

**CDMTCS  
Research  
Report  
Series**

**Search and Enumeration  
Techniques for Incidence  
Structures**

**Paul C. Denny**  
Department of Computer Science  
Univeristy of Auckland

CDMTCS-085  
May 1998

Centre for Discrete Mathematics and  
Theoretical Computer Science

# Abstract

This thesis investigates a number of probabilistic and exhaustive computational search techniques for the construction of a wide variety of combinatorial designs, and in particular, incidence structures. The emphasis is primarily from a computer science perspective, and focuses on the algorithmic development of the techniques, taking into account running time considerations and storage requirements. The search and enumeration techniques developed in this thesis have led to the discovery of a number of new results in the field of combinatorial design theory.



# Acknowledgments

I would like to extend my sincere thanks to a number of people who have given me a great deal of assistance and support throughout the preparation of this thesis.

Firstly, my supervisor Peter Gibbons. I am very grateful for the encouragement and guidance he has given to me. His remarkable enthusiasm and friendliness have helped to make this thesis a most enjoyable experience.

My family, for their constant support and encouragement over the years.

My office mates, for being very supportive and friendly.

The technicians, for their cooperation during my extensive use of the computing resources in the graduate and undergraduate computing laboratories.

Throughout the course of this thesis I have had many useful discussions with a number of researchers, and would like to thank them all for their help and encouragement:

Anton Betten, Gunnar Brinkmann, Charlie Colbourn, Jeff Dinitz, Hans-Dietrich Gronau, Harald Gropp, Reinhard Laue, Rudi Mathon, Brendan McKay, Eric Mendelsohn, Markus Meringer, Alex Rosa and Ted Spence.



# Table of Contents

## CHAPTER ONE:

### INTRODUCTION

1.1	General Introduction . . . . .	1
	- existence problems	
	- enumeration problems	
1.2	Motivation . . . . .	2
1.3	Thesis Overview . . . . .	2

## CHAPTER TWO:

### NON-EXHAUSTIVE CONSTRUCTION TECHNIQUES

2.1	General Introduction . . . . .	5
2.2	Introduction to Hill-Climbing and Simulated Annealing . . . . .	5
2.2.1	Combinatorial Optimisation Problems . . . . .	5
2.2.2	Hill-Climbing . . . . .	7
2.2.3	Simulated Annealing . . . . .	8
2.3	Application of Non-Exhaustive Construction Techniques . . . . .	11
2.3.1	One-Factorisations . . . . .	11
2.3.2	Latin Squares . . . . .	13
2.3.3	Mutually Orthogonal Latin Squares . . . . .	15
2.3.4	Steiner Triple Systems . . . . .	21
	2.3.4.1 Random Steiner Triple System Generation . . . . .	21
	2.3.4.2 Cyclic Steiner Triple System Generation . . . . .	22
	2.3.4.3 3-Cyclic Steiner Triple System Generation . . . . .	24
2.3.5	Weakly Derived Steiner Triple Systems . . . . .	29
	2.3.5.1 Repair Search . . . . .	31
	2.3.5.2 Placement Search . . . . .	37
	2.3.5.3 Non-Isomorphic Base Designs . . . . .	39
	2.3.5.4 Reverse Engineering . . . . .	40
2.3.6	Weakly Union Free Twofold Triple Systems . . . . .	43
	2.3.6.1 Random Twofold Triple System Construction . . . . .	43
	2.3.6.2 Weakly Union Free Design Construction . . . . .	44

## CHAPTER THREE:

### EXHAUSTIVE CONSTRUCTION OF INCIDENCE STRUCTURES

3.1	Introduction . . . . .	47
3.2	Backtracking . . . . .	48
3.3	Incidence Structures . . . . .	49
3.3.1	Balanced Incomplete Block Designs . . . . .	50
3.3.2	Representations . . . . .	50
	- block lists	
	- incidence matrices	
3.4	Balanced Incomplete Block Design Construction . . . . .	51
3.4.1	Brute Force Construction . . . . .	52
3.4.2	Block by Block Backtracking Construction . . . . .	54
	3.4.2.1 Algorithm . . . . .	55
	3.4.2.2 Optimisations . . . . .	57

- point count constraint
- pair count constraint
- 3.4.2.3 Results ..... 59
- 3.4.3 Point by Point Backtracking Construction ..... 60
  - 3.4.3.1 Incidence Matrix Backtracking ..... 60
  - 3.4.3.2 One and Two Level Backtracking ..... 62
  - 3.4.3.3 Incidence Matrix Construction Ordering ..... 64
  - 3.4.3.4 Row by Row Incidence Matrix Construction ..... 65
  - 3.4.3.5 Cell Structure ..... 66
  - 3.4.3.6 Extending and Backtracking ..... 66
    - extend
    - backtrack
  - 3.4.3.7 Optimisations..... 71
  - 3.4.3.8 Results ..... 72
- 3.5 Limitations ..... 73

**CHAPTER FOUR:  
ISOMORPH REJECTION**

- 4.1 Introduction..... 75
- 4.2 Isomorphisms ..... 75
- 4.3 Automorphisms ..... 78
- 4.4 Isomorph Rejection..... 79
  - 4.4.1 Row Ordering ..... 81
  - 4.4.2 Backtrack Halting ..... 82
  - 4.4.3 The Role of Automorphism Groups ..... 83
    - column reordering
    - cell reordering
    - row rejection
- 4.5 Automorphism Group Generation ..... 89
  - 4.5.1 Partitions ..... 90
    - 4.5.1.1 Clique Analysis ..... 91
    - 4.5.1.2 Block Partition ..... 93
    - 4.5.1.3 Point Partition ..... 93
    - 4.5.1.4 Point Pair Partition..... 94
  - 4.5.2 The Group Generation Algorithm..... 95
    - 4.5.2.1 Partition Tightening - Class Vector Juxtapostion ..... 97
      - point partition tightening
      - block partition tightening
    - 4.5.2.2 Source and Target Point Selection..... 101
    - 4.5.2.3 Results ..... 101
  - 4.5.3 Fast Automorphism Group Generation ..... 102
    - 4.5.3.1 Generator Automorphisms ..... 102
    - 4.5.3.2 Automorphism Partition ..... 103
    - 4.5.3.3 Centraliser Subgroups ..... 104
    - 4.5.3.4 Construction of Generator Automorphisms ..... 105
    - 4.5.3.5 Deriving the Automorphism Group from the Generators ..... 108
    - 4.5.3.6 Summary of Fast Group Generation Algorithm ..... 111
    - 4.5.3.7 Results ..... 112
    - 4.5.3.8 Improvements and Observations ..... 115
- 4.6 Conclusion..... 116

**CHAPTER FIVE:  
CONSTRUCTIVE ENUMERATION**

- 5.1 Classical Constructive Enumeration ..... 117
- 5.2 Exhaustive Incidence Matrix Construction..... 117
  - 5.2.1 Optimisations..... 118
    - 5.2.1.1 Packing Constraint ..... 119

5.2.1.2	Strong Partial Isomorph Rejection	122
5.2.1.3	Results	125
5.3	Isomorph Classification	126
5.3.1	List Building	126
5.3.1.1	Incidence Structure Isomorphism Testing	126
5.3.1.2	Results	129
5.3.1.3	Optimisations	130
	- storing the clique analysis results	
	- design signatures	
	- list reordering	
	- list distributing	
5.3.2	Canonicity Testing	139
5.3.2.1	Brute Force Implementation	141
5.3.2.2	Efficient Canonicity Testing	143
5.3.2.3	Results	146
	- the $2-(6,3,\lambda)$ designs	
	- the $2-(7,3,\lambda)$ designs	
5.4	Conclusion	149

## CHAPTER SIX:

### CASE STUDIES

6.1	Case Study One - Weakly Union Free Twofold Triple Systems	151
6.1.1	The Search Tree	151
6.1.2	Weakly Union Free Constraint	153
6.1.3	Estimation of the Size of the Search Tree	153
6.1.3.1	Monte-Carlo Estimation	154
6.1.3.2	Results	156
6.1.4	Estimation of Exhaustive Search Running Time	157
6.1.5	The Effect of Isomorph Rejection	158
6.1.6	Settling the wuf-TTS(12) Existence Question	160
6.1.7	Enumeration of the wuf-TTS(13) Designs	160
6.2	Case Study Two - Parallelisation	161
6.2.1	Introduction	161
6.2.2	Splitting on Backtrack Calls	162
6.2.3	Splitting on Starter Configurations	163
6.2.4	Results	164
6.3	Case Study Three - The $2-(7,3,\lambda)$ Design Family	166
6.3.1	Background	166
6.3.2	Counting Formulas	167
6.3.2.1	Order 6 Triple Systems	167
6.3.2.2	Order 7 Triple Systems	168
6.3.2.3	Proposed Formula	169
6.3.2.4	Discrepancy	170
6.3.3	Storage of Large Catalogues	171
6.3.3.1	Design Representations	172
6.3.3.2	Optimisations	173
6.3.3.3	Implementation	173
6.3.3.4	Buffering	174
6.3.3.5	File Processing	174
6.3.4	Discrepancy Resolution	175
6.3.5	Results	177
6.4	Case Study Four - The $2-(7,3,\lambda) \rightarrow 3-(8,4,\lambda)$ Extension	177
6.4.1	Background	177
6.4.2	Generating $3-(8,4,\lambda)$ Designs from $2-(7,3,\lambda)$ Designs	178
6.4.3	Testing The Constructed $3-(8,4,\lambda)$ Designs For Isomorphism	183



6.4.4	Results	187
6.5	Case Study Five - The Mendelsohn Triple Systems	187
6.5.1	Background	187
6.5.2	Current Results	188
6.5.3	Enumeration Approach	188
6.5.4	Generation of All Underlying TTS Designs	189
6.5.5	Embedded Orientability Constraint	190
6.5.6	Generation of All Inequivalent Orientations of the Underlying Designs	190
6.5.6.1	Producing All Distinct Orientations	191
6.5.6.2	Producing Only Inequivalent Orientations	192
6.5.7	Results	196
	- the MTS(10,1) designs	
	- the MTS(12,1) designs	
6.5.8	Conclusion	198
6.6	Case Study Six - The 2-(10,4,4), 3-(11,5,4), 4-(12,6,4) Design Families	199
6.6.1	Background	199
6.6.2	The 2-(10,4,4) Designs	200
6.6.3	The 3-(11,5,4) Designs	201
6.6.3.1	Motivation	201
6.6.3.2	Results	202
6.6.4	The 4-(12,6,4) Designs	202
6.6.5	The Derived Designs	202
6.6.5.1	Canonical Labelling	203
6.6.5.2	Implementation	204
6.6.6	Results Summary	205
6.7	Case Study Seven - Configurations	206
6.7.1	Introduction to Configurations	206
6.7.2	Construction of Configurations	206
6.7.3	Configuration Enumeration by List Building	207
6.7.4	Configuration Enumeration by Canonicity Testing	210
	- results	
6.7.5	Other Symmetric Configurations	210
6.7.6	Non-Symmetric Configurations	211
6.7.7	Blocking Set Free Configurations	212
	- method	
	- results	
6.7.8	Triangle Free Configurations	213
	- method	
	- results	
6.8	Case Study Eight - Other Combinatorial Designs	215
6.8.1	t-Designs	215
	- 2-designs	
	- 3-designs	
	- 4-designs	
6.8.2	Cubic Multigraphs	219
6.8.3	Symmetric Designs	221
	- symmetric constraint	
	- results	
6.8.4	n-Simple Designs	223
	- implementation	
	- results	

## CHAPTER SEVEN: SUMMARY OF RESULTS

7.1	Block Designs	229
-----	---------------	-----

7.1.1	2-Designs (BIBDs).....	229
7.1.2	3-Designs .....	231
7.1.3	4-Designs .....	231
7.2	Weakly Union Free Twofold Triple Systems .....	231
7.2.1	wuf-TTS(12) .....	231
7.2.2	wuf-TTS(13) .....	231
7.3	Mendelsohn Triple Systems .....	233
7.3.1	The MTS(10,1) Designs .....	233
7.3.2	The MTS(12,1) Designs .....	233
7.3.3	The MTS( $v,1$ ) Designs, $v < 12$ .....	234
7.4	Design Families.....	235
7.4.1	The $2-(7,3,\lambda)$ Design Family .....	235
7.4.2	The $2-(7,3,\lambda) \rightarrow 3-(8,4,\lambda)$ Extension .....	240
7.4.3	The $2-(10,4,4)$ , $3-(11,5,4)$ , $4-(12,6,4)$ Designs .....	247
7.5	Configurations .....	249
7.5.1	Symmetric $v_3$ Configurations .....	249
7.5.2	Blocking Set Free Configurations .....	249
7.5.3	Triangle Free Configurations .....	250
7.6	Miscellaneous .....	251
7.6.1	Cubic Multigraphs .....	251
 <b>CHAPTER EIGHT:</b>		
<b>CONCLUSIONS AND FUTURE WORK</b>		
8.1	Conclusions .....	253
8.2	Future Work .....	253
 <b>REFERENCES</b> .....		
		255

Table of Contents

# Chapter 1

## Introduction

### 1.1 GENERAL INTRODUCTION

This thesis investigates a number of probabilistic and exhaustive computational search techniques for the construction of a wide variety of combinatorial designs, and in particular, incidence structures. The emphasis is primarily from a computer science perspective, and focuses on the algorithmic development of the techniques, taking into account running time considerations and storage requirements. The search and enumeration techniques developed in this thesis have led to the discovery of a number of new results in the field of combinatorial design theory.

An *incidence structure* consists of two finite sets of objects, usually referred to as *points* and *blocks*, with an incidence relation,  $\Psi$ , between them. Given any block of the structure, there is a set of points incident with it as defined by  $\Psi$ . An enormous variety of such designs exist, with many useful applications arising from their particular structural properties. The ability to construct designs with prescribed properties is therefore of considerable importance and is central to the field of combinatorics, which is that branch of mathematics concerned with the arrangement of the objects of a set into patterns satisfying certain constraints. In combinatorial design theory, two important problems are that of the existence and enumeration of certain combinatorial structures.

#### **Existence Problems:**

A given combinatorial design is defined by a set of parameters which essentially encode the constraints of its structure. It is often possible to derive very simple necessary conditions for the existence of a design directly from its parameters. However, it is usually very difficult to prove or disprove that the conditions are in fact sufficient for the existence of such a design. Quite often, existence can only be established by an explicit construction of the design, which may be an extremely involved process. A good example of the difficulty of such a problem is the celebrated search for a 2-(22,8,4) block design. This is the design consisting of the smallest number of points for which existence still remains unsettled, despite an enormous amount of work by many researchers. For example in [50], McKay and Radziszowski devote approximately 15 years of CPU time to this problem.

#### **Enumeration Problems:**

In general, the task of enumeration is to count the number of non-equivalent elements in a given class of combinatorial objects. Constructive enumeration consists of explicitly creating a complete list of configurations with given properties, and is very important for a number of reasons. Firstly, classical counting methods are not applicable to many interesting classes of structures such as strongly regular graphs and block designs, and thus currently the only known way to count such designs is by their explicit construction. Secondly, complete lists of objects often play an important role in the formulation of mathematical conjectures, as well as supporting or providing counter-examples for existing conjectures. Finally, many areas of applied combinatorics require examples of designs with particular properties.

For a given class of structures, usually if the existence question is quite complex, then the corresponding enumeration problem is very difficult.

## 1.2 MOTIVATION

Over the past few decades, combinatorics has experienced a dramatic growth primarily due to the importance of its many applications covering a wide range of disciplines.

Computer science is perhaps the most important field for applications of combinatorial ideas. The analysis of the running times and space demands of computer algorithms essentially requires the calculation of the number of elements in a finite set, which is precisely the goal of enumerative combinatorics. A general survey on the applications of combinatorial designs to computer science is given in [15], by Colbourn and Oorschot. Another important application area is that of cryptography, in which combinatorial structures arise in a natural way. Stinson [62] presents a survey of the important role that combinatorial designs play in the study of such topics in cryptography as secrecy and authentication codes, secret sharing schemes and resilient functions. However, the interaction between combinatorics and computer science is not all one way - in fact both fields have genuinely profited from one another. The advent of computers has fuelled a tremendous growth in combinatorics, and the computer has been indispensable in the construction of combinatorial designs of many types. Their enormous processing power has increased at an exponential rate, making possible the calculation of previously unattainable solutions to immense problems. However, developing algorithms for tackling large combinatorial problems is not trivial and indeed analysis of the storage requirements and to a greater degree the running time requirements of such algorithms requires further combinatorial consideration.

Combinatorics has important applications in many other fields also. An excellent reference for the practical applications of combinatorics to a wide range of subjects is given in [33]. A large number of practical problems covering the areas of operations research, electrical engineering and statics, statistical physics, chemistry, molecular biology, pure mathematics, and computer science are presented and solved using various combinatorial techniques.

## 1.3 THESIS OVERVIEW

This thesis examines in detail both the existence and enumeration aspects of combinatorial design construction. A number of non-exhaustive construction methods, which are probabilistic in nature, are presented and applied to several specific problems. In addition, an exhaustive generation algorithm is developed for the constructive enumeration of general incidence structures. An overview of the organisation of this thesis is given below:

Chapter 2:

This chapter introduces the two well-known non-exhaustive construction techniques of hill-climbing and simulated annealing in the context of combinatorial optimisation. Previously successful uses of these techniques are examined, and several new problems are presented along with probabilistic construction algorithms and search-guiding heuristics which are analysed and refined. A number of the combinatorial structures introduced in this chapter are referred to throughout the thesis.

Chapter 3:

This chapter presents a simple exhaustive algorithm for the construction of incidence structures. A number of schemes are investigated, the most effective of which generates all possible solutions for a particular set of design parameters, and is the core procedure behind the constructive enumeration algorithm developed in this thesis.

Chapter 4:

This chapter presents the powerful technique of isomorph rejection and details its operation in conjunction with the exhaustive algorithm developed in Chapter 3. This

technique significantly reduces the amount of required processing, and hence allows the algorithm to be applied to considerably larger problems.

Chapter 5:

This chapter presents several methods for non-isomorphic design classification and combines these with the efficient design generation algorithm of Chapter 4 to develop an effective constructive enumeration algorithm for incidence structures.

Chapter 6:

This chapter details the application of the constructive enumeration algorithm to a number of specific problems. These problems are presented as a series of eight case studies which describe the specific tasks, detail the particular techniques adopted and summarise the generated results. As well as discovering several errors in currently published results within the literature, a number of new results were computed.

Chapter 7:

This chapter summarises the main results of the thesis, many of which are also presented in their respective case studies and are compiled again in this chapter for convenience. Alongside each result, a reference is given to the relevant section of the thesis explaining its computation.

Chapter 8:

This chapter briefly presents some of the conclusions of the thesis and indicates a number of possible directions for future research.



# Chapter 2

## Non-Exhaustive Construction Techniques

### 2.1 GENERAL INTRODUCTION

In this chapter, two well-known non-exhaustive search techniques are investigated: hill-climbing and simulated annealing. Essentially, both of these techniques can be applied to the construction of combinatorial designs by starting with an approximation to a design and refining it by making random modifications to its components. Non-exhaustive search techniques such as these have proven themselves to be successful in many situations where solutions to the corresponding problems exist. One drawback of any non-exhaustive strategy is that if no solution is found, regardless of the amount of searching performed, then no conclusions can be drawn about the existence of a solution.

This chapter is divided into two main sections. Section 2.2 introduces the two techniques, explaining how they work, and reviewing some of their successful applications in the field of combinatorial design construction.

Section 2.3 presents a series of small case studies, where non-exhaustive search techniques are implemented and utilised to construct several different types of combinatorial designs. The effectiveness of the construction techniques are analysed for each problem.

### 2.2 INTRODUCTION TO HILL-CLIMBING AND SIMULATED ANNEALING

Both hill-climbing and simulated annealing are non-exhaustive state-space search techniques, which control the navigation from state to state around a search space. The characteristics of the search space itself, such as its size and the nature of the states within it, are dependent on the specific problem. Both techniques are probabilistic in nature, for as their execution progresses, random state transitions are made which direct the search. They are non-exhaustive because it cannot be guaranteed that every state within the search space will be examined as a result of the random transitions.

The task of constructing a combinatorial design can be reformulated as a combinatorial optimisation problem, and it is in this context that the behaviour of a hill-climbing or simulated annealing algorithm is best described. Section 2.2.1 describes the important entities and definitions of a generic combinatorial optimisation problem, which will simplify the explanation of the hill-climbing and simulated annealing algorithms for construction of combinatorial designs, in the subsequent two sections.

#### 2.2.1 COMBINATORIAL OPTIMISATION PROBLEMS

A combinatorial optimisation problem consists of a set,  $\Sigma$ , of feasible solutions together with an objective function which associates a profit,  $\text{prf}(S)$ , with each feasible solution  $S$ . The task is to locate an optimal feasible solution, which corresponds to a feasible solution belonging to the set  $\Sigma$  with the largest associated profit. Alternatively, the objective



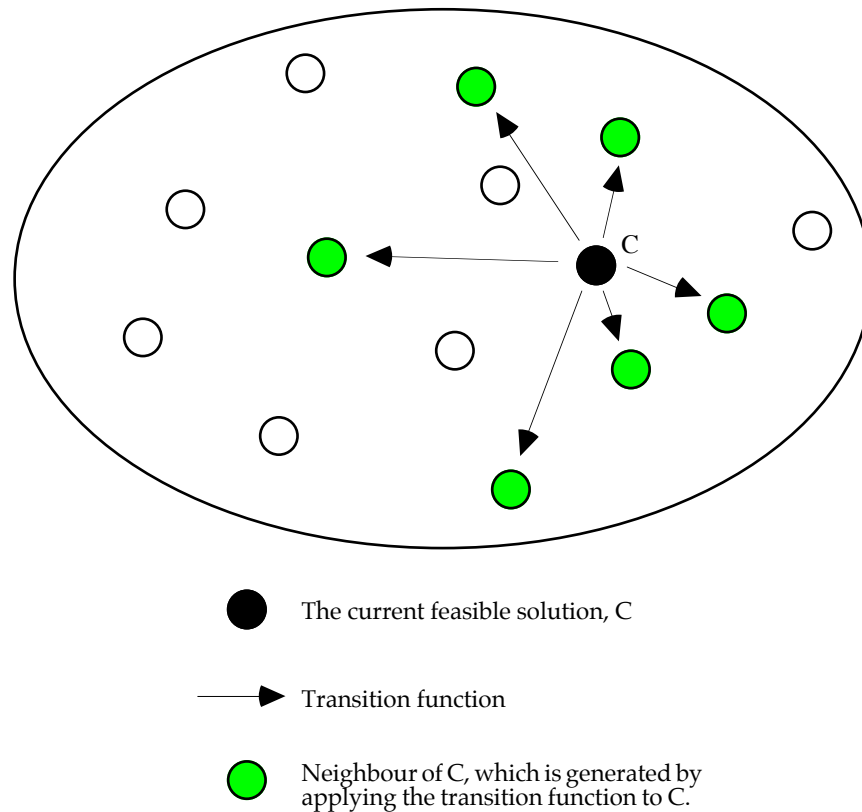
function may associate a cost with each feasible solution in which case the task is to find a feasible solution with the minimum associated cost.

A combinatorial optimisation problem can therefore be represented as a set of all possible feasible solutions, or states, called the state-space of the problem. State-space search techniques, such as hill-climbing and simulated annealing, traverse the state-space of a given problem in an attempt to locate the optimal solution, or at least a close approximation to it.

In order to traverse the state-space, a transition function is introduced which defines the process of moving from one state to the next and imposes a neighbourhood structure on the state-space. State N is a *neighbour* of state C if it is possible to move from state C to state N with a single application of the transition function. The set of all possible states which are reachable, via a single application of the transition function, from state C is called the *neighbourhood* of C. At any point in the execution of a state-space search technique, one particular state is the current state, and the task is to determine which of the neighbours of this state to move to, in order to eventually locate an optimal solution.

A schematic diagram of the effect of the transition function on the state-space is given below:

*State-space of all feasible solutions*



Denote the profit (respectively cost) of a state S by  $\text{prf}(S)$  (respectively  $\text{cs}(S)$ ). The optimal solution has the maximum associated profit, and the minimum associated cost. Let C denote the current state, and let N denote one of the neighbours of C. If  $\text{prf}(C) < \text{prf}(N)$  (equivalently  $\text{cs}(C) > \text{cs}(N)$ ), then a transition from C to N is called a *forward transition*, because it takes the profit and cost of the current feasible solution closer to the profit and cost of the optimal solution. Conversely, a move from C to N which decreases the profit and equivalently increases the cost, is called a *backward transition*. Any transition for which N has the same associated profit or cost as C is called a *level transition*. When applying

the objective function, usually only one of the quantities of profit and cost are considered, the choice of which usually depends on the specific properties of the problem being studied.

This brief background in combinatorial optimisation problems enables the state-space search techniques of hill-climbing and simulated annealing to be introduced in a familiar context. Sections 2.2.2 and 2.2.3 describe each of these techniques with respect to optimisation problems, provide a connection between solving optimisation problems and constructing combinatorial designs, and overview some of their successes in the latter field.

## 2.2.2 HILL-CLIMBING

Hill-climbing is a state-space search technique that consists of making random forward and level transitions between the states within a search space of feasible solutions. Backward transitions are not permitted, and the goal of the search is to locate the optimal state, or optimal solution. The technique of hill-climbing was first used in the field of optimisation problems, to efficiently find close approximations to optimal solutions, and has also achieved success in the construction of combinatorial designs.

As it is unnecessary to consider both profit and cost, the remainder of this section will consider only profit, and assume that it is to be maximised. For many combinatorial design construction problems, it is more natural to think of maximising a profit rather than minimising a cost.

The basic steps of a hill-climbing algorithm for solving a combinatorial optimisation problem are outlined below:

```

Start with a current state, C
while prf(C) is not a local maximum do
  begin
    pick a random neighbour of C, say N, such that  $\text{prf}(N) \geq \text{prf}(C)$ 
    C := N
  end;

```

As indicated by the steps above, random forward and level transitions around the state-space are performed, until a state, M, is reached such that all of M's neighbours have a smaller profit than M. At this point, no further forward or level transitions can be performed, as M is a local maximum, and so the algorithm terminates.

For certain problems, either it may not be known what the profit of the optimal solution is, or simply an approximation to the optimal solution may be required. In such cases, the solution M can be returned as this approximation or possible optimal solution. In other cases, either it may be known what the profit of an optimal solution is, or it may be determined that M is not a good enough approximation. In these cases, the hill-climbing algorithm could start over with a new initial state, and through a different series of random transitions produce a new local maximum, M', until the profit of M' was acceptable.

In the context of combinatorial design construction, a useful feature is that it is trivial to detect an optimal solution. An optimal solution will correspond to a valid design, which often consists of a set of blocks or tuples, over which certain restrictions and conditions must hold. One very effective way of creating the state-space of a combinatorial design construction problem is to define each state to be a partial design. A partial design consists of a possibly incomplete set of blocks or tuples, over which all the restrictions and conditions of a valid design still hold. The collection of all possible partial designs then constitutes the complete state-space. The profit of a state in the state-space is the number of blocks or tuples in the corresponding partial design. Clearly, the profit of a valid design will be known, and so detection of an optimal state is trivial.

The transition function attempts to add a block or tuple to the current partial design. If one can be added without violating the constraints of the design, then the profit of the corresponding partial design increases which represents a forward transition. Otherwise, the profit does not increase, and in this case an effective transition function will generate a new distinct state, which corresponds to a level transition. The number of blocks or tuples in the current partial design never decreases as backward transitions are not permitted

The hill-climbing algorithm will only complete once a valid design has been constructed. If a local maximum is reached which does not correspond to a completed, valid design, then the algorithm must start again, and a different sequence of random transitions will be performed. This restarting process can be repeated until either a valid design is found, or some processing threshold is exceeded.

Hill-climbing has been used successfully to construct numerous combinatorial designs - including strong-starters, one-factorisations and Room squares, Howell designs, leaves of partial triple systems, quadrilateral free Steiner triple systems, enclosings of triple systems, pairwise balanced designs with block sizes 3 and 4, room frames, Steiner triple systems, orthogonal Steiner triple systems and sequencing groups.

### 2.2.3 SIMULATED ANNEALING

Simulated annealing is similar to hill-climbing, in that random transitions are performed to traverse a state-space with the goal of arriving at an optimal state or solution. However, unlike hill-climbing, simulated annealing allows backward transitions in addition to forward and level transitions, reducing the chance of the search getting stuck in a local maximum which does not correspond to the globally optimal solution. It is considerably more complicated than hill-climbing as the backward transitions are only permitted in a highly controlled fashion which models the physical annealing process of solids, described below. Essentially, forward and level transitions are always accepted, yet as the algorithm proceeds, the probability of accepting backward transitions slowly decreases.

The physical annealing process, which is modelled by simulated annealing algorithms, refers to a thermal process in which a solid is first heated to melting point, and then slowly cooled until the low-energy ground state is reached. If the solid is not heated sufficiently to begin with, or the cooling is not performed slowly enough, then the solid will freeze into a meta-stable state rather than the ground state. A complete description of this process is given by Aarts & Korst [1] and Laarhoven [41]. In terms of the optimisation problem, if the transitions are not carefully controlled then a local maximum state (meta-stable state) will be reached which is not the optimal state (ground state). This analogy between the physical annealing process and the task of solving combinatorial optimisation problems was studied in the early 1980's by Kirkpatrick, Gelatt and Vecchi [39] and independently by Cerny [10]. They made a number of correspondences, which are summarised in the table below:

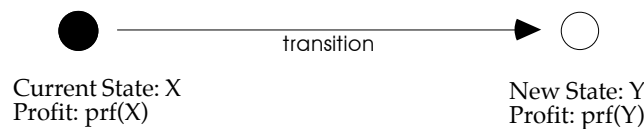
Physical Annealing Process	Optimisation Problem
States of the solid	State-space, or set of feasible solutions
Energy of the states	Profit or cost of feasible solutions as assigned by the objective function
Melting point state	Randomly generated initial feasible solution
Slight distortion of state	Elementary change to feasible solution caused by transition function
Temperature	A control parameter, T

The physical annealing process can be simulated on the computer and used to tackle large combinatorial optimisation problems. Metropolis et al. [54] used a simple Monte Carlo

algorithm which generated a sequence of states of a solid modelled as a collection of particles. Each state had an associated energy, and the goal of the algorithm was to locate the ground state with the lowest energy. The current state was distorted slightly, for example by random displacement of a particle, and the energy of the new state was calculated. A transition to this new state was always accepted if the energy of the new state was less than or equal to the energy of the undistorted state. If the energy of the distorted state increased, it was accepted with probability  $e^{(\Delta Energy/k_B T)}$ , where T was the modelled temperature of the solid which slowly decreased as the algorithm progressed, and  $k_B$  is a physical constant known as the Boltzmann constant. This acceptance rule is known as the Metropolis criterion and the associated algorithm is called the Metropolis algorithm.

When applied to a simulated annealing algorithm for tackling a combinatorial optimisation problem, the Metropolis criterion can be used as a basis for deciding whether to accept transitions between states in the state-space. The objective function is used to assign a profit, or equivalently a cost, to each state, however to be consistent with Section 2.2.2 only the profit of a given state which must be maximised by the algorithm will be considered for the remainder of this section. Following each transition, the difference between the profit of the current state and the profit of the new state is used to determine whether to accept the new state as the current state. The diagram below outlines the use of the Metropolis criterion in such optimisation problems:

### The Metropolis Criterion



The goal of the algorithm is to construct a feasible solution with the maximum profit, and a possible transition from state X → state Y is being considered in the above diagram. The following two rules are used to determine whether or not to accept the transition:

**if**  $\text{prf}(Y) \geq \text{prf}(X)$  **then**  
 accept the current transition to the new feasible solution Y

**otherwise if**  $\text{prf}(Y) < \text{prf}(X)$  **then**  
 accept the current transition to the new feasible solution Y with probability:  

$$e^{\left(\frac{\text{prf}(Y) - \text{prf}(X)}{kT}\right)}$$

where T is the modelled temperature which slowly decreases as the algorithm progresses, and k is some constant. In [54] this was the Boltzmann constant, but a more appropriate constant can be used to suit each particular optimisation problem.

In order to use the acceptance rule above, a temperature is defined as one of the parameters of the simulated annealing algorithm. At the start of the algorithm the temperature parameter is initially high, but as the algorithm progresses and state transitions are performed, the temperature slowly decreases. This has the effect of slowly lowering the probability of accepting backward transitions. The temperature is lowered slowly enough that a certain number of state transitions are made at each temperature level. Such a sequence of transitions at the same temperature is called a Markov chain, and the number of transitions in the sequence is the Markov chain length. A cooling schedule is defined to specify by how much the temperature should decrease following each Markov chain.

So, as in hill-climbing, the state-space is traversed by the transition function. However, unlike hill-climbing, if the search gets trapped in a local maximum there is a certain probability of leaving the local maximum by means of a backward transition which

decreases the profit of the current feasible solution. This probability is determined by the Metropolis criterion and depends on the size of the backward step and the current value of temperature parameter, and slowly decreases as the algorithm progresses. If the temperature starts off sufficiently high, and the cooling schedule reduces the temperature slowly enough, the chance of traversing the state-space to an optimal feasible solution is increased.

A rough outline of the simulated annealing procedure is:

```

Construct an initial feasible solution, X.
Specify the initial temperature
Specify the cooling schedule, which includes the following:
• Markov chain length, L
• a factor for reducing the current temperature following each Markov chain
• a stopping or freezing condition, which indicates that the cooling process has
  completed.

do
  repeat L times
    Generate a neighbour of X, say Y
     $\Delta_{YX} = \text{prf}(Y) - \text{prf}(X)$ 
    if  $\Delta_{YX} \geq 0$  or  $\text{random}(0,1) < e^{(\Delta_{YX}/T)}$ 
      X := Y;

  Reduce the temperature in accordance with the cooling schedule
while the stopping condition is not met.

```

The stopping condition specifies at what point the current cooling process has completed so that the current feasible solution can be thought of as being frozen. Although there are many possible stopping conditions which could be used, a common one is to terminate the algorithm when there is no increase in the profit of the current feasible solution over a number of consecutive Markov chains.

Once the algorithm has frozen, if the resulting state is not a satisfactory solution to the optimisation problem, then the annealing process can be restarted by generating a new initial state, resetting the temperature to its initial value, and performing a new series of state transitions.

This technique can be applied to combinatorial design construction in the same way as hill-climbing. A complete valid design will consist of a set of blocks or tuples satisfying the design conditions, and a partial design will consist of a valid subset of these blocks or tuples. The profit of a feasible solution, whether partial or complete, is then the number of blocks or tuples in the corresponding design. As the profit of a complete design is known, the construction algorithm can terminate as soon as one is found. If the simulated annealing freezes without a valid design being constructed, the algorithm is simply reset.

One possible method of construction would involve the transition function attempting to add a block or tuple to the current partial design. If one can be added without violating the constraints of the design, then the transition is accepted and the profit increases. Otherwise, the new block must conflict with one or more of the blocks already in the partial design, and these must be removed to accommodate the new block. If a transition of this type is accepted, the profit of the partial design may decrease. The Metropolis criterion is used to determine whether or not to accept such a backward transition.

Simulated annealing has been used successfully to construct covering designs, packing designs, block designs, subsquare free Latin squares and antipodal triple systems.

## 2.3 APPLICATION OF NON-EXHAUSTIVE CONSTRUCTION TECHNIQUES

The techniques of hill-climbing and simulated annealing can be successfully applied to the construction of certain types of combinatorial designs. Once the construction processes are formulated in terms of state-spaces and transition functions, useful structures such as Latin squares and one-factorisations of complete graphs can be generated with great efficiency.

This main section presents a series of small case studies, in which the non-exhaustive construction techniques presented in Section 2.2 are applied to the generation of a variety of combinatorial designs. Sections 2.3.1 and 2.3.2 briefly review the successful application of hill-climbing to the construction of one-factorisations of the complete graph and to the construction of Latin squares respectively. Both sections provide excellent examples of how the non-exhaustive techniques can be applied to practical problems, and the performance of each algorithm is analysed to estimate its complexity. In the remaining sections of this chapter, the basic techniques are refined for the construction of more complicated combinatorial structures.

### 2.3.1 ONE-FACTORISATIONS

The algorithm given in this section for the construction of one-factorisations was initially presented by Dinitz and Stinson in [23]. The complete graph on  $n$  vertices,  $K_n$ , has every pair of vertices connected by an edge, thus there are  $\binom{n-1}{2}$  edges.

**Definition:**

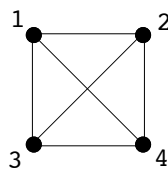
A *one-factor* of  $K_n$  is a set of  $\frac{n}{2}$  edges that partitions the vertex set. For a one-factor to exist,  $n$  must clearly be even.

A *one-factorisation* of  $K_n$  is a set of  $n-1$  one-factors that partitions the edge set.

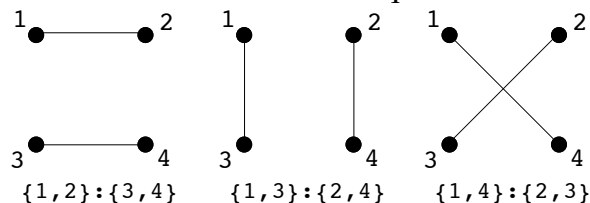
As an example of the practical use of such structures, one factorisations of the complete graph  $K_n$  can be used to construct round robin tournaments amongst  $n$  players, where each of the  $n-1$  one factors correspond to a single round of competition with the edges denoting the players who compete against one another in the given round.

Consider the complete graph  $K_4$  illustrated below, along with three one-factors which partition the edge set:

The complete graph,  $K_4$



One factors of  $K_4$



The set of all three one-factors displayed above partitions the edge set of  $K_4$  and is therefore a one-factorisation. It can be listed explicitly, as below:

One-Factor 1:  $\{ \{1, 2\}, \{3, 4\} \}$

One-Factor 2:  $\{ \{1, 3\}, \{2, 4\} \}$

One-Factor 3:  $\{ \{1, 4\}, \{2, 3\} \}$

or can equivalently be represented as a set of 6 pairs:

$\{(1, \{1, 2\}), (1, \{3, 4\}), (2, \{1, 3\}), (2, \{2, 4\}), (3, \{1, 4\}), (3, \{2, 3\})\}$

Each pair is of the form  $(f, \{x, y\})$ , where  $\{x, y\}$  is an edge in  $K_4$  and  $f$  is the one-factor to which that edge belongs.

For a complete graph on  $n$  vertices, the properties which must be satisfied by the pairs in such a set representing a one-factorisation of the graph are:

- 1) every edge  $\{x, y\}$  of  $K_n$  occurs in a unique pair  $(f, \{x, y\})$
- 2) for every one-factor  $f$  and vertex  $x$ , there is a unique pair of the form  $(f, \{x, y\})$

In the context of the hill-climbing algorithm, a feasible solution in the state-space corresponds to a possibly incomplete, partial set of these pairs. The only restriction on the pairs in a partial set are:

- 1) every edge  $\{x, y\}$  of  $K_n$  occurs in *at most one* pair  $(f, \{x, y\})$
- 2) for every one-factor  $f$  and vertex  $x$ , there is *at most one* pair of the form  $(f, \{x, y\})$

The profit of a feasible solution is the number of pairs in its set, and a valid one-factorisation therefore corresponds to a feasible solution with profit  $n(n-1)/2$ .

In [23], Dinitz and Stinson present two transition functions for traversing the state-space. These are outlined below:

#### TransitionFunction1

A random vertex,  $x_{new}$ , which has not yet appeared once with every one-factor, is chosen. A second random vertex,  $y_{new}$ , which has not yet appeared with  $x_{new}$  in any edge of a one factor is also chosen. Finally a random one-factor,  $f_{new}$ , is chosen such that  $x_{new}$  does not occur in that one-factor.

The new pair,  $(f_{new}, \{x_{new}, y_{new}\})$  is then added to the set of pairs making up the current partial solution. If  $y_{new}$  has not yet appeared in one-factor  $f_{new}$ , then this new pair will not conflict with any existing pair and is simply added to the partial solution, increasing the profit by one. However, if a pair already exists, say  $(f_{new}, \{x', y_{new}\})$  then this pair is removed from the current partial solution to accommodate the new pair, and the profit remains the same.

#### TransitionFunction2

A random one-factor,  $f_{new}$ , which has not yet appeared in  $n/2$  pairs of the current partial solution is chosen. Two random vertices,  $x_{new}$  and  $y_{new}$  are then chosen such that neither have yet appeared with one-factor  $f_{new}$ .

The new pair,  $(f_{new}, \{x_{new}, y_{new}\})$  is then added to the current partial solution. If there already exists a pair  $(f', \{x_{new}, y_{new}\})$ , in the partial solution, this is removed to accommodate the new pair. If not, the addition of the new pair increases the profit of the feasible solution by one.

To construct a random one-factorisation of  $K_n$ , the algorithm begins with an empty feasible solution, and each of the two transition functions above are performed alternately until the profit of the current feasible solution becomes  $n(n-1)/2$ .

#### Performance

The hill-climbing algorithm of Dinitz and Stinson given in [23] and outlined above was implemented in C and executed on a Digital Alpha 2100 4/275 running the Unix operating system. The table on the following page summarises the algorithm's performance, by giving the number of transitions and the execution time required to construct random one-factorisations of the complete graph  $K_v$ , for a selection of orders where  $10 \leq v \leq 1500$ . The rightmost column of the table gives the value of  $v^2 \log(v) / \# \text{Transitions}$ :

Number of nodes, $v$ , in complete graph	Number of transitions through search space to build one-factorisation of the complete graph, $K_v$	Execution time (seconds)	$\frac{v^2 \log(v)}{\# \text{Transitions}}$
10	90	0.00	1.1111
50	4110	0.04	1.0334
100	19915	0.17	1.0043
200	85569	0.93	1.0756
300	203700	2.30	1.0945
400	379987	4.70	1.0956
500	596363	8.50	1.1314
600	896379	13.50	1.1157
700	1,236,054	19.20	1.1279
800	1,639,029	26.60	1.1336
900	2,079,204	36.40	1.1509
1000	2,614,156	45.80	1.1476
1100	3,186,254	55.70	1.1550
1200	3,850,046	67.90	1.1517
1300	4,538,872	82.20	1.1594
1400	5,320,011	97.74	1.1591
1500	6,158,915	114.10	1.1603

The running times quoted above include the time taken to initialise all necessary data structures. In the case of the complete graph  $K_{1500}$ , which consists of 1500 vertices and 1,124,250 edges, the algorithm constructs a random one-factorisation in less than two minutes.

An estimation of the complexity of this algorithm for constructing random one-factorisations of the complete graph  $K_v$  is  $v^2 \log(v)$ . The algorithm has been implemented so that each of the transition functions requires only constant time. This has been done with the aid of large data structures which occupy an amount of memory proportional to  $v^2$ , as suggested by Stinson in [63]. This means that the overall running time of the algorithm is proportional to the number of iterations performed, and the rightmost column of the table above exhibits that the number of iterations performed appears to be proportional to  $v^2 \log(v)$ .

### 2.3.2 LATIN SQUARES

In [63], Stinson outlines a hill-climbing algorithm for the construction of Latin squares, which is implemented and analysed in this section.

**Definition:**

A *Latin Square* of order  $n$  is an  $n \times n$  matrix with entries from the set of  $n$  integers  $I_n = \{1, 2, 3, \dots, n\}$ , such that each integer from the set  $I_n$  occurs exactly once in each row and in each column.

Aside from being interesting mathematical entities in themselves, Latin squares are used in the design of statistical experiments [37], with one of their earliest applications being to experimental agriculture.

A convenient representation for the hill-climbing construction of random Latin squares of order  $n$  is a set of  $n^2$  triples, or 3-tuples. For example, consider the  $3 \times 3$  Latin Square below:

1	3	2
3	2	1
2	1	3



This would be represented by the nine 3-tuples:

(1, 1, 1) (1, 2, 3) (1, 3, 2)  
 (2, 1, 3) (2, 2, 2) (2, 3, 1)  
 (3, 1, 2) (3, 2, 1) (3, 3, 3)

where the general form of each 3-tuple is:

(row position, column position, element value)

For an  $n \times n$  Latin square, each element in the set  $I_n = \{1, 2, \dots, n\}$  must occur exactly  $n$  times in the row position,  $n$  times in the column position and  $n$  times in the element position of the set of tuples. In addition, any given pair of values occurs exactly once together in each tuple. This condition is enforced by the fact that every value occurs exactly once in each row and each column.

