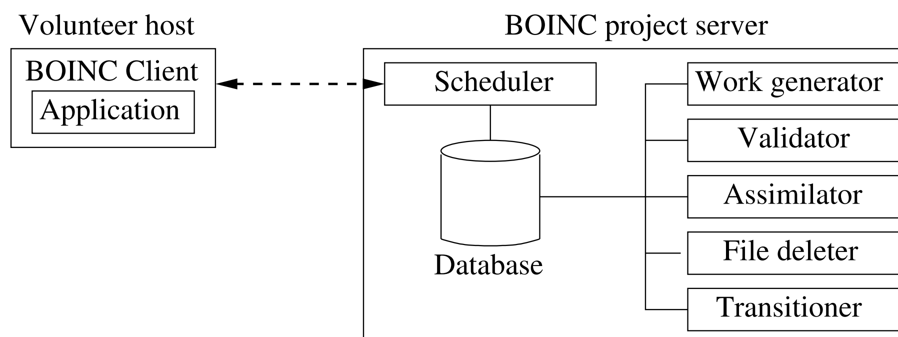



FIGURE 4.2: *An example of the server setup showing the interaction of the MySQL database with the BOINC clients and various server deamons.*

### 4.2.3 Special types of applications

In addition to the basic functionality described in the previous section, BOINC allows applications to display graphics, execute on multiple cores or GPUs through OpenCL [51] and CUDA [72], or even to run entirely within a virtual machine.

#### Graphics applications

Publicly-launched volunteer computing projects usually attempt to provide volunteers with some visualisation of the computation running on their machines, either as screensavers or in windows which can be opened in the BOINC Client Manager. The main reason for providing such graphics is to further engage the public in the goals of and progress made in a scientific project in order to increase involvement. Examples of graphics provided by SETI@Home and WCG may be seen in Figure 4.3.

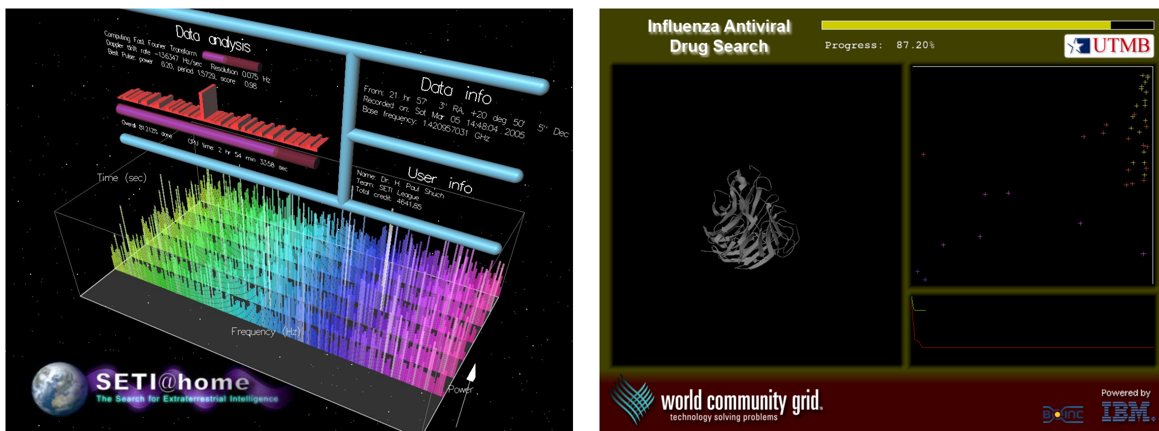



FIGURE 4.3: *Graphical applications in SETI@Home and WCG provide volunteers with an indication of the state of the current computation.*

In a recent survey of 15 627 volunteers by IBM's World Community Grid project [49], however, 72% of volunteers stated that they either disabled the graphics or did not pay attention to it [81], thereby casting some doubt on the actual benefit of graphical applications. Graphics applications are usually separate from the main scientific application, built on a specific BOINC graphics library and make use of OpenGL [27]. Communication with the main application takes place through shared memory — the main application would periodically store a representation of its current state and progress in a block of memory which is accessible to the graphics application, from where it is read and displayed. Graphics applications can also display static images, text or web-based graphics by opening browser windows pointing to specific URLs.

#### Muti-core and GPU applications

CPU manufacturers have been dealing with the limit placed on CPU frequency by increasing the number of cores provided and it seems that this trend will continue in the foreseeable future [44]. In cases where the completion time of individual work units have to be improved, or the memory footprint of the application is too large to allow a separate copy of the application to run on every core, it may be desirable to implement a multi-threaded application developed in OpenCL, MPI [64], OpenMP [73], CUDA or a number of other languages. BOINC also supports

what it calls 'coprocessors,' or GPUs, designed by NVIDIA, AMD or Intel. Available GPUs are reported to the scheduler by the BOINC client, which also keeps track of which instances are currently allocated on every GPU. To prevent system failures, GPU kernels are executed within critical sections, which may not be killed by the BOINC client and BOINC projects may define work units of different size classes to compensate for the potentially dramatic differences in speed between CPU and GPU versions of the same applications, thereby ensuring that a typical work unit will take approximately the same amount of time irrespective of the architecture on which it is executed.

#### Applications which run inside virtual machines

BOINC supports applications which run entirely within virtual machines. This approach has two distinct advantages in that it provides the highest level of security for the host machine, as virtual machines cannot access or modify the host system, and there is no need to build applications for different architectures, as every application will run on a virtual computer with exactly the same runtime environment on all platforms.

There are, however, also some additional complexities that arise from using applications inside virtual machines. An example of such a complication is the fact that hosts require software such as VirtualBox to mount the machine image, yet VirtualBox is not currently available for all processors. Furthermore, GPU applications cannot currently run inside VirtualBox and not all processors are capable of running both 32- and 64-bit virtual machines; images of both architectures therefore have to be provided. Distributing the virtual machine image also increases the size of the first download to approximately 200Mb, which may be prohibitive for volunteers with limited bandwidth.

### 4.2.4 Setting up a server and project maintenance

As described in §4.2.1, a BOINC project consists mainly of a MySQL database, a directory structure and a configuration file which specifies the options, deamons and periodic tasks that must be performed and, as such, it is possible to use almost any computer as a BOINC server [92].

Since reliability and security are of the utmost importance, a server should have a static IP address and at the very least be placed behind a firewall and have a reliable internet connection for connecting with volunteers. Additional hardware-related measures that may be taken to improve reliability include an uninterrupted power supply, automatic backup protocols, adequate cooling and hot-swappable spare server parts. Any Unix or Linux distribution may be used for setting up a server and detailed instructions are available for the configuration [92]. Alternatively, a project may elect to host its server either on the *Amazon Elastic Cloud Computing* (EC2) service, which removes potential concerns involving hardware reliability and security, or host the project within a virtual machine. A virtual machine image is available with all the software packages required to set up a server, as are a number of installation and configuration scripts.

Once a server has been set up, a number of maintenance tasks have to be performed regularly, such as reviewing the deamon logs for errors, deleting files as they are no longer needed and archiving and purging old jobs from the database so as to prevent the database from becoming to large. BOINC provides small programmes for all of these activities. It is possible, due to the popularity of a project, that a server may not be able to keep up with the traffic generated by the volunteers, which may result in dropped connections, slow website access, deamons that

fall behind and very slow database queries. A number of strategies are discussed in the BOINC documentation for upgrading the server in such cases, including upgrading the server hardware, hosting the database on a separate server, and parallelising the deamons and scheduler so that multiple instances run continually.

### 4.2.5 Security concerns

According to [92], a number of security concerns arise from the inherently public nature of volunteer computing, chief amongst which are:

- result falsification,
- credit falsification,
- denial-of-server attacks on the project servers,
- theft of project files or participant account information, including email addresses, and
- the distribution of malicious executables.

Result and credit falsification may be limited by adopting replication and validation protocols and by limiting the number of results for which a user may receive credit in a single day. BOINC protects projects against denial-of-service attacks, in which servers are overrun by requests and transfers from automated programs, by providing a size limit for every file that is uploaded to the server (refer to Table 4.3) and by making use of upload certificates. Every project is responsible for protecting its own users' account information against theft and servers should be subjected to regular security audits. The greatest security risk to volunteer computing project is the potential that a server may be broken into and used to distribute malicious executables that wreak havoc on volunteer hosts. In order to prevent this, code-signing software is used in which every approved and secure application version is authenticated. The computer responsible for code-signing applications should be kept in 'cold storage,' in other words, physically secure and completely disconnected from the network so as to prevent attackers from breaking into it and authenticating their applications.