In [63], Stinson outlines a hill-climbing algorithm for the construction of several types of combinatorial designs, including Latin squares. A random Latin square, represented as a set of  $n^2$  3-tuples, can be constructed a 3-tuple at a time using the following transition function:

- Either the row, column or element set is selected at random to be the initial set
- One of the other two sets is selected at random to be auxiliary set one
- The remaining set becomes auxiliary set two
- An element  $X$ , which has not yet occurred  $n$  times, is selected at random from the initial set
- An element  $Y$ , which has not yet occurred in any 3-tuple in the current partial solution with  $X$ , is selected at random from auxiliary set one
- An element  $Z$ , which has not yet occurred in any 3-tuple in the current partial solution with  $X$ , is selected at random from auxiliary set two
- If  $Y$  and  $Z$  occur together in a 3-tuple in the current partial solution, then this tuple is replaced with the newly found tuple consisting of  $X$ ,  $Y$  and  $Z$ . In this case, the profit - the number of 3-tuples in the partial solution - stays the same.
- Otherwise, the new 3-tuple consisting of  $X$ ,  $Y$  and  $Z$  is simply added to the partial solution, increasing the profit of the partial solution by one.

### Performance

The hill-climbing algorithm outlined above was implemented in C and executed on a Digital Alpha 2100 4/275 running the Unix operating system. The table at the top of the next page summarises the algorithm's performance, by giving the number of transitions and the execution time required to construct a random Latin square of order  $n$  for  $10 \leq n \leq 130$ , in multiples of 10. The rightmost column of the table gives  $n^2 \log(n) / \# \text{Transitions}$  and is used to estimate the algorithm's complexity.

Size of Latin square, n	Number of transition operations required to build random nxn Latin square	Execution time (secs) to build random Latin square	$\frac{n^2 \log(n)}{\# \text{Transitions}}$
10	224	0.00	0.4464
20	1122	0.01	0.4638
30	2608	0.08	0.5097
40	4850	0.21	0.5285
50	8312	0.61	0.5110
60	12408	1.38	0.5159
70	17178	2.61	0.5263
80	24253	4.98	0.5022
90	30418	7.68	0.5204
100	37463	11.70	0.5339
110	47334	18.93	0.5218
120	55996	25.48	0.5347
130	68031	35.23	0.5251

For the 130x130 Latin Square, only 68,031 switching operations were required to construct the 16,900 tuples which represent the elements of the random square.

The estimated complexity of this hill-climbing algorithm for constructing random Latin squares of order  $n$  is  $n^2 \log(n)$ . As with the previous algorithm for the construction of one-factorisations, the implementation allows each transition function to be performed in constant time. This means that the overall running time of the algorithm is proportional to the number of iterations performed, and as exhibited in the table above, the number of iterations performed appears to be proportional to  $n^2 \log(n)$ .

As demonstrated by Sections 2.3.1 and 2.3.2, non-exhaustive hill-climbing algorithms can be very effective for certain problems. A good example of the use of simulated annealing for the construction of combinatorial designs is the work of Elliott and Gibbons in [25], in which subsquare free Latin square of orders 16 and 18 were constructed for the first time.

In the following sections, several other combinatorial design problems have been investigated. In each case, an algorithm has been developed using a variant of the generic hill-climbing or simulated annealing techniques, and the effectiveness of the chosen approaches are analysed.

### 2.3.3 MUTUALLY ORTHOGONAL LATIN SQUARES

**Definition:**

Two Latin squares,  $L_1 = |a_{ij}|$  and  $L_2 = |b_{ij}|$  on  $n$  symbols, say the natural numbers  $\{1,2,3,\dots,n\}$  are said to be *orthogonal* if every ordered pair of symbols occurs exactly once among the  $n^2$  pairs  $(a_{ij}, b_{ij})$ , for  $i = 1,2,3,\dots,n$  and  $j = 1,2,3,\dots,n$ . In other words, all the ordered pairs from  $(1,1)$  up to  $(n,n)$  must be covered exactly once when  $L_1$  is superimposed on  $L_2$ , such that the entries of  $L_1$  and  $L_2$  form the first and second elements of the ordered pairs respectively. In such a case,  $L_1$  is said to be the *orthogonal mate* of  $L_2$ , and vice versa. A set of Latin squares  $L_1, \dots, L_m$  is *mutually orthogonal*, or a set of MOLS, if for every  $1 \leq i < j \leq m$ ,  $L_i$  and  $L_j$  are orthogonal.

To investigate the necessary condition for a given Latin square to have an orthogonal mate, consider two orthogonal Latin squares,  $L_1$  and  $L_2$ . If the set of exactly  $n$  cells in  $L_2$  which contain the same fixed entry, say  $h$  ( $1 \leq h \leq n$ ), are considered, then the entries in the corresponding cells of the Latin square  $L_1$  must all be different. If this were not the case, then not all of the ordered pairs containing the element  $h$  would be covered when the squares were superimposed. Since the symbol  $h$  occurs exactly once in each row and in each column of  $L_2$ , it is easy to see that the set of  $n$  cells of  $L_1$ , corresponding to the cells of  $L_2$

containing  $h$ , all contain different values and also occur once in each row and once in each column. Such a set is called a *transversal* of the Latin square.

It is obvious from the definitions of transversal and orthogonality, that a given Latin square possesses an orthogonal mate if and only if it contains  $n$  disjoint transversals. The goal of this section is to develop a non-exhaustive, probabilistic algorithm for the construction of a pair of orthogonal Latin squares. Two main approaches were considered.

**Approach One**

The random construction of a base Latin square, followed by a probabilistic method for construction of its orthogonal mate.

**Approach Two**

The random construction of a pair of orthogonal Latin squares, simultaneously cell by cell.

The disadvantage of the first approach is that a random base square may not admit an orthogonal mate, in which case any effort spent searching for one is in vain. The second approach overcomes this problem by building both squares simultaneously. The main advantage of the first approach is that instead of a random base square, it is possible to prescribe some other initial square with particular properties. As this becomes useful later on, the first approach is adopted, and discussed for the remainder of this section. In fact, an implementation of the second approach was no more effective at generating random Latin squares than the first.

**Current Results**

An enormous amount of work has gone into the study of mutually orthogonal Latin squares (MOLS) and Latin squares in general. An excellent reference devoted to the theory and applications of Latin squares is given by Denes and Keedwell [20], which updates their previous publication [19] on the same topic. In [13], Colbourn and Dinitz describe the process of building the well known MOLS-table, which gives the best known lower bound on the number of MOLS for all orders  $n \leq 10,000$ .

The question of the existence of a pair of orthogonal Latin squares of order  $n$  was first asked by Euler who showed their existence except when  $n \equiv 2 \pmod{4}$ . In the case of order 6, Euler posed his famous 36 Officers Problem [27], the solution to which was equivalent to finding a pair of orthogonal Latin squares of order 6. In 1782, he conjectured that it was impossible to find any such pair of orthogonal Latin squares of order  $n$  for  $n \equiv 2 \pmod{4}$ . In 1960, the combined efforts of Bose, Shrikhande and Parker [5], proved that the Euler conjecture was false for all odd multiples of two except for  $n=2$  and  $n=6$ , leading to the following theorem:

**Theorem:**

There exists a pair of orthogonal Latin squares of order  $n$  exactly when  $n \neq 2,6$ .

In the context of mutually orthogonal Latin squares (MOLS), the number  $N(n)$  is the maximum number of Latin squares in a set of MOLS of side  $n$ . The table below gives the number  $N(n)$  for  $1 \leq n \leq 10$ :

Order, $n$	1	2	3	4	5	6	7	8	9	10
$N(n)$	-	1	2	3	4	1	6	7	8	$\geq 2$

All entries in the table above are exact, except for  $N(10)$ , in which 2 is the largest known value for this order.

A set of  $n-1$  MOLS of side  $n$  is called a *complete* set of MOLS. The existence of a complete set of MOLS of side  $n$  is equivalent to the existence of a projective plane or affine plane of order  $n$ . The only known projective and affine planes are of prime power order, and likewise a complete set of MOLS is known only for those side  $n$  Latin squares for which  $n$  is a prime power, ie.  $n = p^e$ , for  $p$  prime. The famous search and subsequent non-existence result for projective planes of order 10 was performed by Lam, Thiel and Swiercz in 1989 [42]. Their

result also established the non-existence of a complete set of 9 MOLS of side 10. As the above table exhibits, for Latin squares of side 10, it is currently not known whether there exists 3 Latin squares which are mutually orthogonal. A great amount of effort has gone into searching for such a triad of Latin squares, but without success. One of the closest constructions was that of A.E. Brouwer in 1984, in which 4 Latin squares of side 10 were constructed which were almost pairwise orthogonal, but for the fact that they all shared a common  $2 \times 2$  subsquare.

As part of the analysis in this thesis, for every constructed pair of orthogonal Latin squares of side 10, an exhaustive search is implemented to see whether or not there exists a third Latin square which is orthogonal to them both.

### Implementation

The algorithm developed in this section for the construction of a pair of orthogonal Latin squares involved two main stages. In the first stage, a random Latin square of side  $n$ , the base square, was constructed using the standard hill-climbing algorithm outlined in Section 2.3.2. The second stage then involved building an orthogonal mate to the base square. The diagram below shows the state of the algorithm, part way into the construction of the orthogonal mate. The base square has already been constructed, and the orthogonal mate is being built a cell at a time:

5	3	1	2	4
2	4	5	3	1
1	2	4	5	3
3	1	2	4	5
4	5	3	1	2

*Base square*

	1		3	
4	2	1	5	3
1		3		2
3	4			5
5	3	4	2	1

*Orthogonal mate  
being built*

The orthogonal mate is built by generating random triples of the form:  
(row position, column position, element value)

This is just as in Stinson's hill-climbing algorithm for the construction of Latin squares [63], which was described in Section 2.3.2. Each triple is then added to the current partial solution, which is the partially built orthogonal mate. There are two conditions which must be satisfied by each triple of the current partial solution. The first is the Latin square property, as the orthogonal mate under construction must be a valid Latin square. The other condition concerns the orthogonality of the partial solution, and ensures that no ordered pair is covered more than once as the triples are added to the orthogonal mate. The addition of a given random triple to the current partial solution will create one of four possible types of conflicts with the other triples in the partial solution. A description of each type of conflict, along with a corresponding example relating to the diagram above is given below:

### Conflict Types

1) *No conflict*: The new triple satisfies both the Latin square and the orthogonality conditions, and thus can be added to the current partial solution. This increases the size, and hence profit, of the current partial solution.

- For example, suppose  $(\text{row}, \text{column}, \text{element}) = (1, 1, 2)$ .

The ordered pairs are formed by taking the element from the particular cell of the base square followed by the element in the corresponding cell of the orthogonal mate being constructed. In the diagram above, the ordered pair  $(5, 2)$  which is created by this triple is not yet covered, and both row 1 and column 1 still require the symbol "2". In this case,  $(1, 1, 2)$  would be added to the current partial solution, and its size would increase by one.

2) *Latin square conflict*: The new triple causes a conflict with a previous triple, by adding a symbol to a row or column in which it is already represented. The random triples are generated so that at most one triple of the current partial solution can be violated in this

way. This conflict is resolved by replacing the old triple with the new one, and hence the size of the current partial solution does not change.

- For example, suppose  $(\text{row}, \text{column}, \text{element}) = (1, 1, 4)$ .

This triple satisfies the orthogonality constraint by creating the ordered pair  $(5, 4)$ , which is not yet covered, but it violates the Latin square property by repeating the symbol "4" in column 1. In this case, the triple  $(1, 1, 4)$  would be added and the triple  $(2, 1, 4)$  would be removed from the current partial solution. Clearly, the size of the current partial solution would remain unchanged.

**3) Orthogonality conflict:** The new triple creates an ordered pair with the base square which is already covered by some other triple of the partial solution. Again, only one triple in the current partial solution can conflict with the new random triple in this way, because only a single ordered pair is created with each new random triple. As with a single Latin square conflict, the old triple is replaced with the new one, maintaining the size of the partial solution.

- For example, suppose  $(\text{row}, \text{column}, \text{element}) = (1, 3, 2)$ .

The Latin square property for both the first row and the third column is satisfied by placing a "2" in this position, but the ordered pair  $(1, 2)$ , is already covered in the fifth row and fourth column. In this case, the new triple  $(1, 3, 2)$  would be added to the current partial solution, and the existing triple  $(5, 4, 2)$  would be removed to accommodate it. As before, the size of the current partial solution would remain unchanged.

**4) Latin square and Orthogonality conflict:** The new triple may violate the Latin square constraint by adding a symbol to a row or column in which it already exists, and at the same time create an ordered pair with the base square which is already covered, thus simultaneously violating the orthogonality constraint.

- For example, suppose  $(\text{row}, \text{column}, \text{element}) = (1, 5, 5)$ .

This places the symbol "5" into row 1 and column 5, yet it is already represented in column 5. In addition, it creates the ordered pair  $(4, 5)$  with the base square, which is already covered in row 5 of the first column. This is the least desirable type of conflict. If the triple  $(1, 5, 5)$  was added to the current partial solution, then in order to maintain its validity the two triples  $(4, 5, 5)$  and  $(5, 1, 5)$  would need to be removed. Resolving the conflict in this way reduces the size, and hence the profit of the current partial solution.

Several heuristics were devised to handle the different types of conflicts explained above. These heuristics are detailed below, and their performance is then evaluated.

### Heuristic One

This is basically pure hill-climbing, in that only forward and level transitions are accepted throughout the search. Conflicts of type 1, 2 and 3 do not decrease the size of the partial solution, and so randomly generated triples admitting conflicts of these types are always added to the partial solution and any conflicts are handled as outlined above. Any randomly generated triple which admits a conflict of type 4 is simply discarded.

### Heuristic Two

This heuristic handles conflicts of the first three types in the usual manner, but does not discard triples which admit conflicts of type 4. Any randomly generated triple which violates both the orthogonality and Latin square conditions is always accepted into the partial solution, and is accommodated by removal of the two existing triples it conflicts with. This heuristic therefore always accepts backward transitions.

### Heuristic Three

This heuristic accepts backward transitions in a more controlled manner. As before, forward and level transitions corresponding to conflicts of the first three types are always accepted. A simple probability function is used to decide whether or not to accept backward transitions which correspond to conflicts of type 4. This probability decreases linearly as the algorithm progresses and transitions are performed. Let  $X$  denote the stopping condition of the construction, such that if no solution has been found after  $X$  random triples have been

generated, then the search will be abandoned. The probability of accepting a backward transition when the  $N^{\text{th}}$  random triple is constructed will be  $1-N/X$ . During the early stages of the algorithm, backward transitions will nearly always be accepted, but as the algorithm progresses the probability of their acceptance slowly decreases.

A comparison of these heuristics can be performed by examining how successful they each are at generating orthogonal mates. In the cases where the generation of orthogonal mates is difficult, a further comparison can be made of the profits of the current partial solutions once the algorithm becomes stuck. A particular execution of the hill-climb is deemed to be "stuck" if no improvement in the size, or profit, of the partial solution can be made following a prescribed number of attempts at adding random triples.

The table below summarises the effectiveness of each of the three heuristics for generating the orthogonal mate to a random Latin square of order  $n$ , for  $5 \leq n \leq 10$ . For each heuristic and each order, the algorithm is run 1,000 consecutive times and in each case, if a valid mate has not been found after the generation of 200,000 random triples, then the algorithm is said to be stuck. When either the algorithm becomes stuck, or an orthogonal mate is validly constructed, the final profit of the partial solution is recorded. This is averaged over the 1,000 runs and given in the table below, along with the number of valid mates constructed.

Order	<i>Heuristic One</i>		<i>Heuristic Two</i>		<i>Heuristic Three</i>	
	Average Profit On Completion	Orthogonal Mates Produced	Average Profit On Completion	Orthogonal Mates Produced	Average Profit On Completion	Orthogonal Mates Produced
<b>5</b>	21.19	118	16.24	116	21.55	141
<b>6</b>	-	-	-	-	-	-
<b>7</b>	42.96	0	28.47	0	44.51	3
<b>8</b>	56.96	0	36.10	0	58.57	0
<b>9</b>	72.99	0	44.89	0	74.37	0
<b>10</b>	91.02	0	55.14	0	92.19	0

No pair of orthogonal mates exist for  $n = 6$ , and so this row is left blank in the table above. The first observation which can be made from these results is that none of the heuristics were very effective at generating orthogonal mates of order  $n$  for  $n > 6$ . In fact, no orthogonal mates were constructed for Latin squares of orders greater than 7. Of the three heuristics, the second one was by far the worst, clearly indicating that the total acceptance of backward transitions is a very poor approach. The other two heuristics performed more similarly, with the third one being clearly superior.

It should be noted that the algorithm was able to generate orthogonal mates for orders 8, 9 and 10, using published basis squares which are known to contain at least  $n$  disjoint transversals.

### **Application to an Open Problem**

One of the most celebrated open questions regarding MOLs, is the question of existence of a triad of mutually orthogonal Latin squares of order 10. Despite the work that has gone into this problem, no triad has yet been found.

The following probabilistic method was used in this thesis in an attempt to construct such a triad:

- a basis square  $B$  was constructed (see below for details).
- the hill-climbing algorithm, using heuristic three which accepts backward transitions in a controlled fashion, was used to construct a random orthogonal mate,  $M$ , to  $B$
- an exhaustive search was implemented to find a third Latin square, orthogonal to both  $B$  and  $M$ .

**The Basis Square**

In order to admit an orthogonal mate, a given basis square of side  $n$  must contain  $n$  disjoint transversals, as discussed previously. The following Latin square was constructed by E.T. Parker [55]:

5	1	7	3	4	0	6	2	8	9
1	2	3	4	5	6	7	8	9	0
7	3	4	5	6	2	8	9	0	1
3	4	5	6	7	8	9	0	1	2
4	5	6	7	8	9	0	1	2	3
0	6	2	8	9	5	1	7	3	4
6	7	8	9	0	1	2	3	4	5
2	8	9	0	1	7	3	4	5	6
8	9	0	1	2	3	4	5	6	7
9	0	1	2	3	4	5	6	7	8

This 10x10 Latin square is known to contain 5504 transversals and have approximately one million orthogonal mates. Unfortunately, it has been proven that this particular square does not belong to a triad of MOLS. Therefore, there is no point in using this square as the basis square,  $B$ . However, by applying a slight distortion to the above square which maintains the Latin square property but which in general produces a non-isomorphic square, a candidate basis square,  $B$ , can be constructed. In [25], Elliott and Gibbons present a cycling operation for Latin squares, which involves picking a random row in the square, swapping two random elements in the row, and continuing to cycle the elements of the rows and columns of the square until the structure once again satisfies the Latin square constraints. The new square is usually not much different to the original, but in general is non-isomorphic, and can therefore be used as the basis square,  $B$ . Sometimes, the square which is produced following the random cycling operation is isomorphic to the above square, due to the fact that all occurrences of one symbol have interchanged with some other symbol. These cases are detected and are not used as basis squares. Experience has shown that the Latin squares resulting from this cycling operation are effective basis squares, and a random orthogonal mate can be constructed via the hill-climbing algorithm with little difficulty.

Once the orthogonal mate was constructed, an exhaustive search was implemented to find a third Latin square, orthogonal to both the basis and its mate. If such a square was found, the three squares would complete the triad, and answer the existence question. The exhaustive search attempted to build the third square row by row, in lexicographical order, backtracking when either the Latin square property or the orthogonality property were violated. The general technique of backtracking is presented in Chapter 3. Even though the constraints to this search were quite tight, a complete exhaustive check was a very expensive operation.

**Results**

Unfortunately, no triad of mutually orthogonal Latin squares of order 10 were found. The closest construction is given below, in which the third square has been completed successfully up to and including row 7, but the final three rows could not be validly completed.

**SquareOne:**

5	1	7	3	4	10	6	2	8	9
1	2	3	4	5	6	7	8	9	10
7	3	9	5	6	2	8	4	10	1
3	4	5	6	7	8	9	10	1	2
4	5	6	7	8	9	10	1	2	3
10	6	2	8	9	5	1	7	3	4
6	7	8	9	10	1	2	3	4	5
2	8	4	10	1	7	3	9	5	6
8	9	10	1	2	3	4	5	6	7
9	10	1	2	3	4	5	6	7	8

**SquareTwo:**

2	8	1	5	10	9	7	3	6	4
3	6	8	4	7	5	10	1	9	2
8	2	3	6	4	7	9	5	10	1
7	3	10	1	5	4	6	8	2	9
1	9	6	3	2	8	4	7	5	10
6	10	2	8	1	3	5	9	4	7
9	4	7	2	3	10	1	6	8	5
4	5	9	7	6	2	3	10	1	8
10	7	5	9	8	1	2	4	3	6
5	1	4	10	9	6	8	2	7	3

**Square Three:**

1	2	3	4	5	6	7	8	9	10
3	1	5	2	9	10	6	7	4	8
4	10	9	7	3	5	8	1	2	6
8	6	10	1	7	4	2	3	5	9
9	3	8	10	2	1	5	4	6	7
10	4	7	6	8	2	9	5	1	3
2	9	1	3	4	8	10	6	7	5

## 2.3.4 STEINER TRIPLE SYSTEMS

**Definition:**

A *Steiner Triple System* of order  $n$ , or  $STS(n)$ , is a pair  $(X, B)$  where  $X$  is a set of  $n$  elements called *points*, and  $B$  is a set of 3-element subsets of  $X$  called *blocks*, such that every unordered pair of distinct points is contained in a unique block.

An  $STS(n)$  has  $n(n-1)/6$  blocks. A necessary condition for the existence of an  $STS(n)$  is that  $n \equiv 1, 3 \pmod{6}$ . As demonstrated by Kirkman [38] in 1847 this condition is also sufficient. Thus Steiner Triple Systems exist for orders 7, 9, 13, 15, 19, . . .

A Steiner Triple System is a special type of well known combinatorial design called a balanced incomplete block design, or BIBD. A BIBD is a pair  $(V, B)$  where  $V$  is a set of  $v$  points or elements, and  $B$  is a collection of  $b$   $k$ -element subsets of  $V$  called blocks, such that each element of  $V$  is contained in exactly  $r$  blocks, and any 2-subset of  $V$  is contained in exactly  $\lambda$  blocks. The numbers  $v, b, r, k$ , and  $\lambda$  are the parameters of the BIBD.

An STS is therefore a  $(v, b, r, k, \lambda) = (v, v(v-1)/6, (v-1)/2, 3, 1)$ -BIBD.

An enormous amount of literature exists regarding Steiner Triple Systems, and bibliographies such as [24] have been compiled to organise and reference much of this material. A more up to date publication on triple systems in general is given by Colbourn and Rosa [17].

### 2.3.4.1 RANDOM STEINER TRIPLE SYSTEM GENERATION

In 1985, Stinson [63] described a fast hill-climbing algorithm for the construction of random Steiner Triple Systems, which is very similar to the algorithms of Sections 2.3.1 and 2.3.2 of this chapter. Stinson's method constructs a series of random blocks, or triples, adding each one to a set of blocks comprising a partial STS. A new random block causes a violation if it covers an unordered pair of points which is already covered by another block in the partial STS. In such a case, the block in the partial STS which it conflicts with is removed to accommodate the addition of the new random block. Otherwise each random block is simply added to the partial STS, increasing its profit, and taking it one block closer to completion. The random triples are constructed in such a way that they cannot conflict with



more than one existing block. This method is extremely effective, and can generate large STSs very quickly. For further details of the algorithm, refer to Stinson's paper [63].

An implementation of Stinson's algorithm was used in this thesis to generate random Steiner Triple Systems which were then applied in other problems. For example, Section 2.3.5 of this chapter, on Weakly Derived Steiner Triple Systems, requires a random STS to be constructed as a base design for the main problem. The implementation was very effective, and able to generate random STSs for any order under 400 in less than a second. A random STS(999), which contains 166,167 blocks, was generated in around 10 seconds. This particular construction involved 838,430 switching operations of the randomly generated triples, 672,263 of which replaced an existing triple in the current partial solution. The remaining 166,167 switching operations correspond to random triples which caused no violations with existing triples, and were added to the partial STS, increasing its profit.

### 2.3.4.2 CYCLIC STEINER TRIPLE SYSTEM GENERATION

**Definition:**

A *cyclic* Steiner Triple System of order  $v$ , consisting of block set  $B$  on the point set  $V = \{0, 1, 2, \dots, v-1\}$ , is an STS( $v$ ) with an automorphism  $\theta = (0, 1, 2, \dots, v-1)$ .

Automorphisms are formally introduced in Chapter 4, but for the purposes of this section it is sufficient to observe that the effect of the automorphism partitions the blocks of the cyclic STS( $v$ ) into  $\frac{b}{v} = \frac{(v-1)}{6}$  sets of  $v$  blocks, where each block in a set is formed by cycling the elements of the previous block in the set, mod  $v$ .

Cyclic STSs exist for all orders  $v$ , such that  $v \equiv 1, 3 \pmod{6}$ , except for  $v = 9$ . In the case that  $v \equiv 1 \pmod{6}$ , which is assumed throughout this section, such a design can be represented by a set of  $\frac{(v-1)}{6}$  base blocks, or triples, with the remaining blocks of the design formed by applying the automorphism  $\theta^i$ , for  $i = 1, 2, 3, \dots, v-1$ , ie. point  $x \rightarrow (x+1)$ , to the base blocks.

For example, the unique STS(7) is cyclic. It can be represented by a single base block:  $b = (0 \ 1 \ 3)$ . The other 6 blocks are then formed by applying the cyclic automorphism:

Base block:	<b>(0 1 3)</b>
+1 (mod 7)	(1 2 4)
+2 (mod 7)	(2 3 5)
+3 (mod 7)	(3 4 6)
+4 (mod 7)	(4 5 0)
+5 (mod 7)	(5 6 1)
+6 (mod 7)	(6 0 2)

The generation of a cyclic STS( $v$ ), for  $v \equiv 1 \pmod{6}$ , requires only the construction of the set of  $\frac{(v-1)}{6}$  base blocks, from which the complete design is formed by applying the cyclic automorphism. Just as the construction of a random STS can be effectively tackled with a hill-climbing algorithm, so too can the construction of a cyclic STS. In the latter case, the hill-climbing algorithm explicitly constructs only the base blocks of the design, and this process is described below.

The set of base blocks,  $B_{base}$  must satisfy the property that for each *residue*,  $r \neq 0 \pmod{v}$ , there exists exactly one pair,  $\{x,y\}$ , belonging to a block in  $B_{base}$  such that  $x-y \equiv r \pmod{v}$ . In other words, each of the non-zero differences: 1, 2, 3, .....  $v-1$  must be covered exactly once by the base blocks of the design. The set  $B_{base}$  can be constructed by adapting Stinson's hill-climbing algorithm [63] to hill-climb on the  $v-1$  non zero differences, mod  $v$ .

This adapted algorithm constructs random base blocks of the form  $(0 \ x \ y)$  and attempts to add each one to the set  $B_{base}$ . The complete cycle of any base block formed by the application of the cyclic automorphism is referred to as an *orbit*, and hence the base block is

the orbit representative. The assumption that every base block in the set  $B_{\text{base}}$  contains the point "0" is perfectly valid because any block on a cyclic orbit could be used as the representative for that orbit. Every distinct point must be contained in exactly 3 blocks of each complete orbit, and so for any orbit, one of the 3 blocks containing the point "0" can be selected as its representative.

The differences covered by each base block of the form  $(0 \ x \ y)$  are  $x$ ,  $-x$ ,  $y$ ,  $-y$ ,  $y-x$  and  $x-y$  (all mod  $v$ ).

If  $x \pmod{v}$  is covered by one of the base blocks, this implies that  $-x \pmod{v}$  is also covered. Conversely, if  $x \pmod{v}$  is not yet covered by one of the base blocks, then  $-x \pmod{v}$  will also be uncovered. This implied coverage holds similarly for  $y$  and  $-y \pmod{v}$  and for  $y-x$  and  $x-y \pmod{v}$ .

The algorithm therefore picks a random uncovered difference  $x$  and a random uncovered difference  $y$ , and attempts to add the block  $(0 \ x \ y)$  to the current partial solution, which is the set  $B_{\text{base}}$ . If a block exists in the current partial solution which already covers the difference  $y-x \pmod{v}$  then this block is removed before the new one is added. An outline of this algorithm is given below:

```

Bbase = nil;
while |Bbase| < (v-1)/6 do
  begin
    Pick random difference x, uncovered by blocks in Bbase
    Pick random difference y, uncovered by blocks in Bbase
    if difference (y-x) is already covered by block X in Bbase
      remove X from Bbase
    Add block (0 x y) to Bbase
  end

```

### Results

Just as in the original hill-climbing algorithm for the construction of random STSs, this algorithm is particularly effective because the size of the partial solution never decreases, and valid choices for  $x$  and  $y$  can always be made. The complete set of base blocks for a given cyclic STS is generated very quickly, from which the entire design can be trivially constructed.

An example of the execution of the algorithm for constructing a cyclic STS(13) is given below. Such a design is represented by just 2 base blocks, and the following were generated almost instantly:

```

Base Block 1: 0 9 12
Base Block 2: 0 2 7

```

The complete cyclic STS(13) is constructed by applying the cyclic automorphism to each of the base blocks. The table given on the next page expands each orbit to produce a random STS(13), on the point set  $\{0,1,2,\dots,12\}$ :

Orbit 1:				Orbit 2:			
Base Block 1:	0	9	12	Base Block 2:	0	2	7
Block 1.1:	1	10	0	Block 2.1:	1	3	8
Block 1.2:	2	11	1	Block 2.2:	2	4	9
Block 1.3:	3	12	2	Block 2.3:	3	5	10
Block 1.4:	4	0	3	Block 2.4:	4	6	11
Block 1.5:	5	1	4	Block 2.5:	5	7	12
Block 1.6:	6	2	5	Block 2.6:	6	8	0
Block 1.7:	7	3	6	Block 2.7:	7	9	1
Block 1.8:	8	4	7	Block 2.8:	8	10	2
Block 1.9:	9	5	8	Block 2.9:	9	11	3
Block 1.10:	10	6	9	Block 2.10:	10	12	4
Block 1.11:	11	7	10	Block 2.11:	11	0	5
Block 1.12:	12	8	11	Block 2.12:	12	1	6

### 2.3.4.3 3-CYCLIC STEINER TRIPLE SYSTEM GENERATION

**Definition:**

An STS(v), where  $v = 3t$  and  $v \equiv 3 \pmod{6}$ , is *3-cyclic* if it has an automorphism of the form:

$$\theta = (0,1,2,\dots,t-1) (t,t+1,t+2,\dots,2t-1) (2t,2t+1,2t+2,\dots,3t-1).$$

In other words, the blocks of the design can be partitioned into classes, or orbits, such that each block in a given orbit can be generated by cycling the elements of the previous block in the orbit according to the automorphism above. Giving each cycle a subscript, the automorphism can equivalently be represented as:

$$\theta = (0_0,1_0,\dots,(t-1)_0) (0_1,1_1,\dots,(t-1)_1) (0_2,1_2,\dots,(t-1)_2).$$

A 3-cyclic design can be represented by a set of  $\binom{v-1}{2}$  base blocks, with the remaining blocks of the design obtained by applying the automorphism  $\theta^i$ , for  $i = 1,2,\dots,t-1$  to the base blocks.

In the previous section, STSs with single cycles were constructed, and the difference between two points of a base block was a single residue. However, for a 3-cyclic design, the points within a base block may belong to distinct cycles of the automorphism, and therefore the difference between two points is either a pure difference or a mixed difference. It is a pure difference if the points belong to the same cycle, and it is a mixed difference if the points belong to distinct cycles.

In order to construct a complete set of valid base blocks,  $B_{\text{base}}$ , all pairs of points within the blocks of  $B_{\text{base}}$  must cover all the mixed, and all the non-zero pure differences. The pure difference of zero clearly should not be covered, as this would imply that a single block has exactly the same point, from the same cycle, repeated in it.

To generate a given 3-cyclic STS, the cyclic STS algorithm presented in the previous section is modified to hill-climb on the mixed and non-zero pure differences to produce the necessary  $\binom{v-1}{2}$  base blocks belonging to  $B_{\text{base}}$ .

In order to cover these differences correctly, the base blocks of the design must be constructed so that they each consist of the correct number of points from each of the cycles. In order to maintain this constraint throughout the algorithm, a template is used which specifies to which cycle the points within each of the base blocks must belong. This template is called a *tactical decomposition*.

A brief description of the construction of tactical decompositions is given below, which is followed by an outline of a hill-climbing algorithm for the construction of 3-cyclic Steiner Triple Systems, making use of the tactical decompositions.

**Building Tactical Decompositions**

A tactical decomposition serves as an underlying template for the generation of the base blocks of a design, which in this case is a 3-cyclic triple system. If the number of base blocks

of the design is  $B$ , then the tactical decomposition can be represented as a  $B \times 3$  array of values from the set  $\{0,1,2\}$ , where each value represents one of the cycles of the automorphism of the design. There are a number of properties of the design which constrain how the values must be arranged in the array. These properties and constraints are discussed in more detail shortly.

The generation of tactical decompositions is performed using a backtracking algorithm. The technique of backtracking will be described in much greater detail in Chapter 3, but for the purposes of this section it is sufficient to accept that backtracking is an exhaustive search technique, and is able to construct tactical decompositions in a prescribed lexicographical order. It is worth mentioning that a backtracking algorithm is not only capable of generating a particular tactical decomposition, but unlike a non-exhaustive technique such as hill-climbing, it has the potential to generate *every* tactical decomposition of a particular size.

The points of the base blocks of a 3-cyclic STS( $v$ ) are classified into 3 separate classes, corresponding to the cycles of the automorphism of the design. As previously mentioned, the difference between two points has been extended from a single residue to pure and mixed differences. In fact, with 3 cycles there are 3 types of pure differences and 3 types of mixed differences, which are represented using the scheme below:

The 3 types of pure differences are represented as: 00, 11, 22

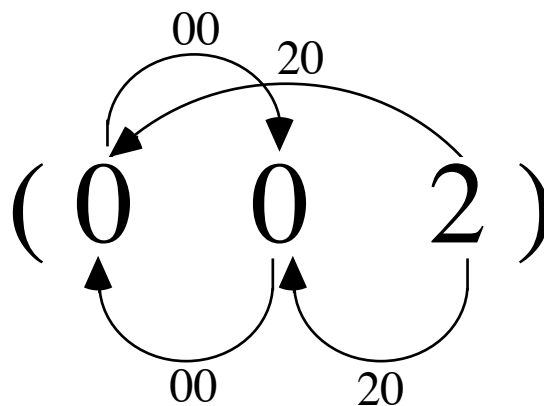
The 3 types of mixed differences are represented as: 10, 20, 21

The first digit in each representation corresponds to the cycle to which the first point belongs, and the second digit corresponds to the cycle to which the second point, which is subtracted from the first point, belongs. Each base block of the design has a corresponding block in the tactical decomposition, and the blocks in the tactical decomposition must cover the pure and mixed differences with the following frequencies:

The pure differences: 00, 11 and 22 must be covered  $t-1$  times.

The mixed differences: 10, 20 and 21 must be covered  $t$  times.

For example, consider the block  $(0 \ 0 \ 2)$  belonging to the tactical decomposition. This particular block covers the pure difference 00 and the mixed difference 20.



The pure and mixed differences covered by  $(0 \ 0 \ 2)$

In fact, 00 is covered twice (once in each direction) and 20 is also covered twice. The other direction for mixed differences (ie. 02) does not need to be considered because if a difference  $x \pmod{t}$  is covered in the direction 20 then that implies the difference  $-x \pmod{t}$  is covered in the direction 02.

There are 10 distinct block types which can be used to generate a valid tactical decomposition for a 3-cyclic STS( $v$ ). Each block in the tactical decomposition is an underlying template for a base block of the 3-cyclic design, and specifies the valid cycle

types allowed in the points of the base block. The 10 block types are given below, along with the corresponding pure and mixed differences covered by each one.

Distinct Block Type	PureDifferencesCovered	MixedDifferencesCovered
000	00, 00, 00, 00, 00, 00	
001	00, 00	10, 10
002	00, 00	20, 20
011	11, 11	10, 10
012		10, 20, 21
022	22, 22	20, 20
111	11, 11, 11, 11, 11, 11	
112	11, 11	21, 21
122	22, 22	21, 21
222	22, 22, 22, 22, 22, 22	

The backtracking algorithm proceeds by systematically constructing partial tactical decompositions which consist of subsets of the distinct block types given above which cover a proportion of the required pure and mixed differences. Block types are added to the current partial tactical decomposition in lexicographical order until a conflict arises. A conflict occurs when a given pure difference is represented more than  $t-1$  times, or when a given mixed difference is represented more than  $t$  times. Conflicts are resolved by removing the most recently added block type and replacing it with the next block type in lexicographical order. A valid tactical decomposition is constructed whenever the partial design contains  $\binom{v-1}{2}$  blocks and there are no conflicts.

Once the first tactical decomposition is constructed, a new one can be generated simply by continuing with the ordered construction process. The advantage of this approach is that the tactical decompositions are constructed in a strict lexicographical order, and if necessary, the algorithm can proceed until all such decompositions have been made.

**Results**

The algorithm described above for constructing tactical decompositions was implemented, and executed for order 21. A 3-cyclic STS(21) contains 3 cycles of length 7, and is represented by 10 base blocks. The first tactical decomposition constructed by the algorithm is listed below:

Order,  $v = 21$ . Cycle length,  $t = 7$ .

- Block 1: [0] [0] [0]
- Block 2: [0] [1] [1]
- Block 3: [0] [1] [1]
- Block 4: [0] [1] [2]
- Block 5: [0] [1] [2]
- Block 6: [0] [1] [2]
- Block 7: [0] [2] [2]
- Block 8: [0] [2] [2]
- Block 9: [1] [1] [2]
- Block 10: [1] [2] [2]

As required by the constraints of the construction process, all the pure differences are covered 6 times, and all the mixed differences are covered 7 times.

**Building 3-cyclic Steiner triple systems**

Using the tactical decompositions generated by the algorithm outlined above, the complete set of base blocks,  $B_{base}$ , of a random 3-cyclic STS can be constructed with the algorithm outlined below:

- On each iteration, a template block, say (a b c), is randomly chosen and removed from the tactical decomposition.
- A random difference, x, of the type ba is chosen which is not yet covered by any of the base blocks of the design.

- A random difference,  $Y$ , of the type  $ca$  is chosen which is not yet covered by any of the base blocks of the design.
- A check is made to determine if the difference  $Y_c - X_b$  is already covered by a block,  $Z$ , in the current set of base blocks,  $B_{\text{base}}$ .
- If it is, then  $Z$  is removed from  $B_{\text{base}}$ , and its template is returned to the tactical decomposition for later use.
- The block  $(0_a \ X_b \ Y_c)$  is added to  $B_{\text{base}}$ .

Not all tactical decompositions admit a complete solution set of base blocks. In practice, for those decompositions that do admit solutions, the hill-climbing algorithm will find one quickly. However, if the tactical decomposition being used as the template does not admit a solution, the hill-climbing will obviously fail. This problem is overcome by restarting the hill-climbing algorithm with a new tactical decomposition whenever finding a solution seems particularly difficult.

An outline of the complete construction process for random 3-cyclic Steiner Triple Systems is summarised below:

- The first tactical decomposition is generated by the backtracking algorithm and is used as the underlying template for the hill-climbing construction.
- The hill-climbing algorithm is then executed, picking random blocks from the underlying template, and attempting to cover all the pure and mixed differences. A threshold is set on the number of consecutive iterations of the algorithm which may be performed without an increase in the size of the partial solution. If this threshold is exceeded, then it can be assumed the underlying tactical decomposition does not admit a solution.
- If a solution is found, the algorithm completes successfully.
- If no solution is found and the threshold is exceeded, the backtracking algorithm is used to construct the next tactical decomposition, and the hill-climbing algorithm is repeated with this new underlying template.

### Results

Most of the time in the construction algorithm is spent backtracking to find tactical decompositions, for as the order of the designs increases, the time taken to generate the tactical decompositions increases exponentially. Once a tactical decomposition is found which admits a solution, the base blocks of the 3-cyclic STS can be built very rapidly. For example, to build a 3-cyclic STS(123), which has 3 cycles of length 41, 61 base blocks and 2,501 blocks, 17 tactical decompositions were built and tested before one of them admitted a complete set of base blocks. The construction of the first 17 tactical decompositions took slightly more than a second, whereas the time taken to actually build the set of 61 base blocks using the 17<sup>th</sup> tactical decomposition was less than a tenth of a second.

The construction of a 3-cyclic STS(21) using the algorithm just described, is now briefly presented. There are only 8 tactical decompositions generated by the algorithm for order 21. The first of these did not admit a complete set of base blocks,  $B_{\text{base}}$ , following 100,000 consecutive iterations without improvement in the size of the set. This tactical decomposition, shown on the left in the table overleaf, was then discarded and the next tactical decomposition was constructed by the backtracking algorithm, which is also displayed.

Tactical Decomposition #1	Tactical Decomposition #2
Block 1: [0] [0] [0]	Block 1: [0] [0] [0]
Block 2: [0] [1] [1]	Block 2: [0] [1] [2]
Block 3: [0] [1] [1]	Block 3: [0] [1] [2]
Block 4: [0] [1] [2]	Block 4: [0] [1] [2]
Block 5: [0] [1] [2]	Block 5: [0] [1] [2]
Block 6: [0] [1] [2]	Block 6: [0] [1] [2]
Block 7: [0] [2] [2]	Block 7: [0] [1] [2]
Block 8: [0] [2] [2]	Block 8: [0] [1] [2]
Block 9: [1] [1] [2]	Block 9: [1] [1] [1]
Block 10: [1] [2] [2]	Block 10: [2] [2] [2]

This second tactical decomposition did admit a complete set of base blocks, and so the algorithm terminated.

This set of 10 base blocks,  $B_{base}$  for the 3-cyclic STS(21) is given below:

- Base Block 1:  $0_0 6_1 0_2$
- Base Block 2:  $0_2 5_2 1_2$
- Base Block 3:  $0_0 1_1 1_2$
- Base Block 4:  $0_0 0_1 3_2$
- Base Block 5:  $0_0 4_1 2_2$
- Base Block 6:  $0_0 2_0 6_0$
- Base Block 7:  $0_0 2_1 6_2$
- Base Block 8:  $0_0 5_1 4_2$
- Base Block 9:  $0_0 3_1 5_2$
- Base Block 10:  $0_1 6_1 2_1$

The subscripted numbers represent the cycles to which each point belongs. It is clear, from examination of the subscripts, how the tactical decomposition was used as the underlying template for these base blocks. Base block 6 contains the template (0 0 0), base block 10 contains the template (1 1 1) and base block 2 contains the template (2 2 2). All the other base blocks contain the template (0 1 2).

Generating the complete orbit of length 7 for each one of these 10 base blocks constructs all 70 blocks in the 3-cyclic STS(21). These blocks are listed below in their 10 orbits. The first, representative block of each orbit is the base block generated by the hill-climbing algorithm:

$0_0 6_1 0_2$	$0_2 5_2 1_2$	$0_0 1_1 1_2$	$0_0 0_1 3_2$	$0_0 4_1 2_2$
$1_0 0_1 1_2$	$1_2 6_2 2_2$	$1_0 2_1 2_2$	$1_0 1_1 4_2$	$1_0 5_1 3_2$
$2_0 1_1 2_2$	$2_2 0_2 3_2$	$2_0 3_1 3_2$	$2_0 2_1 5_2$	$2_0 6_1 4_2$
$3_0 2_1 3_2$	$3_2 1_2 4_2$	$3_0 4_1 4_2$	$3_0 3_1 6_2$	$3_0 0_1 5_2$
$4_0 3_1 4_2$	$4_2 2_2 5_2$	$4_0 5_1 5_2$	$4_0 4_1 0_2$	$4_0 1_1 6_2$
$5_0 4_1 5_2$	$5_2 3_2 6_2$	$5_0 6_1 6_2$	$5_0 5_1 1_2$	$5_0 2_1 0_2$
$6_0 5_1 6_2$	$6_2 4_2 0_2$	$6_0 0_1 0_2$	$6_0 6_1 2_2$	$6_0 3_1 1_2$
$0_0 2_0 6_0$	$0_0 2_1 6_2$	$0_0 5_1 4_2$	$0_0 3_1 5_2$	$0_1 6_1 2_1$
$1_0 3_0 0_0$	$1_0 3_1 0_2$	$1_0 6_1 5_2$	$1_0 4_1 6_2$	$1_1 0_1 3_1$
$2_0 4_0 1_0$	$2_0 4_1 1_2$	$2_0 0_1 6_2$	$2_0 5_1 0_2$	$2_1 1_1 4_1$
$3_0 5_0 2_0$	$3_0 5_1 2_2$	$3_0 1_1 0_2$	$3_0 6_1 1_2$	$3_1 2_1 5_1$
$4_0 6_0 3_0$	$4_0 6_1 3_2$	$4_0 2_1 1_2$	$4_0 0_1 2_2$	$4_1 3_1 6_1$
$5_0 0_0 4_0$	$5_0 0_1 4_2$	$5_0 3_1 2_2$	$5_0 1_1 3_2$	$5_1 4_1 0_1$
$6_0 1_0 5_0$	$6_0 1_1 5_2$	$6_0 4_1 3_2$	$6_0 2_1 4_2$	$6_1 5_1 1_1$

The design can also be represented in a more useful form, on the point set  $\{0,1,2,\dots,20\}$ , rather than in point-cycle notation as above. This is easily achieved by relabelling the points of the design so that point  $x$  in cycle  $y$  maps to the value  $7y+x$ . (ie.  $x_y \rightarrow 7y+x$ ).

The techniques described throughout Section 2.3.4 allow the efficient construction of random Steiner Triple Systems, including cyclic and 3-cyclic varieties. These tools were utilised in other problems tackled in this thesis, such as in the Section 2.3.5 on Weakly Derived Steiner Triple Systems.

### 2.3.5 WEAKLY DERIVED STEINER TRIPLE SYSTEMS

Mendelsohn [52] posed the question of the construction of so-called weakly derived Steiner triple systems. This section defines these structures, and presents and analyses two approaches for their construction using probabilistic algorithms.

**Definition:**

An STS( $v$ ) is *weakly derived* if its triples, or blocks, can be arranged into an array with the following properties. The array consists of  $v$  rows and  $\frac{v(v-1)}{6}$  columns, which is the number of triples in the design. The rows of the array are indexed by the points of the design,  $\{1, 2, \dots, v\}$ . The leftmost  $\frac{(v-1)}{2}$  columns of the  $i^{\text{th}}$  row of the array contain all those triples containing point  $i$  in the design. Clearly every triple of the design appears exactly three times in these leftmost columns, once in each row corresponding to the points of the triple. The final constraint is that every distinct triple not in the design appears exactly once in the array, such that each row of the completed array is a valid STS( $v$ ).

For example, the 7 triples of an STS(7) are listed below:

125
136
147
234
267
357
456

These can be arranged into an array satisfying the above constraints, and is therefore weakly derived:

125	136	147	567	246	237	345
267	215	234	356	146	137	457
316	324	357	256	145	127	467
456	423	417	135	257	126	367
546	512	537	134	167	247	236
645	627	613	157	347	124	235
726	735	714	245	123	156	346

The leftmost  $\frac{(v-1)}{2} = 3$  columns of the array contain all the triples of the original design, arranged so that all the triples containing point  $i$  are in the  $i^{\text{th}}$  row. The remaining columns, on the right hand side of the array, contain all the triples on 7 points which do not belong to the original design, exactly once. In addition, every row of the array is a valid STS(7).

**Definition:**

A *Steiner quadruple system* of order  $v$ , an SQS( $v$ ), is a design on  $v$  points in which the size of each block is 4, and every unordered triple of points occurs exactly once in the blocks of the design. SQS( $v$ )'s exist for all  $v \equiv 2,4 \pmod{6}$ .

A weakly derived STS( $v$ ) can be constructed directly from a given SQS( $v+1$ ). For any SQS( $v+1$ ) and any point  $p$  of this design, the set of blocks containing  $p$ , but with  $p$  removed,



form a Steiner triple system on  $v-1$  points. This triple system is said to be derived from the  $SQS(v)$ . Consider the following theorem:

**Theorem:**

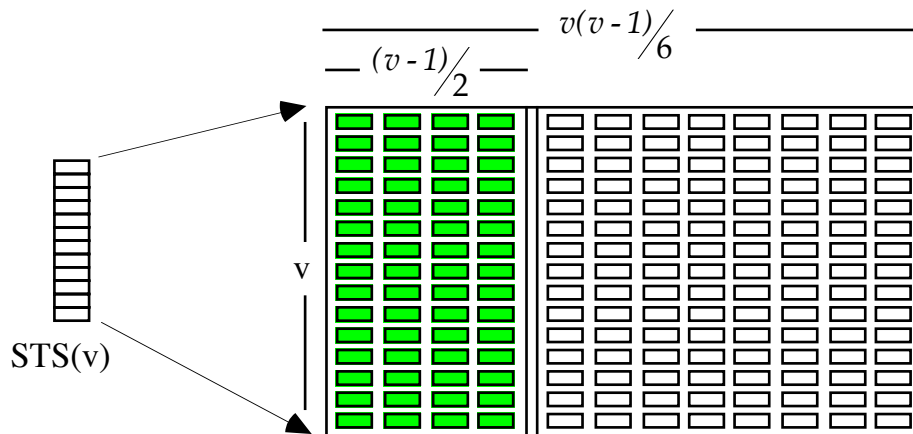
Every Steiner triple system of order  $v \leq 15$  is the derived system of some  $SQS(v+1)$  [14].

Mendelsohn [52] has noted that if the  $STS(v)$  is derived from an  $SQS(v+1)$ , then it is weakly derived. A proof of this statement is summarised briefly below.

Let  $T$  be an  $SQS(v+1)$ , on the point set  $\{1,2,3,\dots,v,v+1\}$ . Take all the blocks of  $T$  which contain point  $j$ , say  $T_j$ , and replace all the triples of the form  $(a, b, v+1)$  by  $(a, b, j)$ . Put these triples into the first  $\binom{v-1}{2}$  columns of row  $j$ , and fill the rest of row  $j$  by the remaining triples of  $T_j$ . An array formed in this manner will satisfy the weakly derived properties.

From the previous theorem, it becomes clear that every Steiner triple system of order  $\leq 15$  is weakly derived, and so can be arranged in an array as described above.

The aim of this section is to develop a non-exhaustive, probabilistic algorithm which can successfully arrange a random  $STS$  into an array so as to show that it is weakly derived. The schematic diagram below helps illustrate the basic approach taken to solving this problem:



A random  $STS(v)$  is constructed using Stinson's hill-climbing algorithm [63] described in Section 2.3.4.1. The left hand side of the array is filled in with the triples of the  $STS$  such that all the triples containing point  $i$  are placed on row  $i$ .

There are exactly  $\binom{v}{3}$  distinct triples on  $v$  points. A given  $STS(v)$  consists of  $v\binom{v-1}{2}/6$  of these, leaving  $\binom{v}{3} - v\binom{v-1}{2}/6$  distinct triples not in the design.

The number of cells in the right hand side of the array, which are not initially filled in by the triples of the design, is exactly:

$$v\left(\binom{v-1}{2} - \frac{v-1}{2}\right)$$

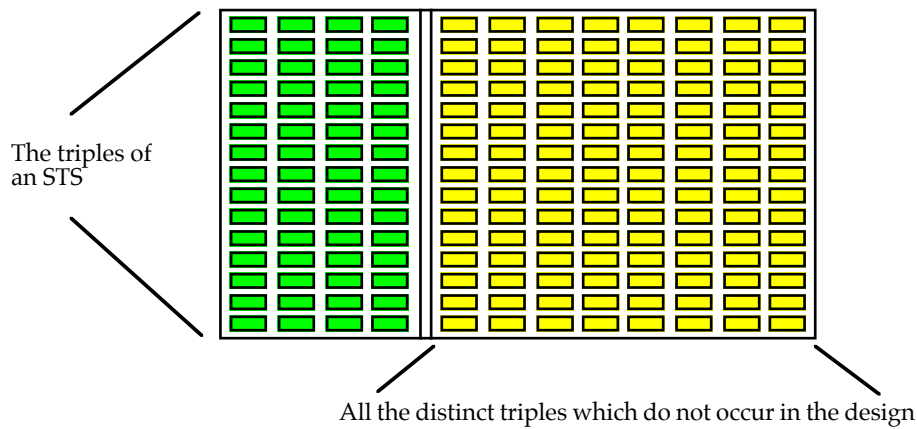
as there are  $v$  rows, and all but  $\binom{v-1}{2}$  of the  $v\binom{v-1}{2}/6$  columns are filled by the triples of the design. This can clearly be seen in the diagram above.

Simple algebraic manipulation shows that:

$$v\left(\binom{v-1}{2} - \frac{v-1}{2}\right) = \binom{v}{3} - v\binom{v-1}{2}/6$$

and so every one of the distinct  $\binom{v}{3}$  triples on  $v$  points which are not triples of the design can be placed exactly once into the remaining right hand cells of the array.

In terms of constructing a weakly derived STS( $v$ ), these unused triples must be arranged in the right hand columns of the array such that every row of the array becomes a valid STS( $v$ ):



Two main approaches have been considered for this construction problem. The first approach uses a "repair" type technique, where all the unused triples are placed randomly into the right hand side of the array, causing a number of violations. The algorithm then attempts to rearrange these randomly placed triples by swapping, or interchanging, them to satisfy the constraints of the problem. The triples in the left hand side of the array can be thought of as being "unswappable" because they correspond to the triples of the original STS and their placement is determined by their constituent points. All the triples in the right hand side of the array are "swappable", because they must be rearranged to satisfy the constraints of the problem. This approach is detailed in Section 2.3.5.1.

The other approach uses a "placement" technique which initially leaves the right hand side the array empty, and attempts to add the unused triples one by one to the rows of the array while at no point violating any of the constraints. This means that at any stage of the algorithm, each row of the array will be a valid partial STS. Section 2.3.5.2 details this approach.

As noted previously, it should be possible to construct an array of the required form for any base STS( $v$ ), with  $v \leq 15$ .

### 2.3.5.1 REPAIR SEARCH

In this first approach to the construction problem, the weakly derived array is initially filled in with the triples of the random STS( $v$ ) as well as all of the unused triples. The initial configuration will almost certainly contain a number of violations, and the search is directed by attempting to remove these from the array.

For example, consider the random STS(7) below:

6	7	4
4	3	5
3	2	6
4	2	1
3	7	1
7	5	2
1	6	5

The triples of this design are arranged into the left hand columns of the array, as illustrated on the next page:

124	137	156	000	000	000	000	0
236	214	257	000	000	000	000	0
345	326	317	000	000	000	000	0
467	435	412	000	000	000	000	0
534	527	516	000	000	000	000	0
647	623	615	000	000	000	000	0
746	713	725	000	000	000	000	0

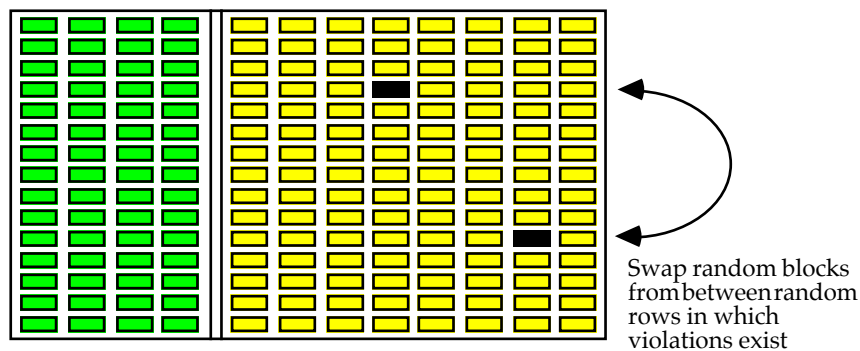
The rightmost column of the above array is simply used to display the number of conflicts in each row. Initially there are no conflicts, because the unused triples have not yet been added to the right hand side of the array. Every unused triple,  $t$ , which is added to one of the right hand columns of the array, has an associated conflict count which specifies how many other triples in the same row contain one of the unordered pairs also contained in  $t$ . The conflict count for each of the triples in the right hand side of the array is summed across the rows, and the total number of conflicts in each row is then displayed in the rightmost column.

Consider the following situation when the unused triples have been added to the right hand side of the array, in lexicographical order:

124	137	156	123	145	237	347	10
236	214	257	125	146	245	356	8
345	326	317	126	147	246	357	7
467	435	412	127	157	247	367	8
534	527	516	134	167	256	456	8
647	623	615	135	234	267	457	5
746	713	725	136	235	346	567	7

The total number of conflicts for the first row of the above array is calculated by summing the individual conflict counts for each of the triples in the row. Triple 123 conflicts with 124, 137 and 237. Triple 145 conflicts with 124 and 156. Triple 237 conflicts with 137, 123 and 347, and triple 347 conflicts with 137 and 237. The total number of conflicts for this row is therefore 10, and this is displayed in the rightmost column of the first row.

The basic operation of the repair technique picks two rows of the array at random which contain conflicts. One triple from each row is then selected, and these are swapped in an attempt to reduce the number of conflicts. Once the total number of conflicts for every row is zero, each row must be a valid STS, and the initial design is shown to be weakly derived. The diagram below illustrates this basic operation:



Three heuristics were developed to control the operations on the weakly derived arrays. These are described below, and the effectiveness and limitations of each heuristic are analysed.

**Heuristic One**

Two rows,  $i$  and  $j$ , which have non-zero conflict values,  $c_i$  and  $c_j$ , are selected at random. For each row, a random triple is selected from its swappable region. Let the random triple from

row  $i$  be  $t_i$ , and from row  $j$  be  $t_j$ . This heuristic then determines whether or not  $t_i$  and  $t_j$  should be swapped, by considering the effect on the total number of conflicts of the two rows. The combined number of conflicts on both rows  $i$  and  $j$  before the swap is calculated as  $c_{\text{bef}} = c_i + c_j$ . If  $t_i$  and  $t_j$  were swapped, the conflicts that would be generated on each row are then calculated as  $c_i'$  and  $c_j'$ . The combined number of conflicts on these rows following the swap is therefore  $c_{\text{aft}} = c_i' + c_j'$ . The triples  $t_i$  and  $t_j$  are only swapped if  $c_{\text{aft}} \leq c_{\text{bef}}$ .

### Results

This heuristic was tested by randomly generating STSs, and trying to construct the corresponding weakly derived array for each design. For any particular run of the algorithm on a given base STS, if 15,000 consecutive swaps of triples were attempted without a decrease in the total number of conflicts of the array, then the particular run was assumed to be frozen, and a new random base STS was generated.

The two orders for which this heuristic was tested were 7 and 9. In both cases, 2000 random base STSs were used. The results for the 2000 runs of the algorithm are given below:

Order	Number of weakly derived arrays successfully constructed	Number of searches frozen, without completing a weakly derived array
7	717	1283
9	1	1999

This heuristic is only moderately effective at producing weakly derived arrays of order 7, and ineffective for order 9. Clearly, only accepting transitions which do not increase the total number of conflicts is too restrictive. This heuristic could be refined by allowing the acceptance of occasional backward transitions, which would present the opportunity to "undo" swaps which although locally were a good choice, created a poor global structure. This refinement is made in a subtle way by the second heuristic.

### Heuristic Two

Again, two random rows with positive conflict values are chosen, and from each row a random triple is selected. Let the two random triples selected be  $t_i$  from row  $i$  and  $t_j$  from row  $j$ . In the first heuristic, the total number of conflicts of the whole row was considered, but now the individual conflict values associated with each triple are used as the criterion for acceptance of a transition. The conflict value of triple  $t_i$  is the number of pairs in  $t_i$  which are contained in other triples on row  $i$ , and let this be denoted by  $c_i(t_i)$ . Similarly,  $c_j(t_j)$  denotes the individual conflict value for triple  $t_j$ , on row  $j$ . Swapping  $t_i$  and  $t_j$  effectively moves  $t_i$  into row  $j$  and  $t_j$  into row  $i$ , and their new conflict values become  $c_j(t_i)$  and  $c_i(t_j)$  respectively. The swap of these two triples is accepted if and only if  $c_j(t_i) + c_i(t_j) \leq c_i(t_i) + c_j(t_j)$ .

This acceptance criterion is less tight than that of the first heuristic, because swaps may be accepted which increase the overall number of conflicts on a given row. This is because  $t_i$  may create new conflicts with other triples on row  $j$ , and similarly for  $t_j$  and row  $i$ . If more new conflicts are created than are destroyed, then the overall number of conflicts in the array will increase. This heuristic attempts to reduce the number of conflicts on two individual triples, and because the total number of conflicts on each row is merely the sum of the individual conflicts of its triples, this does in general guide the search towards an optimal state. Although subtle, this refinement overcomes the main drawback of the first heuristic, and this is reflected in the following results.

### Results

The table on the following page summarises the results of 2000 executions of the algorithm on random base STSs of orders 7 and 9:

Order	Number of weakly derived arrays successfully constructed	Number of searches frozen, without completing a weakly derived array
7	2000	0
9	883	1117

This exhibits a remarkable improvement over the first heuristic. For the smallest order, 7, 100% of the base Steiner triple systems were successfully arranged into the array. For order 9, nearly 50% of the base triple systems were successfully arranged.

This heuristic was then applied to the construction of weakly derived arrays for the next admissible order, 13. The total number of conflicts in all rows of the array is initially around 600, once the right hand columns of the array are randomly filled in with the unused triples.

As the algorithm progresses and the random swapping operations, or switches, are attempted, the total number of conflicts steadily decreases. After approximately 1,000,000 switches, the total number of conflicts reaches a minimum of approximately 75. This behaviour is typical of every execution of the algorithm using this heuristic. The progress has been plotted in Chart 2.1 below, using data taken directly from a single run of the algorithm. Every time the total number of conflicts in the weakly derived array changed value, which was considerably more often during the early stages of the algorithm, the new conflict total was plotted against the total number of switches which had been attempted since the start of the search.

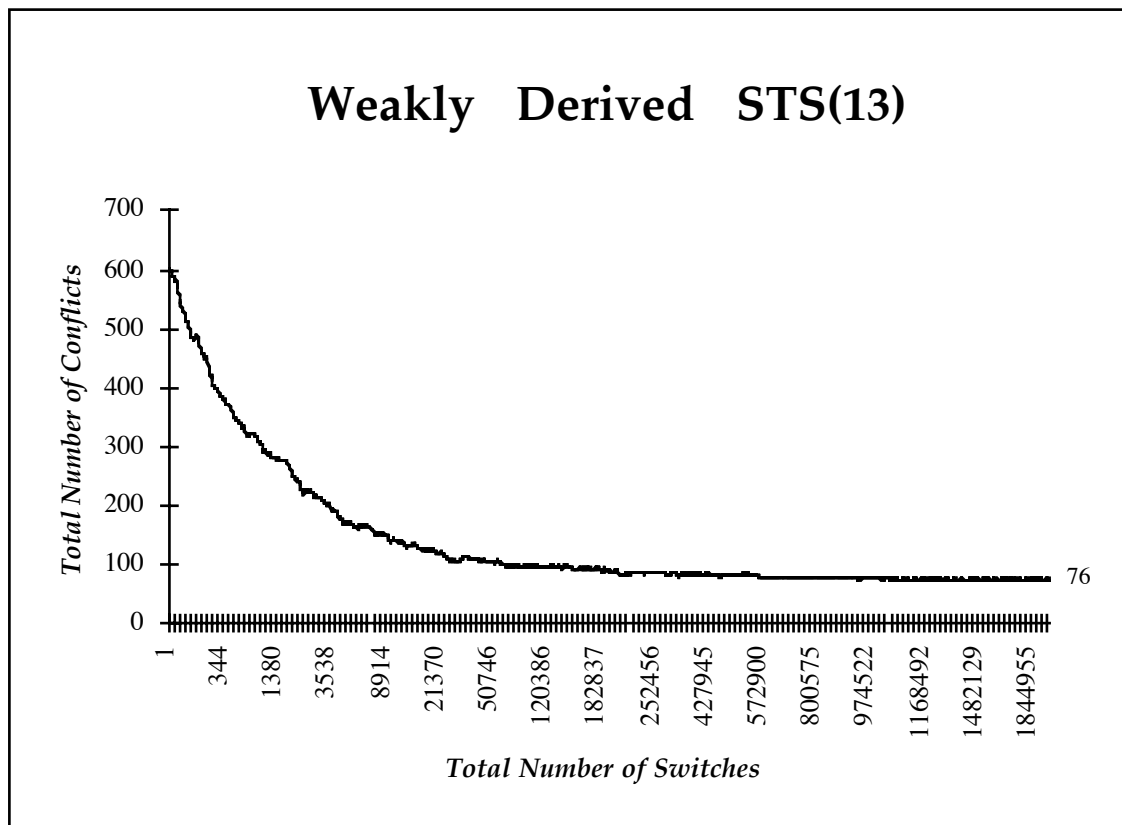


Chart 2.1

Another characteristic of the algorithm's behaviour is the number of consecutive switching operations which are performed without a change to the total number of conflicts in the array. This data is plotted in Chart 2.2 at the top of the next page, and comes from the same run of the algorithm use to produce Chart 2.1.

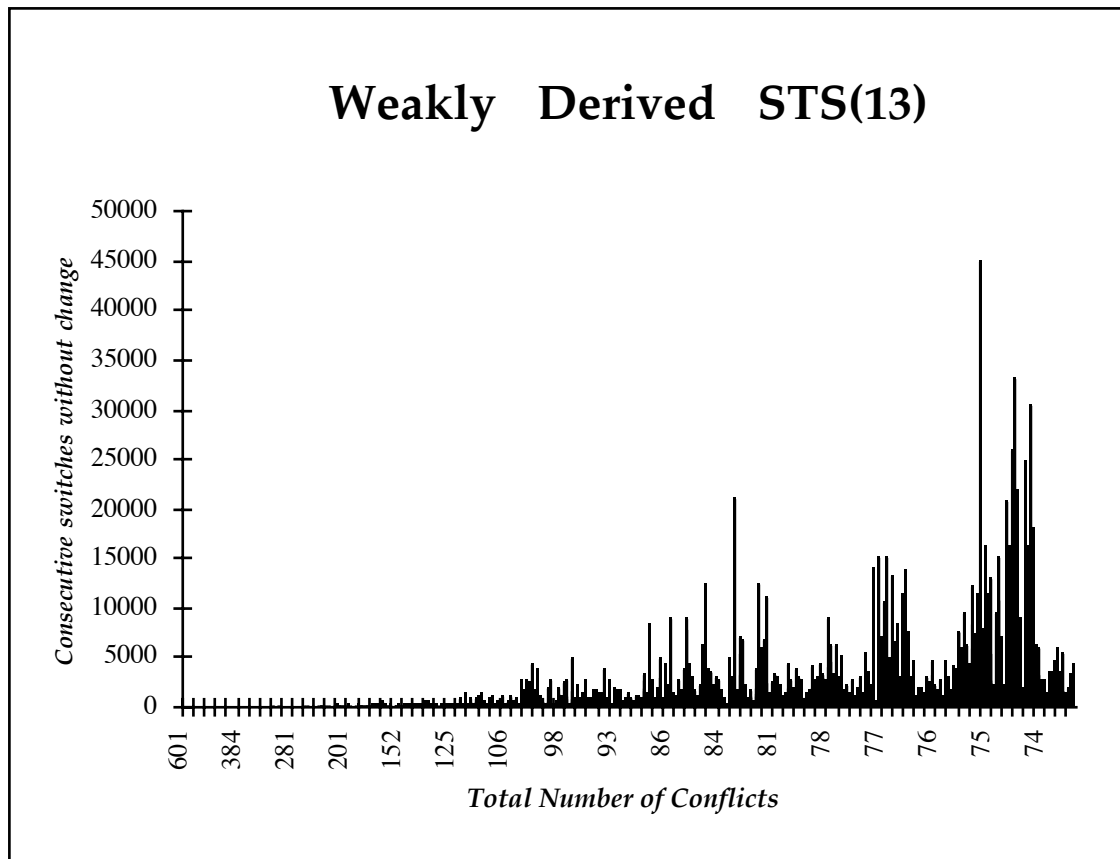


Chart 2.2

Chart 2.2 clearly shows the difficulty in finding useful transitions towards the end of the algorithm. Once the number of conflicts reaches a certain value, there are no locally acceptable switches which improve the global structure of the array.

With so many conflicts remaining when the search gets stuck in a locally optimal configuration, it seems clear that further heuristic refinements will need to be made if a weakly derived STS(13) is to be constructed.

### Heuristic Three

As the size of the weakly derived array increases and the search space gets larger, selecting triples to swap simply by choosing random rows and random triples becomes less effective. This heuristic presents a more desirable way of selecting the triples to swap, and attempts to actively resolve existing conflicts by examining which unordered pairs are required by each row.

In order to implement this heuristic, a more suitable method of measuring the conflicts within the array was required. This was accomplished by defining the "deficiency" value of a given row to be the number of unordered pairs which do not exist in the row. For a design of order  $v$ , there are  $\binom{v}{2}$  unordered pairs. If each of these pairs are covered exactly once by the set of triples in a row, then the set of triples is an STS( $v$ ). Conversely, if a particular row of the array is not a valid STS, then there must be a certain number of the unordered pairs which are not covered by the triples in the row. When all the rows in the array have a deficiency of zero, the array must have the desired properties.

The basic operation of this third heuristic is outlined below:

- A random row,  $r_1$ , is selected which has a non-zero deficiency, thus still requiring at least one unordered pair.
- One of the unordered pairs,  $p$ , still required by row  $r_1$ , is selected at random.

- A second row,  $r_2$ , is selected at random which contains an excess number of pairs of type  $p$ . Such a row is guaranteed to exist if there is at least one row which does not contain the pair  $p$ .
- One of the triples,  $t_p$ , of row  $r_2$ , which contains the pair  $p$  is selected at random. When this triple is removed,  $r_2$  will still contain at least one copy of the pair  $p$ .
- The triple  $t_p$  from row  $r_2$  is swapped with a random triple from row  $r_1$ .

This heuristic is better directed than a purely random approach, because each time a swap is made, a particular deficiency in one of the rows of the array is satisfied by another row which contains an excess of the corresponding unordered pair. Another feature of this heuristic is that once a given row has been completed to form a valid STS, it contains no deficiencies and thus takes no further part in the algorithm. Such rows are said to be *settled*.

**Results**

For the initial array of a weakly derived STS(13), in which the unused triples have been added randomly to the right hand columns, the average number of total deficiencies in the array is about 320. To test the effectiveness of this third heuristic, the algorithm was executed using ten random STS(13)'s, until no improvement could be made in the number of deficiencies. In addition to the number of deficiencies remaining, the number of settled rows were also counted when the algorithm froze. The results are summarised below:

Number of deficiencies when execution "freezes"	Number of rows in which no deficiencies exist, ie. settled rows, when execution "freezes"
40	1
37	1
42	0
42	1
38	1
47	0
38	1
34	2
46	0
42	1
Average: 40.6	Average: 0.8

On average, there were still 40.6 unordered pairs required, equating to each row requiring approximately 3 pairs, when the algorithm froze. In addition, generally no more than one of the rows was completely settled.

In order to impose more control over the acceptance of transitions which increase the number of deficiencies, a simulated annealing strategy was implemented. Here, transitions which decreased the number of deficiencies were always accepted, yet those which increased the number of deficiencies were accepted with a probability which slowly decreased as the algorithm progressed. An exponential acceptance criterion was used, as below:

```

 $\Delta_{cost}$  = number of deficiencies after swap - number of deficiencies before swap
if ( $\Delta_{cost} \leq 0$ ) or ( $random(0,1) < e^{-\Delta_{cost}/temperature}$ )
    accept the transition
    
```

Several different annealing schedules, using various rates of cooling and Markov chain lengths, were implemented and compared. One of the more successful schedules used an initial temperature of 4.0, and a factor of 0.9995 for decreasing the temperature after each Markov chain. The freezing limit was set at 1500 consecutive Markov chains without improvement, and the Markov chain length was increased at the end of each completed chain, so that the search space was more thoroughly examined towards the end of the

algorithm. The Markov chain length,  $L$ , for each temperature level,  $T$ , was calculated according to the formula  $L = 500 + \lfloor 100/T \rfloor$ .

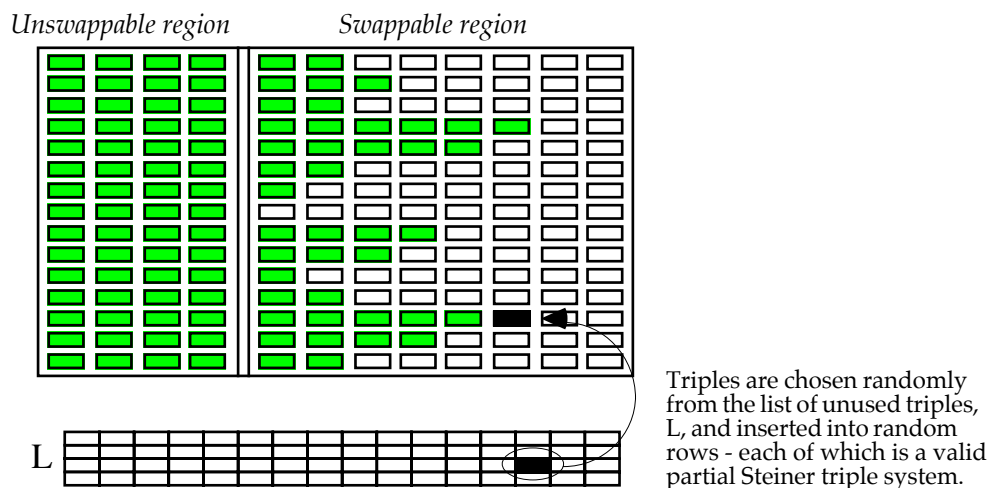
This simulated annealing approach was much more effective than the previous hill-climbing algorithm. A larger proportion of the rows were settled when the execution froze, and the total number of deficiencies were considerably less. The simulated annealing proved most effective when performed several times in succession. Once the annealing had frozen, the temperature was increased again for another run of the algorithm without reinitialising the current solution. Usually after about four successive runs, the most optimal solution was found. The table below gives the minimum number of deficiencies and the number of rows settled following 4 successive applications of the simulated annealing algorithm, for 10 random initial STS(13)'s:

Number of deficiencies when execution "freezes"	Number of rows in which no deficiencies exist, ie. settled rows, when execution "freezes"
19	7
20	6
16	7
19	7
17	7
18	6
18	7
21	7
19	6
17	6
Average: 18.4	Average: 6.6

In conclusion, the repair search technique using the developed heuristics was successful at constructing weakly derived STS(7) and STS(9) designs. Although a number of effective refinements were made to the heuristics, the algorithm was not able to construct weakly derived STSs for larger orders.

### 2.3.5.2 PLACEMENT SEARCH

The other main approach to the construction of weakly derived STSs involved a "placement" technique. The fundamental difference with this approach is that none of the constraints of the problem are violated throughout the search. At any point in the placement algorithm, each row is a valid partial STS, and does not contain more than one distinct unordered pair. A schematic diagram of this technique is given below:





The triples in the left hand side of the array are the triples of the original STS and are fixed, corresponding to the "unswappable" region. The triples in the right hand side of the array can be added and removed, and this is known as the "swappable" region. The algorithm makes use of a list,  $L$ , of all the distinct triples which do not belong to the original design, and must be added to the swappable region. The basic execution of the algorithm is to randomly select a triple from  $L$  and attempt to add it to a random row of the array, without violating any of the constraints. When there are no more triples in  $L$ , the problem is solved.

Each row of the array is a partial STS with a certain subset of fixed triples, and this placement algorithm essentially attempts to complete each of these partial designs. The problem of determining whether a given partial design can be completed, by adding blocks to a fixed subset of given blocks so that each unordered pair is covered exactly once, is NP-complete [12]. A partial STS can be completed using Stinson's standard hill-climbing algorithm [63], with the restriction that some switching operations are not allowed. Stinson examined this problem thoroughly in [63], and stated: "We can try to complete a partial design using the same heuristic as before, except that some switching operations are not allowed - the blocks of the partial design cannot be altered. We suspect that, if a partial design can be completed, this method will either find a completion quite quickly, or reach a *dead-end* from which it cannot escape. Repeated applications of the algorithm should, in most cases, provide a completion of any design which can be completed." This seems to be a promising remark in relation to the current problem, however there is a restriction in the weakly derived STS algorithm, which is that there is a fixed set of triples which can be used to complete each partial design.

The following heuristic was implemented to place the triples randomly from the list  $L$  into the weakly derived array:

#### Triple Placement Heuristic

- A triple,  $t$ , is selected at random from the list of unused triples,  $L$ .
- The triple  $t$  is said to "violate" any block in the weakly derived array which contains one of the unordered pairs also contained in  $t$ .
- A row,  $r$ , is then selected at random with the constraint that  $t$  either violates none of the triples already in  $r$ , or  $t$  violates just one triple on the right hand side (swappable region) of  $r$ .
- In the first case, if no triples are violated, then  $t$  can simply be added to row  $r$ .
- In the second case, if one of the triples on the right hand side of row  $r$  is violated by  $t$ , then this triple is removed from row  $r$  and replaced into the list,  $L$ , of unused triples before  $t$  is added to the row.

This heuristic can be successfully applied for a number of iterations, until the current configuration of the array becomes "frozen". The array is said to be frozen when none of the remaining triples in the list  $L$  can be validly placed into any row of the array. In other words, every triple in the list  $L$  either violates a block in the left hand side (unswappable region) of a row, or it violates more than one block in the row.

Two heuristics were implemented with the aim of overcoming the violations which prevent a triple from being placed once the array becomes frozen. Each heuristic is briefly described below, along with a summary of its effectiveness.

#### Heuristic One

*This heuristic rearranges the triples within the swappable region of the frozen array, so that further triples from the list  $L$  can be added to the new configuration.*

A random triple,  $t$ , is selected from a random row of the swappable region of the frozen array. A second row,  $r$ , is selected at random in which  $t$  can be placed, either without causing any violations, or just causing a single violation with a triple,  $t'$ , in the swappable region of row  $r$ . In the case that triple  $t$  causes no violations, it is simply added to row  $r$ . Otherwise, triple  $t'$  is removed from row  $r$  to accommodate the addition of triple  $t$ , and  $t'$  is

then cycled around the array by repeating this process. This continues until a new valid configuration of the array is constructed.

### Heuristic Two

*This heuristic attempts to place one of the unused triples from the list L into the frozen array, and then attempts to resolve the violations which this may have caused.*

One of the unused triples is selected at random from the list L, and placed into a row of the array such that exactly two triples from the row's swappable region are violated. One of the violated triples is removed from the row and added to the list L of unused triples, and the other violated triple is cycled around the rows of the array using the technique presented in Heuristic One.

In general, both heuristics improve the performance of the algorithm by allowing the placement of additional triples into the weakly derived array once it first becomes frozen. Unfortunately, often the array eventually reaches an even more tightly frozen configuration such that neither of the above heuristics are effective. The results of the algorithm are summarised next.

### Results

The performance of this placement search technique was very similar to that of the earlier repair search. Weakly derived arrays could efficiently be generated for random STS(7) and STS(9) designs, however as before, solutions to larger orders were not able to be constructed.

For order 13, each of the two heuristics described above were applied alternately once the array became frozen. The array was detected to be frozen following 100 iterations without a decrease in the size of the list, L. A particular run of the algorithm was abandoned once 1500 applications of the heuristics were performed without a decrease in the size of the list L.

The algorithm was then run 2000 times for a random base STS(13), and after each run of the algorithm, the number of triples still remaining in the list L was recorded. The table below summarises the maximum, minimum and average number of remaining triples once each run was abandoned, over the 2000 runs:

Order	Maximum number of triples still remaining in the list, L	Minimum number of triples still remaining in the list, L	Average number of triples still remaining in the list, L
13	31	14	22.6

### 2.3.5.3 NON-ISOMORPHIC BASE DESIGNS

In Chapter 4, the concept of isomorphisms and isomorphism classes will be described in much greater detail. It is sufficient at this point to realise that two Steiner triple systems are non-isomorphic if they have distinct underlying structures, regardless of the labelling of the points of the design. It is well known that there are exactly 2 non-isomorphic STS(13)'s and 80 non-isomorphic STS(15)'s. The purpose of this section is to determine whether the underlying structure of each base STS design has any impact on the performance of the algorithm for constructing its corresponding weakly derived array. This was achieved by analysing the performance of the algorithm when each of the 2 STS(13)'s and each of the 80 STS(15)'s were used as base designs.

#### The 2 Non-Isomorphic STS(13) Base Designs

For the STS(13) designs, the algorithm was run to termination 2000 times. For each base design, the maximum, minimum and average number of blocks still remaining in the list L on termination was recorded over the 2000 executions. The results are tabulated on the following page:

	<i>STS(13) Number 1</i>	<i>STS(13) Number 2</i>
Maximum	31	31
Minimum	14	15
Average	22.6	22.6

From the results of this experiment, it is apparent that the performance of the algorithm is independent of which of the 2 non-isomorphic STS(13) designs is used as the base design.

**The 80 Non-Isomorphic STS(15) Base Designs**

The same analysis was performed for each of the 80 non-isomorphic STS(15) designs. For each design, D, 50 complete runs to termination were executed using D as the base design for the algorithm. The results are summarised in Chart 2.3 below, which plots the maximum, minimum and average number of triples remaining in the list L, for each of the 80 STS(15) designs used as the base design:

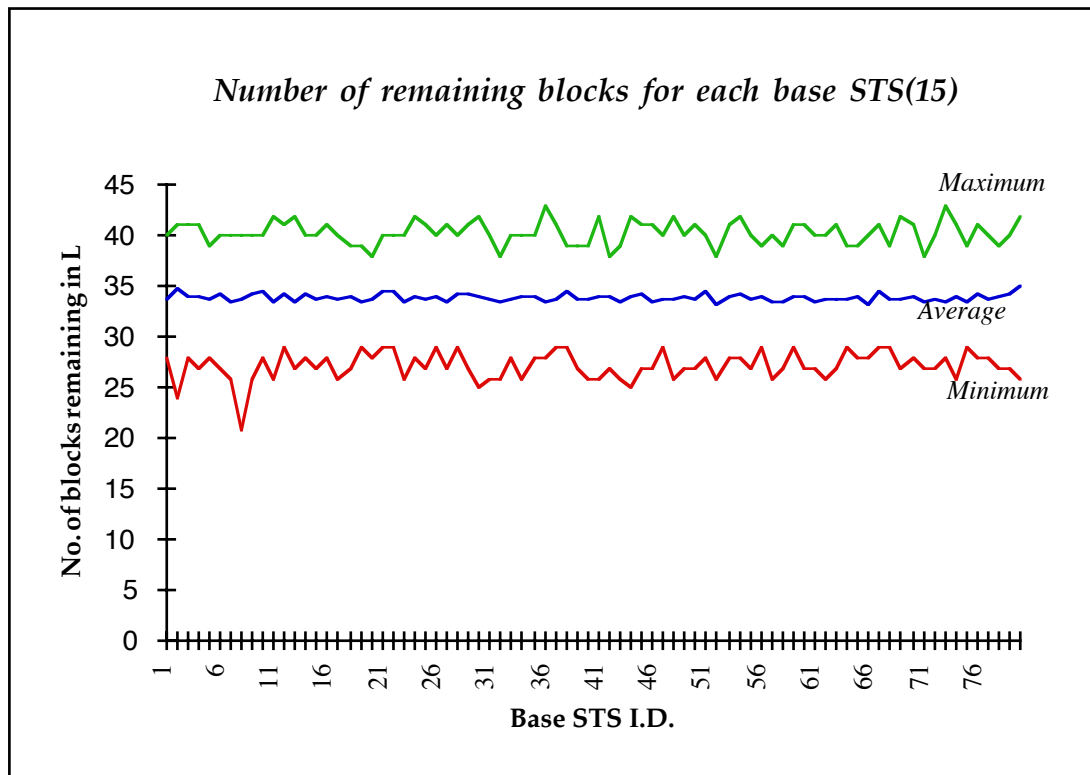


Chart 2.3

The identification of the 80 STS(15) designs, from 1 to 80, corresponds to the complete list of STS(15)'s published in [45]. As Chart 2.3 illustrates clearly, particularly with the average number of blocks remaining in L when the algorithm terminates, the performance of the algorithm is independent of which of the non-isomorphic STS designs are used as the base design.

**2.3.5.4 REVERSE ENGINEERING**

An interesting property of this problem can be illustrated by reverse engineering a solution, ie. a valid weakly derived STS(13) array, and examining the effectiveness of the presented algorithms for rebuilding the solution once a number of triples have been removed from the array and added to the list, L.

This section describes the construction of the weakly derived STS(13) directly from a Steiner quadruple system of order 14. A number of triples,  $T_b$ , are removed at random from

the array and placed into the list L. The algorithm is then executed to determine the values of  $T_b$  for which a valid weakly derived array can be successfully reconstructed.

### Constructing A Weakly Derived STS(13) From An SQS(14)

A weakly derived STS( $v$ ) array can be constructed directly from a Steiner quadruple system of order  $v+1$ , an SQS( $v+1$ ). An SQS has blocks of size 4, which satisfy the condition that every unordered triple occurs exactly once in the blocks of the design. No efficient probabilistic method is yet known for constructing random SQS's, and certainly if such a method was known the construction of weakly derived STS designs could clearly be performed using the same approach.

The SQS(14) constructed in this section was generated from the following set of five base blocks, published in [14]:

$\{0_0 1_0 2_0 4_0\}, \{0_0 1_1 2_1 4_1\}, \{3_0 4_0 3_1 4_1\}, \{4_0 5_0 2_1 3_1\}$  and  $\{5_0 6_0 1_1 2_1\}$

These base blocks are expanded by applying the following automorphisms:

$x_i \rightarrow (3x)_{i+1}$  followed by  $x_i \rightarrow (x+1)_i$  for each block formed using the first automorphism.

The constructed SQS(14) is listed below:

1,2,3,5	14,10,13,12	13,2,5,4	13,9,6,2	6,7,11,12	4,6,14,9	5,6,8,9
8,11,14,13	1,9,10,12	14,3,6,5	14,10,7,3	7,1,12,13	5,7,8,10	10,13,5,1
2,3,4,6	8,4,7,6	4,5,11,12	8,11,1,4	1,2,13,14	6,1,9,11	11,14,6,2
3,4,5,7	2,10,11,13	10,13,3,6	9,12,2,5	2,3,14,8	7,2,10,12	12,8,7,3
4,5,6,1	3,11,12,14	7,2,14,9	1,3,8,10	3,4,8,9	1,3,11,13	13,9,1,4
5,6,7,2	4,12,13,8	5,6,12,13	2,4,9,11	4,5,9,10	6,7,9,10	14,10,2,5
6,7,1,3	5,13,14,9	6,7,13,14	3,5,10,12	14,10,1,4	9,12,4,7	8,11,3,6
7,1,2,4	6,14,8,10	7,1,14,8	4,6,11,13	8,11,2,5	4,6,10,12	5,7,11,13
9,12,8,14	7,8,9,11	1,2,8,9	5,7,12,14	9,12,3,6	7,1,10,11	6,1,12,14
10,13,9,8	9,5,1,7	2,3,9,10	6,1,13,8	10,13,4,7	1,2,11,12	7,2,13,8
11,14,10,9	10,6,2,1	3,4,10,11	5,6,10,11	11,14,5,1	2,3,12,13	1,3,14,9
12,8,11,10	11,7,3,2	11,14,4,7	13,9,7,3	12,8,6,2	3,4,13,14	2,4,8,10
13,9,12,11	12,1,4,3	12,8,5,1	2,4,12,14	3,5,13,8	4,5,14,8	3,5,9,11

A derived STS( $v$ ) is one which has its blocks derived from an SQS( $v+1$ ), by taking all quadruples of the SQS which contain a certain point, and returning all  $\frac{v(v-1)}{6}$  triples resulting when that point is removed from those quadruples. A derived STS which is constructed in this way, is also weakly derived in that its triples can be arranged into the array, subject to the constraints previously discussed. This process is summarised below:

- Let  $T_j$  denote the set of all quadruples of the SQS( $v+1$ ) which contain the point  $j$ . The size of each set  $T_j$  is exactly  $\frac{v(v-1)}{6}$ , which is the number of blocks in an STS( $v$ ).
- For a given set  $T_j$ , replace all quadruples of the form  $(j, a, b, v+1)$  by the triple  $(a, b, j)$ , and place these triples into the first  $\frac{(v-1)}{2}$  columns of row  $j$  of the weakly derived array. For all other quadruples of the form  $(j, a, b, c)$ , where  $c \neq v+1$ , return the triple  $(a, b, c)$  and place these into the remaining  $\frac{(v-1)(v-3)}{6}$  positions of row  $j$ .
- Perform this process for sets  $T_j$  for all  $j \in \{1, 2, 3, \dots, v\}$ .