### 4.2.6 Challenges facing volunteer computing

After a rather bright start, volunteer computing has decreased in the past few years, most recently from approximately 290 000 volunteers in 2012 to 240 000 in 2013 [4]. Volunteer computing projects have also largely stagnated, with only very few, small projects being initiated in recent years. The largest projects, such as GIMPS, SETI@Home and Einstein@Home, are all at least ten years old. Some of this recent stagnation may be attributed to the fact that volunteer computing projects have largely failed to expand beyond their initial target audience of technically minded males working in sectors such as engineering and information technology. Indeed, a recent survey has shown that 90% of WCG's volunteers are male and 70% are involved in some kind of scientific occupation [81]. A number of initiatives have been launched to bridge this gender and technical gap, the most successful of which seems to be a Facebook [98] initiative, called *Progress Thru Processors*, which boasts more than 160 000 likes and a volunteer base split evenly along gender lines [10].

Volunteer computing has traditionally been centred in the United States and Western Europe with only minor contribution from developing countries, such as Brazil, India and China, who

have potentially massive numbers of volunteers [81]. The main reason for this lack of contribution seems to be the language barrier — the majority of projects are inaccessible to non-English-speaking volunteers [35] and so there have been recent additions to the BOINC source code which simplify the translation of projects. Additional problems arise from the quality of hardware, especially in China, where the actual specifications of CPUs and GPUs often differ from the reported names as many hardware brandnames are “unofficially” produced in factories [94].

## 4.3 Chapter summary

This chapter contains a historical overview of the notion of volunteer computing as well as a review of the basic concepts of one of the middleware systems for volunteer computing, BOINC. A brief history of public resource and volunteer computing was presented in §4.1, from its humble beginnings in Xerox’s Palo Alto Research Centre to the middleware system Condor and eventually into the homes of millions by virtue of BOINC. BOINC, the predominant volunteer computing middleware system, was discussed in §4.2. The basic workflow of a BOINC project was reviewed in §4.2.1, while in §4.2.2 the process of grid-enabling an application, which consists of incorporating the BOINC API into an existing application and creating deamons for the project server, was discussed. More advanced functionality which allow applications to facilitate graphical displays or execute on multiple CPUs or GPUs or within a virtual machine was briefly discussed in §4.2.3. Server management protocols were considered in §4.2.4 and the chapter closed with an overview of common security concerns and a discussion of the challenges facing volunteer computing in §4.2.5 and §4.2.6, respectively.

In the next chapter this overview of the components of a volunteer computing project, and BOINC specifically, will be used to create a volunteer computing project for the enumeration of main classes of sets of $k$ mutually orthogonal Latin squares.

CHAPTER 5

# A distributed volunteer project for the enumeration of $k$-MOLS

Contents


5.1 A volunteer project for counting 3-MOLS of order 8 . . . . . . . . . . . . . . . 57

*5.1.1 Server architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58*

*5.1.2 Grid-enabling the exhaustive enumeration algorithm . . . . . . . . . . . 58*

*5.1.3 Deamons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59*

*5.1.4 First enumeration results . . . . . . . . . . . . . . . . . . . . . . . . . . 59*

5.2 Generalising to the enumeration of $k$-MOLS of order $n$ . . . . . . . . . . . . . 60

*5.2.1 Limiting work unit sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . 61*

*5.2.2 Dynamic splitting of work units . . . . . . . . . . . . . . . . . . . . . . . 63*

*5.2.3 Implementing and validating the generalisation . . . . . . . . . . . . . . 64*

5.3 Enumeration results emanating from an implementation . . . . . . . . . . . . 66

5.4 Chapter summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68


This chapter contains a description of the processes involved in designing a distributed volunteer computing project for enumerating $k$-MOLS using the middleware system BOINC. The basic details of grid-enabling the enumeration algorithm of §3.3 are described in §5.1, and some preliminary enumeration results are also reported. A work unit issuing policy aimed at managing work unit sizes uniformly and utilising computing resources more efficiently is proposed and verified in §5.2. The results of a pilot study, in which the main classes of 3-MOLS of order 8 are enumerated, is presented in §5.3, along with partial results of an ongoing enumeration attempt for main classes of 7-MOLS of order 9.

## 5.1 A volunteer project for counting 3-MOLS of order 8

BOINC [3] was selected as middleware for this research project since it is the most prevalent architecture in use by existing volunteer computing projects. There are also active BOINC support forums available online to project developers [92].

### 5.1.1 Server architecture

An initial BOINC server was set up inside a virtual machine running the Debian 6 operating system with an 8Gb virtual hard disk and dedicated access to 384Mb of *random access memory* (RAM) from the host. The machine hosting the virtual machine has a 64-bit Linux distribution with 8Gb of RAM. The virtual machine was run in *bridged ethernet* mode, which means that it shows up as a separate machine on the network with its own static IP address so that the server is always available at the same address for both volunteers signing up and hosts communicating with the server.

### 5.1.2 Grid-enabling the exhaustive enumeration algorithm

The exhaustive enumeration of main classes of $k$-MOLS, as described in §3.3, parallelises trivially — the subtrees rooted at the nodes on any level of the search tree may be enumerated completely independently. The number of branches on the respective levels of the subtree rooted at $\mathcal{P}$ on level $i-1$ is simply the sum of the branches counted on every level in the respective enumerations of the children (on level $i$) of node $\mathcal{P}$.

The main modification required to grid-enable the enumeration of $k$-MOLS main classes is the implementation of checkpointing. As described in §4.2.2, the main goal of causing an application to checkpoint, or save the current state of the computation in such a way that the computation may be restarted from it without any loss in accuracy, is to guard against losing progress through unpredictable volunteer and host behaviour. An application requiring twenty four hours of computing to complete a work unit which does not checkpoint and runs on a host which is only available between 08:00 and 17:00 would, for example, never return results since nine hours of computations are lost every day when the computer is switched off. The enumeration algorithm lends itself naturally to checkpointing, since, once the universal $u_i^{(j)}$ has been fixed for all values of $i$ up to some integer $m$, the number of feasible candidate universals for $u_{m+1}^{(j)}$, prior to testing for orthogonality and class representatives, is completely deterministic. Indeed, the list of candidate universals for $u_{m+1}^{(j)}$ is always generated in exactly the same order, which means that any partial Latin square may be described by specifying the position in the list of candidate universals of the universal selected. All that is required to represent a partially completed $k$-MOLS, and therefore any node of the original search tree in this manner are the positions of the selected universals in each of the Latin squares, since the squares are completely independent prior to the test for orthogonality.

To be able to restart the enumeration from a checkpoint, the original algorithm was adapted so that, after reading the initial, partially completed $k$-MOLS from an input file, it searches for a checkpoint file and, if one is available, reads in the numbers of branches enumerated previously. The enumeration then starts at the designated candidate universals instead of the first candidate universal generated. Care was taken to ensure that branches counted prior to checkpointing are

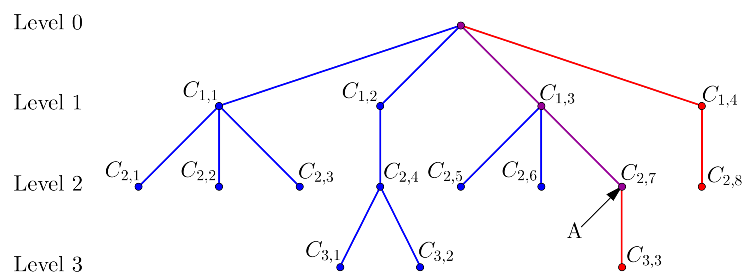



FIGURE 5.1: *A hypothetical enumeration tree showing the checkpointing strategy. The blue part of the tree is traversed prior to checkpointing at the node labelled A, the purple part of the tree is traversed both prior to checkpointing and when restarting from the checkpoint, and the red portion is traversed after restarting from the checkpoint at A.*

not counted twice, although some branches are, of course, considered a second time in returning the computation to its previous state. Figure 5.1 illustrates how the traversal of a hypothetical search tree would checkpoint. The branches in blue were counted before checkpointing and are not revisited, while the purple branches were also counted before checkpointing, but are traversed twice since they had to be visited again in order to return the search to its previous position. The red branches are enumerated once the traversal of the search tree restarts from the checkpoint. In this case the checkpoint file will store the current node at the time of checkpointing (labelled A in Figure 5.1). This node may be reached by choosing child three from four on the first level (*i.e.* $C_{1,3}$) and child three from three on the second (*i.e.* $C_{2,7}$); the traversal has encountered one node on level 0 thus far, three on level 1, seven on level 2 and three on level 3. The checkpoint also registers the time that a computation has taken to date so that the enumeration time reported upon completion reflects the actual total computation time. A new checkpoint replaces all older ones so that there is always at most one checkpoint available per result, thereby ensuring that the application restarts in its most recent location.

A final modification required is to allow the application to read input files from the logical file 'in', write results to the logical file 'out,' and read and write checkpoints to the logical file 'state.' Recall, from §4.2.2, that the result template, along with the BOINC API call `boinc_resolve_filename()`, may be used to convert logical names to actual file names for every individual result.

### 5.1.3 Deamons

Deamons were tasked with performing actions on the server, as described in §4.2.2. Work generation takes place using a bash[1] script, which loops through the forty five partially completed 3-MOLS of order 8 found after inserting the 0-universals into each of the three squares, and creates a corresponding work unit in the project database with an estimated runtime of $10^{14}$ floating-point operations, or approximately 720 hours, and a delay bound of ten days before the scheduler gives up on a dispatched result.

Replication was used to ensure the correctness of results returned: Two results were initially created for every work unit and the quorum was set to 2. Results are validated by a custom validator which compares the number of branches found on every level of the tree. It was not possible to use the default sample bitwise validator of BOINC for this purpose since the enumeration time, which may differ between different hosts, is also included in the result. The default scheduler and sample assimilator could, however, be used, since post-processing only consists of moving the results to a specific folder, which is exactly what the the sample assimilator does.

### 5.1.4 First enumeration results

The main classes of 3-MOLS of order 8 were enumerated by means of a BOINC distributed project involving five hosts on the same platform, namely Intel i7 processors operating within a Linux distribution.

A summary of the enumeration results is presented in Tables 5.1–5.3. Table 5.1 contains the total numbers of results dispatched to hosts, along with the corresponding numbers of validated, invalid and erroneous results received. A particularly high error rate was observed in the first section of the enumeration tree, where six of the work units exceeded their maximum number

[1]See [41] for more information on the bash shell.

Table 5.1: *Results issued in each section of the enumeration tree during the distributed enumeration of main classes of 3-MOLS of order 8, along with the resulting redundancy values.*

| Section | Nodes on level 0 | Work units | Results: Validated | Error | Invalid | Total | Redundancy |
|---|---|---|---|---|---|---|---|
| $z_1 z_2^2 z_3$ | 17 | 23 | 46 | 43 | 0 | 89 | 5.2 |
| $z_1 z_2^1 z_5$ | 14 | 14 | 28 | 1 | 2 | 31 | 2.2 |
| $z_1 z_3 z_4$ | 5 | 5 | 10 | 0 | 0 | 10 | 2 |
| $z_1 z_7$ | 9 | 9 | 18 | 0 | 0 | 18 | 2 |
| Total | 45 | 51 | 102 | 44 | 2 | 142 | 3.3 |

of allowable errors (in this case three) and had to be recreated. These errors played a large part in increasing the overall redundancy rate from 2 to 3.2 and the reason for these errors had to be investigated prior to further enumeration attempts. A break-down of the total number of erroneous, valid and invalid results for each host may be seen in Table 5.2, along with the total credit granted to each host for valid results and the total time required to carry out the computations.

Note that the host with identification number 2 did not connect to the project during this enumeration process and is thus excluded from the summary. It is clear from Table 5.2 that the majority of the errors occurred on host 4 and closer inspection showed that this was due to an error in the work generator which caused some results to be downloaded to volunteer hosts without the necessary accompanying files. The total enumeration time increased by a factor of approximately 5 from the enumeration in §3.3. This increase is considerably more than the overall redundancy of 3.2 would suggest, but is explained by the fact that the redundancy in the first section, which accounts for almost 90% of the total enumeration time reported in §3.3, is 5.2.

Table 5.2: *The number of erroneous (E), valid (V) and invalid (I) results computed by each of the hosts, as well as the credit granted and complete enumeration time required (in seconds) during the enumeration of 3-MOLS of order 8.*

| Host | $E$ | $V$ | $I$ | Credit | CPU Time |
|---|---|---|---|---|---|
| 1 | 4 | 2 | 0 | 1 027.64 | 211 469 |
| 3 | 1 | 36 | 2 | 4 969.13 | 730 630 |
| 4 | 29 | 43 | 0 | 6 104.15 | 1 621 431 |
| 5 | 8 | 14 | 0 | 10 488.90 | 1 352 450 |
| 6 | 2 | 7 | 0 | 5 566.54 | 713 200 |
| | 44 | 102 | 2 | 28 156.36 | 4 629 181 |

Both the smallest and largest successfully completed work units were computed on host 3, with enumeration times of 1.76 and 133 338 seconds, respectively. A work unit took, on average, 31 278.3 seconds to complete (approximately eight and a half hours), which is much more than the ideal work unit length of between one and six hours mentioned in various online sources [19]. The canonical (validated) number of branches counted on every level of the search tree, grouped into cycle structures, may be seen in Table 5.3; these results correspond with the counts in §3.3 and [53, p. 114].

## 5.2 Generalising to the enumeration of $k$-MOLS of order $n$

Although the enumeration of the main classes of 3-MOLS of order 8 described in the preceding section demonstrates that a volunteer computing project can, in principle, be used for the enumeration of main classes of $k$-MOLS, the approach of §5.1 is not suitable for the enumeration

TABLE 5.3: *The number of results issued in each section of the enumeration tree during the distributed enumeration of 3-MOLS of order 8, along with the canonical number of branches on each level of the search tree.*

| | Results | Branches on level | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Section | issued | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $z_1z_2^2z_3$ | 84 | 17 | 12 501 028 | 1 484 518 094 | 18 814 494 | 55 | 23 | 22 | 20 |
| $z_1z_2^1z_5$ | 31 | 14 | 3 358 273 | 61 708 802 | 63 157 | 97 | 92 | 84 | 17 |
| $z_1z_3z_4$ | 10 | 5 | 52 059 | 5 283 | 1 | 0 | 0 | 0 | 0 |
| $z_1z_7$ | 18 | 9 | 37 403 | 9 079 | 82 | 64 | 53 | 53 | 2 |
| Total | 142 | 45 | 15 948 763 | 1 546 241 258 | 318 877 734 | 216 | 168 | 159 | 39 |

of $k$-MOLS of order $n > 8$. For these larger enumeration problems the size and therefore the enumeration time required to traverse a subtree of the search tree is unknown before it has actually been enumerated. In the enumeration of 3-MOLS of order 8, traversing the smallest subtrees required approximately one second of computation time, while the largest ones required more than a day of computation time. For higher orders it is virtually assured that the subtrees rooted on level 0 will require months or even years of computation time. Expecting a single volunteer to complete even a month-long work unit is unreasonable and will most likely lead to a significant amount of wasted computing time due to both incomplete results and replication. In §5.1 the total enumeration time is also bound from below by the computation time required to complete the longest work unit, independently of the number of available hosts. A scenario is therefore conceivable where thousands of volunteers have completed all the available work units, except for a few that are still in progress and which may require excessive computing times. Indeed, this happened in the enumeration of 3-MOLS of order 8: After 60 000 seconds, only six results were still active, which means that the majority of the thirty six CPUs available between the five hosts were idle and unable to contribute for the remainder of the enumeration. Finally, highly variable work unit sizes are also impractical because it prohibits accurate estimations of the work unit size and renders useless the built-in protection of a work unit's `maximum_fpops` attribute against erroneous computations.

The work unit management policy described in the remainder of this section may perhaps be best understood by considering a hypothetical volunteer computing project involving four hosts and a single work unit of fifteen hours, as illustrated in Figure 5.2. In this example, the above-mentioned, potentially fatal concerns are, firstly, that the work unit is too long for computation on a single host and, secondly, that, although additional computing resources are available, the total computing time required is bounded from below by the serial computing time of this single work unit.

### 5.2.1 Limiting work unit sizes

Three ways of limiting work unit sizes are considered in this subsection, namely decreasing the size of work units by precomputation, splitting work units into smaller units and recycling results.

The first potential way of limiting work unit sizes is to perform some amount of precomputation so as to enumerate all the nodes in the search tree from the root down to a certain level, and only to assign volunteers work units from that level downwards. This approach is attractive due to its simplicity and was indeed adopted implicitly in §5.1 when the forty five nodes on level 0 of

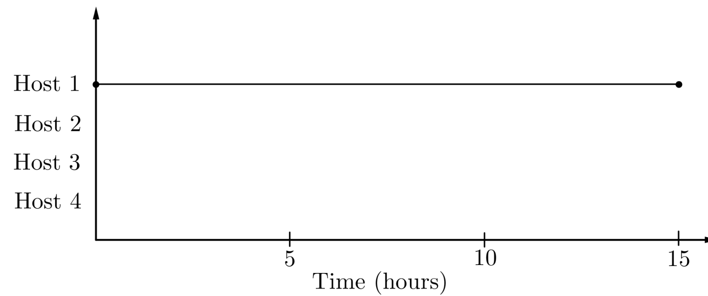



FIGURE 5.2: *A graphical illustration of a hypothetical volunteer project consisting of four hosts and a single result requiring fifteen hours of computation time under the approach of §5.1.*

the search tree for the main classes of 3-MOLS of order 8 were enumerated and used as starting points for the generation of work units, but it also has two serious drawbacks. Firstly, there is no way of knowing in advance how much precomputation would be required to enable the generation of suitably sized work units and for higher orders of MOLS these precomputations, which amount to a breadth-first exploration of the highest levels of the search tree, may be a daunting computational challenge in itself. For example, just inserting the 0-universals into empty 7-MOLS of order 9 required approximately ten days of computing time and for 5-MOLS of order 10 this step could not be completed in thirty days of continuous computation. Secondly, this approach dictates that all nodes at a certain level must be stored so that they may later be used as starting positions for the generation of work units. But just storing the approximately 1.5 billion nodes on level 2 of the enumeration tree for 3-MOLS of order 8 may prove cumbersome and for $k$-MOLS of orders $n > 8$ this problem will be exacerbated.