Using these steps, a valid weakly derived STS(13) array was generated from the SQS(14) given above.

### Reconstructing The Weakly Derived Array

A number of triples,  $T_b$ , were removed at random from this constructed weakly derived STS(13) array, and placed into the list of unused triples, L. The placement search hill-climbing technique described in Section 2.3.5.2 was then applied to reconstruct the design.

The algorithm was performed five times for each value of  $T_b$  ranging from 10 up to 120. The total number of triple replacements required to reconstruct the array were recorded. A triple replacement involves placing a triple from the list L into a row,  $r$ , of the array and returning the triple in row  $r$  which was violated, back to L. This number gives a rough estimate of how difficult the process of array reconstruction was for a given value of  $T_b$ . For the smaller values of  $T_b$ , zero block replacements were required. This means that each of the  $T_b$  triples removed at random from the array were returned to a row, without violation, on the first attempt at placing them. Results are summarised in the following table. The cases in

which the algorithm froze and the blocks could not be returned to the array, are represented by a dash (-) in the table.

$T_b$	Total number of block replacements necessary to reconstruct array				
	<i>Run 1</i>	<i>Run 2</i>	<i>Run 3</i>	<i>Run 4</i>	<i>Run 5</i>
10	0	0	0	0	0
20	0	0	2	0	0
30	4	6	2	4	10
40	4	2	15	8	2
50	38	7	19	8	43
60	24	40	8	59	34
70	58	90	64	64	88
80	69	256	99	83	121
90	84	221	247	89	10114
100	10794	130	-	200	-
110	27621	35017	5356	-	-
120	-	-	-	-	-

As demonstrated by these figures, well over 100 blocks could be removed at random from the weakly derived STS(13), while still allowing it to be easily reconstructed using the standard placement algorithm. Recall that, on average, around 22 blocks were still remaining when the hill-climbing algorithm, which constructed the array from a random base STS, became frozen. It is interesting to note that far in excess of 22 blocks could be removed from the completed structure so that the same hill-climbing algorithm could reconstruct the array.

Another interesting result is that for all the values of  $T_b$  in the table above, the reconstructed array is exactly the same as the original. This suggests that the constraints on the problem are indeed very tight, for even when 110 blocks are removed at random, the probabilistic hill-climbing algorithm reconstructed the array exactly to its original state.

This behaviour tends to suggest that there is some forbidden sub-configuration which is constructed early on when a weakly derived array is being built from a random base STS. The generation of such a sub-configuration prevents the valid construction of the weakly derived array. When a weakly derived STS(13) is constructed directly from an SQS(14), the valid array clearly does not contain such a forbidden sub-configuration, and the removal of any given number of blocks from the array does not introduce one either. As long as the number of blocks removed does not exceed some threshold which depends on the size of the array, the array can be reconstructed without difficulty. If too many blocks are removed from the original array, then it is not possible to reconstruct it, as too much of the original structure is lost. Once the structure is sufficiently damaged, the replacement of randomly selected triples to the rows of the array most likely introduces forbidden sub-configurations as before, and a valid weakly derived array cannot be completed.

### Conclusion

The two main approaches examined for this problem, repair search and placement search, performed very similarly at constructing weakly derived STS arrays. Both techniques, with suitable heuristic refinements, were able to generate weakly derived STS(7) and STS(9) configurations effectively. Neither technique was successful for any larger order, and the analysis of the reverse engineered experiment suggests the existence of some forbidden sub-configuration which is introduced early on into the construction of the array. A more thorough analysis of the types of forbidden sub-configurations may be required before a probabilistic approach is successful at generating these structures for such orders.

### 2.3.6 WEAKLY UNION FREE TWOFOLD TRIPLE SYSTEMS

This section presents a simple hill-climbing algorithm for the construction of twofold triple systems, and extends this with a constraint for the construction for weakly union free designs.

**Definition:**

A *twofold triple system* of order  $v$ , a  $TTS(v)$ , is a pair  $(X, B)$  where  $X$  is a set of  $v$  points, and  $B$  is a set of subsets of  $X$  of size 3, called blocks or triples, such that all unordered pairs of points are covered exactly twice by the blocks of the design.

A  $TTS(v)$  can therefore be considered as a simple extension to an  $STS(v)$ , where each unordered pair of points is covered twice, rather than once, by the blocks of the design. In fact, two copies of a given  $STS(v)$  can clearly be combined to form a valid  $TTS(v)$ .

**Definition:**

A twofold triple system is said to be *weakly union free* if whenever 4 distinct blocks  $B_1, B_2, B_3$  and  $B_4$  are chosen from  $B$ , it is not the case that  $B_1 \cup B_2 = B_3 \cup B_4$ . A weakly union free twofold triple system of order  $n$  is denoted as a  $wuf-TTS(n)$ .

In the paper by Chee, Colbourn and Ling [11], weakly union free twofold triple systems are studied from primarily a mathematical point of view. Their research also outlines an application for such designs to the group testing problem, having numerous real-world applications ranging from multiple access communications to DNA clone isolation. The main theorem of their paper is given below:

**Theorem:**

A  $wuf-TTS(n)$  exists whenever  $n \equiv 0,1 \pmod{3}$  except when  $n \in \{3,4,6,7,9,10\}$  and possibly when  $n \in \{12,15,18,22\}$ .

The aim of this section is to try and solve one of the unknown cases using a probabilistic algorithm such as hill-climbing. If a solution exists to one of the unknown cases, then an efficient non-exhaustive algorithm may be able to construct it. However, with any non-exhaustive approach, no conclusions can be drawn if a solution does not exist.

The problem was tackled in two stages. Firstly, a hill-climbing algorithm was developed to construct random twofold triple systems of a specified order, and this is covered in Section 2.3.6.1. In order to search for weakly union free designs, an extra constraint was then added to the algorithm to detect sets of four blocks violating the weakly union free conditions. This approach and an analysis of its effectiveness is detailed in Section 2.3.6.2.

#### 2.3.6.1 RANDOM TWOFOLD TRIPLE SYSTEM CONSTRUCTION

The hill-climbing algorithm of Stinson [63] for the construction of STSs, described in Section 2.3.4.1, can trivially be extended to construct TTSs. A given design is constructed block by block, and each block is formed a point at a time. Every given point must occur exactly twice with every other point in the design, and so a random point  $x$  is selected which has not yet occurred in exactly two blocks with every other point. There will then be two points,  $y$  and  $z$ , neither of which have occurred in two blocks with  $x$ , and these are selected at random from amongst the possibilities. The random block  $\{x, y, z\}$  is then constructed. The unordered pairs  $\{x, y\}$  and  $\{x, z\}$  are known to not have occurred in two blocks, and so the addition of this new block to the current partial design will not cause an excess of either of these pairs. However, the pair  $\{y, z\}$  may have already been covered in two blocks. If this is the case, one of these blocks is chosen at random and is replaced by  $\{x, y, z\}$ .

This algorithm performs very well. For example, a  $TTS(100)$ , consisting of 3300 blocks and containing each of the 9900 unordered pairs  $\{\{1,2\}, \{1,3\}, \dots, \{99,100\}\}$  in exactly 2 distinct

blocks, can be generated in less than a tenth of a second. Random TTS( $n$ ) designs, for  $n \in \{12, 15, 18 \text{ or } 22\}$ , can be constructed extremely quickly with this method.

An example of a random TTS(12) design, generated by the algorithm outlined above, is given below. Every unordered pair is contained in exactly two blocks. For example, the only two blocks containing the unordered pair  $\{1,2\}$  are highlighted:

8,6,3	11,3,10	11,12,4	7,11,10
8,11,7	11,8,1	7,12,3	5,10,8
9,6,11	1,4,7	8,2,12	6,5,8
7,5,12	10,4,2	2,6,12	<b>2,6,1</b>
1,8,9	11,4,1	10,4,8	4,6,3
2,10,9	3,11,2	11,12,6	1,12,10
1,3,10	1,9,12	6,10,9	9,12,3
5,2,11	6,7,4	4,8,12	9,3,7
1,5,3	2,5,7	3,8,2	9,11,5
5,6,1	4,5,3	8,7,9	12,10,5
9,5,4	<b>1,2,7</b>	2,9,4	7,10,6

### 2.3.6.2 WEAKLY UNION FREE DESIGN CONSTRUCTION

Due to the fact that random TTSs can be generated so quickly, the first approach taken to generating wuf-TTS designs involved simply testing randomly constructed designs for this property.

This approach proved unsuccessful however, because every randomly generated TTS turned out to contain a union violation. For example, in one experiment, 100,000 random TTS(12) designs were constructed, yet none were weakly union free. This is perhaps not surprising, as there are an enormous number of distinct TTS(12) designs which could be constructed by the hill climbing algorithm, most of which if not all of which may contain a union violation. A better approach to this problem is outlined next, in which the partial design for a valid wuf-TTS is constructed block by block, such that each partial design satisfies the weakly union free constraints.

The hill-climbing algorithm of Section 2.3.6.1 for the construction of random TTS( $v$ )'s was used as the basis for the construction of wuf-TTS( $v$ )'s, with the following modifications. Blocks are generated at random using the same schedule as before, however a given block,  $b$ , is only added to the partial design if  $b$  does not violate the weakly union free constraint when considered with any set of three blocks currently in the partial design. The process of adding such a block to the partial design is then exactly the same as before. Using this technique, at any stage of the algorithm the partial design is guaranteed to be weakly union free. Therefore, a valid wuf-TTS( $v$ ) will be constructed once the partial design is complete, ie. once it contains  $v(v-1)/3$  blocks. The only further implementation issue discussed here is the process of determining whether or not a given randomly generated block can be added to the partial design without violating the weakly union free constraint. This test is performed in constant-time by using a large 3-dimensional data structure which explicitly stores all the blocks which are forbidden and cannot be added to a given partial design without causing a weakly union free violation. This forbidden blocks structure, as it is known, is updated whenever a new block is added to the partial design by determining which blocks would cause a weakly union free violation when considered with a set of any 3 blocks currently in the partial design.

### Results

Unfortunately, no wuf-TTS designs were constructed with this method for any of the unknown orders. As random blocks were added to the partial solution, the number of forbidden blocks increased rapidly. Very quickly the algorithm settled into a state from which there were only a small number of possible blocks which could be added to the partial design without creating a weakly union free violation, yet none of these blocks were able to extend the current partial design.

For example, when trying to construct a wuf-TTS(12), the smallest order for which existence is not settled, very quickly the following partial design is constructed containing 33 of the required 44 blocks:

3, 7, 6	4, 9, 8	10, 7, 5
8, 11, 7	10, 4, 3	10, 2, 7
7, 6, 11	8, 3, 5	11, 1, 10
6, 12, 5	11, 12, 4	9, 2, 3
1, 6, 8	11, 12, 8	2, 3, 9
12, 10, 9	2, 4, 1	1, 12, 7
10, 3, 6	12, 4, 2	10, 5, 11
11, 6, 2	6, 9, 12	7, 4, 5
11, 3, 5	10, 4, 8	7, 1, 9
6, 10, 1	9, 8, 1	5, 2, 12
9, 5, 4	6, 2, 5	12, 3, 1

This partial solution is a valid partial wuf-TTS(12), so that no unordered pair is contained in more than two blocks, and the union of any two blocks does not equal the union of any other two blocks. There are exactly  $\binom{12}{3}=220$  distinct blocks of size 3 on 12 points, and of these 220 blocks, only 2 can be added to the partial solution above without generating a weakly union free violation. These two blocks are:  $\{2, 3, 9\}$  and  $\{4, 8, 11\}$ .

Any other block would create a violation if added to the partial design above. For example, consider the random block  $\{6, 8, 10\}$ , and take the following 3 blocks in the partial design above:

$\{3, 4, 10\}$ ,  $\{3, 6, 10\}$  and  $\{4, 8, 10\}$

These satisfy the equality:

$$\{3, 4, 10\} \cup \{6, 8, 10\} = \{3, 6, 10\} \cup \{4, 8, 10\}$$

and so  $\{6, 8, 10\}$  can not be added to the partial design without violating the weakly union free property.

An examination of these only two blocks which can be added to the partial design without causing a weakly union free violation follows:

#### 1) Block $\{2, 3, 9\}$

The block  $\{2, 3, 9\}$  is already contained twice in the current partial solution. For this reason, it can not be generated again by the algorithm, because the unordered pairs  $\{2, 3\}$ ,  $\{2, 9\}$  and  $\{3, 9\}$  are already covered by exactly two blocks.

#### 2) Block $\{4, 8, 11\}$

Again,  $\{4, 8, 11\}$  can not be generated as a random block, because both the pairs  $\{4, 8\}$  and  $\{8, 11\}$  are already covered twice by the following blocks of the current partial solution:  $\{4, 8, 9\}$ ,  $\{4, 8, 10\}$ ,  $\{7, 8, 11\}$ ,  $\{8, 11, 12\}$ .

It is quite clear that the algorithm is completely stuck at this stage. With the 33 blocks listed above making up the partial solution, any random block satisfying the twofold triple system constraint will be forbidden because it causes a weakly union free violation.



**Conclusions**

Although random TTSs of arbitrary orders could be constructed very quickly with a simple extension to Stinson's hill-climbing algorithm for the construction of STSs [63], no wuf-TTS was able to be constructed when the extra weakly union free constraint was included in the search.

A probabilistic algorithm is not an effective approach for a problem such as this, due to the enormous size of the search space and the tiny proportion of solutions within it. Making essentially random transitions around a very large search space is ineffective when solutions are particularly rare.

Chapters 3 and 4 detail the development of an exhaustive design construction algorithm, and its subsequent success on the wuf-TTS problem is presented in Case Study One of Chapter 6.

# Chapter 3

## Exhaustive Construction Of Incidence Structures

### 3.1 INTRODUCTION

Exhaustive search is a technique for constructing or examining all possible states within a given search space. The most well-known general exhaustive search strategy is that of backtracking, in which partial solutions are built up one step at a time in a systematic fashion guaranteeing complete coverage of the search space. In contrast, non-exhaustive search strategies, such as the probabilistic algorithms studied in the previous chapter, traverse the search space more or less at random and thus certain states may never be examined.

The fundamental differences between exhaustive and non-exhaustive strategies are apparent by considering the advantages of each approach: non-exhaustive search is generally faster, whereas exhaustive search is more thorough.

An exhaustive search strategy is usually required for any type of problem which poses questions such as "in how many ways can this be solved?", or "how many designs exist with a particular structure?". In many cases, the only way of answering such questions is to explicitly construct all the solutions. Exhaustive search can also be used to establish the non-existence of particular structures, by exhaustively searching the corresponding search space, without producing any valid designs. Good examples of such applications of exhaustive search are the celebrated work by Lam, Thiel and Swiercz [42] who establish the non-existence of finite projective planes of order 10, the work of McKay and Radziszowski [49] who show that no 4-(12,6,6) designs exist, and even the work presented in Case Study One of Chapter 6 of this thesis, in which it is shown that weakly union free twofold triple systems of order 12 do not exist.

The limited success of the non-exhaustive, probabilistic techniques in the previous chapter, particularly for problems with very large search spaces in which valid solutions are rare or non-existent, motivated the development of an exhaustive search algorithm for general incidence structures. The algorithm was based on the technique of backtracking and was developed in several stages, with extensive analysis and progressive refinements being made at each stage. The final version of the algorithm produced a number of new results in the field of combinatorial design theory, which are covered in their relevant sections of this thesis, and summarised in Chapter 7.

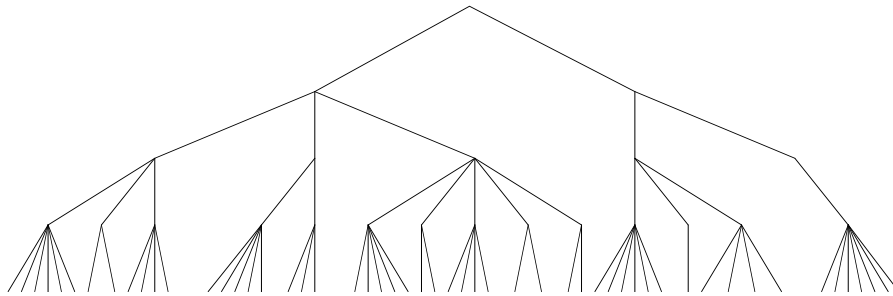
This chapter describes the underlying methods and some of the optimisation techniques of the incidence structure backtracking algorithm. In particular, the algorithm developed in this chapter is able to exhaustively generate all possible incidence structures of a given type, thus answering the question "how many distinct designs are there satisfying a particular set of properties?"

Section 3.2 presents an overview of the technique of backtracking that formalises its methods and outlines its limitations. This is followed by Section 3.3 which introduces incidence structures in general and presents the special type of incidence structures which are used as examples throughout the thesis, profiling their common representations.

## 3.2 BACKTRACKING

Backtracking is a general term representing a technique for performing exhaustive search. In addition to its practical usefulness as an exhaustive search strategy, a basic backtracking algorithm is usually fairly intuitive and so can be easily understood.

An important entity in the context of a backtracking algorithm is a *feasible solution*, which is a partially or completely constructed solution which does not violate any of the constraints of the corresponding problem. Backtracking constructs a series of feasible solutions by extending and collapsing partial feasible solutions one step at a time in a systematic fashion, until a complete feasible solution is constructed which then satisfies all the constraints of the problem being investigated. Thus, the execution of a backtracking algorithm can be viewed as a depth first traversal of a search tree, in which the nodes of the tree correspond to partial feasible solutions and the branches represent the systematic extension and backtracking operations from which one feasible solution can be constructed from another. A typical search tree may look like the one below:



The extension of a partial feasible solution corresponds to a traversal of the search tree to the next deepest level, whereas backtracking is represented by an upward traversal of a branch in the tree. The leaves at the deepest level of the tree correspond to complete feasible solutions which satisfy all the constraints of the given problem and these are generally the nodes of interest. However, traversing the search tree to find these nodes is often difficult, because the tree can become extremely large, even for moderately sized problems. In general, the size of the search tree grows exponentially as the order of the corresponding problems increases, and this is the major drawback of an exhaustive search strategy such as backtracking.

Although backtracking algorithms do in general exhibit exponential behaviour as the problem sizes grow, the use of intelligent pruning techniques, such as rejecting partial feasible solutions which either cannot extend to valid solutions or are equivalent in some way to feasible solutions already considered, can significantly reduce the required execution time of such an algorithm. Just as the performance of a basic backtracking algorithm is exponential, defining intelligent pruning heuristics to speed up the search can result in an exponential saving in the required running time. The implementation of such heuristics to speed up the backtracking algorithm developed in this thesis is central to this chapter and to Chapter 4.

Backtracking can be formulated by considering a solution to a problem as a vector of length  $n$ , of the form  $(a_1, a_2, a_3, \dots, a_n)$ , which satisfies the constraints of the problem. In the most general case this vector may be of variable length, but combinatorial designs are often highly structured and so a fixed length vector is usually sufficient. Each element  $a_i$  of the solution vector is a member of a finite, linearly ordered set  $A_i$ . Thus the exhaustive search must consider the elements of the product space  $A^n = A_1 \times A_2 \times A_3 \times \dots \times A_n$  as potential solutions.

Initially, backtracking starts with the null vector as the partial feasible solution and the constraints of the problem specify, at each level  $i$ , which members of the set  $A_i$  need to be considered as candidates for the vector at position  $i$ . More formally, for any  $k \in \{1, 2, 3, \dots, n\}$ , a boolean valued feasibility property  $F_k : A_1 \times A_2 \times \dots \times A_k \rightarrow \{true, false\}$  is defined. The feasibility property of any complete vector of length  $n$  which is a solution to the corresponding problem, must evaluate to *true*.

The important attribute that a feasibility property must possess to be effective is that for any partial feasible solution  $(a_1, a_2, \dots, a_k)$ ,  $F_k(a_1, a_2, \dots, a_k) = true \Rightarrow F_j(a_1, a_2, \dots, a_j) = true$ , for all  $1 \leq j < k$ . With such a feasibility property, if the partial solution  $(a_1, a_2, \dots, a_k)$ ,  $1 \leq k < n$ , has been constructed and the feasibility property  $F_k(a_1, a_2, \dots, a_k) = false$ , then it is known for any extension  $(a_1, a_2, \dots, a_k, \dots, a_h)$  of the partial feasible solution,  $F_h(a_1, a_2, \dots, a_k, \dots, a_h)$  will also evaluate to *false*. In other words, no extension of the partial solution  $(a_1, a_2, \dots, a_k)$  can possibly form a valid complete solution. As soon as this situation is detected, it is then possible to eliminate from consideration  $|A_{k+1}| \times |A_{k+2}| \times \dots \times |A_n|$  infeasible solutions.

The most effective feasibility properties are ones which often evaluate to *false*, ie. of the form  $F_k(a_1, a_2, \dots, a_k) = false$ , particularly for small values of  $k$ , as these eliminate enormous numbers of partial solutions from consideration.

When applied to combinatorial design construction, backtracking algorithms are generally used to solve either existence questions or enumeration problems. With an existence question, the task is to find just one solution  $(a_1, a_2, \dots, a_n)$  such that  $F_n(a_1, a_2, \dots, a_n) = true$ . With an enumeration problem, the task is to find all such solutions  $(a_1, a_2, \dots, a_n)$ .

The general incidence structure backtracking algorithm developed in this thesis has been applied to a wide range of both existence and enumeration problems, and a summary of the new results obtained with it are given in Chapter 7. In theory, a backtracking algorithm can easily be implemented to solve literally any sort of design construction or enumeration problem. In practice, the exponential behaviour of backtracking considerably restricts the sizes of the problems to which it can be applied.

### 3.3 INCIDENCE STRUCTURES

An *incidence structure* is a pair  $D = (V, B)$  where  $V$  is a finite set of elements or points  $\{p_1, p_2, p_3, \dots, p_v\}$  and  $B$  is a collection of not necessarily distinct subsets of  $V$ ,  $\{b_1, b_2, b_3, \dots, b_b\}$  called blocks. There is an incidence relation,  $\Psi$ , between the points and blocks of the structure such that for any given block there is a particular set of points incident with it. Graphs, configurations, and classical geometries such as linear spaces and affine and projective planes are all examples of incidence structures.

Block designs, or  $t$ -designs, are one of the most well known types of incidence structures and have their roots in statistical design theory [37]. The subset of block designs for which  $t=2$ , ie. the 2-designs, are better known as balanced incomplete block designs or BIBDs, and are defined in Section 3.3.1. For the remainder of this chapter, and for Chapters 4 and 5, the development of the incidence matrix backtracking algorithm is described primarily in the context of BIBDs. The examples and specific design constraints correspond almost exclusively to BIBDs, so that the development of the algorithms are presented in a uniform manner. It must be noted however that the algorithms are not restricted to block designs alone, and Chapter 6 presents a series of case studies in which a wide variety of incidence structures are investigated.

There are a number of reasons for choosing BIBDs as the example class of designs. The primary reason is that they are very well-known, and in fact special types of BIBDs such as the Steiner triple systems and the twofold triple systems have already been introduced in Chapter 2 of this thesis. BIBDs are also the underlying structures of many other interesting designs, such as the weakly union free twofold triple systems, which were introduced in

Section 2.3.6, and the Mendelsohn triple systems which are presented in Case Study Five of Chapter 6.

### 3.3.1 BALANCED INCOMPLETE BLOCK DESIGNS

**Definition:**

A  $t$ - $(v,k,\lambda)$  design is a pair  $(X,B)$  where  $X$  is a  $v$ -element set of points and  $B$  is a collection of  $k$ -subsets of  $X$  called blocks, with the property that every  $t$ -element subset of  $X$  is contained in exactly  $\lambda$  blocks.

A *balanced incomplete block design*, or BIBD, is a  $2$ - $(v,k,\lambda)$  design, and so every unordered pair of points from the set  $\{1,2,3,\dots,v\}$  must occur in exactly  $\lambda$  blocks of the design. A BIBD is defined by 5 parameters,  $v$ ,  $b$ ,  $r$ ,  $k$  and  $\lambda$  - only 3 of which are independent. The 3 parameters which are usually used to define a given BIBD are  $v$ ,  $k$  and  $\lambda$ . From these the other two parameters can be calculated as:

$$r = \frac{\lambda(v-1)}{k-1} \quad \text{and} \quad b = \frac{vr}{k}.$$

Trivial necessary conditions for the existence of a BIBD are:

- 1)  $vr = bk$ , and
- 2)  $r(k-1) = \lambda(v-1)$

A set of parameters which satisfies these conditions is said to be admissible, however the conditions are not sufficient and so BIBDs do not exist for all admissible parameter sets. The parameters for a given BIBD, and the information they represent, are summarised below:

Parameter	Represents
$v$	The number of points in the design
$b$	The number of blocks in the design
$r$	The replication number. The number of distinct blocks in which each point must occur
$k$	The block size
$\lambda$	The number of blocks in which each unordered pair of points must occur

### 3.3.2 REPRESENTATIONS

Block designs are generally represented as either block lists or incidence matrices.

**Block Lists**

The block list of a given block design,  $D$ , explicitly lists the contents of each block of  $D$ . The block list of a  $2$ - $(6,3,2)$  design is given below:

1	2	3
1	2	4
1	3	5
1	4	6
1	5	6
2	3	6
2	4	5
2	5	6
3	4	5
3	4	6

### IncidenceMatrices

The incidence matrix of a  $t$ -( $v,k,\lambda$ ) block design consisting of  $b$  blocks is a  $v \times b$  matrix  $A = (a_{ij})$  such that if a 1 exists in the  $r^{\text{th}}$  row and the  $c^{\text{th}}$  column, ie. if  $a_{rc} = 1$ , then point  $r$  is incident to block  $c$  in the design. All other entries of the array are zero. The rows and columns of the matrix correspond to the points and blocks of the design, respectively. The incidence matrix of the 2-(6,3,2) design given as a block list above is the following:

```

1111100000
1100011100
1010010011
0101001011
0010101110
0001110101

```

The Steiner triple systems, which were constructed in Section 2.3.4, have block size 3, and satisfy the requirement that every unordered pair of points is covered by exactly one block. An STS is therefore a 2-( $v,3,1$ ) design. The twofold triple systems, which were also studied in Chapter 2, are 2-( $v,3,2$ ) designs as every unordered pair of points is contained in exactly two blocks.

## 3.4 BALANCED INCOMPLETE BLOCK DESIGN CONSTRUCTION

The construction of BIBDs can be implemented in a number of ways, and three different techniques are studied in this chapter. The first is a brute force approach, which systematically examines all possible block combinations. The reason for studying this approach is to introduce some of the important ideas of design construction in an uncomplicated context, and also to highlight the limitations of such approaches and to provide some comparisons for the other techniques studied.

The other two techniques are both backtracking approaches. The first of these is a block by block backtracking method, which in a natural way can be thought of as constructing the block lists of the corresponding designs. The other is a point by point backtracking method, which is most naturally considered as a construction of the incidence matrices of the designs. As a specific example, each of the three methods will be described and implemented to generate all the twofold triple systems, or 2-( $v,3,2$ ) designs, for small orders. The reason for constructing such designs may be to help plan a statistical experiment, or they may be used as the base designs in some other construction technique.

Before the three exhaustive methods are presented, an important observation should be made regarding the ordering of the points within the blocks and the ordering of the blocks of the design to be constructed. For example, consider the block list of the following random TTS(6) design which was constructed by the probabilistic hill-climbing algorithm given in Section 2.3.6.1 of Chapter 2:

Block 1:	6	5	4
Block 2:	6	2	1
Block 3:	4	6	1
Block 4:	5	2	1
Block 5:	4	3	2
Block 6:	2	5	4
Block 7:	2	3	6
Block 8:	3	5	6
Block 9:	3	5	1
Block 10:	3	1	4

The design above was constructed a block at a time, by adding randomly generated blocks to a partial feasible solution. The blocks are listed in the random order in which they were constructed, as are the points within the blocks. Recall from the definition that a block design is a collection of  $k$ -subsets of the point set, thus neither the blocks nor the points within each block are in any particular order. As a consequence, any other reordering of the

blocks or of the points within the blocks of the design above will represent exactly the same design. It is therefore desirable to define some sort of ordered form for a given design. Any block design which has the points within its blocks lexicographically ordered and has the blocks themselves in lexicographical order, is said to be *completely ordered*. Given this complete ordering, any design constructed can unambiguously be classified as being less than, greater than or equal to any other given design. This is important because it allows the adopted exhaustive search strategy to have a well defined direction - all designs are constructed in either a strictly increasing or strictly decreasing lexicographical order. This offers the particular advantage that no design will be constructed twice, as any given design has a unique completely ordered form which is only constructed once in the algorithm. It also significantly reduces the size of the search space to be examined, because all blocks which are merely combinations of the same points have exactly the same ordered representation. For example, each one of the following blocks:

123 132 213 231 312 321

has exactly the same ordered form representation, namely 123, and so the number of distinct block types which need to be considered is reduced by a factor of  $k!$  by imposing such an ordering.

The example TTS(6) given previously is displayed below, along with its corresponding completely ordered form:

Original random TTS(6)		Completely Ordered TTS(6)	
Block 1:	6 5 4	Block 1:	1 2 5
Block 2:	6 2 1	Block 2:	1 2 6
Block 3:	4 6 1	Block 3:	1 3 4
Block 4:	5 2 1	Block 4:	1 3 5
Block 5:	4 3 2	Block 5:	1 4 6
Block 6:	2 5 4	Block 6:	2 3 4
Block 7:	2 3 6	Block 7:	2 3 6
Block 8:	3 5 6	Block 8:	2 4 5
Block 9:	3 5 1	Block 9:	3 5 6
Block 10:	3 1 4	Block 10:	4 5 6

The exhaustive search strategies presented in the next three sections, all generate the TTS(6) designs as an example. For any design on  $v$  points with block size  $k$ , there are exactly  $\binom{v}{k}$  distinct block types, when the points within each block are ordered. For the TTS(6) designs, there are exactly  $\binom{6}{3} = 20$  distinct block types, which are given in lexicographical order by column, in the table below:

1 2 3	1 3 4	1 4 6	2 3 6	3 4 5
1 2 4	1 3 5	1 5 6	2 4 5	3 4 6
1 2 5	1 3 6	2 3 4	2 4 6	3 5 6
1 2 6	1 4 5	2 3 5	2 5 6	4 5 6

A valid TTS(6) consists of any 10 of these 20 distinct blocks which satisfy the constraint that every unordered pair of points occurs exactly twice. For any collection of 10 blocks which satisfies this constraint, it will also be the case that every distinct point occurs in exactly 5 of the blocks, as this is the replication number of the TTS(6)'s. The three exhaustive strategies for generating all completely ordered designs of a particular type are presented in the following three sections.

### 3.4.1 BRUTE FORCE CONSTRUCTION

A simple minded, brute force approach to constructing all possible TTS(6) designs would be to generate all possible collections of 10 blocks from the set of 20 distinct blocks given above,

and to test each one to see whether it satisfies the twofold triple system constraints. It is a simple matter to calculate exactly how many such collections would need to be tested. Each collection of 10 blocks is a multiset (a set which may contain repeated elements), selected from the set of 20 possible block types. The number of  $k$ -multisets which can be made from an  $n$ -set,  $R(n,k)$ , is given by the formula:

$$R(n,k) = \binom{n+k-1}{k}$$

Thus the number of collections of 10 blocks from the set of 20 block types is  $\binom{29}{10} = 20,030,010$ . Using a reasonably fast algorithm which could generate and test 100,000 10-block collections a second, every possibility could be constructed and tested in a little over 3 minutes.

The brute force approach described above is perfectly valid, however it is not practical for larger problems and this can be illustrated by performing a similar analysis for the twofold triple systems of orders 7 and 9. These are the next two admissible orders for the TTS designs, as TTS(8)'s do not exist because the replication number,  $r = \frac{\lambda(v-1)}{k-1}$ , is not an integer. For each order 6, 7 and 9, the table below summarises the total number of block collections which would be generated by the brute force algorithm.

Order ( <b>n</b> )	Number of Distinct Block Types ( <b>D<sub>b</sub></b> )	Number of Blocks in TTS( <b>n</b> ) ( <b>B</b> )	Total number of collections of B blocks from the set of D <sub>b</sub> distinct blocks = $\binom{D_b+B-1}{B}$
6	$\binom{6}{3} = 20$	10	$\binom{29}{10} = 20,030,010$
7	$\binom{7}{3} = 35$	14	$\binom{48}{14} = 482,320,623,240$
9	$\binom{9}{3} = 84$	24	$\binom{107}{24} = 501,030,714,391,939,000,000,000$

The enormous numbers above show that the brute force approach is not practical for constructing the TTS(7) or TTS(9) designs, and this fact can be highlighted by estimating the required running time for each search, assuming as before that 100,000 B-block collections can be generated and tested every second. At this rate, it would take a little under 56 days to generate and test all the TTS(7) designs. The TTS(9) designs could not be exhaustively generated with this method. Even at a rate of 100,000 generations and tests every second, the algorithm would run for well over 1 and a half billion centuries.

Although the brute force generate-and-test algorithm described in this section is not a backtracking algorithm, it still serves as a good illustration of the general limitation of exhaustive search techniques. As the order of most problems increases, the size of the corresponding search spaces increases exponentially. As a result, the enumeration of a particular class of combinatorial designs may be performed comfortably for order  $v$ , yet the search space for order  $v+1$  may be too large to search exhaustively.

Sections 3.4.2 and 3.4.3 of this chapter present the two backtracking approaches to the design construction problem. The remainder of this section briefly describes why these backtracking algorithms will outperform the brute force method described in this section. The main reason is that an enormous proportion of the processing performed by the brute force strategy is wasted effort. This is demonstrated by examining exactly how many distinct TTS(6), TTS(7) and TTS(9) designs exist compared to the number of block collections which are generated and tested by the brute force approach. This information is given in the table on the following page.



Order(n)	Exact number of distinct TTS(n) designs	Total number of block collections generated and tested by brute force approach
6	12	20,030,010
7	465	482,320,623,240
9	4,409,916	501,030,714,391,939,000,000,000

Clearly, a huge proportion of the designs generated and tested by the brute force approach are simply invalid designs. A backtracking approach exhibits much greater efficiency than this, because rather than building a series of complete designs and testing each one for validity, it constructs the designs by adding constituent pieces, either points or blocks, to a partial design. A partial design is only extended with a point or a block if it is valid, and does not violate any of the constraints of the design. As soon as a partial design is detected to be invalid, backtracking takes place, and the next partial design in sequence is constructed. All extensions generated to an invalid partial design would be invalid themselves, and thus the only complete designs constructed are in fact valid designs.

Possibly the most natural type of backtracking algorithm for the construction of block designs is one in which the designs are built block by block. This approach is presented in the next section.

### 3.4.2 BLOCK BY BLOCK BACKTRACKING CONSTRUCTION

In a block-by-block backtracking algorithm, each design is constructed by systematically adding a block at a time to a partial design. A partial design is a collection of blocks, taken from the set of distinct blocks, which represent the design being constructed and which do not violate any of the constraints of the problem.

The only constraint which needs to be satisfied when constructing a TTS(6) design is that all unordered pairs of points need to be covered twice. Any design which satisfies this condition is valid, and to enumerate all the distinct TTS(6) designs, it is necessary to produce the set of all designs which satisfy this condition.

Backtracking was formulated previously in Section 3.2. The main entity described in that section was the solution vector  $(a_1, a_2, \dots, a_n)$  which initially starts off as the null vector and is extended and backtracked at each level  $i$  by having each element  $a_i$  of the linearly ordered set  $A_i$ , systematically added to it. The other important entity was a feasibility property which was used to prune from the search tree those partial solution vectors which could not be extended to valid solutions.

In terms of the TTS(6) problem, the solution vector is the 10-element vector  $(a_1, a_2, \dots, a_{10})$ , where each element of the solution vector corresponds to a block of the design. The linearly ordered sets  $A_i$ , for all  $1 \leq i \leq 10$ , consist of the complete set of 20 distinct block types in lexicographical order. Each element of the solution vector is one of the blocks from these sets, and the elements themselves are arranged in lexicographical order within the vector. The feasibility property is defined as  $F_k(a_1, a_2, \dots, a_k) = \text{true}$ , if no unordered pair of points is covered more than twice by the set of the first  $k$  blocks in the partial solution vector, and *false* otherwise.

The advantages of backtracking in this way over the brute force approach become immediately clear. The first complete design considered by the brute force approach is a collection of 10 blocks of type  $\{123\}$ , because this is the lexicographically smallest multiset which can be made from the set of distinct blocks. This violates the constraints of the TTS(6) design, because each pair  $\{12\}$ ,  $\{13\}$  and  $\{23\}$  are covered 10 times, rather than twice.

With block by block backtracking, the lexicographically smallest block,  $\{123\}$ , is added to the partial solution. At this point the feasibility property evaluates to *true* because no unordered pair is covered more than twice. The block  $\{123\}$  is again added to the partial solution, and once again the feasibility property evaluates to *true*, because each of the

unordered pairs  $\{12\}$ ,  $\{13\}$  and  $\{23\}$  are covered exactly twice. When, for the third time, the block  $\{123\}$  is added to the partial solution, the feasibility property evaluates to *false*, indicating that the unordered pairs of points have been covered too many times and this last block addition is rejected.

The next block in lexicographical order,  $\{124\}$ , is then added to the partial solution, but because the unordered pairs  $\{12\}$ ,  $\{13\}$  and  $\{23\}$  are already covered twice, any block containing these pairs will be rejected. Therefore the next block which can be added to the partial solution is  $\{145\}$ .

In summary, the first 3 valid partial solutions constructed by the block by block backtracking algorithm are:

```
123
123  123
123  123  145
```

Only 7 blocks are considered by the backtracking algorithm in order to extend the partial solution to include the block  $\{145\}$ . As a comparison, the brute force approach would have generated:

```
all  $\binom{26}{7} = 657800$  distinct extensions to 123  123  123 ...
then all  $\binom{25}{7} = 480700$  distinct extensions to 123  123  124 ...
then all  $\binom{24}{7} = 346104$  distinct extensions to 123  123  125 ...
then all  $\binom{23}{7} = 245157$  distinct extensions to 123  123  126 ...
then all  $\binom{22}{7} = 170544$  distinct extensions to 123  123  134 ...
then all  $\binom{21}{7} = 116280$  distinct extensions to 123  123  135 ...
then all  $\binom{20}{7} = 77520$  distinct extensions to 123  123  136 ...
```

for a total of 2,094,105 distinct block collections, before the partial solution containing the block  $\{145\}$  would be considered. Each of these block collections certainly violate the constraints of the twofold triple system, because in each case either the pair  $\{12\}$  or the pair  $\{13\}$  is covered more than twice.

### 3.4.2.1 ALGORITHM

The basic structure for a backtracking algorithm to construct block designs is the following:

```
repeat
  while (partial solution is valid)
    Extend partial solution
  while (partial solution is invalid)
    Backtrack partial solution
until all possibilities exhausted
```

Above, extending and backtracking the partial solution correspond to adding the next block in lexicographical order to the partial solution, and removing the last block added to the partial solution, respectively.

A brief pseudo-code outline of the algorithm implemented in this section to perform block by block backtracking is presented overleaf. The feasibility property is used to test the validity of the current partial solution. A partial solution is invalid if it contains more than  $\lambda$  unordered pairs of points, otherwise it is valid:

```

partial solution = { }
Db = set of all  $\binom{v}{k}$  distinct block types
Blast = the lexicographically largest block in the set Db
repeat forever begin
  while (partial solution is valid according to feasibility property) begin
    EXTEND LOOP:
    if (partial solution is complete)
      print("Design constructed")
      set feasibility property to invalid, to enter backtrack loop
    else
      add next block from Db in lexicographical order to partial solution
    end
  while (partial solution is invalid according to feasibility property) begin
    BACKTRACK LOOP:
    while (the last block, Lb, in the partial solution is Blast) begin
      remove Lb
      if (partial solution is empty)
        print("All designs constructed")
        halt
      end
      remove the last block, Lb, from the partial solution
      in Lb's place, add the next block in lexicographical order
    end
  end

```

This basic algorithm would continue building block designs in lexicographical order until all possibilities were exhausted. In other words, it would exhaustively enumerate all distinct design of the specified type. Also, because the searching is exhaustive, the non-existence of a particular class of designs is indicated if the algorithm completes without generating any designs.

This algorithm was used to enumerate the TTS(6) designs. The set of 20 distinct block types, as given in the table below, were initialised and the algorithm was run until each design had been constructed. The following 12 distinct designs, listed in increasing lexicographical order, were generated:

#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1 2 3	1 2 3	1 2 3	1 2 3	1 2 3	1 2 3	1 2 4	1 2 4	1 2 4	1 2 4	1 2 5	1 2 5
1 2 4	1 2 4	1 2 5	1 2 5	1 2 6	1 2 6	1 2 5	1 2 5	1 2 6	1 2 6	1 2 6	1 2 6
1 3 5	1 3 6	1 3 4	1 3 6	1 3 4	1 3 5	1 3 4	1 3 5	1 3 4	1 3 5	1 3 4	1 3 4
1 4 6	1 4 5	1 4 6	1 4 5	1 4 5	1 4 5	1 3 6	1 3 6	1 3 5	1 3 6	1 3 5	1 3 6
1 5 6	1 5 6	1 5 6	1 4 6	1 5 6	1 4 6	1 5 6	1 4 6	1 5 6	1 4 5	1 4 6	1 4 5
2 3 6	2 3 5	2 3 6	2 3 4	2 3 5	2 3 4	2 3 5	2 3 4	2 3 5	2 3 4	2 3 4	2 3 4
2 4 5	2 4 6	2 4 5	2 4 6	2 4 5	2 4 5	2 3 6	2 3 6	2 3 6	2 3 5	2 3 6	2 3 5
2 5 6	2 5 6	2 4 6	2 5 6	2 4 6	2 5 6	2 4 6	2 5 6	2 4 5	2 5 6	2 4 5	2 4 6
3 4 5	3 4 5	3 4 5	3 4 5	3 4 6	3 4 6	3 4 5	3 4 5	3 4 6	3 4 6	3 5 6	3 5 6
3 4 6	3 4 6	3 5 6	3 5 6	3 5 6	3 5 6	4 5 6	4 5 6	4 5 6	4 5 6	4 5 6	4 5 6

Any TTS(6) design, on the point set  $\{1,2,\dots,6\}$ , constructed by any technique, must be one of the 12 designs given above. For example, consider the random TTS(6) design given in Section 3.4, generated with a hill-climbing algorithm. This is given again at the top of the next page, along with its completely ordered form:

<i>Original random TTS(6)</i>		<i>Completely Ordered TTS(6)</i>	
Block 1:	6 5 4	Block 1:	1 2 5
Block 2:	6 2 1	Block 2:	1 2 6
Block 3:	4 6 1	Block 3:	1 3 4
Block 4:	5 2 1	Block 4:	1 3 5
Block 5:	4 3 2	Block 5:	1 4 6
Block 6:	2 5 4	Block 6:	2 3 4
Block 7:	2 3 6	Block 7:	2 3 6
Block 8:	3 5 6	Block 8:	2 4 5
Block 9:	3 5 1	Block 9:	3 5 6
Block 10:	3 1 4	Block 10:	4 5 6

This random TTS(6) is in fact TTS(6) #11 in the lexicographically ordered table. Statistics for the constructive enumeration of all 12 distinct TTS(6) designs using this block by block backtracking algorithm are summarised below:

Type of Loop	Number of times the loop was iterated
Extend loop	43512
Backtrackloop	91755

The execution time for the complete enumeration was 0.13 seconds.

### 3.4.2.2 OPTIMISATIONS

There are several optimisations which can be made to this algorithm, by incorporating more of the design constraints. The only design constraint currently exploited in the feasibility property is that every pair of points must occur in exactly 2 blocks. Backtracking is only performed on those partial solutions for which an unordered pair is covered more than two times.

#### PointCountConstraint

The replication number,  $r$ , of a block design is another constraint which can be used to significantly prune the search tree. In a valid design, every point must occur exactly  $r$  times.

Blocks are added to the partial solution in lexicographical order, and the points within each block are also ordered. For any ordered block of the form  $\{a, b, c\}$ , the lexicographically smallest point within this block is  $a$ . In addition, any ordered block which is lexicographically larger than  $\{a, b, c\}$  can not contain any points less than  $a$ .