A second approach toward limiting work unit size involves splitting the subtree rooted at node $\mathcal{P}$ into a number of different subtrees, which remain rooted at $\mathcal{P}$. This can easily be accomplished by introducing a limit, or upper bound, on the position of the last candidate which should be considered by the current work unit alongside the starting position in the list of candidate universals (recall that the starting position was introduced for checkpointing purposes in §5.1.2). A single work unit consisting of a node with, say, 100 candidate universals on level $i$ may be split into five work units by specifying the [start, limit]-values for successive work units as $[0, 21], [20, 41], [40, 61], [60, 81]$ and $[80, 101]$. Note that, in this case, candidate universals 0 to 20 will be considered in the first work unit, since an upper bound of 21 was enforced. This allows for the generation of smaller work units without the need for storing further nodes of the search tree, but it does nothing to address the problems of wildly varying work unit sizes — the distribution of work unit sizes will be exactly what it was before, but scaled down by a constant factor (in the above example this constant factor would be five). Since the size of the search tree grows exponentially as a function of the MOLS order $n$, such a constant factor decrease in work unit size is unlikely to be effective as $n$ grows.

A third method of limiting work unit sizes, and the one which seems to hold the most potential, is to limit the computation time allocated to or the number of floating point operations allowed per work unit before returning a result. The approach of limiting the computing time allotted to a work unit introduces complications, since the actual amount of work done depends largely on host specifications and it will be difficult to validate computations that were interrupted at different stages of progress. It does, however, seem feasible to limit the amount of computation allowed per work unit by restricting the number of calls to a certain function. Limiting the number of calls to the function for testing whether a candidate universal is orthogonal to the current partially completed MOLS (part of step 9 of Algorithm 3.2 in §3.3) to, for example,

$5 \times 10^9$ ensures work units of approximately one hour in length and seems to be a practical approach toward limiting work unit size. After reaching this limit, the application is forced to checkpoint and return this checkpoint as a result, together with a tag signalling to a script on the server that the result is incomplete and should be reinserted into the project database as the starting position of a new work unit. If the work unit is completed before reaching the call limit, then the result is returned together with a tag signalling that it is a final result and that there is thus no need for recycling.

The effects of employing this policy of recycling work units after an hour of computation in the hypothetical volunteer computing project is illustrated in Figure 5.3. Recycling is successful in ensuring work units of a uniform size independently of the initial size of the subtree, but the hypothetical enumeration makes very poor use of the four available hosts — the total time to completion remains fifteen hours, since the work units are all computed in series. A further parallelisation is thus required to remove this dependence on the sizes of the original subtrees.

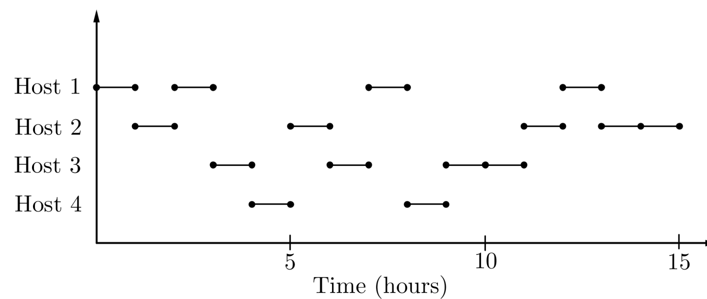



FIGURE 5.3: *The effect of the recycling policy on the hypothetical volunteer computing project consisting of four volunteer hosts and a single work unit requiring fifteen hours of computation time.*

### 5.2.2 Dynamic splitting of work units

The practice of traversing the different parts of a subtree of unknown size in series is likely to be impractical, but thanks to the way in which work units are repeatedly recycled, it is possible to improve on this dramatically on the serial enumeration time of a subtree.

A method for splitting subtrees into multiple smaller subtrees rooted at the same node was discussed in the previous section, but was discarded as a viable approach because it only offered a constant factor decrease in size, which is negligible if the subtree size grows exponentially. However, by constantly recycling results, repeated opportunities for such a splitting of subtrees are presented, which may lead to both significantly smaller work units and a further parallelisation in the traversal of a single subtree. The proposed solution is thus to split work units into two parts, as described in §5.2.1, before recycling them. Although work units may easily be split into more parts, it is only required to repeatedly split them into two parts so as to decrease the expected completion time (measured from the start of the traversal to the moment that the last portion of the traversal concludes, and

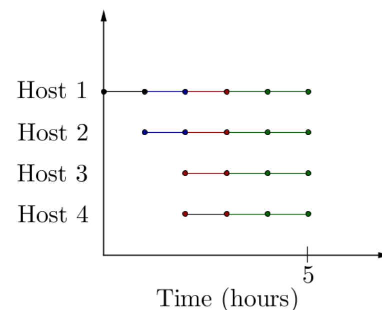



FIGURE 5.4: *A visual representation of the work performed by four hosts to complete a work unit of 15 hours under the recycling and splitting policy. Every line segment represents a single work unit, and work units of the same colour were created during the same round of recycling.*

not as the sum of the traversal times) exponentially, given sufficiently many hosts. Instead of the entire enumeration being bounded from below by the time that the traversal of the largest subtree takes, say $t$, it is thus now bounded from below by $\log_2 t$.

Figure 5.4 illustrates the impact of this approach towards the splitting of subtrees, when used in conjunction with the recycling policy, in the same hypothetical volunteer computing project as considered in Figures 5.3 and 5.2. It is clear that the subtree will be fully traversed much earlier under this work unit management policy, and also that it leads to a much more efficient utilization of the available resources.

### 5.2.3 Implementing and validating the generalisation

In order to facilitate the recycling and dynamic splitting of subtrees, the formats of the input, checkpoint and result files were changed in such a way that checkpoints could be returned as results and that results could be re-issued as starting positions. An example of a file in this format is shown in Table 5.4. An A on the first line means that the work unit has been completed, while a B signals that it is a checkpoint for recycling. Furthermore, the file sections containing the positions, current branch counts, time, *etc.* (shown in grey) are optional components.

It was decided to split work units only when the remaining number of unsent results do not exceed the total that could potentially be assigned to the hosts had they all requested work simultaneously, in other words, whenever there was a chance of under-utilizing some of the hosts. This practice typically prevents a large backlog of work units building up due to the incessant splitting of work units. The policy of recycling and dynamic splitting of work units was

Table 5.4: *The general file format used for starting positions, checkpoints and completed results in the enumeration of main classes of $k$-MOLS of order $n$. Everything after the starting position (coloured gray) is optional.*

| General file format | Description |
|---|---|
| `@B` | A - completed, B - recycle |
| `8 3` | $n$ $k$ |
| `01234567 02143675 03412756` | Starting position |
| `#Positions` | Start, limit and number of candidates for every universal and every Latin square |
| `-1 -1 -1 -1 -1 -1 -1 -1 -1` | |
| `284 2120 2119 592 2120 2119 1001 2120 2119` | |
| `634 784 783 146 793 792 92 792 791` | |
| `-1 255 254 -1 251 250 -1 250 249` | |
| `-1 69 68 -1 68 67 -1 68 67` | |
| `-1 15 14 -1 14 13 -1 10 9` | |
| `-1 -1 -1 -1 -1 -1 -1 -1 -1` | |
| `-1 -1 -1 -1 -1 -1 -1 -1 -1` | |
| `705032706` | Number of calls to `isOrthogonal` |
| `#Branchcounts` | Number of branches counted on every level of the tree |
| `0 0 0 55 6365 25137 423490...` | |
| `#Total time` | Enumeration time for this portion of the tree |
| `1239.4 seconds` | |
| `#MOLS found` | Number of MOLS found in this portion of the tree |
| 0 | |

**Algorithm 5.1**: Split and recycle results

**input** : A list, $\mathcal{L}$, of received output files
**output**: New work units are created where needed

```
1 begin
2   for every output file f in L do
3     if f is a checkpoint then
4       if the number of unsent results is below a certain level then
5         Split the remaining work into two parts and create a new work unit for each
6       else
7         Recycle the checkpoint file by creating a new work unit
8     Move f to the a folder of completed results
9 end
```

implemented on the BOINC project server in the form of a Python [80] script which periodically executes after the assimilator has moved all the completed results to a folder. Every completed result marked as a checkpoint may be split, depending on the current level of unassigned work units, and is recycled as a new work unit. Completed work units are moved to a different folder where the current traversal state of the search tree is updated. A pseudo-code version of this script is given in Algorithm 5.1.

The policy was tested for feasibility at the ninth node on level 0 of the enumeration tree for main classes of 3-MOLS of order 8, since this was the node on level 0 producing the largest subtree. The details of the resulting enumeration, with a redundancy factor of 2, may be seen in Table 5.5. Note that the length of the longest work unit was approximately eighty minutes, compared to the more than thirty seven hours required during the serial enumeration. The computation time associated with an average work unit decreased to approximately forty minutes which, although perhaps somewhat short, carries much less risk of lost computation time than before. Very little overhead was involved in handling multiple work units, since the average computation time per replication of 134 977.7 seconds compares rather favourably to the serial enumeration time of 132 480.7 seconds. Furthermore, the computation was divided much more evenly between the hosts under the recycling and dynamic splitting policy described above (§5.2.1 – §5.2.2) and the final result was received approximately twenty one hours after the generation of the first work unit — a significant improvement on the thirty eight hours required by the serial enumeration.

Table 5.5: *A comparison of the traversal of the subtree from the ninth node on level 0 of the enumeration search tree for main classes of 3-MOLS of order 8.*

| Policy | Host | Valid | Credit | CPU Time | Smallest wu | Largest wu | Avg. wu |
|---|---|---|---|---|---|---|---|
| §5.1.2 | 3 | 2 | 1 868.22 | 265 818.7 | 132 480.7 | 133 338 | 132 909.35 |
| §5.2 | 1 | 46 | 1 377.08 | 111 586.0 | 0.01 | 3 856.28 | 2 425.78 |
| | 3 | 24 | 516.17 | 50 022.0 | 0.01 | 4 570.62 | 2 084.25 |
| | 5 | 2 | 15.76 | 1 174.1 | 585.52 | 588.54 | 587.03 |
| | 6 | 34 | 1 887.41 | 107 173.4 | 0.01 | 4 266.06 | 3 152.16 |
| | Total | 106 | 3 796.43 | 269 955.4 | 0.01 | 4 570.62 | 2 546.75 |

## 5.3 Enumeration results emanating from an implementation

Following the successful test on the ninth node on level 0 of the enumeration tree of 3-MOLS of order 8 described in §5.2.3, the recycling and dynamic splitting policy was employed in a complete enumeration of main classes of 3-MOLS of order 8, the results of which are summarised in Tables 5.6–5.7. It may be seen in Table 5.6 that the number of errors, and therefore also the overall replication rate, has decreased dramatically from §5.1. The 19 results that reported errors were all due to volunteers manually aborting results. As expected, the policy of recycling and dynamically splitting work units lead to significantly more work units being created than in the original distributed enumeration summarised in Table 5.2. When employing this work unit management policy in the traversal of larger search trees, care must be taken to ensure that sufficient storage space is available for new work units. The length of the largest work unit is less than three hours, well within the accepted range, and the average work unit length of approximately 45 minutes may be slightly increased in future enumerations to lighten the load on the server.

This success of this pilot study suggests use of this volunteer computing project to attempt at finding novel enumeration results. To start this ambitious endeavour, the enumeration instance with the smallest expected computation time was selected. Based on the information in Table 3.6 and §3.4, it is expected that the main classes of 8-MOLS of order 9, or those of 7-MOLS of order 9, will be the easiest to enumerate. As was, however, mentioned in §3.2, the number of main

TABLE 5.6: *The number of results issued in each section of the enumeration tree during the distributed enumeration of 3-MOLS of order 8 under the recycling and splitting work unit management policy, along with the resulting redundancy.*

| Section | Nodes on level 0 | work units | Results | | | | Redundancy |
|---|---|---|---|---|---|---|---|
| | | | Validated | Error | Invalid | Total | ($r$) |
| $z_1z_2^2z_3$ | 17 | 474 | 948 | 19 | 12 | 979 | 2.06 |
| $z_1z_2^1z_5$ | 14 | 50 | 100 | 0 | 0 | 100 | 2 |
| $z_1z_3z_4$ | 5 | 5 | 10 | 0 | 0 | 10 | 2 |
| $z_1z_7$ | 9 | 9 | 18 | 0 | 0 | 18 | 2 |
| Total | 45 | 538 | 1 076 | 19 | 12 | 1 110 | 2.06 |

TABLE 5.7: *A summary of the work done by every host that contributed to the enumeration of the main classes of 3-MOLS of order 8, including the total computation time performed by every host and the length of the shortest, average and longest work unit it computed.*

| Host | Errors | Valid | Invalid | CPU Time | Smallest wu | Largest wu | Avg. wu |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 277 | 0 | 756996.0 | 0.01 | 4536 | 2732 |
| 3 | 19 | 110 | 1 | 477098.9 | 1335 | 9445 | 3669 |
| 4 | 0 | 261 | 2 | 781637.5 | 0.01 | 6731 | 2972 |
| 5 | 0 | 121 | 1 | 267248.4 | 0.01 | 5003 | 2190 |
| 6 | 0 | 94 | 6 | 357733.1 | 0.01 | 8950 | 3577 |
| 7 | 0 | 61 | 2 | 160880.3 | 0.00 | 6612 | 2553 |
| 9 | 0 | 87 | 0 | 193267.3 | 0.01 | 4391 | 2221 |
| 10 | 0 | 65 | 0 | 97090.8 | 0.01 | 4394 | 1493 |
| Total | 19 | 1076 | 12 | 3091952.4 | 0 | 9 445 | 2 786 |

classes of 8-MOLS of order 9 is known due to a result by Owens and Preece [76] and so it was decided to launch a much larger pilot study with the long-term goal of enumerating the main classes of 7-MOLS of order 9.

Thanks to the generosity of the computing divisions of the Faculties of Natural Sciences and Economics at Stellenbosch University, it was possible to add a further 190 computers with Intel i5 processors to the twelve volunteer hosts originally used in the distributed enumeration of main classes of 3-MOLS of order 8. The maximum daily throughput of this distributed computing project is approximately 2 000GHz-days, compared to the estimated daily throughput of 1 250GHz-days of the high performance cluster at Stellenbosch University.

The search tree for main classes of 7-MOLS of order 9 consists of six sections and a total of 6 670 346 nodes on level 0, significantly more than the 45 nodes on level 0 in the enumeration search tree for main classes of 3-MOLS of order 8. The distribution of the nodes on level 0 over the different sections of the enumeration tree for 7-MOLS of order 9 may be seen in Table 5.8 and for all $k$-MOLS of order 9 in Table 3.11 in §3.4.

Table 5.8: *The distribution of nodes on level 0 over the different sections of the enumeration search tree for main classes of 7-MOLS of order 9.*

| Section | Nodes |
|---|---|
| $z_1z_2^4$ | 926 486 |
| $z_1z_2^2z_4$ | 4 727 973 |
| $z_1z_2z_3^2$ | 776 991 |
| $z_1z_2z_6$ | 238 243 |
| $z_1z_3z_5$ | 647 |
| $z_1z_4^2$ | 6 |
| $z_1z_8$ | 0 |
| 9 | 6 670 346 |

The two sections corresponding to the cycle structure representatives $z_1z_3z_5$ and $z_1z_4^2$, containing 647 and 6 nodes on level 0, respectively, have been traversed completely without finding any 7-MOLS of order 9, or even a single node on level 1 of the search tree. The absence of a 7-MOLS of order 9 (which are known to exist) in these two sections is not surprising, because together they contain only 653 of the 6 670 346 nodes on level 0 of the search tree.

Traversal of the sections of the enumeration search tree for main classes of 7-MOLS of order 9 corresponding to the cycle structure representatives $z_1z_2z_6$ and $z_1z_2z_3^2$ is currently in progress and a summary of the current state of the enumeration in these sections may be seen in Table 5.9. The subtrees of approximately 34% of the nodes on level 0 of section $z_1z_2z_6$ have been traversed without uncovering a main class 7-MOLS of order 9. The subtrees rooted at a tenth of the nodes in the section $z_1z_2z_3^2$ have also been traversed, leading to the discovery of three complete 7-MOLS of order 9, however, all three the completed 7-MOLS had smaller paratopes and were eventually discarded. Although no main class representatives have thus yet been found, this does mean that there is at least one main class of 7-MOLS of order 9 (this is, of course, also implied by the existence of 8-MOLS of order 9).

It is hoped that this enumeration may be continued until completion. Insights gained form this enumeration attempt may prove pivotal to the success of using volunteer computing to obtain further novel enumeration results.

Table 5.9: *The aggregated computation time and number of nodes encountered thus far on every level within the two sections of the enumeration search tree for main classes of 7-MOLS of order 9 corresponding to the cycle structure representatives $z_1z_2z_6$ and $z_1z_2z_3^2$ currently being computed.*

| | Level | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Section | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time ($s$) |
| $z_1z_2z_3^2$ | 82 872 | 20 860 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 581 566 |
| $z_1z_2z_6$ | 225 140 | 26 571 | 34 | 3 | 3 | 3 | 3 | 3 | 0 | 2 827 864 |

## 5.4 Chapter summary

This chapter contains a description of the process of grid-enabling the exhaustive algorithm of §3.3 for enumerating main classes of $k$-MOLS of order $n$. A simple BOINC project for the enumeration of main classes of 3-MOLS of order 8 was described in §5.1. This description included the specifications of the BOINC server in §5.1.1, the changes required to the original enumeration algorithm in §5.1.2 and a summary of the deamons implemented in §5.1.3. The implementation of the distribution of Algorithm 3.1 was verified by replicating the results in Table 3.4.