This means that if a block of the form  $\{a, b, c\}$  is added to the partial solution, yet there is a point,  $p$ , lexicographically less than  $a$  which has not occurred in  $r$  blocks, no extension to the partial solution can possibly be a valid design because the point  $p$  will not occur  $r$  times. A good example of this is the following partial solution, which consists of 5 of the 10 required blocks for a TTS(6) design:

1 2 3
1 2 3
1 4 5
1 4 5
2 4 6
...

The pairs  $\{1, 2\}$ ,  $\{1, 3\}$ ,  $\{2, 3\}$ ,  $\{1, 4\}$ ,  $\{1, 5\}$ ,  $\{4, 5\}$  are all covered exactly twice. The pairs  $\{2, 4\}$ ,  $\{2, 6\}$  and  $\{4, 6\}$  are all covered once, and no other pair is covered at all. This partial design is perfectly valid according to the feasibility property which checks that no pair is covered more than twice. Using only this feasibility test, no violation is

detected and the above partial solution would be extended. However, searching the extensions of the above partial design is pointless, because no extension can possibly lead to a valid design. This is because the point 1 is only covered 4 times, yet the replication number for a TTS(6) is 5. Blocks are added in strictly increasing lexicographical order and so no block added to and extending the partial solution above can contain the point 1, so the replication number constraint will always be violated.

To prune the search tree of such profitless branches, the following simple feasibility property can be used. Whenever a block of the form  $\{a, b, c\}$  is added to the partial solution, no extension to the partial solution can add any more points which are lexicographically less than  $a$ . Therefore, if any point less than  $a$  has not yet occurred in  $r$  blocks, backtracking can begin immediately.

This optimisation considerably improves the performance of the algorithm, because many small partial solutions consisting of only a few blocks are rejected. Such rejections early on in the search tree are very effective because they prune a large number of branches at the deeper levels of the tree which otherwise would have unnecessarily been searched.

The practical advantages of this optimisation are well illustrated by the results of another generation of the 12 distinct TTS(6) designs:

	TTS(6) generation without the point count constraint	TTS(6) generation with the point count constraint
# Extend loop iterations	43512	664
# Backtrack loop iterations	91755	5781
Execution time (secs)	0.131	0.0066

For this search, the number of extend and backtrack operations have decreased considerably - particularly the number of extend operations - because the point count constraint detects violations and performs backtracking on partial solutions which without the constraint would be extended. As a direct consequence of the reduced amount of searching performed, the algorithm ran approximately 20 times faster with the point count constraint.

For larger searches, the savings offered by this point count constraint are much greater. In fact, as will be seen later, counting the distinct TTS(9) designs would be infeasible without this constraint, yet with it all the designs are generated in just over two hours.

### Pair Count Constraint

In a similar fashion to the point count constraint, another feasibility property can easily be defined which counts the pairs covered in a given partial solution. For any BIBD, every unordered pair of points must be covered by exactly  $\lambda$  blocks of the design. For the twofold triple systems which are being examined in this section,  $\lambda = 2$ .

The idea behind the pair count constraint for the twofold triple system construction is as follows. No partial solution should be extended if there exists an unordered pair which is covered by less than two blocks, if that pair cannot be covered by any of the remaining block types which are to be considered following the extension.

For example, say the block  $\{a, b, c\}$  is added to the current partial solution. The lexicographically smallest unordered pair of points covered by this block is  $\{a, b\}$ . Every other block considered as an extension to this partial solution cannot be lexicographically smaller than  $\{a, b, c\}$  because the blocks are added in increasing lexicographical order. This also means that no pair lexicographically smaller than  $\{a, b\}$  can be added to the partial solution by any extension. Therefore, whenever a block  $\{a, b, c\}$  is added, it should be checked that all pairs lexicographically less than  $\{a, b\}$  are already covered exactly twice. If there exists a pair which is not covered twice, then the new extension can not be accepted, and backtracking can begin immediately.

An example of such a situation in the TTS(6) search is the partial solution given on the next page, which has just had the block  $\{2,4,5\}$  added to it:

1 2 3
1 2 5
1 3 5
1 4 6
1 4 6
2 4 5
...

The pair  $\{2, 3\}$  has only been covered once by the first block, and as soon as the block  $\{2, 4, 5\}$  is added,  $\{2, 3\}$  can no longer be covered by any extension. Without the pair count constraint, no violation can be detected and the next block in lexicographical order will be considered to extend the partial solution. However, with the pair count constraint, backtracking can take place immediately and the block  $\{2, 4, 5\}$  will be removed from the partial solution.

The size of the TTS(6) search tree, already considerably reduced by the point count constraint, more than halves when this pair count constraint is also used. For larger designs, the combination of these two constraints is extremely effective.

### 3.4.2.3 RESULTS

This section summarises the results of the block by block backtracking algorithm when applied to the exhaustive construction of the TTS(v) designs.

#### The 12 Distinct TTS(6) Designs

The table below compares the effectiveness of the two optimisations discussed in Section 3.4.2.2 on the block by block backtracking construction of the 12 distinct TTS(6) designs:

	No Optimisations	Point Count Constraint Only	Point Count AND Pair Count Constraints
# Extend loop iterations	43512	664	278
# Backtrack loop iterations	91755	5781	2364
Execution time (secs)	0.131	0.0066	0.0030

#### The 465 Distinct TTS(7) Designs

The block by block backtracking algorithm was also used to generate the 465 distinct TTS(7) designs. The table below compares the effectiveness of the backtracking optimisations on each of these enumerations:

	No Optimisations	Point Count Constraint Only	Point Count AND Pair Count Constraints
# Extend loop iterations	17,130,010	16,858	7,264
# Backtrack loop iterations	44,936,682	198,013	78,123
Execution time (secs)	59.5	0.233	0.117

The combination of the constraints reduces the time taken to enumerate the designs by a factor of more than 500.

#### The 4,409,916 Distinct TTS(9) Designs

For the TTS(9) enumeration, the extra constraints of the point and pair counting optimisations reduce an infeasible problem into one which can be completed quickly. There are 4,409,916 distinct TTS(9) designs. When the block by block backtracking was performed without either of the optimisations, it took approximately 2 hours to construct the first 300 designs. This is because the search was very often exploring paths of the search tree which

could not lead to feasible solutions because either a point or an unordered pair of points had not been covered by sufficiently many blocks. If this rate of construction continued for the remaining 4,409,616 designs, it would take more than 105,000,000 seconds to generate them all. This corresponds to approximately 1224 days or 3.4 years - which is prohibitively long. When the first of the optimisations, that of the point count constraint, was used by itself, all 4,409,916 were constructed in 7,667 seconds - approximately 2.1 hours.

When both the point count and pair count optimisations were combined, all the designs were constructed in 3,004 seconds - just over 50 minutes. The enormous reduction in construction times, by a factor of more than 35,000 with and without the constraints, illustrates in general how important the use of appropriate feasibility properties are to performing efficient backtrack searches.

Although the algorithm has performed well on fairly small designs, it does not take much effort to discover the limitations of the current approach.

There are 3,061,998,240 distinct TTS(10) designs, and for the next largest admissible order, the total number of distinct TTS(12) designs is astronomical. Even with the optimisations presented, construction of all distinct TTS(10) designs is pushing the limits of the current algorithm. On average, roughly 350 distinct TTS(10) designs are constructed each second, meaning a complete enumeration would take approximately 100 days. Certainly, construction of all distinct TTS(12) designs using this method would be infeasible.

In order to look at larger, more interesting problems, a more powerful approach than block by block backtracking was developed. This is the subject of the last main section, 3.4.3, of this chapter which introduces point by point backtracking.

### **3.4.3 POINT BY POINT BACKTRACKING CONSTRUCTION**

Section 3.4.2 introduced the technique and discussed the limitations of block by block backtracking. Although it is an analogous procedure, point by point backtracking has been utilised in all of the powerful versions of the algorithms developed in this thesis, for a number of good reasons. The aim of this main section is to compare the block by block and point by point approaches, in particular highlighting the advantages of the latter. An efficient implementation of point by point backtracking, performed on the incidence matrix of the corresponding structure, is also detailed.

#### **3.4.3.1 INCIDENCE MATRIX BACKTRACKING**

Incidence matrices are a convenient way of representing block designs, and indeed incidence structures in general. The block by block backtracking approach described in Section 3.4.2, although initially presented as generating a design's block list, can equivalently be thought of as backtracking on the columns of the incidence matrix, which correspond to the blocks of the design.

Similarly, the generation of block designs using point by point backtracking can be considered from the point of view of constructing either the block list or the incidence matrix of the design. For a partial design which is being constructed point by point, the incidence matrix will be completed down to a particular row, whereas the block list may contain some complete, partially complete and even empty blocks.

The two backtracking techniques are now compared, by considering their actions on the incidence matrix of the design being constructed. For example, consider the block list of the following TTS(9):

1 2 3	1 6 7	2 4 6	2 7 9	3 5 7	4 7 8
1 2 3	1 6 9	2 4 6	2 7 9	3 5 9	4 8 9
1 4 5	1 7 8	2 5 8	3 4 7	3 6 8	5 6 7
1 4 5	1 8 9	2 5 8	3 4 9	3 6 8	5 6 9

The incidence matrix for this design is given below, with the rows numbered from 1 to 9 representing the points of the design and the columns numbered 1 to 24 representing the blocks of the design:

```

          1      2
123456789012345678901234
1: 111111110000000000000000
2: 110000001111110000000000
3: 110000000000001111110000
4: 001100001100001100001100
5: 001100000011000011000011
6: 000011001100000000110011
7: 000010100000111010001010
8: 000000110011000000111100
9: 000001010000110101000101

```

This TTS(9), and more specifically its incidence matrix, can be constructed either block by block or point by point. Block by block construction would build a column of the incidence matrix at a time, whereas point by point construction would build the matrix row by row. Halfway through a block by block construction, with the partial solution consisting of the first 12 blocks of the design, the incidence matrix would appear as below:

```

          1      2
123456789012345678901234
1: 111111110000000000000000
2: 110000001111000000000000
3: 110000000000000000000000
4: 001100001100000000000000
5: 001100000011000000000000
6: 000011001100000000000000
7: 000010100000000000000000
8: 000000110011000000000000
9: 000001010000000000000000

```

The next valid extension of this partial design would add the block  $\{2, 7, 9\}$ , completing the 13<sup>th</sup> column.

The incidence matrix of the design approximately halfway through a point by point construction, say when the first 5 of the 9 points had been added, is given below:

```

          1      2
123456789012345678901234
1: 111111110000000000000000
2: 110000001111110000000000
3: 110000000000001111110000
4: 001100001100001100001100
5: 001100000011000011000011
6: 000000000000000000000000
7: 000000000000000000000000
8: 000000000000000000000000
9: 000000000000000000000000

```

The next valid extension to this partial solution would be the addition of a valid row 6.

For any non-symmetric block design, the number of blocks is larger than the number of points, and for most designs the number of blocks is considerably larger. In terms of the construction of the incidence matrix of a design, backtracking would be performed on considerably fewer levels using a point by point technique. The completion of each level of the backtrack usually requires that a number of data structures be updated, and provided that the addition of a given point at each level could be performed efficiently, backtracking on a fewer number of levels is a great advantage.

Most importantly, the canonicity tests performed and the calculation of automorphism groups which were vital to the success of the algorithm (and are described in detail in Chapters 4 and 5), are naturally suited to partial designs which have been built point by point rather than block by block.



Another of the major advantages of a point by point approach is that it is able to exploit a second level of backtracking to a much greater degree than the block by block approach. This idea is described in detail next.

### 3.4.3.2 ONE AND TWO LEVEL BACKTRACKING

For the block by block backtracking algorithm described in Section 3.4.2.1, each time the partial design was extended or backtracked, a complete block was added to or removed from it. In the context of the incidence matrix of the design, this is equivalent to adding or removing complete columns of the matrix in single operations. Such an approach is termed a 1-level backtrack, because the blocks themselves are the only entities involved.

A more sophisticated approach for the block by block backtrack would be to use a 2-level technique, in which backtracking is performed not only on the blocks but also on the points within the blocks. If such a 2-level technique can be exploited properly, it can significantly improve the performance of a backtracking algorithm, in this case by reducing the amount of time spent searching for blocks with which to extend the design's partial solution. It is shown below why such a 2 level approach is exploited much more effectively by a point by point rather than a block by block technique.

Consider a valid partial TTS(6) consisting of the four blocks:

```
1 2 4
1 2 6
1 3 4
1 3 5
```

This partial TTS(6) is to be extended by a block by block backtracking procedure. A 1-level backtracking technique would try to add the following 3 blocks as the fifth block of the partial design, all of which would be rejected for the reasons stated below:

BlockAdded	Reason for rejecting the block, using the pair count feasibility property
1 3 6	The pair {1, 3} is already covered twice
1 4 5	The pair {1, 4} is already covered twice
1 4 6	The pair {1, 4} is already covered twice

The next block in lexicographical order to be added is {1, 5, 6}, which is perfectly valid and in fact leads to a completed TTS(6) design.

Now consider the conversion of this 1-level backtracking approach to a 2-level approach. The designs are still built block by block, but rather than adding complete blocks in one operation, each block is constructed by backtracking a point at a time.

Again, the previous example can be used to illustrate this, just as the fifth block is to be added to the partial TTS(6):

```
1 2 4
1 2 6
1 3 4
1 3 5
```

The new block is built point by point. The first point added to the empty block is "1", and this is in fact necessary as each point must occur in 5 blocks of the design. The designs are still built in increasing block order, so the next point to add to the second position of the fifth block will be "3", as each successive block must be at least as large as the previous block.

At this stage, before the final point of the fifth block is even considered, backtracking within the fifth block can occur because the pair {1, 3} has already been covered twice. The point "3" is removed and replaced by the next possible point, "4". Again a violation is

detected and the "4" is replaced by a "5", which is acceptable. Only one possibility then remains for the final point of the block, "6", and this is then added to the last position.

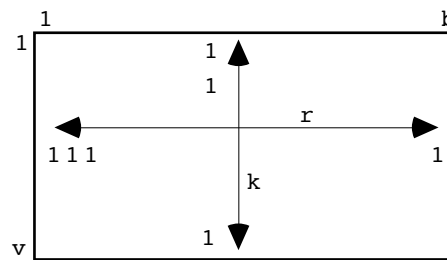
When the number of distinct block types,  $\binom{v}{k}$ , is relatively small, as it is for the TTS(6) designs, there is no real advantage in using a 2-level approach. This is because the amount of extra work performed by attempting to add each block in lexicographical order to the partial solution rather than building the blocks point by point is not significant. The 2-level block by block approach would perform notably better than the 1-level block by block approach on larger designs where the number of distinct block types is considerably greater.

Even for the TTS(12) designs, which are far too numerous to construct by a block by block approach, there are only 220 distinct block types. When adding a block to the partial solution, equivalent to adding a complete column to the incidence matrix, only a very small number of these distinct possibilities will need to be tried before one can successfully be added, or before it is determined that no block can be added.

When the partial solution contains only a small number of blocks, there are less design constraints which can be violated, and adding the next block in lexicographical order can be done rapidly. On the other hand, as the size of the partial solution grows and there are more possibilities for violating the design constraints, there are also fewer blocks to be considered as possible additions because the blocks are added in increasing lexicographical order. In conclusion, as long as the number of distinct block types is relatively small, there will be essentially no advantage in performing a 2-level block by block backtrack over a single level approach.

The constructive enumeration of all designs for a given set of parameters is limited to a great extent by the order, or number of points in the design. As the order increases, the number of designs tends to increase exponentially. Therefore the classes of incidence structures which can be completely enumerated tend to be those of relatively small order,  $v$ , and consequently relatively small block size,  $k$ . As  $\binom{v}{k}$  is fairly small for such structures, a 2-level backtracking approach simply cannot be well exploited by a block by block implementation.

However, even for small designs, the use of a 2-level backtracking algorithm coupled with point by point construction is very effective. In general, point by point backtracking can exploit a 2-level approach to a much greater extent than block by block backtracking can, and this is one of the main reasons it was chosen for the algorithms developed in this thesis. Consider the general form of the incidence matrix of a BIBD, given below:



The matrix consists of  $v$  rows and  $b$  columns, with  $k$  ones in each column and  $r$  ones on each row. The number of distinct block types has already been defined, and is calculated as  $\binom{v}{k}$ . This is equivalent to the number of distinct types of columns which can exist in the incidence matrix. The number of distinct row types for the incidence matrix can also be calculated in a similar way, as  $\binom{b}{r}$ .

One of the trivial necessary conditions for existence of BIBDs is  $vr=bk$ . For symmetric BIBDs,  $v=b$  and  $r=k$ . For all others,  $v < b$ , which also implies  $k < r$ . Therefore, for any

parameter set,  $\binom{v}{k} \leq \binom{b}{r}$ . In other words, the number of distinct column types is less than or equal to the number of distinct row types. For most designs, the number of distinct row types is significantly greater.

Point by point backtracking builds the incidence matrix of a design row by row. A 1-level point by point backtracking algorithm would try to place each of the possible distinct row types into each row. However, most of the rows placed would be invalid due to one of the trivial constraints of the design, and such a 1-level approach would be very inefficient. If however a 2-level approach were adopted, each row,  $r$ , of the incidence matrix would itself be built using a backtracking technique, pruning from consideration a large number of invalid row types as row  $r$  was being constructed.

In conclusion, a point by point backtracking algorithm can exploit 2-levels of backtracking to a much greater degree than a corresponding block by block approach. The reason for this is simply that the number of distinct row types of the incidence matrix of a given design will in general be much greater than the number of distinct column types, and so backtracking on the rows will in general prune a larger number of invalid configurations from consideration.

### 3.4.3.3 INCIDENCE MATRIX CONSTRUCTION ORDERING

In the block by block backtracking algorithm, a completely ordered form was defined for a given block list which allowed two designs represented as block lists to be lexicographically compared, and also defined an ordering over the construction of designs. The conversion of a block list to its completely ordered form required the following two steps:

- 1) A lexicographical sorting of the points within each block
- 2) A lexicographical sorting of the blocks of the design

A perfectly analogous ordering can be defined on incidence matrices, so that an incidence matrix is in *ordered form* if the points within each block and the blocks themselves are sorted into lexicographical order.

The blocks of a design correspond to the columns of the incidence matrix, and so the points within each block of a design represented as an incidence matrix are already sorted. For example, the block  $\{1, 2, 4\}$  can be reordered on its points in 6 different ways, yet each permutation has exactly the same form as a column of the incidence matrix, namely:

Row 1: 1  
 Row 2: 1  
 Row 3: 0  
 Row 4: 1

An ordering on the columns of a  $v \times b$  incidence matrix is defined such that column  $X=x_i$  ( $1 \leq i \leq v$ ) is less than column  $Y=y_i$  ( $1 \leq i \leq v$ ) if for some  $n \leq v$ ,  $x_i = y_i$  for  $1 \leq i < n$ , but  $x_n = 0$  whereas  $y_n = 1$ . With this standard lexicographical ordering, the sorting of the blocks of the design is easily achieved by enforcing that each column of the incidence matrix be in increasing lexicographical order. For example, the following TTS(6):

1	2	3	1	3	5	1	5	6	2	4	5	3	4	5
1	2	4	1	4	6	2	3	6	2	5	6	3	4	6

can be represented by the following column ordered incidence matrix:

```

1111100000
1100011100
1010010011
0101001011
0010101110
0001110101

```

This definition of ordering also adds direction to the construction of a class of designs. The point by point backtracking algorithm, which builds the incidence matrices of the designs row by row, constructs each incidence matrix in strictly increasing ordered form.

### 3.4.3.4 ROW BY ROW INCIDENCE MATRIX CONSTRUCTION

This section outlines the operation of the point by point backtracking algorithm for construction of a generic BIBD. Such a design contains  $v$  points and  $B$  blocks of size  $k$  with the constraint that each point must occur in  $r$  blocks and each unordered pair of points must occur in exactly two blocks.

The empty incidence matrix, before construction takes place, has the form:

```
Columns: 1234..... B
Row 1:  00000000000000000000 ..... 00000
Row 2:  00000000000000000000 ..... 00000
Row 3:  00000000000000000000 ..... 00000
.....
Row v:  00000000000000000000 ..... 00000
```

The blocks of the design, represented by the columns of the matrix, are to be constructed in lexicographical order. In other words, block  $a$  must be lexicographically less than or equal to block  $b$  for all blocks  $a$  and  $b$  represented respectively by columns  $c_a$  and  $c_b$ , such that  $c_a$  is to the left of  $c_b$ .

The first point to be placed in the design, point "1", must be added to exactly  $r$  blocks. In fact, this point must be added to the first  $r$  blocks of the design, ie. into the leftmost  $r$  columns. The reason for this can be illustrated by assuming they were not added to the leftmost  $r$  columns of row 1. Then row 1 may look something like:

```

      ← r+1 →
Row1:11111011110000000000000000000000
      |
      x

```

Above, point 1 is added to the first  $r+1$  blocks of the design, except for block  $x$ . Now for any solution design constructed as an extension to this initial row, block  $x$  will not contain point "1". Therefore, block  $x$  will be lexicographically larger than all of the blocks to its right which do contain point "1", and so the matrix will not be in ordered form.

This notion of ordering on the columns is relevant for every row of the incidence matrix, in a similar way to above. If the incidence matrix has been partially completed down to row  $n$  (for  $n < v$ ), and a set of adjacent partial blocks exist which are equal down to row  $n$ , then any addition of point  $(n+1)$  to this set of blocks must be added to the leftmost columns. For example, consider the following partial incidence matrix to row  $n$  for a certain adjacent subset of blocks labelled  $a$ ,  $b$  and  $c$  such that  $(a < b < c)$ :

```
Columns:  abc
Row 1:    111
Row 2:    000
Row 3:    111
.....
Row n:    111
```

Each of the columns  $a$ ,  $b$  and  $c$  represent the partial block  $\{1, 3, \dots, n\}$ . If point  $(n+1)$  is added to this set of blocks, it must be added to block  $a$  first. If another copy is to be added, it would then extend block  $b$ . Block  $c$  would only get a copy of the point  $(n+1)$  if both blocks  $a$  and  $b$  already contained it. If block  $b$  or  $c$  contained point  $(n+1)$  but block  $a$  did not, this would violate the ordering of the incidence matrix. The adjacent columns  $a$ ,  $b$  and  $c$  above form what is termed a *cell*, and its structure determines how points may be added to the set of columns. Cell structure is an extremely important property of a partial incidence matrix,

and is essential to many of the optimisations which have been developed for the algorithm. This property is outlined in the following section.

### 3.4.3.5 CELL STRUCTURE

The explicit lexicographical ordering of the columns of the incidence matrix gives rise to the concept of *cell structure*, which is central to many of the optimisations introduced later. At each row,  $r$ , of the incidence matrix, all the blocks containing exactly the same set of treatments down to row  $r$  belong to the same cell. Due to the lexicographical ordering, all those blocks which belong to the same cell down to any given row are represented by adjacent columns in the incidence matrix. The partial incidence matrix down to row 4 of a TTS(9) is given below:

```

1:  111111110000000000000000
2:  110000011111100000000000
3:  110000000000011111100000
4:  0011000110001100001100

```

There are two cells at row 1 of the incidence matrix, dividing the columns into a set of blocks containing the point "1", and a set not containing the point "1". There are 4 cells at row 2, 5 cells at row 3 and 9 cells at row 4. To illustrate better the division of the matrix into cells, it has been divided at cell boundaries down to row 4 below:

```

Cells:   1  2  3   4  5   6  7   8  9
1:       11 11 1111 00 0000 00 0000 00 00
2:       11 00 0000 11 1111 00 0000 00 00
3:       11 00 0000 00 0000 11 1111 00 00
4:       00 11 0000 11 0000 11 0000 11 00

```

When the construction of the next row is considered, each point placed on row 5 will fall within one of the cells at the previous level, row 4:

```

5:       00 11 0000 00 1100 00 1100 00 11

```

Due to the ordering of columns, any set of points on row 5 which lie within the same cell of the row 4 cell structure must be placed as far to the left within that cell as possible. There is no choice about where within a cell to place the points - only the number of points placed within each cell is of importance.

### 3.4.3.6 EXTENDING AND BACKTRACKING

The row by row construction process involves the use of two primitive operations: extending and backtracking. These are analogous to the same operations described in Section 3.4.2.1 for the block by block backtracking algorithm.

Extending occurs when a valid partial design has been constructed down to a particular row of the incidence matrix, and the matrix should be extended to the next level. Backtracking occurs when the points placed within a row have violated a constraint or when all extensions to a particular row have been calculated, and thus the points need to be reordered in a new way.

In order to direct the search, an ordering must be defined on the placement of points within each row. Although a simple lexicographical ordering would be sufficient, a slightly more sophisticated ordering is used which is essential for some of the optimisations introduced in Chapter 4. With each distinct placement of points on a given a row, a unique *cell fill vector* is associated which represents the number of points in the placement which lie in each cell determined by the cell structure of the previous row.

For example, consider again the first four rows of the incidence matrix of a TTS(9) design:

```

Cells:   1  2  3   4  5   6  7   8  9
1:       11 11 1111 00 0000 00 0000 00 00
2:       11 00 0000 11 1111 00 0000 00 00
3:       11 00 0000 00 0000 11 1111 00 00
4:       00 11 0000 11 0000 11 0000 11 00

```

There are 9 cells defined by the cell structure down to row 4. This cell structure is used to calculate the cell fill vectors for point placements on row 5. Two distinct, valid point placements for row 5 of the incidence matrix are given below:

```

Possible row 5 A: 000011001010001010000011
Possible row 5 B: 000011001010000011001010

```

When each of the two possible rows above are separated into the 9 cells given by the cell structure of the first 4 rows of the matrix, the number of points falling within each cell becomes apparent:

```

Cells:   1  2  3   4  5   6  7   8  9
1:       11 11 1111 00 0000 00 0000 00 00
2:       11 00 0000 11 1111 00 0000 00 00
3:       11 00 0000 00 0000 11 1111 00 00
4:       00 11 0000 11 0000 11 0000 11 00

```

```

Row A:    00 00 1100 10 1000 10 1000 00 11
Row B:    00 00 1100 10 1000 00 1100 10 10

```

The cell fill vectors for each of the possible row 5's, A and B, are calculated directly by counting the number of points which lie within each cell. Each vector consists of 9 elements, corresponding to the cell structure of the first 4 rows:

```

Cell Fill Vector for Row A: 0 0 2 1 1 1 1 0 2
Cell Fill Vector for Row B: 0 0 2 1 1 0 2 1 1

```

Two distinct point placements for a given row can then be compared lexicographically by their corresponding cell fill vectors.

Let  $P$  and  $Q$  be two distinct point placements for a given row  $r$ . Their cell fill vectors calculated using the cell structure of the first  $r-1$  rows are denoted by  $CFP_i$  and  $CFQ_i$  respectively, where  $1 \leq i \leq n$  and  $n$  is the number of cells in the cell structure. It is defined that  $P < Q$  if and only if there exists a  $j$ ,  $1 \leq j \leq n$  such that  $CFP_i = CFQ_i$  for all  $1 \leq i < j$ , but  $CFP_j > CFQ_j$ .

In other words, the earlier in the algorithm the point placement is constructed, the "larger" the corresponding cell fill vector. As the backtracking construction proceeds, the cell fill vectors of the rows of the incidence matrix steadily decrease. The point placements generated later in the construction therefore have smaller cell fill vectors, and are said to be lexicographically greater than those point placements generated earlier in the construction - since the search proceeds in increasing lexicographical order.

Given this definition, it can be determined that of the two possibilities, A and B, for row 5 given above,  $A < B$ . This means that A would be generated earlier in the backtrack construction of row 5 than B.

The row by row exhaustive construction is now outlined, by detailing the extend and backtrack operations. The generic backtracking procedure which was introduced in Section 3.4.2.1 again describes the underlying process:

```

repeat
  while (partial solution is valid)
    Extend partial solution
  while (partial solution is invalid)
    Backtrack partial solution
until all possibilities exhausted

```

The following two subsections detail the behaviour of the primitive extend and backtrack operations.

### Extend

A partial design consists of points  $\Sigma = \{1, 2, \dots, w\}$ , ( $w \leq v$ ). Again using the example of a BIBD, such a partial design is valid if each point,  $p \in \Sigma$ , occurs exactly  $r$  times, and every unordered pair of points,  $\{a, b\}$ ,  $a, b \in \Sigma$ ,  $a \neq b$ , occurs exactly  $\lambda$  times. In addition, no block may contain more than  $k$  points - in other words there must be no more than  $k$  1's in any column of the partial incidence matrix. A valid partial incidence matrix, constructed down to the first 4 rows only, for a TTS(9) design is given below:

```
Row 1:      111111110000000000000000
Row 2:      110000001111110000000000
Row 3:      110000000000001111110000
Row 4:      001100001100001100001100
```

Each row contains exactly 8 points - the replication number for a TTS(9). The columns contain either 0, 1, 2 or 3 points. The first two blocks of the design are both  $\{1, 2, 3\}$ , and are complete in the sense that no further points may be added to them. All other blocks are incomplete and must be augmented with further points as the deeper rows of the incidence matrix are generated. Every unordered pair  $\{i, j\}$ , for  $1 \leq i < j \leq 4$ , must be covered exactly  $\lambda = 2$  times, because no more copies of these points can be added to any extension of this partial design. The dot product, or intersection count, between any two rows is therefore exactly 2. This valid partial design will be used as the example for this section.

The *extend* operation is performed on a partial incidence matrix which is perfectly valid down to a given row, say  $x-1$ , and which needs to be extended. The next row,  $x$ , is given the distinct point placement having the largest possible corresponding cell fill vector. This is therefore the lexicographically smallest possible arrangement of points on the new row.

Row  $x$  is extended by starting at column 1 of the incidence matrix, and working left to right along row  $x$ , examining each column systematically up to the last column,  $B$ . For each examined column,  $c$ , if the placement of point "x" in the corresponding block does not violate any of the design constraints, then a 1 will be placed in column  $c$ .

In other words, the result of an extend operation is that the 1's are placed as far as possible to the left hand side of the row being extended. This will obviously generate the lexicographically smallest possible row  $x$ , ie. the one with the largest possible cell fill vector.

To help explain this operation, the effect of extending row 5 of the 4-row partial TTS(9) design given earlier is illustrated. The incidence matrix has been divided along cell boundaries below so that the effect on each cell of extending row 5 can be shown more clearly.

```
Cells:      1  2  3   4  5   6  7   8  9
1:          11 11 1111 00 0000 00 0000 00 00
2:          11 00 0000 11 1111 00 0000 00 00
3:          11 00 0000 00 0000 11 1111 00 00
4:          00 11 0000 11 0000 11 0000 11 00
```

The extension process for row 5 is described in the steps below, in which each cell is considered in turn from 1 up to 9. At each step, the current cell of the new row 5 being generated is given in bold.

```
Cell 1: 00 00 0000 00 0000 00 0000 00 00
```

The columns are examined from left to right. The first two columns, corresponding to the first cell, represent complete blocks and so clearly can not have point "5" added to them.

```
Cell 2: 00 00 0000 00 0000 00 0000 00 00
```

The blocks in the second cell contain only two points: {1,4}, and so point 5 can be added to both columns in this cell. The unordered pairs {1,5} and {4,5} are therefore both covered twice.

Cell 3: 00 11 **0000** 00 0000 00 0000 00 00

The pair {1,5} is already covered in two blocks at this stage, and so cannot be contained in any more blocks. Point 5 is therefore not added to any of these blocks.

Cell 4: 00 11 0000 **00** 0000 00 0000 00 00

Again, point 5 cannot be added to the blocks in this cell because the pair {4,5} is already covered twice.

Cell 5: 00 11 0000 00 **0000** 00 0000 00 00

The pair {2,5}, which will be covered by any addition of point "5" to these blocks, has not yet been covered. Therefore the two leftmost columns of this cell can be set to 1.

Cell 6: 00 11 0000 00 1100 **00** 0000 00 00

No additions can be made, as the pair {4,5} is already covered.

Cell 7: 00 11 0000 00 1100 00 **0000** 00 00

The pair {3,5} is not yet covered, and so point "5" is added twice to this cell.

Cell 8: 00 11 0000 00 1100 00 1100 **00** 00

No additions can be made, as the pair {4,5} is already covered twice.

Cell 9: 00 11 0000 00 1100 00 1100 00 **00**

This cell represents two empty blocks. Up to this stage, point "5" has only been added to 6 blocks, however the replication number of a TTS(9) is 8, and so it must be added to another 2 blocks. Both columns in this cell therefore have the point "5" added, and obviously no new unordered pairs are covered by this last addition.

The partial incidence matrix, after extending row 5, is given below:

Cells:	1	2	3	4	5	6	7	8	9
1:	11	11	1111	00	0000	00	0000	00	00
2:	11	00	0000	11	1111	00	0000	00	00
3:	11	00	0000	00	0000	11	1111	00	00
4:	00	11	0000	11	0000	11	0000	11	00
5:	00	11	0000	00	1100	00	1100	00	11

The cell fill vector for row 5, under the cell structure imposed by the first four rows, is:

0 2 0 0 2 0 2 0 2

This is the largest possible cell fill vector for any valid point placement on row 5, given the first four rows.

### Backtrack

The *backtrack* operation is performed on an already initialised row, which needs to be rearranged to form the lexicographically next largest possible point placement, which will accordingly have the next smallest cell fill vector.

Backtracking on the current point placement of a given row, say  $x$ , starts at the rightmost column of the incidence matrix,  $B$ , and systematically works towards the leftmost, first column. Let column  $c$  be the first column on row  $x$  encountered in this fashion containing a "1". The block design represented by the incidence matrix therefore contains the point " $x$ " within block  $c$ . A check is then performed to determine whether this point can be moved to any column to the right of  $c$ . If it cannot be moved to any column further to the right, the point is discarded and the process continues on the remaining columns of the row, working from right to left. Otherwise, let  $c_r$  be the nearest column to the right of column  $c$  to which the point can be moved without violating the design constraints.

Clearly, under the cell structure of the first  $(r-1)$  rows,  $c$  and  $c_r$  must not both lie within the same cell, as no point can be moved amongst the columns within a cell. All points within a given cell must reside in the leftmost columns, and there is no choice about how these may be rearranged.

Let  $\text{Cell}_{\text{old}}$  be the cell which contains column  $c$ , in which the point currently resides, and let the cell which contains column  $c_r$  be denoted  $\text{Cell}_{\text{new}}$ .



The point is removed from column  $c$  and replaced into column  $c_r$ , further to the right. This has the effect of decreasing by one the value of  $\text{Cell}_{\text{old}}$ , and increasing by one the value of  $\text{Cell}_{\text{new}}$ . Overall, this reduces the value of the cell fill vector - producing a lexicographically larger point placement.

Consider again the partial incidence matrix for the TTS(9) design given below, for which row 5 has just been extended:

```
Cells:   1  2  3   4  5   6  7   8  9
1:      11 11 1111 00 0000 00 0000 00 00
2:      11 00 0000 11 1111 00 0000 00 00
3:      11 00 0000 00 0000 11 1111 00 00
4:      00 11 0000 11 0000 11 0000 11 00
5:      00 11 0000 00 1100 00 1100 00 11
```

Backtracking on this partial design would systematically remove points from row 5 of the matrix, working from right to left, until one of the points removed could be repositioned in a column further to the right. This process is described step by step below, and the cell being examined at each step is given in bold.

Cell 9: 00 11 0000 00 1100 00 1100 00 **11**

Neither of the two points in this cell can be shifted further to the right, and so they are both discarded. There are no points to consider in cell 8, so backtracking skips these two columns.

Cell 7: 00 11 0000 00 1100 00 **1100** 00 00

If the rightmost of the two points in this cell was removed, the nearest column to the right to which it could validly be shifted would be the leftmost column of cell 9. Therefore the backtracking procedure would remove this point, and reposition it in the 9th cell, as given below:

00 11 0000 00 1100 00 **1000** 00 **10**

Once the first point which can be repositioned has been moved further to the right, the remainder of the row is then reconstructed using the same process as row extension. In the case above, the last column of the 9<sup>th</sup> cell would also have point 5 added.

This backtracking operation has therefore reduced the cell fill vector from:

0 2 0 0 2 0 2 0 2

to

0 2 0 0 2 0 1 0 2

Although the new point configuration on row 5 is indeed lexicographically greater than before, it is not a valid placement because point "5" only occurs in 7 blocks, instead of the required 8. Backtracking would repeatedly be performed on this row until a valid point placement was constructed, or until all possibilities on the row had been exhausted. After several more backtracking operations on row 5, the very next valid point placement becomes:

```
Cells:   1  2  3   4  5   6  7   8  9
1:      11 11 1111 00 0000 00 0000 00 00
2:      11 00 0000 11 1111 00 0000 00 00
3:      11 00 0000 00 0000 11 1111 00 00
4:      00 11 0000 11 0000 11 0000 11 00
5:      00 10 1000 10 1000 00 1100 00 11
```

For which row 5 has the cell fill vector:

0 1 1 1 1 0 2 0 2

What becomes apparent after stepping through such an example, is that very often a lot of the work performed during a series of backtracking steps is wasted. For example above, using the currently described method of backtracking each point on row 5 is moved to the very next valid cell to its right then the rest of the row is reconstructed without success, a number of times. It is not until one of the points in cell 2 is removed that the row can be reconstructed validly.

Particularly for larger searches, an enormous amount of unnecessary work can be saved if it can quickly be determined during backtracking whether a row can be validly reconstructed once one of the points on the row is shifted to the right. In fact, there are a very large number of optimisations which can be applied to both the extending and backtracking procedures, and a selection of these are presented next.

### 3.4.3.7 OPTIMISATIONS

Recall from the block by block algorithm, the point count and pair count optimisations which were explicitly implemented. One counted the points in the design and backtracked as soon as a partial design did not contain enough copies of a particular point to be completed validly. The other optimisation counted pairs, causing backtracking as soon as the partial design did not contain enough copies of a particular unordered pair to be completed validly.

The first of these, the point count constraint, is automatically incorporated into the point by point construction. Each point is added to the partial design until it is covered exactly  $r$  times, and unless this is satisfied the next point is not considered. The detection of a point which has not been covered enough times is therefore trivial, and is a further advantage of a point by point construction technique.

The second optimisation, which counts pairs, is very simple to implement. For any given unordered pair,  $\{i, j\}$ , the total number of blocks it has been covered by is simply the dot product of rows  $i$  and  $j$  of the corresponding incidence matrix. If a point is placed on row  $j$  such that no more pairs of the form  $\{i, j\}$ , (for  $1 \leq i < j$ ) can be added to the current partial design, and if the pair  $\{i, j\}$  has not yet been covered exactly  $\lambda$  times, then a violation can be detected and backtracking should begin.

To describe a simple check for such a pair count optimisation, the notion of the *cover* of any given point must be defined. For any point represented as a "1" on row  $p$  and in column  $c$  of the incidence matrix of a design, the *cover* of that point is defined to be the lexicographically smallest point already contained in the corresponding block  $c$ . If the point is already the lexicographically smallest one in the block, then it does not have a cover. For example, consider the partial TTS(9) below:

Row 1:	111111110000000000000000
Row 2:	110000001111110000000000
Row 3:	110000000000001111110000
Row 4:	001100001 <u>1</u> 00001100001100

The cover of the point which is underlined is the point "2", as the block represented by this column is the incomplete block  $\{2,4\}$ , and 2 is the smallest point in the block.

The pair count optimisation is implemented as follows. Consider the construction of row  $p$  of the incidence matrix of a design, and let point "p" be the newest point placed on this row. It needs to be determined whether the addition of point "p" has caused a pair count violation. Let  $\text{cov}(p)$  denote the cover of point "p". The lexicographically smallest pair covered by this new point placement is the pair  $\{\text{cov}(p), p\}$ . No pair lexicographically smaller than  $\{\text{cov}(p), p\}$  can be covered by any further placements of points on the given row and so if it is not already covered  $\lambda$  times, no extension to the current partial design can be valid. Once this violation is detected, backtracking can occur immediately.

As a consequence of the above constraint, a further optimisation can be defined. Consider the operation of backtracking on a given row, say  $r$ , of the incidence matrix. Let the point on row  $r$  which is being considered for shifting to the right reside in column  $c$ . It is already established that this point cannot be shifted to any other column which is in the same cell as the column  $c$ .

The new optimisation insists that the point can only be shifted to another column in which it will have precisely the same cover as it has in its current position. In other words, the point in column  $c$  should not be shifted to the right during backtracking to another column which has a different cover than that of column  $c$ . The reason for this is clear. Let "p" be

the point which is shifted, and let  $\text{cov}(p)$  be the cover of point "p" in column  $c$ , its position before shifting. If point "p" is shifted to another column in which it has a different cover, then a pair of the form  $\{\text{cov}(p), p\}$  will be removed from the partial design, and this pair cannot be covered sufficiently many times by any further extension. In this case the shift of point "p" should not be accepted, and point "p" can in fact be discarded. This optimisation considerably restricts the amount of backtracking which is performed on any given row.

### 3.4.3.8 RESULTS

The point by point backtracking algorithm has been used to construct all distinct designs of the same twofold triple system classes that were generated by the block by block backtrack, in Section 3.4.2, and this section compares the results of each approach.

#### The 12 Distinct TTS(6) Designs

The table below compares the performance of the block by block algorithm without any optimisations and with the point counting optimisation, with the performance of the point by point algorithm without any of the explicit optimisations. The point by point approach implicitly performs the point counting constraint as the incidence matrix is constructed row by row, and as the results below indicate, it clearly outperforms the block by block approaches. In each case, all 12 distinct TTS(6) designs were constructed.

	Block by Block (without optimisations)	Block by Block (with point counting optimisation)	Point by Point (without optimisations)
# Extend loop iterations	43512	664	24
# Backtrack loop iterations	91755	5781	578
Execution time (secs)	0.131	0.0066	0.0015

The table below compares the block by block and point by point approaches, using all the optimisations presented in Sections 3.4.2.2 and 3.4.3.7 respectively, for enumeration of the distinct TTS(6) designs. The point by point technique is clearly superior.

	Block by Block (fully optimised)	Point by Point (fully optimised)
# Extend loop iterations	278	24
# Backtrack loop iterations	2,364	115
Execution time (secs)	0.0030	0.00056

#### The 465 Distinct TTS(7) Designs

The advantages of the point by point algorithm become even more apparent on larger designs. The table below compares the performance of both fully optimised methods on the enumeration of the 465 distinct TTS(7)'s.

	Block by Block (fully optimised)	Point by Point (fully optimised)
# Extend loop iterations	7,264	607
# Backtrack loop iterations	78,123	3,552
Execution time (secs)	0.117	0.019

#### The 4,409,916 Distinct TTS(7) Designs

The TTS(9) enumeration was completed using the point by point algorithm, with and without the presented optimisations.

Without any explicit optimisations, all 4,409,916 distinct TTS(9)'s were constructed in 2908 seconds, just under 50 minutes. With the optimisations, the same design generation took just 305 seconds, approximately 5 minutes. This yields a construction rate of more than 14,400 designs a second.

In comparison, the block by block algorithm would have taken an estimated time of more than 3 years without any optimisations, and even the fully optimised block by block algorithm required around 50 minutes.

These results are strong indicators of the superiority of a point by point backtracking approach, and this is why it has been chosen as the underlying backtracking technique in the fast enumeration algorithms developed in this thesis.

### **3.5 LIMITATIONS**

All three of the exhaustive generation algorithms presented in this chapter, from the brute force approach to the two backtracking approaches, were designed to generate all distinct block designs for a particular set of parameters. As an example, all distinct twofold triple systems were generated for orders 6, 7 and 9.

There is a major limitation to any algorithm which attempts to generate all distinct incidence structures of a given type. This limitation is simply the fact that as the order of the structures increases, the number of distinct designs which must be generated grows exponentially, and this severely restricts those classes of designs which can be examined.

The next chapter introduces the concept of isomorphisms, and the technique of isomorph rejection, which is critical to any effective backtracking algorithm. One of the key improvements is that not all distinct designs need to be generated explicitly, as was performed by the algorithms of this chapter, but rather the idea is to construct a considerably smaller set containing only non-isomorphic designs. Such a set contains exactly one representative for any given distinct design, and from these representatives all the distinct designs can be induced. The techniques presented in the next chapter were used to modify the incidence matrix backtracking algorithm developed in Section 3.4.3, and are essential to performing large constructive enumerations.



# Chapter 4

## Isomorph Rejection

### 4.1 INTRODUCTION

In the previous chapter, a general purpose incidence matrix backtracking algorithm was developed to generate all distinct solutions for a given parameter set. Throughout the chapter, the construction of twofold triple systems was used as an example to evaluate the algorithm and its optimisations. Despite the effectiveness of the optimisations, this approach is severely limited in the types of designs which can be generated. The number of distinct solutions grows exponentially as the order of the designs increases and the complete generation of twofold triple systems could only be performed for orders 6, 7 and 9 using the developed algorithm.