Although this first enumeration attempt successfully distributed the computation to volunteers, the distribution schema is impractical for higher-order search spaces, chiefly due to the effect of subtrees of unknown sizes in the enumeration search tree. A work unit management policy was proposed in §5.2 firstly to limit work unit sizes by recycling results, as described in §5.2.1, and secondly to improve the utilisation of available resources, as described in §5.2.2. This is accomplished by reducing the time required to traverse a large subtree from $t$ seconds, if performed serially, to $\log_2 t$ seconds using parallelism as a result of the splitting of work units.

A pilot study incorporating this work unit management policy was launched for the enumeration of main classes of 3-MOLS of order 8 in §5.3. This pilot study was a success, both in terms of the actual enumeration results and the suitability of the work unit sizes assigned to volunteers. This confirmed the potential that volunteer computing holds for the enumeration of equivalence classes of MOLS and resulted in the launch of a much larger and ongoing enumeration of main classes of 7-MOLS of order 9. This enumeration makes use of computing resources volunteered by the Faculty of Commerce at Stellenbosch University, and partial results obtained thus far were reported.

# CHAPTER 6

# Conclusion

Contents

6.1 Overview of the work contained in this thesis . . . . . . . . . . . . . . . . . . . 69

6.2 An appraisal of the contributions of this thesis . . . . . . . . . . . . . . . . . . 71

6.3 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

In the first section of this final chapter, the work contained in this thesis is briefly recounted. This is followed by an appraisal of the contributions of this thesis. The chapter closes with a brief reflection on possible future avenues for research arising from the work documented in this thesis.

## 6.1 Overview of the work contained in this thesis

A volunteer computing project was designed in this thesis for the enumeration of main classes of $k$-MOLS of order $n$ in an attempt to overcome the current computational barrier that previous enumeration attempts have encountered.

The prerequisite mathematical notions required for this endeavour were reviewed in Chapter 2, in partial fulfilment of Research Objective I of §1.3. In §2.1, the notion of a permutation was introduced and this was followed by a very brief review of a number of group theoretic notions which have a direct bearing on the theory of Latin squares. The relevant theory underlying Latin squares was discussed in §2.3, with a specific emphasis on the notion of orthogonality between Latin squares and sets of Latin squares in §2.3.2 and the various main class invariant operations which may act on a single Latin square or $k$-MOLS in §2.3.3.

In Chapter 3, the problem of counting structurally distinct Latin squares and MOLS was considered. In §3.1, it was shown how Latin squares (and MOLS) may be partitioned into equivalence classes based on transformations of specific types which act on them. This, together with the historical overview of previous work on the enumeration of the various equivalence classes of Latin squares and MOLS in §3.2, fulfils Research Objective II of §1.3. An exhaustive backtracking algorithm was developed for the enumeration of main classes of $k$-MOLS of order $n$ in §3.3. This algorithm finds a single representative in each of the main classes of a $k$-MOLS

of order $n$ by traversing a search tree and pruning away branches whenever possible. Even for MOLS of relatively small orders such as $n = 9$ and $n = 10$, however, this search approach becomes computationally very expensive. Numerical results on the enumeration of main classes of MOLS obtained by this algorithm were also reported in §3.3. These results served to validate the correctness of the algorithmic implementation and to provide empirical evidence showing that the implementation is, at least in specific cases, more efficient than previous implementations found in the literature. The enumeration of main classes of MOLS of orders 9 and higher, however, remain computationally infeasible using the algorithmic approach of §3.3. Techniques for estimating the size of the relevant search trees were explored in §3.4. Estimates of the sizes of the search trees and potential run-times associated with the enumeration of main classes of $k$-MOLS of order 9 and higher were provided in §3.4. Research Objectives IV and V of §1.3 were therefore fulfilled in §3.3 and §3.4, respectively.

The notion of volunteer computing was introduced in Chapter 4 as a potential way of overcoming the computational barrier currently prohibiting the enumeration of main classes of MOLS of orders 9 and, more importantly, order 10. The history of distributed computing in general, and specifically volunteer computing, was reviewed in §4.1. The components and working of BOINC, the predominant middleware used in volunteer computing today, were examined in some detail in §4.2. Attention was also given to special types of applications which make use of paralellism or GPUs, as well as setting up a server for a volunteer computing project. The chapter closed with an exposition on the security concerns and challenges related to the widespread adoption of volunteer computing. This chapter further contributed to the fulfilment of Research Objective I of §1.3.

In Chapter 5, a volunteer computing project was designed for the enumeration of main classes of MOLS. A project server was set up in a virtual machine and Algorithm 3.1 was modified to incorporate checkpointing and the relevant calls to the BOINC application programming interface. The main classes of 3-MOLS of order 8 were successfully enumerated, but a number of factors related to the way in which subtrees of unknown size were traversed raised concerns about the feasibility of this approach towards enumerating main classes of MOLS of higher orders. A work unit management policy was therefore proposed in §5.2 to ensure that work units do not exceed a certain size, that the available computing resources are efficiently used and that the overall computation time is not bounded from below by the time it takes to traverse the largest subtree on a given level of the enumeration search tree. The policy, together with its implementation, was verified by traversing the largest subtree rooted on level 1 of the search tree for 3-MOLS of order 8. The design of the project, together with the work management policy, fulfil Research Objective VI of §1.3.

Following the success of this test, a pilot study was launched locally to fully enumerate the main classes of 3-MOLS of order 8 again and to confirm that the work unit management policy allows the volunteer computing project to generalize to $k$-MOLS of any order while making efficient use of resources and without placing unnecessary loads on the project server. The successful pilot study stands in fulfilment of Research Objective VII of §1.3. Because using volunteer computing for the enumeration of main classes of mutually orthogonal Latin squares seems wholly feasible, the lengthy process of enumerating main classes of 7-MOLS of order 9 was initiated, since this is the smallest instance for which the number of main classes remains unknown. Although some preliminary results were obtained for this enumeration instance in §5.3, the enumeration attempt is still ongoing and makes use of approximately 150 computers from the Faculty of Commerce at Stellenbosch University. The success of the pilot study and the ongoing enumeration of main classes of 7-MOLS of order 9 are in fulfilment of Research Objective VIII of §1.3.

## 6.2 An appraisal of the contributions of this thesis

The main contributions of this thesis are threefold. The first contribution is an extension to the work of Kidd [53], who enumerated the main classes of $k$-MOLS of orders $3 \leq n \leq 8$. In addition to verifying the enumeration results reported in [53], estimates for the sizes of the enumeration search trees for $k$-MOLS of orders 9 and 10 were provided in this thesis, along with a discussion on the shape and nature of these search trees. Evidence was also provided in §3.4 that the universal $u_0^{(1)}$ plays an important, hitherto unrecognised role in determining the structure of the subtrees containing that cycle structure representative. This contribution was published in [8].

The second contribution of this thesis is the design of a distributed volunteer computing project aimed at overcoming the computational barrier encountered in previous attempts at enumerating main classes of $k$-MOLS. The volunteer computing project was designed using the BOINC middleware system and incorporates a work unit scheduling policy making it possible to enumerate search trees of arbitrary sizes effectively. This volunteer computing project has the added benefit that the work unit management policy is likely to be equally valid for recursive enumeration algorithms commonly associated with a number of other combinatorial problems.

The final contribution of this thesis is a pilot study launched as a proof-of-concept experiment to show that volunteer computing may indeed be a viable approach towards enumerating main classes of $k$-MOLS of order $n \geq 9$. This finding increases the tools available to researchers attempting to answer the celebrated question of the existence of a 3-MOLS of order 10. A journal paper on this pilot study is in its final stages of preparation, and is expected to be submitted soon for publication.

## 6.3 Future work

This section contains a number of suggestions for future work related to Latin square enumeration that emerged while this research was conducted. The first two suggestions are focused on the volunteer computing project for the enumeration of main classes of MOLS.

**Proposal 6.1.** *Publicly launch the volunteer computing project designed in Chapter 5 for the enumeration of main classes of MOLS, potentially in collaboration with a university department or a large corpration.*

Various challenges related to the security of a volunteer computing project were safely ignored in this thesis, since the project was only launched on Stellenbosch University's internal network and therefore not exposed to potentially malicious external attacks. These security concerns, as well as challenges related to attracting and retaining volunteers, need to be investigated before a public launch is to be successful.

**Proposal 6.2.** *Investigate whether the search may be sped up significantly by further parallellising the computation of work units.*

As was mentioned in §4.2.3, BOINC facilitates parallellised CPU and GPU processing, although the current application, documented in Chapter 5, only makes use of a single CPU at a time. It is possible that the computation time may be decreased by further parallelisation.

The following four potential avenues for further research deal with various aspects of the existence, classification and enumeration of the equivalence classes of sets of mutually orthogonal Latin squares considered in this thesis.

**Proposal 6.3.** *Attempt to generalise recent state-of-the art algorithms for the problem of group isomorphims to quasigroups and, by extension, MOLS.*

Recent developments in computer science, chiefly by Rosenbaum [84], have led to a decrease in the complexity of the best-known test for whether two groups are isomorphic, in other words, whether their respective Cayley tables may be relabelled to be equal. If a similar generalisation to quasigroups is possible, an algorithm of the same order of complexity will apply to MOLS. If such an algorithm is available, it suggests an alternative enumeration approach for the equivalence classes of MOLS. The algorithm may then be used to construct a list of smallest partial MOLS from each of the different transformation classes encountered up to that point on every level of the enumeration tree. New partial MOLS may either be inserted into the list or pruned away. If sufficient storage space is available, such an algorithm may well rival Algorithm 3.1.

**Proposal 6.4.** *Pursue better estimates of the enumeration search tree sizes for MOLS of higher orders to gain further insight into any potentially exploitable structures of these trees.*

A number of techniques exist for finding accurate, early predictions for branch-and-bound tree sizes, some of which may be adapted for general backtrack search trees. An example of such a technique is that of Cournuéjols *et al.* [26].

**Proposal 6.5.** *Study the properties of the transformation class generated by* $(\pi_r, \pi_c, \epsilon)$*-type transformations of MOLS.*

The author is unaware of any previous or current investigations into the transformation class generated by the $(\pi_r, \pi_c, \epsilon)$-transformations, which are principal isotopisms and additionally allow a conjugate operation.

**Proposal 6.6.** *Compare the computation time of techniques which only find a single solution (as opposed to enumerating all solutions), such as certain linear and constraint programming techniques, or any number of metaheuristics, to the time Algorithm 3.1 takes to find the first class representative.*

A number of construction techniques for $k$-MOLS were not considered in this thesis, but may be included in a study primarily focussed on attempting to prove the existence (or otherwise) of a 3-MOLS of order 10.

# References


[1] Allen B, 2005, *Einstein@Home*, [Online], [Cited June 12$^{th}$, 2014], Available from http://einstein.phys.uwm.edu/.

[2] Allenby RBJT, 1991, *Rings, fields, and groups: An introduction to abstract algebra*, Butterworth-Heineman, Oxford.

[3] Anderson DP, 2004, *BOINC: A system for public-resource computing and storage*, Proceedings of the 5$^{th}$ IEEE/ACM International Workshop on Grid Computing, Pittsburgh (PA), pp. 4–10.

[4] Anderson DP, 2013, *BOINC status and plans*, Proceedings of the 9$^{th}$ BOINC workshop, Grenoble.

[5] Anderson DP, 2003, *Public computing: Reconnecting people to science*, Proceedings of the Conference on Shared Knowledge and the Web, Madrid, pp. 17–19.

[6] Anderson DP, Cobb J, Korpela E, Lebofsky M & Werthimer D, 2002, *SETI@home: An experiment in public-resource computing*, Communications of the Association for Computing Machinery, **45(11)**, pp. 56–61.

[7] Bachet CG, 1884, *Problèmes plaisants & délectables: Qui se font par les nombres*, Gauthier-Villars, Paris.

[8] Benadé JG, Burger AP & van Vuuren JH, 2013, *The enumeration of k-sets of mutually orthogonal Latin squares*, Proceedings of the 42$^{th}$ Conference of the Operations Research Society of South Africa, Stellenbosch, pp. 40–49.

[9] BitTorrent, 2013, *BitTorrent*, [Online], [Cited July 21$^{st}$, 2013], Available from http://www.bittorrent.com.

[10] Blumberg M, 2013, *GridRepublic and Progress Thru Processors status report*, [Online], [Cited September 25$^{th}$, 2013], Available from http://boinc.berkeley.edu/trac/wiki/WorkShop13.

[11] Bóna M, 2004, *Combinatorics of permutations*, CRC Press, Boca Raton (FL).

[12] Bose RC, 1938, *On the application of the properties of Galois fields to the problem of constrution of hyper-graeco-latin squares*, Sankhya: The Indian Journal of Statistics, **3**, pp. 323–338.

[13] Bose RC, Shrikhande SS & Parker ET, 1960, *Further results on the construction of mutually orthogonal Latin squares and the falsity of Euler's conjecture*, Canadian Journal of Mathematics, **12**, pp. 189–203.

[14] Bose RC & Shrikhande SS, 1959, *On the falsity of Euler's conjecture about the non-existence of two orthogonal Latin squares of order* $4t + 2$, Proceedings of the National Academy of Sciences of the United States of America, **45(5)**, pp. 734–737.

[15] Bose R & Nair K, 1941, *On complete sets of Latin squares*, Sankhya: The Indian Journal of Statistics, **5(4)**, pp. 361–382.

[16] Brown JW, 1968, *Enumeration of Latin squares with application to order 8*, Journal of Combinatorial Theory, **5(2)**, pp. 177–184.

[17] Bruck RH & Ryser HJ, 1949, *The nonexistence of certain finite projective planes*, Canadian Journal of Mathematics, **1(191)**, pp. 88–93.

[18] Brunner J, 1975, *The Shockwave Rider*, Harper & Row, New York (NY).

[19] Buck PD, 2011, *Workunit size*, [Online], [Cited September 29th, 2013], Available from http://blog.gmane.org/gmane.comp.distributed.boinc.user/month=20110501.

[20] Burger AP, Kidd MP & Van Vuuren JH, 2010, *Enumerasie van self-ortogonale Latynse vierkante van orde 10*, LitNet Akademies (Natuurwetenskappe), **7(3)**, pp. 1–22.

[21] Cayley A, 1890, *On Latin squares*, Messenger of Mathematics, **19**, pp. 135–137.

[22] CERN, 2013, *LHC@Home*, [Online], [Cited June 12th, 2013], Available from http://lhcathome.web.cern.ch/.

[23] climateprediction.net, 2003, *climateprediction.net — The world's largest climate forecasting experiment for the 21st century*, [Online], [Cited June 12th, 2013], Available from http://www.climateprediction.net/.

[24] Colbourn CJ & Dinitz JH, 2006, *Handbook of combinatorial designs*, Chapman & Hall/CRC, Boca Raton (FL).

[25] Computation Institute, 2013, *Globus Toolkit*, [Online], [Cited July 21st, 2013], Available from http://www.globus.org/.

[26] Cornuéjols G, Karamanov M & Li Y, 2006, *Early estimates of the size of branch-and-bound trees*, INFORMS Journal on Computing, **18(1)**, pp. 86–96.

[27] Corporation SGI, 1997, *OpenGL — The industry's foundation for high performing graphics*, [Online], [Cited June 14th, 2013], Available from http://www.opengl.org/.

[28] Dénes J & Keedwell AD, 1991, *Latin squares: New developments in the theory and applications*, North Holland, New York (NY).

[29] Dénes J & Keedwell AD, 1974, *Latin squares and their applications*, English Universities Press, London.

[30] distributed.net, 1997, *distributed.net*, [Online], [Cited June 12th, 2013], Available from http://www.distributed.net/.

[31] Erdös P, 1977, *A note of welcome*, Journal of Graph Theory, **1(1)**, pp. 3.

[32] Euler L, 1782, *Recherches sur une nouvelle espéce de quarrés magiques*, Verhandelingen uitgegeven door het zeeuwsch Genootschap der Wetenschappen te Vlissingen, **9**, pp. 85–239.

[33] Euler L, 1849, *De quadratis magicis*, Commentationes Arithmeticae, **2**, pp. 593–602.

[34] Evans D, 2011, *The internet of things — How the next evolution of the internet is changing everything*, (Unpublished) Technical Report, Cisco Internet Business Solutions Group.

[35] Fang W, 2013, *BOINC volunteer community in China*, [Online], [Cited September 25th, 2013], Available from http://boinc.berkeley.edu/trac/wiki/WorkShop13.

[36] Fisher RA, 1942, *Completely orthogonal* $9 \times 9$ *squares. A correction*, Annals of Eugenics, **11**, pp. 402.

[37] Fisher RA, 1925, *Statistical methods for research workers*, Oliver & Boyd, Edinburgh.

[38] Fisher RA, 1935, *The design of experiments*, Oliver & Boyd, Edinburgh.

[39] Fisher RA & Yates F, 1934, *The* $6 \times 6$ *Latin squares*, **30**, pp. 492–507.

[40] Foster I & Kesselman C (Eds), 1999, *The grid: Blueprint for a new computing infrastructure*, Morgan Kaufmann Publishers Inc., San Francisco (CA).

[41] Free Software Foundation, 2007, *Bash 4.2*, [Online], [Cited September 29th, 2013], Available from http://ftp.gnu.org/gnu/bash/bash-3.2.48.tar.gz.

[42] Frolov M, 1890, *Recherches sur les permutations carrees*, Chateau de la Grave, Bonzac.

[43] Gardner M, 2000, *Modeling mathematics with playing cards*, The College Mathematics Journal, **31(3)**, pp. 173–177.