This chapter introduces the concept of isomorphisms, and explains how they can be used to improve the performance of the incidence matrix backtracking algorithm by identifying and rejecting equivalent parts of the search space. This technique was termed *isomorph rejection* by Swift [64] in 1958, and is crucial to the efficiency of an exhaustive search strategy.

As was the case in Chapter 3, the descriptions and examples in this chapter are presented in terms of balanced incomplete block designs. This is only for consistency however, and it should be noted that the definitions and techniques are relevant for all incidence structures.

### 4.2 ISOMORPHISMS

**Definition:**

Let  $D_1$  and  $D_2$  be two designs of the same type, with point sets  $V_1$  and  $V_2$  and block sets  $BL_1$  and  $BL_2$  respectively. An *isomorphism* from  $D_1$  to  $D_2$  is a mapping  $\varnothing : V_1 \rightarrow V_2$  such that for all  $b_1 \in BL_1$ ,  $\varnothing(b_1) \in BL_2$ .

If such a mapping exists, then designs  $D_1$  and  $D_2$  are said to be isomorphic - otherwise they are non-isomorphic. In other words, two designs are isomorphic if one can be formed by relabelling the points of the other.

Clearly, if designs  $D_1$  and  $D_2$  are isomorphic (say by mapping  $M_1$ ), and designs  $D_1$  and  $D_3$  are isomorphic (say by mapping  $M_2$ ), then designs  $D_2$  and  $D_3$  are also isomorphic. The isomorphism mapping  $D_2$  to  $D_3$  can be constructed by composing mappings  $M_1^{-1}$  and  $M_2$ .

A classical problem in the field of combinatorial design theory is the constructive enumeration problem. This involves generating a complete list, or catalogue, of non-isomorphic designs of a given class. In the previous chapter, all distinct designs of a given class were constructed, even though many of these were in fact isomorphic. For any given class, the total number of distinct designs can be calculated directly from a catalogue of all non-isomorphic designs.

Consider, for example, all of the 12 distinct TTS(6) designs which were constructed in the previous chapter.

#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1 2 3	1 2 3	1 2 3	1 2 3	1 2 3	1 2 3	1 2 4	1 2 4	1 2 4	1 2 4	1 2 5	1 2 5
1 2 4	1 2 4	1 2 5	1 2 5	1 2 6	1 2 6	1 2 5	1 2 5	1 2 6	1 2 6	1 2 6	1 2 6
1 3 5	1 3 6	1 3 4	1 3 6	1 3 4	1 3 5	1 3 4	1 3 5	1 3 4	1 3 5	1 3 4	1 3 4
1 4 6	1 4 5	1 4 6	1 4 5	1 4 5	1 4 5	1 3 6	1 3 6	1 3 5	1 3 6	1 3 5	1 3 6
1 5 6	1 5 6	1 5 6	1 4 6	1 5 6	1 4 6	1 5 6	1 4 6	1 5 6	1 4 5	1 4 6	1 4 5
2 3 6	2 3 5	2 3 6	2 3 4	2 3 5	2 3 4	2 3 5	2 3 4	2 3 5	2 3 4	2 3 4	2 3 4
2 4 5	2 4 6	2 4 5	2 4 6	2 4 5	2 4 5	2 3 6	2 3 6	2 3 6	2 3 5	2 3 6	2 3 5
2 5 6	2 5 6	2 4 6	2 5 6	2 4 6	2 5 6	2 4 6	2 5 6	2 4 5	2 5 6	2 4 5	2 4 6
3 4 5	3 4 5	3 4 5	3 4 5	3 4 6	3 4 6	3 4 5	3 4 5	3 4 6	3 4 6	3 5 6	3 5 6
3 4 6	3 4 6	3 5 6	3 5 6	3 5 6	3 5 6	4 5 6	4 5 6	4 5 6	4 5 6	4 5 6	4 5 6

These designs are listed in strictly increasing lexicographical order. Consider the following point mapping, which swaps points 5 and 6:  
 (1→1) (2→2) (3→3) (4→4) (5→6) (6→5)

This maps design #1 directly onto design #2. In other words, if the first TTS(6) design in the list above is relabelled by swapping all occurrences of point 5 with point 6 and vice versa, the resulting design is in fact the second TTS(6) design in the list.

It will be useful to examine the effect of this mapping on each of the common design representations, particularly on the incidence matrix, as this is the representation used in the actual backtracking algorithm.

Firstly, consider the effect of the mapping on the block list of design #1 given above. If every occurrence of point "5" is replaced with point "6" and vice versa, it is clear that the resulting design will be precisely design #2. In fact the order of the blocks within the list of design #2 is almost exactly the same as the order of the blocks in the list of design #1, following the mapping. Only the last two blocks, {3,4,5} and {3,4,6} actually need to be reordered once the mapping is applied.

Equivalently, the effect of the mapping on the incidence matrix of the design can be examined. The incidence matrix of the first TTS(6) design in the list is given below:

```

Row 1: 1111100000
Row 2: 1100011100
Row 3: 1010010011
Row 4: 0101001011
Row 5: 0010101110
Row 6: 0001110101
    
```

The blocks of the design which contain point "5" are specified by those columns of the incidence matrix containing a "1" on row 5. Similarly, the columns which contain a "1" on row 6 indicate which blocks of the design contain point "6". The point mapping swaps all occurrences of points "5" and "6", and its effect on the design can therefore be represented precisely by interchanging rows 5 and 6 of the incidence matrix.

Swapping rows 5 and 6 of the incidence matrix above forms the new, row permuted incidence matrix below:

```

Row 1: 1111100000
Row 2: 1100011100
Row 3: 1010010011
Row 4: 0101001011
Row 5: 0001110101
Row 6: 0010101110
    
```

This incidence matrix is not in ordered form however. When examining the effect of the mapping on the block list of the design, recall that completely ordering the relabelled block list required interchanging the last two blocks. Equivalently, only the last two

columns of the incidence matrix above must be swapped to convert it to ordered form. The resulting ordered incidence matrix is given below:

```

Row 1: 1111100000
Row 2: 1100011100
Row 3: 1010010011
Row 4: 0101001011
Row 5: 0001110110
Row 6: 0010101101

```

This is indeed the incidence matrix of the second TTS(6) design in the list.

Clearly, the first two distinct TTS(6) designs in the list are isomorphic. In fact it can easily be shown using a similar process to that just described, that any pair of TTS(6) designs from the list are isomorphic. In other words, the 12 distinct TTS(6) designs are simply point relabellings of the same underlying block structure. There is just a single, unique TTS(6) design up to isomorphism.

In a classical constructive enumeration, the aim is to produce a complete list of only non-isomorphic designs. Such a list constructed for the TTS(6) designs would therefore contain just a single design, which could be any one of the 12 distinct designs. As will be demonstrated later, the other 11 distinct TTS(6) designs can be generated directly from this single non-isomorphic representative.

The constructive enumeration of a given class of designs can be performed using the incidence matrix backtracking algorithm introduced in Chapter 3. The simplest way of extending the current algorithm to perform this task is to generate all distinct designs as before - but then classify each of these designs for isomorphism. This approach can be implemented by attempting to place each constructed design into a list with the restriction that no two designs in the list can be isomorphic. Each distinct design generated by the backtracking algorithm is tested against every other design in the list to determine whether or not a point mapping, or isomorphism, exists between the two designs. If such a mapping exists, an isomorphic copy of the new design already exists in the list and so it can be discarded. However, if no mapping exists between the new design and any other design already in the list, then it must be added to the list as a new non-isomorphic design. If this process were performed on the TTS(6) designs, the first design generated, which is the lexicographically smallest, would be added to the list and all 11 subsequent designs would be discarded as they are isomorphic to this one.

The isomorphism testing between the generated designs and the designs in the list can be performed in a number of ways. The simplest yet most expensive approach of testing two designs for isomorphism is to examine all possible point mappings to determine whether or not one design can be formed as a relabelling of the other design. Although perfectly valid, this approach is infeasible for large orders, because for designs on  $v$  points there are  $v!$  possible point mappings to be considered.

The next chapter, on constructive enumeration, will describe several methods for efficient isomorphism testing and cover in more detail the process of building a non-isomorphic catalogue. For the purposes of this chapter, which focuses on efficient design construction, it is sufficient to assume that such a list can be constructed from a sequence of distinct designs.

#### **Advantages of Non-Isomorphic Design Classification**

For a given class of designs, there are a number of advantages to constructing a catalogue of only non-isomorphic designs rather than a catalogue of all distinct designs. In particular, there is a significant space saving. A non-isomorphic catalogue will in general be considerably smaller than a catalogue of all distinct designs, particularly as the order of the designs increases, making it much more desirable if the catalogue is to be stored on disc.

There is also a significant time saving. The exhaustive construction algorithm can be considerably optimised if only non-isomorphic designs are to be generated by the use of isomorph rejection techniques. These allow huge sections of the search space to be



eliminated from consideration, leading to enormous reductions in the running time of the algorithm.

In addition to the efficiency of building a non-isomorphic catalogue, all information about the corresponding distinct designs is still retained and they can easily be constructed or simply counted using Burnside's lemma and the automorphism groups of the non-isomorphic designs. This process is described in the next section.

### 4.3 AUTOMORPHISMS

**Definition:**

Let  $D$  be a design on the point set  $V$ , and the block set  $BL$ . An *automorphism* of  $D$  is a mapping,  $\theta : V \rightarrow V$  such that for all  $b \in BL$ ,  $\theta(b) \in BL$

**Definition:**

The set of all automorphisms of a design  $D$ , under the operation of function composition, form a group  $G$ , called the *automorphism group* of  $D$  (or simply the group of  $D$ ).

Clearly, the identity mapping,  $(i \rightarrow i)$  for all  $1 \leq i \leq v$ , belongs to the automorphism group of every design, because it leaves the set of blocks of the design unchanged.

The automorphism group of a design is important because it measures the symmetry of the design. Certain designs may be classified depending on the properties of their automorphism group. For example, a  $(v,k,\lambda)$  BIBD with automorphism group  $G$  is said to be:

- cyclic                    if  $G$  contains a cycle of length  $v$
- $k$ -rotational        if some automorphism of  $G$  has one fixed point and  $k$  cycles each of length  $(v-1)/k$
- transitive            if for any two points,  $x$  and  $y$ , there is an automorphism mapping  $x$  to  $y$

As an example, consider again the unique TTS(6) design. The mapping:

$$\alpha : (1 \rightarrow 2) (2 \rightarrow 6) (3 \rightarrow 3) (4 \rightarrow 5) (5 \rightarrow 1) (6 \rightarrow 4)$$

is an automorphism of this design, and its effect on the set of blocks of the design is illustrated below:

Consider the lexicographically smallest labelling of the unique TTS(6) design:	Below is the resulting block list from applying the mapping, $\alpha$ :	After sorting the relabelled design, the completely ordered block list becomes:
1 2 3	2 6 3	1 2 3
1 2 4	2 6 5	1 2 4
1 3 5	2 3 1	1 3 5
1 4 6	2 5 4	1 4 6
1 5 6	2 1 4	1 5 6
2 3 6	6 3 4	2 3 6
2 4 5	6 5 1	2 4 5
2 5 6	6 1 4	2 5 6
3 4 5	3 5 1	3 4 5
3 4 6	3 5 4	3 4 6

The point mapping above is clearly an automorphism of the TTS(6) design, because it maps the design exactly back onto itself. The set of all different point mappings with this property constitute the automorphism group of the design.

There are exactly  $6! = 720$  different possible point mappings on 6 points. An examination of the effect of each one on the unique TTS(6) design reveals that exactly 60 of the 720 distinct

point mappings map this design onto itself, and thus are automorphisms of the design. Therefore, the automorphism group size of the unique TTS(6) design is 60.

As the order,  $v$ , of the designs increases, generating the automorphism group by examining the effect of each of the  $v!$  possible point mappings is not feasible. The efficient calculation of automorphism groups is a difficult task, and along with related isomorphism problems has undergone a great deal of research. Accepted as the most practical tool for graph, and equivalently design, isomorphism is the nauty software developed by McKay [48], which includes procedures for determining the automorphism group of a vertex-coloured graph. Automorphism groups are required extensively in this thesis, and an algorithm for fast group generation has been developed which updates the algorithm of Gibbons in [32]. This process is described in detail in Section 4.5.3.

#### Computing the Number of Distinct Designs

For any given class of designs, the total number of distinct designs can be calculated from a complete catalogue of non-isomorphic designs using *Burnside's lemma*. The formula for the number of distinct designs,  $D$ , is given below:

$$D = \sum_{i=1}^{N_d} \frac{v!}{G_i}$$

where  $N_d$  is the number of non-isomorphic designs,  $v$  is the order of the designs, and  $G_i$  is the automorphism group size of the  $i^{\text{th}}$  non-isomorphic design.

For example, the complete catalogue of non-isomorphic TTS(6) designs consists of a unique design with an automorphism group size of 60. Using the formula of Burnside's Lemma, the number of distinct TTS(6) designs is calculated below:

$$\begin{aligned} &= \sum_{i=1}^1 \frac{720}{G_i} \\ &= \frac{720}{60} \\ &= 12 \end{aligned}$$

This in fact agrees exactly with the number of distinct TTS(6) designs which were generated by the exhaustive backtracking algorithms of the previous chapter.

## 4.4 ISOMORPH REJECTION

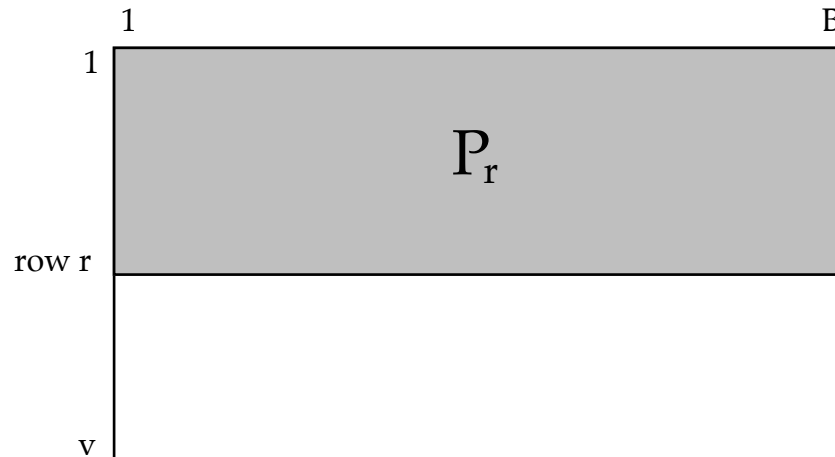
The backtracking algorithms developed in the previous chapter were designed to generate all distinct designs of a given class using a number of feasibility properties, or optimisations, to reduce the amount of searching performed. The purpose of the feasibility properties was to reject partial designs which could not be extended to complete solutions.

Now that only non-isomorphic designs are to be constructed, the feasibility properties can be greatly strengthened. Once a design,  $D$ , has been generated and classified in the non-isomorphic catalogue, there is no need to generate any other design which belongs to the isomorphism class of  $D$  - which is the set of all designs to which  $D$  is isomorphic. Any design generated after the construction of  $D$ , which is isomorphic to it, is simply discarded and so any time spent constructing such designs is wasted.

The feasibility properties can also be extended to reject any partial design which is isomorphic to another partial design that has already been generated earlier in the search. In effect, this eliminates from consideration large chunks of the search space which are isomorphic to pieces already searched, and can lead to enormous improvements in efficiency. However, the implementation of such *isomorph rejection* techniques is not trivial.

Consider the point by point backtracking algorithm which exhaustively generates the incidence matrices of a given class of designs. At some point in the algorithm, a partial design is constructed validly down to some row of the incidence matrix. All partial and complete designs are constructed in increasing lexicographical order, so if a mapping exists which maps the current partial design to a lexicographically smaller partial design, then the current partial design can be rejected and no extensions to it need to be considered. This is because it is isomorphic to a partial design already constructed, and all extensions to it will be isomorphic to the extensions already performed earlier in the search.

For example, the incidence matrix for a partial design,  $P_r$ , validly constructed down to row  $r$ , is given below:



To perform isomorph rejection on  $P_r$ , it is necessary to determine whether or not there exists a mapping of the points corresponding to the first  $r$  rows which map  $P_r$  to another partial design which is lexicographically less than  $P_r$ . If such a mapping exists, then  $P_r$  is isomorphic to a partial design which was considered earlier in the search.  $P_r$  can therefore be rejected from consideration and backtracking can begin on row  $r$ . If  $P_r$  is not rejected, then all extensions to it would be isomorphic to extensions already examined earlier in the search. Moreover, if any extensions of  $P_r$  led to a complete design, this design would be isomorphic to a previous design already constructed, and could not be added to a non-isomorphic catalogue.

A simple way of performing such a test, would be to examine the effect of all possible point mappings on the first  $r$  rows of the incidence matrix, and see if one of these mappings produced a partial design which was lexicographically less than  $P_r$ . If so, then  $P_r$  would be rejected, pruning all extensions of it from the search space. However if such a mapping did not exist, then  $P_r$  would be the lexicographically smallest partial design in its isomorphism class, and so could not be rejected. Such an approach can be termed *total isomorph rejection*, because at each level of the search all isomorphs are rejected from consideration. As a result, if the rejection was performed right up to the last row of the incidence matrix, exactly one design, the lexicographically smallest, would be generated from each isomorphism class.

Although ideal, a naive implementation of such total isomorph rejection becomes impractical as the order of the designs increases. An efficient implementation of this technique is described in Chapter 5, when canonicity testing is introduced. However, for many designs, even an efficient total isomorph rejection strategy is too costly for certain stages of the search, and a much quicker although slightly less effective method of isomorph rejection is required. The development of such *partial isomorph rejection* techniques in which a large number, but not all, isomorphs can be quickly rejected, is the focus of this section. The first technique, presented in Section 4.4.1, enforces an ordering on the rows of the incidence matrix which is very effective at rejecting isomorphs and which in fact leads to the second technique described in Section 4.4.2 which greatly limits the

amount of backtracking performed on each row. The final, very powerful technique, makes use of the automorphism groups of the partial designs and is outlined in Section 4.4.3.

#### 4.4.1 ROW ORDERING

The idea behind isomorph rejection in general is the detection of point mappings which map a partial design to a lexicographically smaller one, which has therefore already been considered in the search.

This notion allows an ordering to be enforced upon the rows of the incidence matrix at all times. In particular, any row  $r$  must be lexicographically greater than all rows  $i$ , for  $1 \leq i < r$ . Two rows are compared lexicographically using their corresponding cell fill vectors.

The reason for enforcing this ordering is illustrated by considering a partial design with row  $r$  lexicographically less than row  $q$ , for some  $q < r$ . In this case, a point mapping which swapped rows  $r$  and  $q$ , but fixed all the other rows of the incidence matrix, would map the matrix to a lexicographically smaller partial design.

For example, consider the partial incidence matrix: for a TTS(9) design:

```
Row1: 111111110000000000000000
Row2: 110000001111110000000000
Row3: 101000001000001111100000
Row4: 010100000100001100011100
Row5: 000110000011001010000011
Row6: 001001000000110010011010
Row7: 000000110010100101010001
Row8: 000010101000010000101101
Row9: 000001010101000001100110
```

When this design is divided into the cell structure imposed by the first 4 rows, there are exactly 12 cells:

```
Row 1:  1  1  1  1 1111 0 0 0000 00 000 000 00
Row 2:  1  1  0  0 0000 1 1 1111 00 000 000 00
Row 3:  1  0  1  0 0000 1 0 0000 11 111 000 00
Row 4:  0  1  0  1 0000 0 1 0000 11 000 111 00
Row 5:  0  0  0  1 1000 0 0 1100 10 100 000 11
Row 6:  0  0  1  0 0100 0 0 0011 00 100 110 10
Row 7:  0  0  0  0 0011 0 0 1010 01 010 100 01
Row 8:  0  0  0  0 1010 1 0 0001 00 001 011 01
Row 9:  0  0  0  0 0101 0 1 0100 00 011 001 10
```

The cell fill vector for row 5 under this cell structure is:

```
0 0 0 1 1 0 0 2 1 1 0 2
```

The cell fill vector for row 6 under this cell structure is:

```
0 0 1 0 1 0 0 2 0 1 2 1
```

Row 6 has a greater cell fill vector than row 5, under the cell structure of the first 4 rows, and is therefore lexicographically less than row 5. If the point mapping:

```
(1→1) (2→2) (3→3) (4→4) (5→6) (6→5) (7→7) (8→8) (9→9)
```

is applied to the incidence matrix above, rows 5 and 6 would be swapped, and all other rows would remain unchanged. Once the resulting matrix is completely ordered, the relabelled design would be lexicographically less than the original. In other words, the above partial design is isomorphic to a lexicographically smaller design.

An implementation of this partial isomorph rejection would check the row ordering property as the rows were being constructed. No point would ever be placed in a row which would make it lexicographically less than any previous row, under the corresponding cell structure of that row. In the example above, the first point on row 6, which is in cell 3 of the cell structure, would never be placed. Instead, a row ordering violation would be detected and the next possible block in which the point could be placed would be examined.

This strategy has very little overhead, because the check is only necessary when a row is being extended, or initialised, for the first time. Once a row has been extended, and is lexicographically greater than all previous rows, all backtracking performed on the row only makes it lexicographically larger, and so it can never become less than some previous row.

A very important observation can be made directly from the definition of this heuristic, and it leads to another extremely powerful isomorph rejection constraint.

For any class of designs with block size  $k$ , the first block in every generated design must be  $\{1,2,\dots,k\}$ . In other words, the first column of the incidence matrix of every design must consist of exactly  $k$  1's in the first  $k$  rows, and 0's in the remaining rows. For example, consider the TTS(6) design below:

```

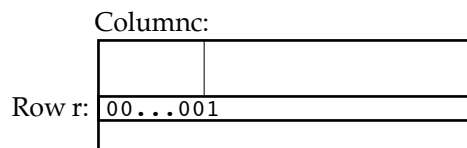
1111100000
1100011100
1010010011
0101001011
0010101110
0001110101
    
```

The first column should never be changed as the backtracking algorithm progresses. Otherwise, if the first column did change, then not all of the first  $k$  rows would contain a point in the first column. In order to complete the first block of the design, some row  $r$ , for  $r > k$  would need to contain a 1. A point mapping which swapped row  $r$  with the first row not containing a point in the first column, would map the partial design to a lexicographically smaller one. This observation can in fact be made more general, and leads to the second of the partial isomorph rejection strategies, backtrack halting.

### 4.4.2 BACKTRACK HALTING

As a direct extension of the previous observation, the first point placed on any row need never be shifted further to the right during the backtracking procedure. In other words, as soon as backtracking reaches the first point on any row, it can halt, and the row can be rejected.

The justification for this constraint is very similar to the previous argument. For a given row,  $r$ , being backtracked, let column  $c$  be the column which contains the first point on row  $r$ . This is illustrated below:



If the backtracking removed the point from column  $c$ , and shifted it further to the right, then the block represented by column  $c$  must become incomplete, ie. column  $c$  must contain less than  $k$  points, regardless of whether or not it was complete before the shift.

Therefore, if the partial design is to extend to a complete design, at least one further point must be placed in column  $c$  by some row  $q$ , where  $q > r$ . All the points in row  $r$  following the shifting of the point from column  $c$ , lie strictly to the right of this column - yet row  $q$  contains a point in column  $c$ . This violates the row ordering, and a mapping which swaps rows  $r$  and  $q$  would therefore construct an isomorphic, lexicographically smaller, design.

This constraint also enforces that the first block of every generated design must be  $\{1,2,\dots,k\}$ , because no point can be shifted out of the first column during backtracking.

In general, the effectiveness of the different construction optimisations and isomorph rejection constraints depends largely on the design being constructed. The backtrack halting constraint just described performs well on all designs, because there is very little overhead in performing the test and it will prune a large portion of the search space for almost any design.

For example, consider the 12 distinct TTS(6) designs which were constructed using the point by point incidence matrix backtracking algorithm. Only the first 6 of these contain the block {1,2,3}, so the backtrack halting constraint by itself would prune at least half of the search space.

For a much more startling improvement, the effect of the backtrack halting constraint on the incidence matrix backtracking algorithm has been examined for the symmetric 2-(19,19,9,9,4) designs. There are exactly 6 non-isomorphic 2-(19,19,9,9,4) designs, with the following group sizes:

<i>The 2-(19,19,9,9,4) Designs</i>	Design #1	Design #2	Design #3	Design #4	Design #5	Design #6
<i>Automorphism Group Size</i>	6	8	9	24	72	171

The number of distinct 2-(19,19,9,9,4) designs can be calculated using Burnside's Lemma, yielding an astronomical 56,465,379,210,240,000 distinct designs. This is an excellent example of a case where the enumeration of all distinct designs is not possible, yet the exact number of such designs can be computed from the corresponding catalogue of non-isomorphic designs, which can be generated quickly.

The point by point incidence matrix backtracking algorithm can be used to generate the 2-(19,19,9,9,4) designs, from which a non-isomorphic catalogue can be constructed. One of the final versions of the algorithm developed in this thesis has been used to generate these designs with and without the backtrack halting constraint. The particular level of rejection used in the search meant that exactly 86 complete designs were generated. From these, the 6 non-isomorphic designs could be classified. The execution time required to exhaustively generate these designs with and without the backtrack halting constraint is given in the table below:

	Without backtrack halting	With backtrack halting
Execution Time (secs)	2052.5	1.54

The use of the backtrack halting constraint for this particular class of designs clearly makes an enormous difference.

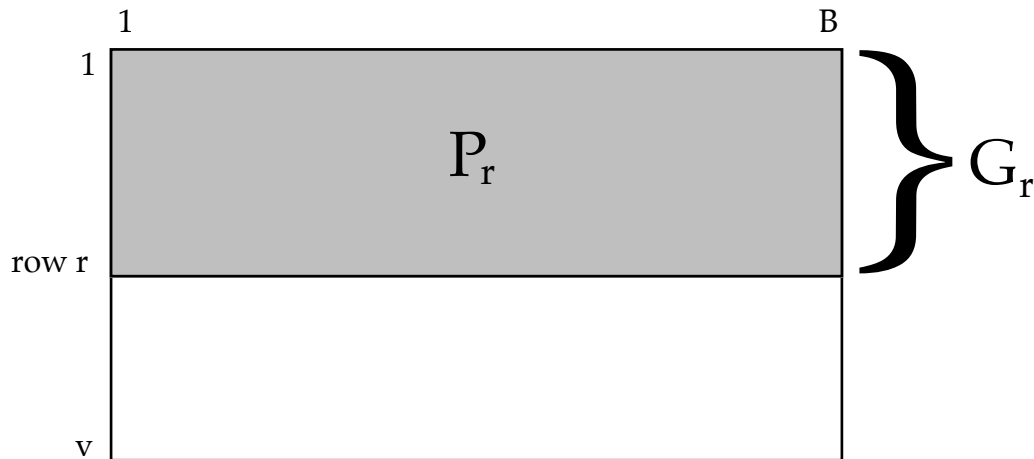
In fact, in many cases, the correct use of even relatively simple constraints can determine the feasibility of a particular enumeration, depending on how effectively unnecessary search is avoided. For example, this was the case for the enumeration of the 3-(11,5,4) designs, which was performed for the first time in this thesis and is described in more detail in Case Study Six of Chapter 6. The backtrack halting constraint was in fact necessary in order for this enumeration to be completed.

The final partial isomorph rejection technique to be discussed in this chapter is more complicated than the previous two. It makes use of the automorphism groups of the partial designs, and more specifically of their actions on the incidence matrix. This strategy is presented next, in Section 4.4.3.

### 4.4.3 THE ROLE OF AUTOMORPHISM GROUPS

The automorphism group of a partial design can be used to implement a very powerful form of partial isomorph rejection. Not only is it very effective at pruning isomorphic sections of

the search space but it can be calculated very rapidly. Consider the following incidence matrix, which has been completed down to row  $r$ .



Let  $P_r$  be the partial design represented by the first  $r$  rows of the incidence matrix, and let  $G_r$  be the automorphism group of  $P_r$ .  $G_r$  is therefore the set of point mappings which map  $P_r$  back onto itself. As illustrated in Section 4.2, the effect of a point mapping on a given partial design is to reorder the rows of the incidence matrix. If a given point mapping is an automorphism of the partial design and the mapping is applied to permute the rows of the incidence matrix, then a particular set of column permutations must also exist which reconstruct the original incidence matrix exactly.

In terms of  $P_r$  then,  $G_r$  is the set of all row permutations of the first  $r$  rows of the matrix, which when applied to  $P_r$  generate a row permuted partial incidence matrix  $P_r'$ , for which a set of column permutations exist which reconstruct  $P_r$  exactly from  $P_r'$ . These column permutations are of particular importance and are in fact the only information which is required to perform this partial isomorph rejection technique. However, calculating the column permutations requires calculating the automorphism group of the design, and this is not a straightforward task.

**Column Reordering**

To illustrate the reorderings of the columns which are required to reconstruct an original incidence matrix once its rows have been permuted by an automorphism, consider the partial TTS(6) design below, constructed to row 3:

	Columns:
	1234567890
Row 1:	1111100000
Row 2:	1100011100
Row 3:	1010010011

The identity mapping (1→1) (2→2) (3→3) is obviously an automorphism of this partial design. The partial incidence matrix which results after applying the identity automorphism is exactly the same as the original, and since the design has not changed, the set of column permutations which reconstruct the original design is also the identity:

(1→1) (2→2) (3→3) (4→4) (5→5) (6→6) (7→7) (8→8) (9→9) (0→0)

As a less trivial example, consider the next automorphism in the group of this partial design:

(1→1) (2→3) (3→2)

This automorphism maps point 2 onto point 3 and vice versa. The effect this has on the partial incidence matrix is to swap rows 2 and 3, which results in the matrix at the top of the following page:

```

Columns:
1234567890
Row 1: 1111100000
Row 2: 1010010011
Row 3: 1100011100

```

To return this row permuted matrix to the original, the following column permutations must be applied:

(1→1) (2→3) (3→2) (4→4) (5→5) (6→6) (7→9) (8→0) (9→7) (0→8)

The diagram below illustrates these column operations:

	Column:	1	2	3	4	5	6	7	8	9	0
Rowpermuted incidencematrix:	Row 1:	1	1	1	1	1	0	0	0	0	0
	Row 3:	1	0	1	0	0	1	0	0	1	1
	Row 2:	1	1	0	0	0	1	1	1	0	0
Columnpermuted, reconstructed incidencematrix:	Row 1:	1	1	1	1	1	0	0	0	0	0
	Row 2:	1	1	0	0	0	1	1	1	0	0
	Row 3:	1	0	1	0	0	1	0	0	1	1

### Cell Reordering

A disadvantage of considering complete column permutations is that for a given row permuted partial incidence matrix, there may be several distinct column reorderings which could reconstruct the design. This is simply because any set of  $n$  adjacent identical columns can be reordered in  $n!$  ways - each way producing exactly the same matrix. Such a set of columns is called a cell, and was introduced in Section 3.4.3.4.

Therefore, instead of column permutations, considering the unique cell permutation which reconstructs a given row permuted incidence matrix leads to a more concise and less ambiguous representation.

For example, consider the same initial design again, this time divided at cell boundaries.

```

Cells:    1 2 3 4 5 6 7
Row 1:    1 1 1 11 0 00 00
Row 2:    1 1 0 00 1 11 00
Row 3:    1 0 1 00 1 00 11

```

The 10 columns are divided into 7 distinct cells. It is obvious that any given row permutation will preserve the cell structure. Consider once again the effect of the automorphism:

(1→1) (2→3) (3→2)

which swaps rows 2 and 3:

```

Cells:    1 2 3 4 5 6 7
Row 1:    1 1 1 11 0 00 00
Row 2:    1 0 1 00 1 00 11
Row 3:    1 1 0 00 1 11 00

```

Now for this automorphism, the unique cell permutation which reconstructs the incidence matrix is:

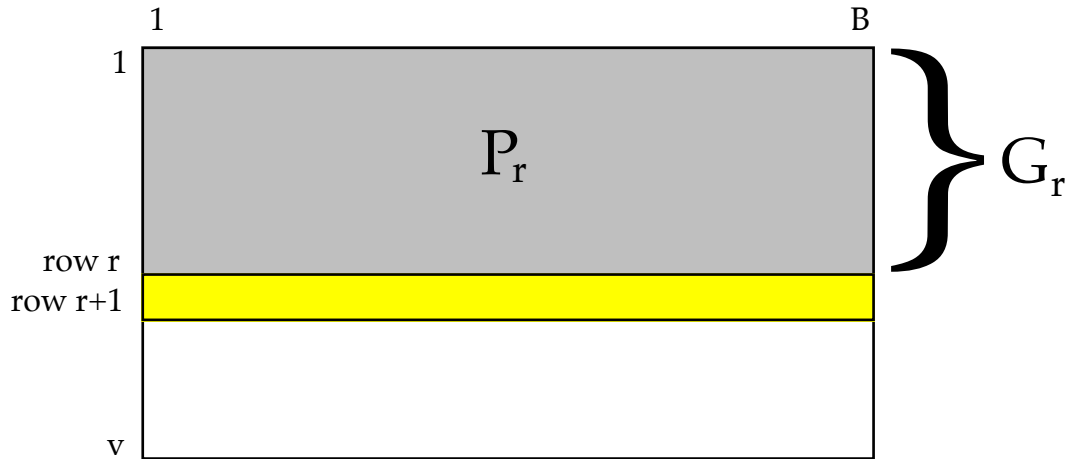
(1→1) (2→3) (3→2) (4→4) (5→5) (6→7) (7→6)

### RowRejection

The partial isomorph rejection technique can now be described. Consider again the partial incidence matrix,  $P_r$ , constructed validly down to row  $r$ , with automorphism group  $G_r$ .



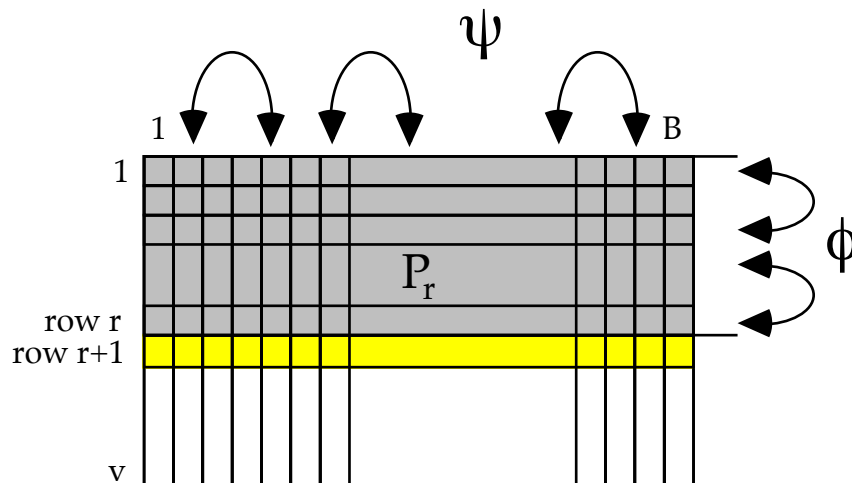
Now assume that  $P_r$  is extended, and that a valid row  $r+1$  is constructed which does not violate any of the design constraints or any of the other optimisations. For example, the number of points on row  $r+1$  must equal the replication number of the design, it must intersect every other row in exactly  $\lambda$  places, and it must be lexicographically larger than all previous rows.



The idea behind the strategy is to use the group of the first  $r$  rows to determine whether or not the partial incidence matrix on the first  $r+1$  rows is isomorphic to a lexicographically smaller partial incidence matrix. If it is, then row  $r+1$  can be rejected, and all extensions to the above partial incidence matrix can be eliminated from the search. Particularly if the row is at a shallow level, the amount of searching which can be avoided if the row is rejected can be very substantial.

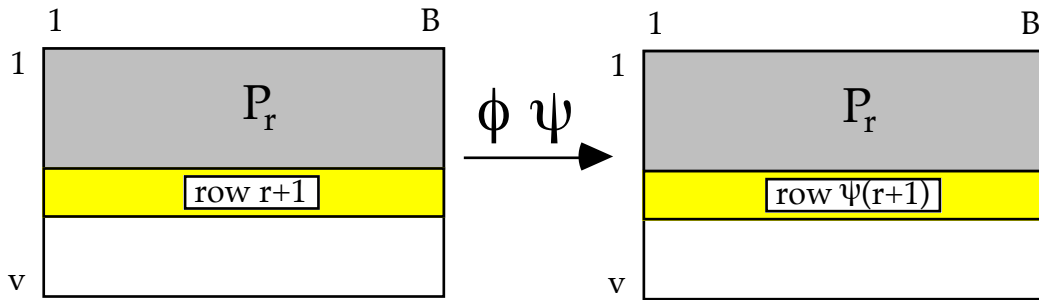
Every point mapping belonging to the group  $G_r$  will permute the first  $r$  rows of the design, producing a new partial incidence matrix,  $P_r'$ , on the first  $r$  rows. In addition, for all permuted partial designs  $P_r'$ , there exists a unique permutation of the cells which reconstruct  $P_r$ .

Let  $\phi$  be an automorphism from the group  $G_r$ , and let  $\psi$  be the unique cell permutation such that  $\psi(\phi(P_r)) = P_r$ . Schematically, this can be represented as:



Now if the mapping  $\phi$  is performed, followed by  $\psi$ , then the partial incidence matrix  $P_r$  will be completely unchanged. However, the cell permutation,  $\psi$ , permutes the entire columns of the incidence matrix, and hence the structure of the new row  $r+1$  may be changed. Under  $\psi$ , the current point placement of row  $r+1$  would map to a new configuration,  $\psi(r+1)$ , which must then be compared lexicographically to the original point placement of row  $r+1$ . This comparison would be performed by calculating the cell fill vector of both rows  $r+1$  and

$\psi(r+1)$  under the cell structure imposed by the first  $r$  rows of the incidence matrix. Row  $\psi(r+1)$  would be lexicographically less than row  $r+1$  if its cell fill vector was greater than the corresponding cell fill vector of row  $r+1$ .

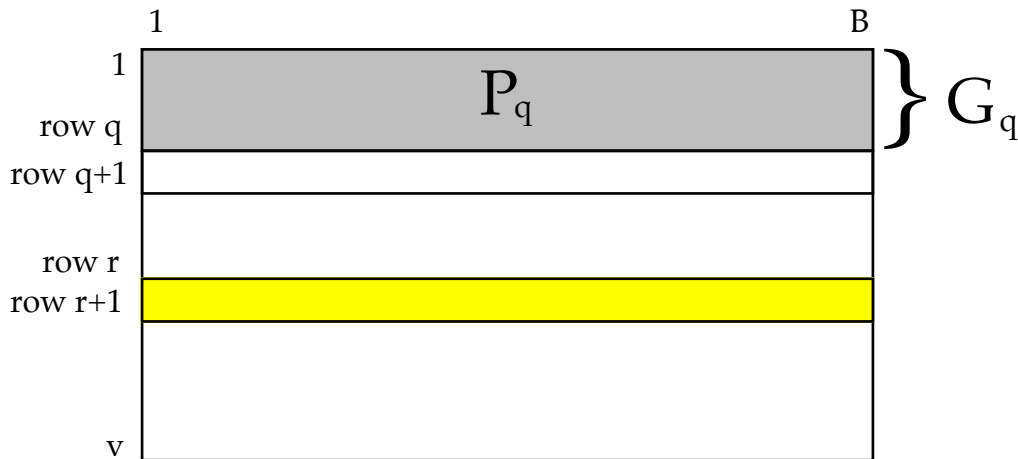


If row  $\psi(r+1)$  is lexicographically less than row  $r+1$ , then the mapping  $\phi$ , along with the point mapping  $((r+1) \rightarrow \psi(r+1))$ , clearly maps the current partial design on the first  $r+1$  rows to a lexicographically smaller configuration.

In other words, the partial design consisting of  $P_r$  with the extension of row  $r+1$  is isomorphic to a lexicographically smaller partial design, consisting of  $P_r$  with the extension of row  $\psi(r+1)$ , which consequently has been considered at an earlier stage of the ordered search. Once this is detected, row  $r+1$  is rejected, and backtracking can begin immediately.

The basic idea above can be trivially extended to not only use the group of the first  $r$  rows, but in fact to use the group information of any initial number of rows in order to test whether a newly constructed row  $r+1$  can be rejected.

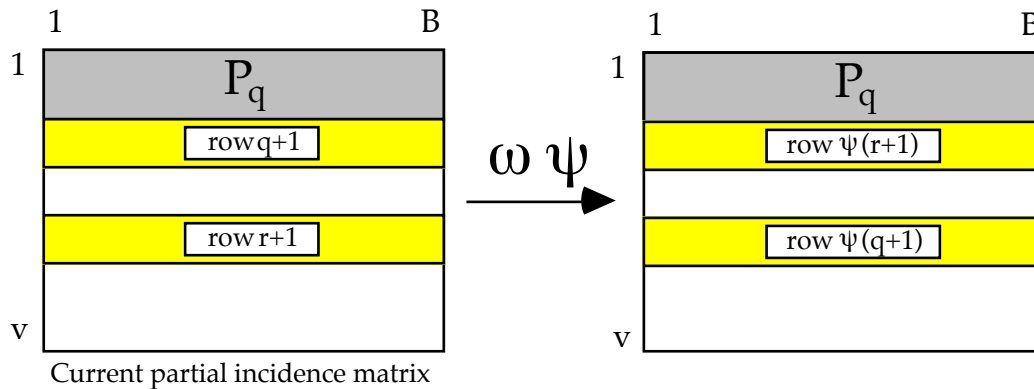
Assume that the current partial incidence matrix below has been validly constructed down to row  $r$ , and that a potential row  $r+1$  has just been generated. Let  $G_q$  denote the automorphism group of the partial design on the first  $q$  rows,  $P_q$ , where  $q < r$ . If the corresponding cell permutations of an automorphism in  $G_q$  maps row  $r+1$  to a lexicographically smaller configuration than row  $q+1$ , then row  $r+1$  can be rejected.



More formally, let  $\phi$  be an automorphism of the group  $G_q$  which permutes the first  $q$  rows of the partial incidence matrix,  $P_q$ , in some way. Let  $\psi$  be the set of cell permutations which reconstruct  $\phi(P_q)$  to its original form. Assume that the permutation of the columns defined by  $\psi$  maps row  $r+1$  to a lexicographically smaller configuration than row  $q+1$ , under the cell structure of the first  $q$  rows of the incidence matrix. Then it can easily be shown that the current partial incidence matrix on the first  $r+1$  rows is isomorphic to a lexicographically smaller partial incidence matrix, by defining an isomorphism which has the following effect on the rows of the matrix:

- Rows  $\{1, 2, \dots, q\}$  are mapped according to  $\phi$
- Row  $(q+1) \rightarrow \text{row } (r+1)$
- Rows  $\{q+2, q+3, \dots, r\}$  are mapped amongst themselves in any arbitrary fashion
- Row  $(r+1) \rightarrow \text{row } (q+1)$

Let this mapping be denoted by  $\omega$ . Essentially,  $\omega$  applies the automorphism  $\phi$  to the first  $q$  rows of the design, and interchanges row  $q+1$  and row  $r+1$ . Its effect on the other rows of the incidence matrix are unimportant. Recall that  $\psi$  is the set of cell permutations which restore the first  $q$  rows of the incidence matrix following the application of the automorphism  $\phi$ .

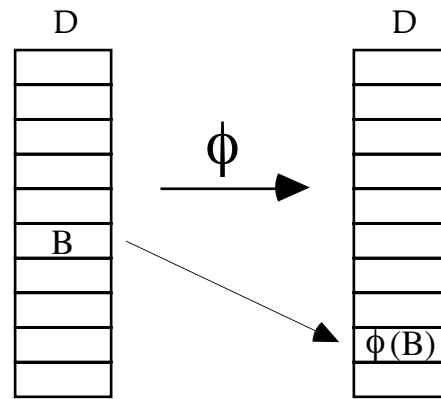


After the row permutations of  $\omega$  and the column permutations of  $\psi$  have been applied, the first  $q$  rows of the partial incidence matrix,  $P_q$  will be exactly the same as before, because the mapping of these rows is an automorphism of  $G_q$  and the columns of the matrix have been ordered once again.