[44] Gepner P & Kowalik MF, 2006, *Multi-core processors: New way to achieve high system performance*, Proceedings of the International Symposium on Parallel Computing in Electrical Engineering, Bialystok, pp. 9–13.

[45] Gunther S, 1876, *Mathematisch-historiche Missellen II, Die magischen quadrate bei Bauss*, Journal of Mathematical Physics, **21**, pp. 61–64.

[46] Hall M, Swift JD & Walker RJ, 1956, *Uniqueness of the projective plane of order eight*, Mathematical Tables and Other Aids to Computation, **10(56)**, pp. 186–194.

[47] Hayes B, 1998, *Computing science Collective wisdom*, American Scientist, **86(2)**, pp. 118–122.

[48] Hsiao MY, Bossen DC & Chien RT, 1970, *Orthogonal Latin square codes*, IBM Journal of Research and Development, **14(4)**, pp. 390–394.

[49] International Business Machines Corporation, 2013, *World Community Grid — Technology solving problems*, [Online], [Cited June 12th, 2013], Available from http://www.worldcommunitygrid.org/.

[50] Keedwell AD, 2000, *Designing tournaments with the aid of Latin squares: a presentation of old and new results*, Utilitas Mathematica, **58**, pp. 65–86.

[51] Khronos Group, 2013, *The open standard for parallel programming of heterogeneous systems*, [Online], [Cited June 12th, 2013], Available from http://www.khronos.org/opencl/.

[52] Kidd MP, 2010, *A tabu-search for minimising the carry-over effects value of a round-robin tournament*, ORiON, **26(2)**, pp. 125–141.

[53] Kidd MP, 2012, *On the existence and enumeration of sets of two or three mutually orthogonal Latin squares with application to sports tournament scheduling*, PhD thesis, Stellenbosch University, Stellenbosch.

[54] Knuth DE, 1975, *Estimating the efficiency of backtrack programs*, Mathematics of computation, **29(129)**, pp. 122–136.

[55] Lam CWH, 1991, *The search for a finite projective plane of order 10*, American Mathematical Monthly, **98(4)**, pp. 305–318.

[56] Laywine CF & Mullen GL, 1998, *Discrete mathematics using Latin squares*, Wiley & Sons, New York (NY).

[57] MacInnes C, 1907, *Finite planes with less than 8 points on a line*, American Mathematics Monthly, **14**, pp. 171–174.

[58] MacMahon P, 1898, *A new method in combinatory analysis, with applications to Latin squares and associated questions*, Transcripts of the Cambridridge Philosophical Society, **16**, pp. 262–290.

[59] Mann HB, 1942, *The construction of orthogonal Latin squares*, The Annals of Mathematical Statistics, **13(4)**, pp. 418–423.

[60] Max Planck Instute for Gravitaional Physics, 2013, *Einstein@Home on Android devices*, [Online], [Cited July 18th, 2013], Available from http://www.aei.mpg.de/287641/boinc_android.

[61] McKay BD, 2014, *Combinatorial data*, [Online], [Cited February , Available from http://cs.anu.edu.au/~bdm/data/latin.html.

[62] McKay BD, Meynert A & Myrvold W, 2007, *Small Latin squares, quasigroups, and loops*, Journal of Combinatorial Designs, **15(2)**, pp. 98–119.

[63] Mersenne Research I, 2013, *Great Internet Mersenne Prime Search*, [Online], [Cited March 28th, 2013], Available from http://www.mersenne.org/.

[64] Message Passing Interface Forum, 2013, *MPI*, [Online], [Cited July 21st, 2013], Available from http://www.mpi-forum.org/.

[65] Minnick RC, Elspas B & Short RA, 1963, *Symmetric Latin squares*, IEEE Transactions on Electronic Computers, **12(2)**, pp. 130–131.

[66] Minnick RC & Haynes JL, 1962, *Magnetic core access switches*, IRE Transactions on Electronic Computers, **11(3)**, pp. 352–368.

[67] mobiThinking, 2013, *Global mobile statistics 2013 Part A: Mobile subscribers; handset market share; mobile operators*, [Online], [Cited May 6th, 2013], Available from http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/.

[68] Napster L, 2001, *Napster*, [Online], [Cited July 21st, , Available from http://www.napster.com.

[69] NorduGrid, 2013, *Advanced Resource Connector*, [Online], [Cited July 22nd, 2013], Available from http://www.nordugrid.org/arc/.

[70] Norton DA, 1952, *Groups of orthogonal row-Latin squares*, Pacific Journal of Mathematics, **2(3)**, pp. 335–341.

[71] Norton HW, 1939, *The* $7 \times 7$ *squares*, Annals of Eugenics, **9(3)**, pp. 269–307.

[72] NVIDIA Corporation, 2013, *CUDA Parallel computing platform*, [Online], [Cited June 12th, 2013], Available from http://www.nvidia.com/object/cuda_home_new.html.

[73] OpenMP Architecture Review Board, 2013, *The OpenMP API specification for parallel programming*, [Online], [Cited July 21st, 2013], Available from http://www.openmp.org/wp/.

[74] Oracle, 2013, *Oracle Grid Engine*, [Online], [Cited July 22nd, 2013], Available from http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html.

[75] Oram A, 2001, *Peer-to-peer: Harnessing the benefits of a disruptive technologies*, O'Reilly Media, Inc., Newtown (MA).

[76] Owens P & Preece D, 1995, *Complete sets of pairwise orthogonal Latin squares of order 9*, Journal of Combinatorial Mathematics and Combinatorial Computing, **18**, pp. 83–96.

[77] Pande Lab, 2013, *Folding@Home*, [Online], [Cited June 12th, 2014], Available from http://folding.stanford.edu/.

[78] Parker ET, 1959, *Construction of some sets of mutually orthogonal Latin squares*, Proceedings of the American Mathematical Society, **10(6)**, pp. 946–949.

[79] Purdom PW, 1978, *Tree size by partial backtracking*, SIAM Journal on Computing, **7(4)**, pp. 481–491.

[80] Python Software Foundation, 1990, *Python programming language — Official website*, [Online], [Cited September 29$^{th}$, 2013], Available from `http://www.python.org/`.

[81] Reed K, 2013, *Results of the WCG User Participation Study*, [Online], [Cited September 25$^{th}$, 2013], Available from `http://boinc.berkeley.edu/trac/wiki/WorkShop13`.

[82] RenderFarming.net, 2004, *BURP — The big and ugly rendering project*, [Online], [Cited June 14$^{th}$, 2014], Available from `http://burp.renderfarming.net/`.

[83] Robinson DF, 1981, *Constructing an annual round-robin tournament played on neutral grounds*, Mathematical Chronicle, **10**, pp. 73–82.

[84] Rosenbaum DJ, 2013, *Bidirectional Collision Detection and Faster Deterministic Isomorphism Testing*, arXiv preprint arXiv:1304.3935.

[85] Sade A, 1948, *Enumeration des carrés latins: Application au 7e ordre, conjecture pour les ordres supérieurs*, Published privately.

[86] Schönhardt E, 1930, *Über lateinische Quadrate und Unionen*, Journal für die reine und angewandte Mathematik, **163**, pp. 183–230.

[87] Shirky C, 2000, *What is P2P... and what isn't*, [Online], [Cited June 14$^{th}$, 2013], Available from `http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html`.

[88] Shoch JF & Hupp JA, 1982, *The "worm" programs — Early experience with a distributed computation*, Communications of the Association of Computing Machinery, **25(3)**.

[89] Stinson DR & Kreher DL, 1999, *Combinatorial algorithms: generation, enumeration, and search*, CRC Press, Boca Raton (FL).

[90] Tarry G, 1900, *Le problème des 36 officiers*, Association Française pour l'avancement des sciences.

[91] Thain D, Tannenbaum T & Livny M, 2005, *Distributed computing in practice: The Condor experience*, Concurrency and Computation: Practice and Experience, **17(2)**, pp. 323–356.

[92] University of California, 2013, *BOINC: Open-source software for volunteer computing and grid computing*, [Online], [Cited March 30$^{th}$, 2013], Available from `http://boinc.berkeley.edu/`.

[93] University of Wisconsin-Madison, 2013, *HTCondor — High throughput computing*, [Online], [Cited November 7$^{th}$, 2013], Available from `http://research.cs.wisc.edu/htcondor/`.

[94] Uplinger K, 2013, *Informal discussion on the future of BOINC in China,* 9th BOINC workshop, Grenoble.

[95] Veblen O & Maclagan-Wedderburn J, 1907, *Non-Desarguesian and non-Pascalian geometries*, Transactions of the American Mathematical Society, **8(3)**, pp. 379–388.

[96] Wallis WD & George JC, 2011, *Introduction to combinatorics*, CRC Press, Boca Raton (FL).

[97] Wells MB, 1967, *The number of Latin squares of order eight*, Journal of Combinatorial Theory, **3(1)**, pp. 98–99.

[98] Zuckerberg M, 2004, *Facebook*, [Online], [Cited October 14$^{th}$, 2013], Available from `http://www.facebook.com`.