The next row in the mapped partial incidence matrix is  $\psi(r+1)$ , as illustrated in the diagram above. If this row is lexicographically less than row  $q+1$ , then the current partial incidence matrix, mapped by the isomorphism  $\omega$ , is isomorphic to a lexicographically smaller partial incidence matrix, so it can be rejected, and backtracking can immediately begin on row  $r+1$ .

The isomorph rejection technique described above is very effective, but one which can only be fully exploited if the automorphism groups of the partial designs can be constructed efficiently. Each time a new valid partial incidence matrix is constructed, its group must be calculated, from which the corresponding cell permutations can be generated, which can then be used to reject isomorphic extensions. The automorphisms themselves need not be stored, for it is only the effect of the cell permutations on the rows being tested which are important. For a partial incidence matrix validly constructed to row  $r$ , these cell permutations are stored for each partial design on the first  $i$  rows of the incidence matrix,  $P_i$ , for  $2 \leq i \leq r$ . Each set of cell permutations are tested in an attempt to reject extensions to the current partial design.

In order to calculate the complete set of cell permutations for a given partial design,  $D$ , each automorphism in the group of  $D$  must be explicitly constructed. As indicated by the illustration at the top of the following page, this requires generating all point mappings  $\phi$ , such that when  $\phi$  is applied to any block,  $B$ , in  $D$ , the resulting block,  $\phi(B)$ , is also contained in  $D$ .



For a partial design on  $v$  points, a simple implementation would test all  $v!$  possible permutations, returning those which preserved the blocks. However, such an implementation has obvious limitations, especially considering that this operation must be performed many times during the course of a search.

The number of mappings which must be tested can be considerably reduced by performing an initial analysis, which partitions the points of the design into classes, such that only point mappings in which the target and the source points belong to the same class need to be considered. This technique is combined with a backtracking algorithm for the systematic construction of all automorphisms of a partial design, and is described in detail in the next section.

## 4.5 AUTOMORPHISM GROUP GENERATION

The group generation algorithm implemented in this thesis is based on the algorithm presented by Gibbons in [32]. A series of partial automorphisms are constructed and extended a point at a time, using an exhaustive backtracking approach which generates each complete automorphism belonging to the group.

For example, at a particular stage of the algorithm, the partial automorphism may be defined on the first 3 points as  $(1 \rightarrow 1) (2 \rightarrow 8) (3 \rightarrow 5)$ :

Partial Automorphism,  $\emptyset$

Source points:	1	2	3	4	5	...	$v$
Target points:	↓ 1	↓ 8	↓ 5				

The source points and the target points are the points mapped from and mapped to, respectively, in the current partial automorphism.

A partial automorphism,  $\emptyset$ , is said to be *incidence preserving* when for any subset,  $S$ , of the source points already mapped, if  $S$  is contained in a block of the design, then  $\emptyset(S)$  is also contained in a block of the design. The backtracking algorithm only extends the current partial automorphism,  $\emptyset$ , by adding a new (point  $\rightarrow$  point) mapping if  $\emptyset$  is incidence preserving. If at any stage, the current partial isomorphism is not incidence preserving, then any extension to it of additional point to point mappings will also not be incidence preserving. Clearly a valid automorphism of any design must be incidence preserving.

Let  $D$  be the partial design on  $v$  points which is to have its group calculated. To reduce the size of the search space for this backtracking algorithm, the points are partitioned into classes such that the only point mappings which need to be considered are those for which the source and the target points belong to the same class. If the sizes of the classes are small enough, the number of combinations of point mappings which need to be considered falls

drastically. Not only do the partitions reduce the size of the search space, but they also provide an elegant way of testing whether a given partial mapping is incidence preserving. The next section describes the formulation and uses of the partitions in detail.

### 4.5.1 PARTITIONS

An *invariant* of a design is a property of the design which remains unchanged under isomorphism. Relabelling a given design by applying a mapping to the points of the design in no way affects its invariant properties, which represent some labelling-independent, structural information. As a consequence, any two designs which are isomorphic necessarily share the same invariants.

To compute an invariant, some subsystem of the design can be defined, such as a subset of the blocks satisfying some property. Then, for each point of the design, the number of subsystems to which the point belongs is counted. If  $M$  is the maximum number of subsystems to which an individual point belongs, then an  $M$ -element vector  $INV$  can be constructed such that  $INV_i$  is the number of points belonging to  $i$  subsystems. The vector  $INV$  is an invariant of the design, for no matter what point mapping is applied to the design, the frequency of points belonging to a given number of subsystems cannot change.

Invariants themselves are extremely valuable to the process of building non-isomorphic catalogues, and Chapter 5 studies this in greater detail. For the purpose of this section, invariants are useful for constructing partitions of the points and blocks of a design which assist the group generation process. Two points belong to the same class of a partition if and only if they belong to exactly the same number of subsystems. In an analogous way, a partitioning of the blocks can be induced.

For any individual point mapping ( $a \rightarrow b$ ) of an automorphism of the design, the points  $a$  and  $b$  must belong to the same class of the partition. If they did not, then once the mapping was applied to the design, point  $b$  would no longer belong to the same number of subsystems as it did in the original design, and so the mapping could clearly not be an automorphism.

Similarly, for any block,  $b$ , of the original design which is mapped to a new block,  $b'$ , by an automorphism, both  $b$  and  $b'$  must belong to exactly the same class of the block partition. If they were in different classes, then block  $b'$  would belong to a different number of subsystems in the mapped design than it did in the original design, and thus the mapped design could not be the same as the original.

It is worth noting that the previous two observations also have implications in terms of the isomorphism of two designs, say  $A$  and  $B$ . If an isomorphism exists between these designs, then the point partition of  $A$  must contain exactly the same number of classes, of the same sizes, as the point partition of  $B$ . Similarly the block partitions of  $A$  and  $B$  must contain the same number of classes, of the same sizes. In other words, two designs can only be isomorphic if their point and block partitions are in one-to-one correspondence. These ideas are put to use in Chapter 5, as part of the design isomorphism algorithm.

The automorphism group generation backtracking algorithm relies on the following three partitions:

- *a partition of the blocks*
- *a partition of the points*
- *a partition of the point pairs*

These partitions are generated by counting a particular type of subsystem within the block intersection graph of the design. The *block intersection graph*,  $G_v$ , is a graph in which the nodes are the blocks of the design, and two nodes are adjacent if and only if their corresponding blocks contain exactly  $i$  points in common. Isomorphism of block intersection graphs is a necessary condition for isomorphism of the corresponding designs. As a result,

any invariant computed for the block intersection graph of a design also provides an invariant for the design itself.

#### 4.5.1.1 CLIQUE ANALYSIS

The subsystems within the block intersection graph, which have been selected in the implementation of this thesis to compute the partitions, are cliques of varying sizes. A clique is a subset of the nodes of the graph which are fully connected, ie. in which each node within the clique is adjacent to every other node within the clique. To establish the partitions, a clique analysis of the block intersection graph,  $G_i$ , is performed using an exhaustive backtracking algorithm. This analysis counts, for each block represented by a node of  $G_i$ , the number of cliques less than or equal to size  $c$  to which the block belongs. This process is called a  $(c,i)$ -clique analysis and was first employed by Rudi Mathon, whose algorithm was published in [32]. An outline of the algorithm implemented in this thesis for performing such a  $(c,i)$ -clique analysis, which is based on the original algorithm of Mathon, is presented next.

The data structures used in the algorithm are summarised in the following table:

Data Structure	Purpose
CV[i][j]	A matrix used to indicate the number of $j$ -cliques containing vertex $i$ for the block intersection graph.
Current_Clique	A set containing all the nodes in the block intersection graph, $G_i$ , which belong to the current clique being examined.
Current_Block	The identifier for the current node of the block intersection graph which is having its clique properties calculated.
MAX_CLIQUÉ_SIZE	Specifies the maximum clique size to be analysed. This is the value of $c$ in the $(c,i)$ -clique analysis.
B	The number of blocks in the design, and hence the number of nodes in the block intersection graph of the design. Each node has a unique identifier from 1 to B, and this is therefore the range of possible values for Current_Block.

The backtracking algorithm is given on the next page. The actual representation of the block intersection graph,  $G_i$ , of the corresponding design is not specified. Any common graph representation in which the adjacency of two nodes can be tested efficiently is adequate.

```

// Initialise the structures.
for each node, j, of the block intersection graph, Gi
  for all allowable clique sizes, i, from 1 to MAX_CLIQUE_SIZE
    CV[j][i] = 0;
Current_Block = 0;      // The first valid identifier for
                       // Current Block is 1, and this is
                       // incremented by the algorithm
i = 1;                 // Initially look for a clique of size 1
Current_Clique = empty; // The current clique is initialised to be
                       // empty.

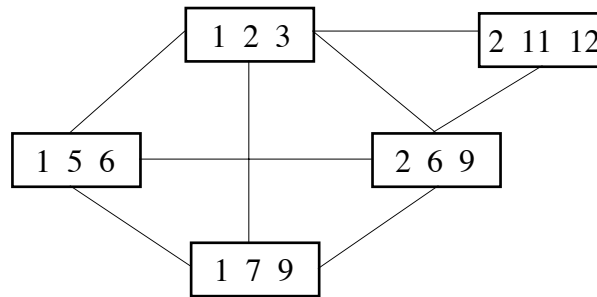
// Perform the clique analysis:
do forever:
  while (Current_Block < B)
    begin
      Current_Block++;
      if (Current_Block is adjacent to every block in the
          Current_Clique, then it can be added to the
          Current_Clique, to produce a larger clique)
        begin
          Add Current_Block to the Current_Clique
          for every point, h, in the new clique
            CV[h][i]++;
          // if the new clique is not yet the maximum clique size, then
          // repeat the process with the next possible clique size
          if (i < MAX_CLIQUE_SIZE)
            i++;
          end
          // Now, all possible Current_Blocks have tried to be added to
          // a clique of size i.
          // Backtrack to search for the next (i-1) clique.
          i--;
          if (i > 0)
            Current_Block = last block added to the Current_Clique.
            This will be incremented on the next iteration so that
            the Current_Block becomes the very next block which was
            not involved in the current clique
          else
            exit;
        end
    end

```

The results of the clique analysis of the block intersection graph immediately induces a partition on the blocks of the design. The point and point pair partitions are then derived from the initial block partition. The next three sections, 4.5.1.2, 4.5.1.3 and 4.5.1.4, describe the construction of each of the partitions in relation to the following example design:

{1, 2, 3}
{1, 5, 6}
{1, 7, 9}
{2, 6, 9}
{2, 11, 12}

The graph at the top of the facing page is the corresponding block intersection graph  $G_1$ , for this design. In  $G_1$ , the nodes correspond to the blocks of the design, and two nodes are adjacent if and only if their corresponding blocks contain exactly 1 point in common:



### 4.5.1.2 BLOCK PARTITION

Each node in the block intersection graph corresponds to a block of the design, and following the clique analysis, the number of cliques of varying sizes containing each node is known. This therefore partitions the corresponding blocks of the design into classes such that only blocks with identical clique properties belong to the same class.

In the example block intersection graph given at the end of Section 4.5.1.1, blocks  $\{1, 2, 3\}$  and  $\{2, 6, 9\}$  belong to 4 cliques of size 2 (ie. are connected to 4 other blocks), 4 cliques of size 3 (ie. belong to 3 distinct triangles of the graph) and 1 clique of size 4. Blocks  $\{1, 5, 6\}$  and  $\{1, 7, 9\}$  belong to 3 cliques of size 2, 3 cliques of size 3 and a single clique of size 4. Block  $\{2, 11, 12\}$  belongs to 2 cliques of size 2 and a single clique of size 3.

The table below summarises the clique involvements of each block:

Blocks	Clique Sizes		
	2	3	4
1) $\{1, 2, 3\}$	4	4	1
2) $\{1, 5, 6\}$	3	3	1
3) $\{1, 7, 9\}$	3	3	1
4) $\{2, 6, 9\}$	4	4	1
5) $\{2, 11, 12\}$	2	1	0

The clique analysis induces a block partition consisting of 3 distinct classes. Class 1 contains blocks 1 and 4, class 2 contains blocks 2 and 3, and block 5 is in a class of its own.

The block partition is represented by a class vector containing 5 elements, where the entry in element  $i$  of the vector is the integer assigned to the class containing block  $i$ . The class vector representing the above partition would therefore be:  $(1\ 2\ 2\ 1\ 3)$ .

### 4.5.1.3 POINT PARTITION

Once the blocks have been partitioned, the points also need to be classified, and this can be derived directly from the block partition. Assume the block partition contains  $C$  classes. Then for each point,  $p$ , in the design, a *count vector* of length  $C$  is calculated in which element  $i$  contains the number of blocks belonging to class  $i$  of the block partition which contain point  $p$ .

Returning to the previous example, the class vector for the block partition is given below:  
 $(1\ 2\ 2\ 1\ 3)$

Examining the blocks of the design, point 1 is contained in the 3 blocks  $\{1, 2, 3\}$ ,  $\{1, 5, 6\}$  and  $\{1, 7, 9\}$ , which belong to classes 1, 2 and 2 of the block partition respectively. Therefore, point 1 is contained in one class 1 block, and two class 2 blocks. From a similar analysis, point 2 belongs to two class 1 blocks and one class 3 block. The following table gives the count vector for each point of the design:



Points	Block Classes		
	1	2	3
1	1	2	0
2	2	0	1
3	1	0	0
5	0	1	0
6	1	1	0
7	0	1	0
9	1	1	0
11	0	0	1
12	0	0	1

The count vectors of points 5 and 7 are identical, as are the count vectors for points 6 and 9, and for points 11 and 12. The count vectors partition the point set into 6 distinct classes, where each class contains the following sets of points:

Point Class	1	2	3	4	5	6
Points within class	{1}	{2}	{3}	{5, 7}	{6, 9}	{11, 12}

The class vector for the point partition is therefore:  
 (1 2 3 4 5 4 5 6 6)

### 4.5.1.3 POINT PAIR PARTITION

This partition places each pair of points of the design into classes, such that two distinct pairs of points are in the same class if they are contained in exactly the same number of blocks of each distinct block type. A point pair partition is represented as a 2-dimensional class matrix.

The point pair partition is induced from the block partition in a similar way to the individual point partition, except only those blocks which contain both points in the given pair are used in the analysis. Returning once again to the example, the blocks of the design and their classes in the block partition are shown below:

Block:	{1, 2, 3}	{1, 5, 6}	{1, 7, 9}	{2, 6, 9}	{2, 11, 12}
Class:	1	2	2	1	3

The pairs {1, 2} and {1, 3} occur in just one block of type 1, and so they are placed into class 1 of the point pair partition. The pair {1, 5} occurs in just one block of type 2, and so it is placed into a new class, 2.

This process is performed for each possible pair in the design, yielding the complete point pair partition represented in the following symmetric class matrix:

	1	2	3	5	6	7	9	11	12
1	0	1	1	2	2	2	2	3	3
2	1	0	1	3	1	3	1	4	4
3	1	1	0	3	3	3	3	3	3
5	2	3	3	0	2	3	3	3	3
6	2	1	3	2	0	3	1	3	3
7	2	3	3	3	3	0	2	3	3
9	2	1	3	3	1	2	0	3	3
11	3	4	3	3	3	3	3	0	4
12	3	4	3	3	3	3	3	4	0

The point pairs of the example design can therefore be partitioned into 4 distinct classes.

The block, point and point pair partitions for the block intersection graph of a design  $D$ , are used to reduce the size of the search space for the backtracking algorithm which constructs the automorphism group of  $D$ . The ways in which this is done are outlined below:

- 1) The point partition provides a means of quickly producing the set of target points which must be mapped to a given source point.
- 2) The block partition provides a means of quickly checking whether a given partial automorphism is incidence preserving.
- 3) The point pair partition provides an elegant way of strengthening the point and block partitions as the search progresses.

The remaining sections of this chapter focus on the uses of these partitions and on the details of the group generation algorithm.

## 4.5.2 THE GROUP GENERATION ALGORITHM

The automorphism group generation algorithm constructs a series of partial automorphisms by backtracking on all admissible point mappings, which are determined by the classes of the point partition. As the partial automorphism grows, the remaining allowable point mappings become more restricted, and indeed the point classes themselves can usually be divided further. For this purpose, the algorithm makes use of two point partitions - one for the source points, and one for the target points of the constructed mapping. Initially, the source and target point partitions are exactly the same, because all the class information is derived from the clique analysis. As the algorithm progresses and individual point mappings are accepted, these partitions are refined independently to reflect any new restrictions. In a similar way, a source and a target block partition are used, and are refined after each new point mapping is accepted.

The detailed description of the algorithm which follows shortly makes use of the important data structures presented below:

Data Structure	Purpose
<b>VSourcePart, VTargetPart</b>	These structures store the class vectors of the source and target point partitions at each level of the search. The partitions are stored at each level of the partial automorphism construction so that they can be recalled rather than recalculated if the partial mapping is backtracked.
<b>BSourcePart, BTargetPart</b>	These structures store the class vectors of the source and target block partitions at each level of the search. As with the point partitions, these are stored at each level and retrieved when backtracking occurs.
<b>VV</b>	This structure represents the class matrix of the point pair partition, and is not updated as the algorithm progresses. It is nevertheless extremely important for refining the point partitions efficiently.
<b>INC</b>	This is the incidence matrix of the design for which the group is being calculated. This structure also remains unchanged throughout the algorithm, although it is used for the efficient refinement of the block partitions.
<b>L</b>	This is the current level to which the partial automorphism has been constructed. In other words, it specifies how many source points have had valid mappings to target points associated with them.
<b>SP</b>	This is the set of all remaining unmapped source points. Once a valid mapping is accepted at a given level of the search, the next source point to map is selected from this set.
<b>TP(L)</b>	This is the set of all target points which are still to be considered for the mapping from the source point selected at level $L$ of the construction.
<b><math>P_{src}(L), P_{tgt}(L)</math></b>	These indicate the source and target points selected at level $L$ . $P_{src}(L)$ is selected from $SP$ , and $P_{tgt}(L)$ is selected from $TP(L)$ . Once a complete valid automorphism is constructed for a design on $v$ points, it can be accessed from these structures as follows: $(P_{src}(1) \rightarrow P_{tgt}(1)) (P_{src}(2) \rightarrow P_{tgt}(2)) \dots (P_{src}(v) \rightarrow P_{tgt}(v))$

For example, say the partial automorphism:

$$\emptyset: (1 \rightarrow 1) (2 \rightarrow 6) (3 \rightarrow 5)$$

has been validly constructed, and the source point chosen at the next level of the search is 4, ie.  $P_{src}(4) = 4$ . (It is worth noting that the source points do not necessarily have to be selected in lexicographical order, and in fact a better order for their selection is presented in Section 4.5.2.2)

Partial Automorphism,  $\emptyset$

Source points:	1	2	3	4	5	...	v
	↓	↓	↓	↓			
Target points:	1	6	5	?			

At this stage of the algorithm, assume that the source and target point partitions are given by the following class vectors:

	1	2	3	4	5	6	...	v
Class vector for source points:	1'	2'	3'	4	5	4		

	1	2	3	4	5	6	...	v
Class vector for target points:	1'	4	4	5	3'	2'		

The classes 1', 2' and 3' are marked with a ' because they correspond to classes of points which have already been mapped. Looking at the target point partition, points "2" and "3", belonging to class 4, are the only possible points to which the source point "4" may be mapped.

The values of the structures  $L, P_{src}, TP(L)$  and  $SP$  at this stage of the construction would be:

$$L = 4$$

$$P_{src}(L) = 4$$

$$TP(L) = \{2,3\}$$

$$SP = \{4,5,6,7,\dots,v\}$$

One at a time, potential target points,  $P_{tgt}(L)$ , are removed from the set  $TP(L)$  and are examined as possibilities for the target mapping at level 4.

An outline of the complete group generation algorithm is given at the top of the facing page. Note the two starred (\*) operations, which are described in greater detail following the algorithm:

```

L = 1;
SP = {1,2,3,...v}

while (TRUE) {
  valid_mapping = FALSE;
  Select a Psrc(L) from SP;
  Set TP(L) = all possible target points at level L*
  while ((L>0) && (valid_mapping == FALSE)) {
    while ((TP(L) is non-empty) && (valid_mapping == FALSE)) {
      Select and remove a Ptgt(L) from TP(L), and add to partial
      automorphism,  $\emptyset = (P_{src}(1) \rightarrow P_{tgt}(1)) \dots (P_{src}(L) \rightarrow P_{tgt}(L))$ 
      if (partial automorphism  $\emptyset$  is incidence preserving*) {
        if (L == v)
          Store  $\emptyset$  as an automorphism of the design
        else {
          Remove Psrc(L) from SP;
          valid_mapping = TRUE;
        }
      }
    }
    if (valid_mapping == FALSE) {
      L = L-1;
      if (L>0)
        Add Psrc(L) to SP(L)
    }
  }
  if (valid_mapping == FALSE)
    Group completely calculated. Exit algorithm.
  else
    L = L+1;
}

```

There are two main operations, denoted with a star (\*) in the algorithm outline, which should be elaborated upon.

- 1) *Setting TP(L) to all possible target points at level L.*  
 When this operation is executed, a source point at level L, P<sub>src</sub>(L), has already been selected. The point partitions are used at this stage to significantly reduce the number of possible target candidates. The class of the partition to which the source point belongs is stored in the VSourcePart structure. Let this class be C. Only those target points which also belong to the same class, C, should be added to the set TP(L). The VTargetPart structure maintains the class information for the target points.
- 2) *Checking whether the current partial automorphism,  $\emptyset$ , is incidence preserving.*  
 When the new mapping at level L is added to the partial automorphism,  $\emptyset$ , this new information allows the point and block partitions to be strengthened. The process of strengthening, or tightening, these partitions is quite involved, and described in detail in the Section 4.5.2.1. The source and target point partitions are tightened independently, as are the source and target block partitions. Following this strengthening, determining whether or not the new mapping at level L just added to  $\emptyset$  is incidence preserving requires a trivial test that the class vectors for the block partitions are in one-to-one correspondence.

#### 4.5.2.1 PARTITION TIGHTENING: CLASS VECTOR JUXTAPOSITION

The initial partitions which are induced from the results of the clique analysis divide the points and blocks of the design into classes, which place restrictions on the mappings considered during the algorithm. The initial partitions alone significantly reduce the

search space, however, as each new point mapping is accepted into the partial automorphism, the partitions can be further refined. The partial automorphism must always maintain the incidence preserving property, and the method of juxtaposition which is used to refine the partitions provides a simple way of performing this test. The next two subsections detail the tightening process for the point and block partitions.

### Point Partition Tightening

*The purpose of tightening the point partitions is to reduce the number of possible mappings which must be considered at each level of the automorphism construction.*

Assume the partial automorphism is incidence preserving up to level  $L$ , and is then extended to level  $L+1$  by the mapping  $P_{src}(L+1) \rightarrow P_{tgt}(L+1)$ .

$V_{SourcePart}(L+1)$  and  $V_{TargetPart}(L+1)$  are the class vectors of the source and target point partitions at level  $L+1$ . The class of  $P_{src}(L+1)$ , say  $C$ , was calculated from  $V_{SourcePart}(L+1)$ , and the set of possible candidate target points, from which  $P_{tgt}(L+1)$  was selected, consisted of those points belonging to class  $C$  in  $V_{TargetPart}(L+1)$ . Before this new point mapping can be accepted as part of the automorphism, the point partition must be tightened.

The class matrix of the point pair partition of the design,  $VV$ , plays an important role in the point partition tightening. Let  $VV_{ij}$  denote the class to which the pair  $\{i,j\}$  belongs in this partition. As a useful example, assume the current partial automorphism consists of the single point mapping  $(1 \rightarrow 3)$ . When this is extended to level 2, assume the point mapping  $(2 \rightarrow 4)$  is added. This partial automorphism will clearly map the pair  $\{1,2\}$  onto the pair  $\{3,4\}$ , and so for this new mapping to be valid, the pairs  $\{1,2\}$  and  $\{3,4\}$  must belong to the same class of the point pair partition, ie.  $VV_{12} = VV_{34}$ . In fact for any pair of points, say  $x$  and  $y$ , if  $\emptyset$  is a valid automorphism it must be true that  $VV_{xy} = VV_{\emptyset(x)\emptyset(y)}$ .

This general fact can be related to the automorphism construction process. For an individual point mapping  $(P_{src}(L) \rightarrow P_{tgt}(L))$  to be part of an automorphism  $\emptyset$  of a design  $D$ ,  $VV_{P_{src}(L)p} = VV_{P_{tgt}(L)\emptyset(p)}$  must hold for all points  $p$  of the point set of  $D$ . In other words, for any given point  $p$ , let  $VV_p$  denote the vector of point pair classes  $VV_{pi}$  for all  $1 \leq i \leq v$ . The vectors  $VV_{P_{src}(L)}$  and  $VV_{P_{tgt}(L)}$  must be in one-to-one correspondence for  $(P_{src}(L) \rightarrow P_{tgt}(L))$  to be part of a valid partial automorphism,  $\emptyset$ . If they were not, then no set of future point mappings could validly complete  $\emptyset$ .

The method for updating the point partitions is naturally defined recursively:

- 1) The initial point partitions are induced from the clique analysis, and stored in the vectors  $V_{SourcePart}_1$  and  $V_{TargetPart}_1$
- 2) When a mapping  $(P_{src}(L) \rightarrow P_{tgt}(L))$  is chosen at level  $L$ , the new class vectors for the next level are calculated as follows:  
 $V_{SourcePart}_{L+1}$  is the result of juxtaposing  $V_{SourcePart}_L$  and  $VV_{P_{src}(L)}$   
 $V_{TargetPart}_{L+1}$  is the result of juxtaposing  $V_{TargetPart}_L$  and  $VV_{P_{tgt}(L)}$

To illustrate the process of vector juxtaposition, consider the following example.

Assume, at some level  $L$  in the construction process, the mapping  $(1 \rightarrow 2)$  is chosen, ie.  $P_{src}(L)=1$  and  $P_{tgt}(L)=2$ . Let the class vectors of the source and target point partitions, and the relevant rows of the class matrix of the point pair partition be as below:

$$\begin{aligned} V_{SourcePart}_L &= && 1 & 1 & 1 & 1 & 3 & 2 & 2 & 2 & 2 \\ VV_{P_{src}(L)} &= && 0 & 1 & 1 & 1 & 5 & 2 & 2 & 3 & 3 \\ V_{TargetPart}_L &= && 2 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 3 \\ VV_{P_{tgt}(L)} &= && 2 & 0 & 1 & 3 & 3 & 2 & 1 & 1 & 5 \end{aligned}$$

To create the new, tightened point partitions at level  $L+1$ , the sets of vectors are juxtaposed by placing them side by side and considering the distinct types of pairs formed, which correspond to the new classes of the tightened vectors.

For example, the juxtaposition of these 2 pairs of vectors forms the following 5 unique pairs:  $(1, 0)$ ,  $(1, 1)$ ,  $(3, 5)$ ,  $(2, 2)$  and  $(2, 3)$ , where the first value of each pair corresponds to the level  $L$  point partitions, and the second pair corresponds to the point pair partitions.

For each of these 5 pairs, a unique number is assigned as the new class identifier for that pair. The table below gives the new assignments:

<i>Pair</i>	<i>New class identifier</i>
$(1, 0)$	1
$(1, 1)$	2
$(3, 5)$	3
$(2, 2)$	4
$(2, 3)$	5

So the new point partitions at level  $L+1$  become:

$$\begin{aligned}
 V_{\text{SourcePart}_L} &= 1\ 1\ 1\ 1\ 3\ 2\ 2\ 2\ 2 \\
 VV_{P_{\text{src}}(L)} &= \underline{0\ 1\ 1\ 1\ 5\ 2\ 2\ 3\ 3} \\
 V_{\text{SourcePart}_{L+1}} &= 1\ 2\ 2\ 2\ 3\ 4\ 4\ 5\ 5 \\
 V_{\text{TargetPart}_L} &= 2\ 1\ 1\ 2\ 2\ 2\ 1\ 1\ 3 \\
 VV_{P_{\text{tgt}}(L)} &= \underline{2\ 0\ 1\ 3\ 3\ 2\ 1\ 1\ 5} \\
 V_{\text{TargetPart}_{L+1}} &= 4\ 1\ 2\ 5\ 5\ 4\ 2\ 2\ 3
 \end{aligned}$$

Two important observations can be made:

- 1) The point partitions now consist of 5 distinct classes at level  $L+1$ , whereas at level  $L$  there were only 3. This juxtaposition has tightened the partitions, reducing the number of possible point mappings for future selections.
- 2) The new point partitions,  $V_{\text{SourcePart}_{L+1}}$  and  $V_{\text{TargetPart}_{L+1}}$  are in one-to-one correspondence - both containing one class 1 point, three class 2 points, one class 3 point, two class 4 points and two class 5 points.

This second observation is particularly important, for the point mapping  $(P_{\text{src}}(L) \rightarrow P_{\text{tgt}}(L))$  can only be accepted if the new point partitions are in one-to-one correspondence. If the new point partitions at level  $L+1$  were not in one-to-one correspondence, then the mapping  $(P_{\text{src}}(L) \rightarrow P_{\text{tgt}}(L))$  could not be accepted, for it would not be possible to complete the automorphism as there would be some points which could not be included in future point mappings.

### Block Partition Tightening

*The purpose of tightening the block partitions is to ensure that the partial automorphism is incidence preserving.*

As already mentioned, just as two point partitions are used in the algorithm, two block partitions are also used - a source and a target block partition.

These partitions initially divide the blocks of the design into classes, such that two blocks are in the same class if they have similar properties following the clique analysis. As the partial automorphism is constructed, each time a point mapping  $(P_{\text{src}}(L) \rightarrow P_{\text{tgt}}(L))$  is added at level  $L$ , all those blocks containing the point  $P_{\text{src}}(L)$  get split into new classes in the source block partition, and all those blocks containing the point  $P_{\text{tgt}}(L)$  get split into new classes in the target block partition. This ensures that for any subset,  $S$ , of the source points contained within a block of the design belonging to class  $C$  of the source block partition, the subset  $\emptyset(S)$  of target points will also be contained within a block of the design belonging to class  $C$  of the target block partition.

Consider the following incidence preserving partial automorphism,  $\emptyset$ , constructed to level  $L$ :

$$\emptyset : (P_{\text{src}}(1) \rightarrow P_{\text{tgt}}(1)) (P_{\text{src}}(2) \rightarrow P_{\text{tgt}}(2)) \dots (P_{\text{src}}(L) \rightarrow P_{\text{tgt}}(L))$$

Let  $BSourcePart_L$  and  $BTargetPart_L$  denote the source and target block partitions at level  $L$  of the automorphism construction, respectively.

Let  $BSourcePart_{L,B}$  and  $BTargetPart_{L,B}$  denote the actual class to which block  $B$  belongs in the source and target block partitions at level  $L$  respectively.

Formally then, for the incidence preserving partial automorphism  $\emptyset$  constructed to level  $L$ , if  $S$  is any subset of  $\{P_{src}(1), P_{src}(2), \dots, P_{src}(L)\}$ , and  $S$  is contained within a block  $B_1$  of the design, then  $\emptyset(S)$  must be contained in any block  $B_2$  of the design for which  $BSourcePart_{L+1,B_1} = BTargetPart_{L+1,B_2}$ . If this property does not hold, then the partial automorphism,  $\emptyset$ , must not be incidence preserving to level  $L$ .

The source and target block partitions are updated at each level in such a way that as long as the partial automorphism is incidence preserving, the two block partitions will be in one-to-one correspondence. If a point mapping which violates the incidence preserving property is added to the partial automorphism, then the updated block partitions will not be one-to-one correspondence.

Whenever a point mapping ( $P_{src}(L) \rightarrow P_{tgt}(L)$ ) at level  $L$  is added to the partial automorphism, the block partitions are tightened to form new partitions at level  $L+1$ . This is achieved by juxtaposing the source block partition with row  $P_{src}(L)$  of the incidence matrix and juxtaposing the target block partition with row  $P_{tgt}(L)$  of the incidence matrix.

For example, assume the design consists of 10 blocks, and the initial source and target block partitions induced from the clique analysis separate the blocks into 4 classes, as given below:

```
Block:           1 2 3 4 5 6 7 8 9 10
BSourcePart1:   1 1 1 2 2 3 1 2 1 4
BTargetPart1:   1 1 1 2 2 3 1 2 1 4
```

Now, assume that the point mapping ( $a \rightarrow b$ ) is accepted at level 1 of the automorphism construction. Further assume that point "a" belongs to blocks 1, 2 and 4, and that point "b" belongs to blocks 7, 8 and 9. This mapping is clearly incidence preserving because both the source point "a" and the target point "b" belong to two blocks of class 1 and one block of class 2. The new source and target block partitions for the next level must then be calculated.

The source block partition can be split into 2 further classes, by creating a new class for each existing block class corresponding to those blocks containing the point "a". For example, blocks 1 and 2, which both contain point "a", will be placed into a new class, 5. Block 4, which also contains point "a", will be placed into another new class, 6. The source block partition then becomes:

```
Block:           1 2 3 4 5 6 7 8 9 10
BSourcePart2:   5 5 1 6 2 3 1 2 1 4
```

This partition tightening can be achieved by juxtaposing row a of the incidence matrix with the current block partition. Row a of the incidence matrix will have a 1 in columns 1, 2 and 4:

```
Row a:           1 1 0 1 0 0 0 0 0 0
```

Juxtaposing this with the current source block partition allows the new classes to be formed:

```
Block:           1 2 3 4 5 6 7 8 9 10
BSourcePart1:   1 1 1 2 2 3 1 2 1 4
Row a:           1 1 0 1 0 0 0 0 0 0
```

The classes of the new block partition are calculated by assigning a unique class identifier to all distinct pairs of the type (*class*, *row a*) above:

Pair:	(1 1)	(1 0)	(2 1)	(2 0)	(3 0)	(4 0)
New Class Identifier	5	1	6	2	3	4

The new source block partition then becomes:

Block:	1 2 3 4 5 6 7 8 9 10
BSourcePart <sub>1</sub> :	1 1 1 2 2 3 1 2 1 4
Row a:	<u>1 1 0 1 0 0 0 0 0 0</u>
BSourcePart <sub>2</sub> :	5 5 1 6 2 3 1 2 1 4

Using the same new class identifiers, the target block partition can also be tightened. This is achieved by juxtaposing the current target block partition with row b of the incidence matrix:

Block:	1 2 3 4 5 6 7 8 9 10
BTargetPart <sub>1</sub> :	1 1 1 2 2 3 1 2 1 4
Row b:	<u>0 0 0 0 0 0 1 1 1 0</u>
BTargetPart <sub>2</sub> :	1 1 1 2 2 3 5 6 5 4

So after the mapping ( $a \rightarrow b$ ) is accepted at level 1, the new tightened block partitions at level 2 become:

BSourcePart <sub>2</sub> :	5 5 1 6 2 3 1 2 1 4
BTargetPart <sub>2</sub> :	1 1 1 2 2 3 5 6 5 4

Now, for any block,  $B_1$ , containing the subset  $\{a\}$  of the source points, the set  $\{\emptyset(\{a\})\} = \{b\}$  is contained in any block,  $B_2$ , for which  $B_{\text{SourcePart}_2, B_1} = B_{\text{TargetPart}_2, B_2}$ . This is exactly what the example has illustrated: for any block containing the point "a", ie. blocks 1, 2 and 4, the point  $\emptyset(a) = "b"$  is contained in any block belonging to the same class in the target block partition.

If, following this juxtaposition, the new source and target block partitions are in one-to-one correspondence, then the partial automorphism  $\emptyset$  must be incidence preserving. In the example above,  $B_{\text{SourcePart}_2}$  and  $B_{\text{TargetPart}_2}$  are in one-to-one correspondence, which indicates that the mapping ( $a \rightarrow b$ ) must be incidence preserving.

A non incidence preserving mapping, say  $((P_{\text{src}}(L) \rightarrow P_{\text{tgt}}(L)))$ , will be one in which  $P_{\text{src}}(L)$  and  $P_{\text{tgt}}(L)$  belong to a different number of blocks of each class. Such a mapping will clearly generate tightened block partitions which are not in one-to-one correspondence. If this is the case, the point mapping will not be accepted, and backtracking will commence on the partial automorphism.

### 4.5.2.2 SOURCE AND TARGET POINT SELECTION

Once an incidence preserving partial automorphism has been constructed to level  $L$ , the source point for the next level,  $L+1$ , must be selected from the set,  $SP$ , of unmapped source points. Although there are a number of ways in which this can be selected, the strategy adopted here is that of Gibbons [32], in which the source point,  $P_{\text{src}}(L)$ , selected at level  $L$  is the one which belongs to the smallest class of the source point partition. Clearly this approach reduces the breadth of the search for there will be fewer possible target points to be considered at level  $L$ .

It should be noted that once a source point,  $P_{\text{src}}(L)$ , has been selected at level  $L$ , it remains the source point at level  $L$  for the duration of the backtrack. This means that the partitions  $V_{\text{SourcePart}}$  and  $B_{\text{SourcePart}}$  only need to be computed once during the algorithm.

### 4.5.2.3 RESULTS

A detailed description has now been given stating how the point and block partitions are tightened, how each partial automorphism is tested for being incidence preserving, and in what order the source points should be selected. Using these techniques, a simple algorithm for the construction of the automorphism group of a given design has been implemented. The



algorithm was used to generate the automorphism groups of several designs, and its performance is summarised in this section.

For example, there are 2 non-isomorphic Steiner Triple Systems on 13 points. The execution time of the algorithm to generate the groups of these designs are given in the table below:

Design	Group Size	Execution Time (secs)
STS(13) Number 1	36	0.0155
STS(13) Number 2	6	0.0021

These execution times include the time required to induce the initial point, block and point pair partitions, and the time taken to store every automorphism of each group in memory. For designs with moderately small group sizes such as these, the algorithm performs well.

Another example, which deals with slightly larger group sizes, are the 5 non-isomorphic symmetric BIBDs with the parameter set  $(v,b,r,k,\lambda) = (15,15,7,7,3)$ . The group sizes of these designs are given below, along with the time taken to induce the initial partitions, then backtrack and store each automorphism of the group in memory:

Design	Automorphism Group Size	Execution Time (secs)
1	20160	3.663
2	576	0.350
3	168	0.117
4	168	0.083
5	96	0.100

The algorithm still performs reasonably well, although the running times are considerably longer for the designs with large group sizes because every automorphism in the group is formed explicitly by backtracking.

### 4.5.3 FAST AUTOMORPHISM GROUP GENERATION

The automorphism group generation algorithm just described performs well for the designs analysed, however it still carries out a lot of unnecessary backtracking. The focus of the remainder of this chapter is the development of optimisations which reduce the amount of backtracking performed. In particular, the use of centraliser subgroups as employed by Gibbons in [32], allows a majority of the group to be derived from a set of generator automorphisms rather than being constructed explicitly as part of the backtracking algorithm. This reduces the amount of backtracking performed enormously, and leads to a considerable improvement in performance.

#### 4.5.3.1 GENERATOR AUTOMORPHISMS

For any design, a certain amount of time is required to construct and store each automorphism in its group. However, only a small proportion of the automorphisms actually need to be constructed by the backtracking algorithm, as the remainder can be derived quickly from these.

For example, consider the partial automorphism of a given design,  $D$ :

$$\mathcal{O}_1 = (1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3)$$

The presented backtracking algorithm would extend  $\mathcal{O}_1$  to exhaustively generate all automorphisms, belonging to the group of  $D$ , which fix the first 3 points. This process may take some time, depending on the structure of  $D$ .

Once all the extensions to  $\mathcal{O}_1$  have been constructed, backtracking would occur on the mapping (3→3). Assume that the source point "3" can also be validly mapped to the target point "4". The new partial automorphism then becomes:

$$\mathcal{O}_2 = (1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 4)$$

All extensions to  $\mathcal{O}_2$  are then constructed by the backtracking algorithm. However, a large proportion of these extensions may be able to be derived from the automorphisms constructed as extensions to  $\mathcal{O}_1$ , in addition to knowing that (3 → 4) is a valid mapping.

The results of such an optimisation are considerable. For example, design number 1 in the table in Section 4.5.2.3, with group size 20160, can in fact have all of its automorphisms constructed and saved in just 0.82 seconds by deriving as many as possible rather than explicitly building them by backtracking. In fact, only 10 automorphisms need to be constructed by the backtracking algorithm, with the remaining 20150 being generated as compositions of these 10 mappings. These 10 automorphisms are called generator automorphisms, as they are able to generate the automorphism group of the design. The backtracking construction of the generator automorphisms is very fast - all 10 generators for the group of this design above can be constructed in 0.0024 seconds.

In fact, for designs with extremely large groups, it may not be feasible to generate and store every automorphism. In such cases, it may be desirable to construct only the generator automorphisms, from which the complete group could be derived quickly if necessary.

The modifications required to the group generation backtracking algorithm in order to construct only the generator automorphisms and to derive the complete group from these are described in the following sections. Section 4.5.3.2 introduces a new class vector, called the *automorphism partition*. This partition is very important as it reduces the amount of searching required for the construction of the generator automorphisms and plays an important role in the process of deriving the remaining automorphisms. Section 4.5.3.3 formalises the optimisations by defining the *centraliser subgroups* and relates the generation of the subgroups to the construction of the complete automorphism group of a given design. This is followed by Section 4.5.3.4, which outlines the process of the backtracking construction of only the generator automorphisms, and describes how the automorphism partition is used for this task. The following section, 4.5.3.5, details how the complete automorphism group is derived as each new generator automorphism is constructed, and provides a useful example which ties together the ideas of the previous sections. The chapter concludes with an overview of the entire process, a summary of the results of the new algorithm, and a brief mention of further possible improvements and other observations.

The first modification to the currently presented algorithm which needs to be made is to control the order of selection of source and target points at each level of the search. If the source point selected at level L is the smallest point from within the smallest class, and if in addition the target point selected at level L is the smallest point from within the set, TP(L), of possible target points, then the first automorphism generated will always be the identity mapping.

### 4.5.3.2 AUTOMORPHISM PARTITION

The optimised group generation algorithm makes use of an automorphism partitioning vector, AUTPT, which partitions the points of the design into classes such that points within the same class of AUTPT can be mapped onto one another to form valid automorphisms of the design. Initially, nothing is known about the automorphism classes to which each point belongs, and hence the initial automorphism partition places each point within its own class, as the diagram overleaf illustrates:

	1	2	3	4	<i>Points</i>	(v-1)	v
AUTPT	1	2	3	4	<i>Classes</i>	(v-1)	v

The first automorphism generated by the backtracking algorithm will always be the identity mapping:

$$(1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) \dots ((v-1) \rightarrow (v-1)) (v \rightarrow v)$$

As each new automorphism is constructed, the new mapping information is used to update the automorphism partition classes, in the vector AUTPT.

For example, once backtracking begins on the identity mapping, the point mapping  $(v \rightarrow v)$  will be removed, as will the mapping  $((v-1) \rightarrow (v-1))$ . The new point mapping  $((v-1) \rightarrow v)$  will then be tested to see if it can be added to the partial automorphism. Assume that it can, so that the very next automorphism constructed maps point  $(v-1)$  onto  $v$  and point  $v$  onto  $(v-1)$ :

$$(1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) \dots ((v-1) \rightarrow v) (v \rightarrow (v-1))$$

In this case, an automorphism exists in the group of the design which maps point  $(v-1)$  and point  $v$  onto each other. These points therefore belong to the same class of the automorphism partition, and this information is updated in the partition vector as shown below:

	1	2	3	4	<i>Points</i>	(v-1)	v
AUTPT	1	2	3	4	<i>Classes</i>	(v-1)	(v-1)

Points  $(v-1)$  and  $v$  are merged into a single class in AUTPT. This merging of classes is performed throughout the algorithm, whenever an automorphism is discovered which contains a point mapping between two points belonging to different classes in AUTPT.

### 4.5.3.3 CENTRALISER SUBGROUPS

The fast group generation algorithm, which is the subject of this main section, 4.5.3, derives as many automorphisms as possible rather than generating them explicitly by backtracking. For every pair of distinct points,  $a$  and  $b$ , as soon as the first automorphism is found which maps  $(a \rightarrow b)$ , no more mappings which also map  $(a \rightarrow b)$  and which fix the same number of initial points need to be generated by backtracking. All the rest of these automorphisms are able to be constructed from the current set of generators by composition. This idea is formalised with the definition of centraliser subgroups.



$H_n$  is the centraliser subgroup of order  $n$ , and corresponds to the set of all automorphisms in the automorphism group of a design in which the first  $n$  points map onto themselves. Formally, the centraliser subgroup,  $H_n$ , of a design  $D$  with automorphism group  $G$  is defined as follows:

$$H_n = \{\theta \in G : \theta(P_{src}(m)) = P_{src}(m), (1 \leq m \leq n)\}$$

For example, given a design on  $v$  points,  $H_v$  simply consists of the identity automorphism, as it maps all  $v$  points onto themselves.

As another example, if the first 3 source points selected in the backtracking process for a design were 1, 2 and 3, then  $H_3$  would consist of all automorphisms of the form:

$(1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) \dots$

Clearly,  $H_v = H_{v-1}$ , because if the first  $(v-1)$  points map onto themselves, then the last mapping is forced, and the remaining unmapped point must map onto itself. In addition,  $H_0 = G$ , the full automorphism group of the design, because in this set of automorphisms there is no restriction on how the initial points must map.

The optimised algorithm being presented generates a series of sets,  $S_v, S_{v-1}, \dots, S_2, S_1, S_0$ . Each set consists of the generator automorphisms of the corresponding centraliser subgroup. In other words,  $H_i = \langle S_i \rangle$ , meaning the centraliser subgroup  $H_i$  is generated by the set of automorphisms,  $S_i$ . In addition, each set  $S_i$  is contained in the set  $S_{i-1}$ ,  $0 < i \leq v$ .

The aim of the algorithm is therefore to produce the set  $S_0$ , which corresponds to the set of generator automorphisms from which  $H_0$  - the complete automorphism group,  $G$  - can be constructed.

#### 4.5.3.4 CONSTRUCTION OF GENERATOR AUTOMORPHISMS

This section details, with the aid of an example, the construction of the complete set of generator automorphisms for a given design. The automorphism partition, represented by the class vector AUTPT, plays a pivotal role in reducing the number of mappings which are considered at each level.

The first automorphism constructed is the identity, which is always the first generator. The backtracking algorithm then proceeds as before, removing point mappings systematically and testing new mappings until a new automorphism is found. Due to the fact that the process of backtracking removes point mappings in the opposite order to which they were selected, the next automorphism will fix a certain number,  $n$ , of initial points. This automorphism will belong to the set,  $S_n$ , of generators for  $H_n$ . An example will illustrate this concept:

Consider the TTS(9) below:

1 2 3	1 2 4	1 3 4	1 5 6	1 5 7	1 6 8
1 7 9	1 8 9	2 3 4	2 5 6	2 5 7	2 6 9
2 7 8	2 8 9	3 5 8	3 5 9	3 6 7	3 6 8
3 7 9	4 5 8	4 5 9	4 6 7	4 6 9	4 7 8

The complete automorphism group of this design consists of the following 8 automorphisms:

- $(1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 5) (6 \rightarrow 6) (7 \rightarrow 7) (8 \rightarrow 8) (9 \rightarrow 9)$
- $(1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 5) (6 \rightarrow 7) (7 \rightarrow 6) (8 \rightarrow 9) (9 \rightarrow 8)$
- $(1 \rightarrow 2) (2 \rightarrow 1) (3 \rightarrow 4) (4 \rightarrow 3) (5 \rightarrow 5) (6 \rightarrow 6) (7 \rightarrow 7) (8 \rightarrow 9) (9 \rightarrow 8)$
- $(1 \rightarrow 2) (2 \rightarrow 1) (3 \rightarrow 4) (4 \rightarrow 3) (5 \rightarrow 5) (6 \rightarrow 7) (7 \rightarrow 6) (8 \rightarrow 8) (9 \rightarrow 9)$
- $(1 \rightarrow 3) (2 \rightarrow 4) (3 \rightarrow 1) (4 \rightarrow 2) (5 \rightarrow 5) (6 \rightarrow 8) (7 \rightarrow 9) (8 \rightarrow 6) (9 \rightarrow 7)$
- $(1 \rightarrow 3) (2 \rightarrow 4) (3 \rightarrow 1) (4 \rightarrow 2) (5 \rightarrow 5) (6 \rightarrow 9) (7 \rightarrow 8) (8 \rightarrow 7) (9 \rightarrow 6)$
- $(1 \rightarrow 4) (2 \rightarrow 3) (3 \rightarrow 2) (4 \rightarrow 1) (5 \rightarrow 5) (6 \rightarrow 8) (7 \rightarrow 9) (8 \rightarrow 7) (9 \rightarrow 6)$
- $(1 \rightarrow 4) (2 \rightarrow 3) (3 \rightarrow 2) (4 \rightarrow 1) (5 \rightarrow 5) (6 \rightarrow 9) (7 \rightarrow 8) (8 \rightarrow 6) (9 \rightarrow 7)$

The group generation algorithm of Section 4.5.2 would have explicitly constructed each of the 8 automorphisms above by backtracking. The optimised algorithm being presented in this main section actually generates only 4 of the above automorphisms by backtracking. These comprise the complete set of generator automorphisms from which the entire group can be derived. The operation of the algorithm is detailed overleaf.

The initial automorphism constructed is the identity:

- (1→1) (2→2) (3→3) (4→4) (5→5) (6→6) (7→7) (8→8) (9→9)

This identity automorphism is in fact the centraliser subgroup for levels  $v$  and  $(v-1)$ . The algorithm then proceeds in the same way as before, backtracking on the point mappings until the next automorphism is found, which is given below:

- (1→1) (2→2) (3→3) (4→4) (5→5) (6→7) (7→6) (8→9) (9→8)

This automorphism can be rewritten as a two row matrix, with the source points along the top row and the corresponding target points along the bottom row:

Source:	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>
Target:	1	2	3	4	5	<b>7</b>	<b>6</b>	<b>9</b>	<b>8</b>

In this form, it is easier to see for which centraliser subgroup this new mapping is the generator. Only the first 5 points of the automorphism are now fixed. This also indicates that no automorphisms exist which keep the first 6 or 7 points fixed, and hence the centraliser subgroups for these orders are empty. ie.  $H_6$  and  $H_7$  are empty.

$H_5$  is not empty however, and it is necessary to construct all the generator automorphisms making up the set  $S_5$  from which  $H_5$  can be constructed. The new automorphism just constructed clearly belongs to  $S_5$ , but there may be other generators as well which fix the first 5 points but map point "6" to a different target point, other than "7".

Instead of performing backtracking in the usual way, the new automorphism is immediately backtracked to level 6 to search for any further mappings of the form  $(6 \rightarrow p)$ , in which "6" and "p" belong to different classes of the automorphism partition. Any such mappings which can be extended to complete automorphisms would therefore also belong in the set  $S_5$ . There is no point trying to construct any more generators containing a mapping of the form  $(6 \rightarrow p)$  where both "6" and "p" belong to the same class in the automorphism partition. If these points were in the same class, this would indicate a set of automorphisms must have already been found which can be composed to map point "6" to point "p", and another generator automorphism containing this mapping would be redundant.

This method exhibits a number of savings over the previous algorithm. Rather than backtracking in the usual fashion, the generator is immediately backtracked to the current centraliser level. Also, when the next possible point mapping at the centraliser level is being built, only those points belonging to a different class in the automorphism partition to the source point need to be considered. Moreover, say the source point at the centraliser level  $L$  is  $P_{src}(L)$ , and all other possible mappings  $(P_{src}(L) \rightarrow p)$  are being considered, for those points "p" in a different automorphism class than  $P_{src}(L)$ . For a given point, "p", if it is discovered that  $(P_{src}(L) \rightarrow p)$  cannot be validly extended to a complete automorphism, then no further mappings of the form  $(P_{src}(L) \rightarrow q)$  need to be considered if p and q are in the same class of the automorphism partition. The automorphism partition can therefore also be used to assist in rejecting invalid mappings.

In summary, the conditions for remapping the source point,  $P_{src}(L)$ , at the centraliser level,  $L$ , become: if  $P_{src}(L)$  belongs to class  $C$  in the automorphism partition, then only point mappings of the form  $(P_{src}(L) \rightarrow p)$  need to be considered, where "p" is a single representative from each automorphism class different to  $C$ .

The automorphism partition must be updated whenever a new generator automorphism is constructed. Returning to the current example, only the identity automorphism has been processed and so the automorphism partition is still the original, in which each point is in a class of its own:

AUTPT	<i>Point:</i>	1	2	3	4	5	6	7	8	9
	<i>Class:</i>	1	2	3	4	5	6	7	8	9

In the example, the automorphism below was constructed:

$$\cdot (1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 5) (6 \rightarrow 7) (7 \rightarrow 6) (8 \rightarrow 9) (9 \rightarrow 8)$$

This automorphism belongs to the set of generators for the centraliser subgroup at level 5,  $H(5)$ , and contains two cycles of length 2, namely:

$$(6 \rightarrow 7) (7 \rightarrow 6) \text{ and } (8 \rightarrow 9) (9 \rightarrow 8)$$

The automorphism partition is updated by merging the classes of points belonging to the same cycle. Points 6 and 7 have their classes merged as do points 8 and 9. Therefore, the new automorphism partition becomes:

AUTPT	Point:	1	2	3	4	5	6	7	8	9
	Class:	1	2	3	4	5	6	6	7	7

The centraliser level, 6, is then backtracked to directly, and new mappings at this level of the form  $(6 \rightarrow p)$  must be considered. The source point, "6", at this level belongs to class 6 of the automorphism partition. Therefore, only a single representative point from each class different to class 6 needs to be considered for this point mapping. Point "8" is such a representative for the class 7. However, there is no automorphism fixing the first 5 points but mapping  $(6 \rightarrow 8)$ , and so the set of generator automorphisms for the centraliser subgroup of the first 5 levels is complete.  $S_5$  consists of the two automorphisms:

$$\begin{aligned} &\cdot (1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 5) (6 \rightarrow 6) (7 \rightarrow 7) (8 \rightarrow 8) (9 \rightarrow 9) \\ &\cdot (1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 5) (6 \rightarrow 7) (7 \rightarrow 6) (8 \rightarrow 9) (9 \rightarrow 8) \end{aligned}$$

This is in fact the complete centraliser subgroup of the first 5 levels,  $H_5$ .

Backtracking then continues in the usual fashion, and the next automorphism constructed is:

$$\cdot (1 \rightarrow 2) (2 \rightarrow 1) (3 \rightarrow 4) (4 \rightarrow 3) (5 \rightarrow 5) (6 \rightarrow 6) (7 \rightarrow 7) (8 \rightarrow 9) (9 \rightarrow 8)$$

No initial points are fixed here, so this generator automorphism belongs to the set  $S_0$ , which can be used to generate  $H_0$  - the complete automorphism group.

There are 3 cycles generated by this automorphism:

$$(1 \rightarrow 2) (2 \rightarrow 1), (3 \rightarrow 4) (4 \rightarrow 3) \text{ and } (8 \rightarrow 9) (9 \rightarrow 8).$$

The classes of points 1 and 2, 3 and 4, and 8 and 9 are therefore merged to form the new automorphism partition:

AUTPT	Point:	1	2	3	4	5	6	7	8	9
	Class:	1	1	2	2	3	4	4	5	5

Ordinary backtracking does not proceed, because no more automorphisms containing the initial mapping  $(1 \rightarrow 2)$  need to be generated. Instead, the backtrack immediately returns to the first level, and a new mapping for the first point is searched for. The possible target points,  $p$ , for the point mapping  $(1 \rightarrow p)$  are those points belonging to a different automorphism class to point 1. Once again, only a single representative from each such class is required. Therefore, a possible set of target points to be considered are:  $p = \{3, 5, 6, 8\}$

When  $p=3$  is tried, the initial mapping  $(1 \rightarrow 3)$  does extend to a valid automorphism:

$$\cdot (1 \rightarrow 3) (2 \rightarrow 4) (3 \rightarrow 1) (4 \rightarrow 2) (5 \rightarrow 5) (6 \rightarrow 8) (7 \rightarrow 9) (8 \rightarrow 6) (9 \rightarrow 7)$$

This is therefore also a generator automorphism for the centraliser subgroup  $H_0$ . This generator contains 4 cycles:

$$(1 \rightarrow 3) (3 \rightarrow 1), (2 \rightarrow 4) (4 \rightarrow 2), (6 \rightarrow 8) (8 \rightarrow 6) \text{ and } (7 \rightarrow 9) (9 \rightarrow 7)$$

The classes of each point within each cycle are merged, and the automorphism partition is updated to:

AUTPT	Point:	1	2	3	4	5	6	7	8	9
	Class:	1	1	1	1	2	3	3	3	3

Again, the backtrack returns immediately to level 1, to search for a new mapping for the initial point. However, the set of possible target points is now  $p = \{5, 6\}$ , where 5 is the class 2 representative and 6 is the class 3 representative.

Neither of these target points lead to a valid automorphism, and so the backtracking algorithm finishes at this point. A complete set of generator automorphisms has been constructed for the design - two generators for the centraliser subgroup of the first 5 levels, and two additional generators for the centraliser subgroup fixing none of the initial points.

The complete set of generators is therefore:

- (1→1) (2→2) (3→3) (4→4) (5→5) (6→6) (7→7) (8→8) (9→9)
- (1→1) (2→2) (3→3) (4→4) (5→5) (6→7) (7→6) (8→9) (9→8)
- (1→2) (2→1) (3→4) (4→3) (5→5) (6→6) (7→7) (8→9) (9→8)
- (1→3) (2→4) (3→1) (4→2) (5→5) (6→8) (7→9) (8→6) (9→7)

These mappings make up the generator sets  $S_5$  and  $S_0$  of the centraliser subgroups  $H_5$  and  $H_0$  respectively, as illustrated below:

$$\begin{array}{l}
 (1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 5) (6 \rightarrow 6) (7 \rightarrow 7) (8 \rightarrow 8) (9 \rightarrow 9) \\
 (1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 5) (6 \rightarrow 7) (7 \rightarrow 6) (8 \rightarrow 9) (9 \rightarrow 8) \\
 (1 \rightarrow 2) (2 \rightarrow 1) (3 \rightarrow 4) (4 \rightarrow 3) (5 \rightarrow 5) (6 \rightarrow 6) (7 \rightarrow 7) (8 \rightarrow 9) (9 \rightarrow 8) \\
 (1 \rightarrow 3) (2 \rightarrow 4) (3 \rightarrow 1) (4 \rightarrow 2) (5 \rightarrow 5) (6 \rightarrow 8) (7 \rightarrow 9) (8 \rightarrow 6) (9 \rightarrow 7)
 \end{array}
 \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} \begin{array}{l} S_5 \\ S_0 \end{array}$$

Once all the generators have been found, AUTPT is the class vector of the automorphism partitioning of the design. If two points belong to the same class of the final automorphism partition, then an automorphism must exist in the group of the design which maps one point onto the other.

The final automorphism partition of the example design above is given below:

AUTPT	Point:	1	2	3	4	5	6	7	8	9
	Class:	1	1	1	1	2	3	3	3	3

Point "5" is in a class of its own, and indeed every automorphism in the group of the design maps this point to itself. The first four and the last four points of the design are in separate classes, and the interaction between these points is illustrated by examining the complete group of the design.

From the constructed set of generators, the complete automorphism group of the design can be derived. In the implementation of the algorithm in this thesis, the derivation process is performed as each new generator is found, making use of the current state of the automorphism partition. The next section describes this process in detail, and works through an example to clarify some of the ideas.

### 4.5.3.5 DERIVING THE AUTOMORPHISM GROUP FROM THE GENERATORS

The previous section described the backtracking construction of the generators of the automorphism group of a given design. This section details how the complete group is derived as each new generator automorphism is constructed.

Assume that at some point in the execution of the backtracking algorithm, the following generator automorphism is constructed:

$$(1 \rightarrow 1) (2 \rightarrow 2) (3 \rightarrow 3) (4 \rightarrow 4) (5 \rightarrow 6) (6 \rightarrow 5) (7 \rightarrow 9) (8 \rightarrow 10) (9 \rightarrow 7) (10 \rightarrow 8)$$

This can be represented as follows in a two row matrix, where the source points correspond to the values in the top row and the target points correspond to the values in the bottom row of the matrix:

1	2	3	4	5	6	7	8	9	10
1	2	3	4	6	5	9	10	7	8

**1**
↑
**v**  
 non\_fix

Let `non_fix` denote the level at which the first source point is not fixed, and let `non_fix_source` be the source point at level `non_fix`. The generator above fixes the first 4 points, so `non_fix = 5`. Also, `non_fix_source = 5`, because the source points have been selected in increasing numerical order.

This generator is used to derive all the automorphisms in the group of the given design which keep the first four elements fixed. This is done by scanning the point mappings from level `non_fix` up to the last level, `v`, and comparing the classes of the source and target points in the automorphism partition. If a point mapping is discovered for which the source and target points belong to different classes, yet one of them belongs to the same class as the source point at level `non_fix`, `non_fix_source`, then their classes are merged and all resulting mappings are generated.

Let `L` be the level at which such a point mapping is discovered, and let `K` be the point which belongs to the same class as `non_fix_source`. If `K` is the source point at level `L`, then let `PK` be the target point at level `L`, otherwise `K` must be the target point and so `PK` becomes the source point at level `L`.

Firstly, the set of currently generated automorphisms (which will at least include the identity) are searched to find one which maps point `non_fix_source` to point `K`. Let this automorphism, containing the mapping (`non_fix_source` → `K`) be  $\emptyset$ . A new automorphism is then constructed from  $\emptyset$  which maps (`non_fix_source` → `PK`). This is done by composing  $\emptyset$  with either the generator automorphism or the inverse of the generator automorphism, depending on whether `K` is the source or the target point at level `L`. Let this newly constructed automorphism, mapping (`non_fix_source` → `PK`) be denoted by  $P_1$ .

Now that an automorphism mapping (`non_fix_source` → `PK`) has been constructed, all points from 1 up to `v` are scanned and any point, `PPK`, in the same class as `PK`, is used to search for a previously constructed automorphism mapping (`PK` → `PPK`). Once this is found, it is composed with  $P_1$  to produce an automorphism mapping (`non_fix_source` → `PPK`). All such automorphisms are found for every point `PPK` in the same class as `PK`. Each one is composed with all the automorphisms in the current centraliser subgroup, which consists of all the automorphisms currently generated which fix at least the first (`non_fix-1`) initial points. Every resulting automorphism is added to the group of the design.

On its own, the above description is fairly complicated. The example given below illustrates the automorphism derivation, and in particular the merging of the automorphism classes, and should help to clarify this process.

Consider the first four automorphisms in the group of a 2-(7,3,12) design:

(1→1) (2→2) (3→3) (4→4) (5→5) (6→6) (7→7)  
 (1→1) (2→2) (3→4) (4→3) (5→6) (6→5) (7→7)  
 (1→1) (2→3) (3→2) (4→5) (5→4) (6→7) (7→6)  
 (1→1) (2→3) (3→5) (4→2) (5→7) (6→4) (7→6)  
 .....

The generation of these automorphisms using the presented algorithm will be illustrated. The first automorphism constructed is always the identity, and this is always added directly to the group of the design. Assume that the source points are mapped in the specified order:

(1→1) (2→2) (7→7) (3→3) (6→6) (4→4) (5→5)



Each point will still be within its own class of the automorphism partition at this stage of the algorithm. The class vector, AUTPT, is given below:


AUTPT	<i>Point:</i>	1	2	7	3	6	4	5
	<i>Class:</i>	1	2	7	3	6	4	5

The backtracking algorithm then proceeds, and the very next generator automorphism constructed is:

$$(1 \rightarrow 1) (2 \rightarrow 2) (7 \rightarrow 7) (3 \rightarrow 4) (6 \rightarrow 5) (4 \rightarrow 3) (5 \rightarrow 6)$$

This generator fixes the first 3 points, as illustrated below:

$P_{src}$	1	2	7	3	6	4	5
$P_{tgt}$	1	2	7	4	5	6	3


  
 non\_fix

Any automorphism added to the group must be explicitly derived by the following process. The algorithm scans this mapping from level non\_fix up to level v - in this case from level 4 up to level 7. If it finds any point mapping where the source and target points are in different classes, yet one of them is in the same class as the source point at level non\_fix, then their classes are merged and the automorphism is composed with the previous automorphisms to extend the group.

The source point at level non\_fix belongs to class 3 of the automorphism partition. The steps performed at each level of the scanning process are given below, where  $AUTPT_i$  is the class of point i in the automorphism partition:

Level 4:

$$P_{src}(4) = 3. \quad AUTPT_3 = 3$$

$$P_{tgt}(4) = 4. \quad AUTPT_4 = 4$$

These are in different classes, and  $P_{src}$  is in class 3, so these two points have their classes merged, and this mapping is composed with the previously generated identity automorphism - to produce itself - which is then added to the group of the design. The automorphism partition now becomes:

AUTPT	<i>Point:</i>	1	2	7	3	6	4	5
	<i>Class:</i>	1	2	7	3	6	3	5

Level 5:

$$P_{src}(5) = 6. \quad AUTPT_6 = 6$$

$$P_{tgt}(5) = 5. \quad AUTPT_5 = 5$$

Neither point in the mapping at this level belongs to class 3, and so nothing is done.

Level 6:

$$P_{src}(6) = 4. \quad AUTPT_4 = 3$$

$$P_{tgt}(6) = 3. \quad AUTPT_3 = 3$$

The classes of these two points were just merged at level 4, and so again nothing further can be done.

Level 7:

$$P_{src}(7) = 5. \quad AUTPT_5 = 5$$

$$P_{tgt}(7) = 6. \quad AUTPT_6 = 6$$

Just as at level 5, neither point belongs to class 3, and so nothing can be done.

The scanning is now complete. At this stage, if there are any cycles in the point mapping for which the corresponding points still belong to different classes, these are now combined. This is the case for points 5 and 6, which participate in the cycle:  $(6 \rightarrow 5) (5 \rightarrow 6)$ . The classes of these two points are merged, to give the final automorphism partition:

AUTPT	<i>Point:</i>	1	2	7	3	6	4	5
	<i>Class:</i>	1	2	7	3	6	3	6

Automorphisms are only added to the group of the design once they have been explicitly constructed by composing a derived automorphism with a previous automorphism belonging to the current centraliser subgroup. The generator automorphisms themselves, which are constructed by the main backtracking algorithm, are always added to the group when they are composed with the identity automorphism, and this is exactly what happened in the example above.

The backtracking algorithm would continue at this point, searching for the next generator automorphism, which is given below:

$(1 \rightarrow 1) (2 \rightarrow 3) (7 \rightarrow 6) (3 \rightarrow 2) (6 \rightarrow 7) (4 \rightarrow 5) (5 \rightarrow 4)$

This only fixes the first point, and is composed with the identity automorphism to give itself, which is then added to the group. Without showing the steps in detail, it is also composed with the second generator automorphism, which was previously added to the group, to generate a new automorphism:

Compose:  $(1 \rightarrow 1) (2 \rightarrow 2) (7 \rightarrow 7) (3 \rightarrow 4) (6 \rightarrow 5) (4 \rightarrow 3) (5 \rightarrow 6)$   
 $(1 \rightarrow 1) (2 \rightarrow 3) (7 \rightarrow 6) (3 \rightarrow 2) (6 \rightarrow 7) (4 \rightarrow 5) (5 \rightarrow 4)$   
 $(1 \rightarrow 1) (2 \rightarrow 3) (7 \rightarrow 6) (3 \rightarrow 5) (6 \rightarrow 4) (4 \rightarrow 2) (5 \rightarrow 7)$

This newly composed automorphism is the fourth automorphism in the group of the design. In fact this must be constructed by composition, because the backtracking algorithm will not generate more than one automorphism containing the mapping  $(2 \rightarrow 3)$ .

The next section presents a detailed summary of the fast group generation algorithm, giving a useful overview of the entire process. This is followed by a selection of results in Section 4.5.3.7, in which the performance of the algorithm is analysed.

#### 4.5.3.6 SUMMARY OF THE FAST GROUP GENERATION ALGORITHM

- The main backtracking algorithm produces one of the generator automorphisms, and this is then used to extend the group of the given design.

The process of extending the current centraliser subgroup using the newly constructed generator automorphism is summarised below:

- If the automorphism is the identity, then it is simply added to the group.
- Otherwise, the important information about the new automorphism is the centraliser level, or level of the first non-fixed point mapping: `non_fix`, and the automorphism partitioning: `AUTPT`.
- The automorphism generator itself is a series of (source→target) point mappings stored in the vectors  $P_{src}$  and  $P_{tgt}$ .
- `non_fix_source` is the first non fixed source point, ie. the source point at level `non_fix`, and is simply calculated as  $P_{src}(non\_fix)$ .
- The class of `non_fix_source` in the automorphism partition is calculated as  $AUTPT_{P_{src}(non\_fix)}$ .

- Let  $j$  be a loop variable, starting at level `non_fix` and incrementing up to level  $v$ , that of the last point mapping in the generator. The group can only be extended if  $P_{src}(j)$  and  $P_{tgt}(j)$  are in different classes, yet one of them is in the same class as `non_fix_source`. Clearly, the generator automorphism will always be added to the group. This can easily be shown by letting  $C$  be the class of the point `non_fix_source`. When  $j$  is `non_fix` (at the start of the loop),  $P_{src}(\text{non\_fix})$ , which is `non_fix_source`, will obviously be in class  $C$ , and  $P_{tgt}(\text{non\_fix})$  must be in a different class to  $C$ , or else the generator would not have been constructed.
- When such a point mapping is detected,  $K$  becomes the element which is in the same class as `non_fix_source`, and  $PK$  is the element in the different class, which either maps to  $K$  or is mapped to from  $K$ .
- As it is known that `non_fix_source` and  $K$  are in the same class, there must be a previously generated automorphism,  $\emptyset$ , mapping (`non_fix_source`  $\rightarrow$   $K$ ). If  $K = \text{non\_fix\_source}$ , which happens at the start of the loop, then  $\emptyset$  is the identity automorphism. Otherwise,  $\emptyset$  is located in the current group. Once it is found, it is composed with the new generator, or the inverse of the new generator, to produce an automorphism mapping (`non_fix_source` $\rightarrow$  $PK$ ) and this is stored as  $P_1$ .
- All points from 1 up to  $v$  are then examined, for a point in the same class as  $PK$ . Let such a point be  $PPK$ . Once  $PPK$  is found, there must exist an automorphism in the current centraliser subgroup which maps ( $PK \rightarrow PPK$ ). This automorphism is searched for, and then composed with  $P_1$  to produce an automorphism mapping (`non_fix_source` $\rightarrow$  $PPK$ ). Let this automorphism be  $P_2$ . The class of  $PPK$  is set to be the same as the class of `non_fix_source`.
- Each  $P_2$  generated contains a mapping (`non_fix_source` $\rightarrow$  $PPK$ ) for  $PPK$  in the same class as the point mapped to by the point in the same class as `non_fix_source`. Every constructed  $P_2$  is therefore composed with each automorphism in the current centraliser subgroup which was constructed prior to the current generator. These automorphisms clearly keep `non_fix_source` fixed (ie. mapping to itself). All the automorphisms produced in this manner are added to the group.

### 4.5.3.7 RESULTS

The automorphism group generation algorithm was implemented again, this time using the techniques discussed throughout Section 4.5.3. For a given design, a set of generator automorphisms were constructed by the backtracking algorithm, with the rest of the group being derived from these, and the complete group was then stored in memory. It has been tested on a number of classes of designs, and some of the results are examined in this section.

There are exactly 2 non-isomorphic STS(13)'s. One has an automorphism group size of 6, and the other has a group size of 36. For both designs, 3 generators are sufficient for defining the group. The execution time to completely construct each group, and to store each automorphism in memory, is given below. In addition, the execution time of the previous algorithm in which the entire group was constructed by backtracking is given for comparison.

Design	Group Size	Number of generators	Execution Time (secs) for fast group generation	Execution Time (secs) of previous algorithm
1	36	3	0.00607	0.01550
2	6	3	0.00193	0.00217

In both the cases, the running times given include the time taken to induce the block, point and point pair partitions from a (4,1)-clique analysis of the block intersection graph of the design.

There are 80 non-isomorphic Steiner Triple Systems on 15 points. The group sizes of these designs are well known, and they are analysed in this section in the same order as the designs are listed in Stinson's paper on hill-climbing [63]. The fast group generation algorithm was used to generate the automorphism group of each STS(15). The results are given in the table on the next page. For each of the 80 designs, three execution times are quoted in the columns labelled A, B and C:

- A: This is the time taken to construct the set of generators for the group of the design. From these generators the complete group can be derived.
- B: This is the time taken to construct the set of generators and to derive the complete group from them, by expanding the centraliser subgroups. Each automorphism in the group is explicitly stored in memory.
- C: This is the time taken to construct the group using the previous algorithm in which every automorphism is constructed by backtracking. Again, each automorphism in the group is stored in memory.

In each case, the running times include the time taken to calculate the block, point and point pair partitions from a (4,1)-clique analysis. To produce accurate figures, all the quoted running times are averaged over 1000 executions of the group generation algorithm.

Design	G	Number of generators	A Time to construct generators (secs)	B Execution time of fast group generation algorithm (secs)	C Execution time of group generation by backtracking (secs)
1	20160	10	0.00545	4.2720	9.8000
2	192	8	0.00408	0.0440	0.0800
3	96	7	0.01870	0.0388	0.0630
4	8	4	0.00238	0.0038	0.0038
5	32	6	0.00353	0.0100	0.0140
6	24	5	0.00316	0.0081	0.0110
7	288	8	0.00736	0.0677	0.1910
8	4	3	0.00205	0.0027	0.0024
9	2	2	0.00168	0.0019	0.0017
10	2	2	0.00166	0.0019	0.0016
11	2	2	0.00195	0.0021	0.0019
12	3	2	0.00185	0.0023	0.0023
13	8	4	0.00238	0.0038	0.0037
14	12	4	0.00388	0.0062	0.0077
15	4	3	0.00213	0.0028	0.0025
16	168	6	0.03588	0.0714	0.3540
17	24	5	0.00505	0.0099	0.0165
18	4	2	0.00190	0.0025	0.0029
19	12	3	0.00231	0.0046	0.0066
20	3	2	0.00196	0.0024	0.0025
21	3	2	0.00185	0.0023	0.0023
22	3	2	0.00183	0.0025	0.0022
23	1	1	0.00153	0.0015	0.0015
24	1	1	0.00135	0.0013	0.0013
25	1	1	0.00138	0.0013	0.0013
26	1	1	0.00140	0.0014	0.0014
27	1	1	0.00140	0.0014	0.0014
28	1	1	0.00130	0.0013	0.0013
29	3	2	0.00185	0.0022	0.0024
30	2	2	0.00181	0.0028	0.0018
31	4	3	0.00245	0.0030	0.0029
32	1	1	0.00140	0.0014	0.0013
33	1	1	0.00130	0.0013	0.0013
34	1	1	0.00140	0.0014	0.0013
35	3	2	0.00201	0.0024	0.0027
36	4	3	0.00230	0.0029	0.0028
37	12	4	0.00350	0.0058	0.0083
38	1	1	0.00140	0.0014	0.0013
39	1	1	0.00130	0.0013	0.0013
40	1	1	0.00130	0.0013	0.0014
41	1	1	0.00130	0.0013	0.0013
42	2	2	0.00188	0.0021	0.0018
43	6	3	0.00240	0.0034	0.0039
44	2	2	0.00183	0.0020	0.0018
45	1	1	0.00130	0.0013	0.0013
46	1	1	0.00130	0.0013	0.0013
47	1	1	0.00130	0.0013	0.0013
48	1	1	0.00140	0.0014	0.0013
49	1	1	0.00130	0.0013	0.0013
50	1	1	0.00130	0.0013	0.0013
51	1	1	0.00130	0.0013	0.0013
52	1	1	0.00130	0.0013	0.0013
53	1	1	0.00130	0.0013	0.0013
54	1	1	0.00130	0.0013	0.0013
55	1	1	0.00140	0.0014	0.0013
56	1	1	0.00140	0.0014	0.0013
57	1	1	0.00140	0.0014	0.0014
58	1	1	0.00130	0.0013	0.0013
59	3	2	0.00178	0.0022	0.0022
60	1	1	0.00130	0.0013	0.0013
61	21	3	0.00310	0.0073	0.0171
62	3	2	0.00183	0.0022	0.0023
63	3	2	0.00178	0.0022	0.0023
64	3	2	0.00318	0.0036	0.0038
65	1	1	0.00140	0.0014	0.0013
66	1	1	0.00130	0.0013	0.0013
67	1	1	0.00130	0.0013	0.0013
68	1	1	0.00140	0.0014	0.0013
69	1	1	0.00140	0.0014	0.0013
70	1	1	0.00130	0.0013	0.0013
71	1	1	0.00130	0.0013	0.0013
72	1	1	0.00130	0.0013	0.0013
73	4	3	0.00225	0.0029	0.0027
74	4	3	0.00220	0.0028	0.0026
75	3	2	0.00186	0.0022	0.0023
76	5	2	0.00208	0.0029	0.0044
77	3	2	0.00315	0.0036	0.0048
78	4	3	0.00230	0.0029	0.0027
79	36	5	0.00338	0.0107	0.0185
80	60	5	0.08191	0.0951	0.7170

Statistics summarising the performance of the algorithm can be calculated from these results. For example, a complete set of generator automorphisms for all of the 80 STS(15)'s can be generated in a total time of 0.292 seconds, which includes the time taken to induce the block, point and point pair partitions for each design, and then backtrack for the generators.

A design,  $D$ , is  $n$ -transitive if for any two  $n$ -tuples  $(x_1, x_2, \dots, x_n)$  and  $(y_1, y_2, \dots, y_n)$  of the point set of  $D$ , there exists an automorphism  $\emptyset$  of  $D$  such that  $\emptyset(x_1, x_2, \dots, x_n) = (y_1, y_2, \dots, y_n)$ . For the STS(15)'s, design number 1 with group size 20160 is 2-transitive, and design number 80 with group size 60 is 1-transitive. Section 4.5.3.8 briefly discusses the difficulty of group generation for such designs.

Every automorphism in the group of each design is explicitly constructed and stored in a large data structure. The complete groups of all STS(15)'s can be constructed and stored in 4.7 seconds, with 4.2 seconds being required for the 2-transitive design, and half a second for the remaining 79 designs.

The following section presents several observations of the current algorithm relevant to its role in the constructive enumeration algorithm, and suggests possible optimisations.

### 4.5.3.8 IMPROVEMENTS AND OBSERVATIONS

The efficiency of the group generation algorithm depends to a large extent on the point pair partition induced from the clique analysis. The point pair partition, stored in the structure  $VV$ , is used to tighten the point partition as each new point mapping is accepted into the partial automorphism.

For designs which are 2-transitive, such as the STS(15) #1 given in the table of Section 4.5.3.7, for any 2 pairs  $(a,b)$  and  $(c,d)$  of the point set, there exists an automorphism  $\emptyset$  in the group of the design such that  $\emptyset(a,b) = (c,d)$ . Clearly, for such a design, no pair of points can be partitioned into a different class than any other pair of points, and so the point pair partition can not contain any useful information. Consequently, the construction of the automorphism group of such designs by the presented backtracking algorithm will be relatively expensive.

Another drawback is encountered when symmetric designs are considered. A symmetric design has the extra property that any pair of blocks intersect in exactly  $\lambda$  points. The clique analysis of the block intersection graph of a design, from which the partitions of the search are induced, only takes account of pairwise block intersections. For this reason, it can not produce any distinct classes for the partitions of symmetric designs.

Such difficulties can be overcome by using more sophisticated block intersection analyses, such as the  $N$ -way block intersection analysis used by Gibbons in [32].

Despite these drawbacks, the current implementation of the algorithm performs the task it is designed to do extremely well. The main purpose of the automorphism group generation algorithm in this thesis is to quickly calculate the groups of partial designs - ie. designs for which only an initial set of rows of the incidence matrix are constructed. In general, partial designs consist of a number of complete, incomplete and even empty blocks, regardless of the structure of the design being generated. This allows very strong initial partitions to be calculated. Any blocks which contain a different number of points can certainly be placed into different classes of the partitions. Even for symmetric designs, it is often only once the design is completed that calculating strong partitions becomes difficult.

Partial designs also usually consist of a number of cells, which contain identical partial blocks. Only one representative from each cell takes part in the group generation algorithm, as the effect of a point mapping on a single block is exactly the same as its effect on a set of identical blocks. Not only does this speed up the algorithm, but cell structure information can also be used to strengthen the partitions, because any two blocks

representing cells of different sizes must belong to different classes of the initial block partition.

## 4.6 CONCLUSION

Using the fast automorphism group generation algorithm presented in Section 4.5.3, the isomorph rejection techniques discussed in Section 4.4 can be implemented. A given partial design can have its automorphism group quickly calculated, from which the unique set of cell permutations which reconstruct the original matrix can be generated.

The next chapter deals with constructive enumeration, in which a catalogue of all pairwise non-isomorphic designs are constructed for a given set of parameters. The isomorph rejection techniques discussed in this chapter are used to make the incidence matrix backtracking algorithm considerably more powerful, and two different techniques are studied for the non-isomorphic cataloguing of the generated designs. One method examines the explicit construction of lists of pairwise non-isomorphic designs, and the other method involves the development of an efficient canonicity test which detects non-isomorphic designs and has a number of advantages over the list generation method.

# Chapter 5

## Constructive Enumeration

### 5.1 CLASSICAL CONSTRUCTIVE ENUMERATION

A classical problem in design theory is that of constructive enumeration, in which all the non-isomorphic designs of a particular class are generated and then processed in various ways. The procedure often involves cataloguing, or storing the designs, so that an analysis of their properties can easily be performed by examination of the catalogue. Such analysis can be useful for providing counter-examples to existing mathematical conjectures on the corresponding design class, or to establish patterns which can lead to the formulation of new conjectures. For very large classes of designs, which can not feasibly be stored, the analysis can take place as the designs are generated. At the very least a count of all the non-isomorphic designs, and possibly some simple classification statistics, would usually be performed.

A great deal of research has gone into the constructive enumeration of combinatorial objects. The most popular approaches are those which generate a single representative of each isomorphism class, a technique which was pioneered independently by Faradzhev [28] and Read [56]. Read gave the name *orderly algorithms* to such approaches, and these methods have been used successfully for the constructive enumeration of a wide range of structures. Good examples of such applications are the recent algorithms developed by Meringer [53] for the fast generation of regular graphs and by Brinkmann [9] for the fast generation of cubic graphs. Another celebrated result is that of Dinitz, Garnick and McKay [22] who enumerated all 526,915,620 non-isomorphic one-factorisations using an orderly algorithm. Section 6.2.3 of Case Study Two of Chapter 6 compares the enumeration algorithm developed in this thesis with a true orderly algorithm as defined by Read.

### 5.2 EXHAUSTIVE INCIDENCE MATRIX CONSTRUCTION

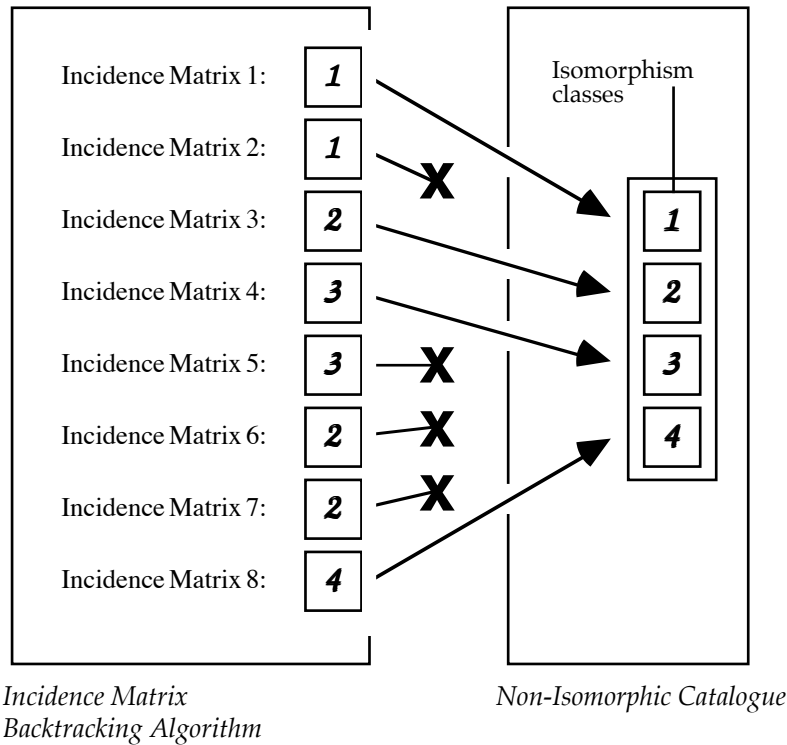
The exhaustive incidence matrix backtracking algorithm, which was developed in the previous chapters, is a powerful underlying tool for performing the constructive enumeration of incidence structures.

Initially, it was used to construct all distinct designs of a given class, however the limitations of such an approach became quickly apparent. The isomorph rejection techniques covered in Chapter 4 considerably reduced the number of designs constructed by the algorithm, by pruning large sections of the search space which were isomorphic to parts already searched. The only designs eliminated from construction are those which are isomorphic to designs already built, and at least one design from each isomorphism class is guaranteed to be generated. Moreover, because the algorithm constructs designs in strictly increasing lexicographical order, each design generated which is non-isomorphic to all previously constructed designs is the lexicographically smallest design in its isomorphism class.

A schematic diagram of one approach to the constructive enumeration process is given overleaf. Assume that there exist exactly 4 isomorphism classes for the designs being



enumerated. The non-isomorphic catalogue should contain exactly one representative design from each isomorphism class - 4 designs in total. Further, assume that the incidence matrix construction process generates 8 incidence matrices in the order given, which belong to the isomorphism classes as specified in the diagram.



The first design generated in each isomorphism class is added to the non-isomorphic catalogue. All subsequently generated designs which belong to the same class as a design already catalogued are simply rejected. Clearly, the catalogue built by such a process will contain the lexicographically smallest representative design from each isomorphism class.

The main focus of this chapter is the development of cataloguing tools which enable the incidence matrix backtracking algorithm to classify all non-isomorphic designs of a particular class. Before such techniques are discussed, two effective optimisations to the backtracking algorithm are presented which further improve its performance.

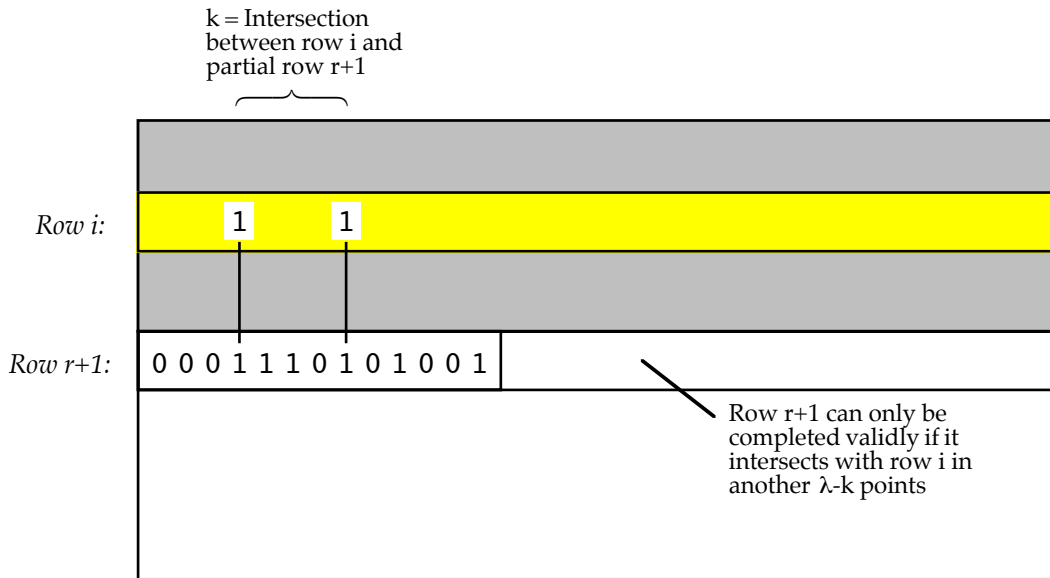
### 5.2.1 OPTIMISATIONS

This section introduces two new optimisations for the algorithm. The first optimisation detects early on in the construction of a row of the incidence matrix whether or not it can validly be completed, and is therefore a "lookahead" feasibility property - very similar to the ones introduced in Chapter 3. This optimisation was developed from the packing constraint used by Gibbons in [32]. The second optimisation is an extension of the automorphism group isomorph rejection, and eliminates a lot of unnecessary backtracking once a row has been rejected, by calculating the leftmost position within the row at which backtracking can immediately begin.

Following the description of the optimisations is a short section of results which summarise their effectiveness.

### 5.2.1.1 PACKING CONSTRAINT

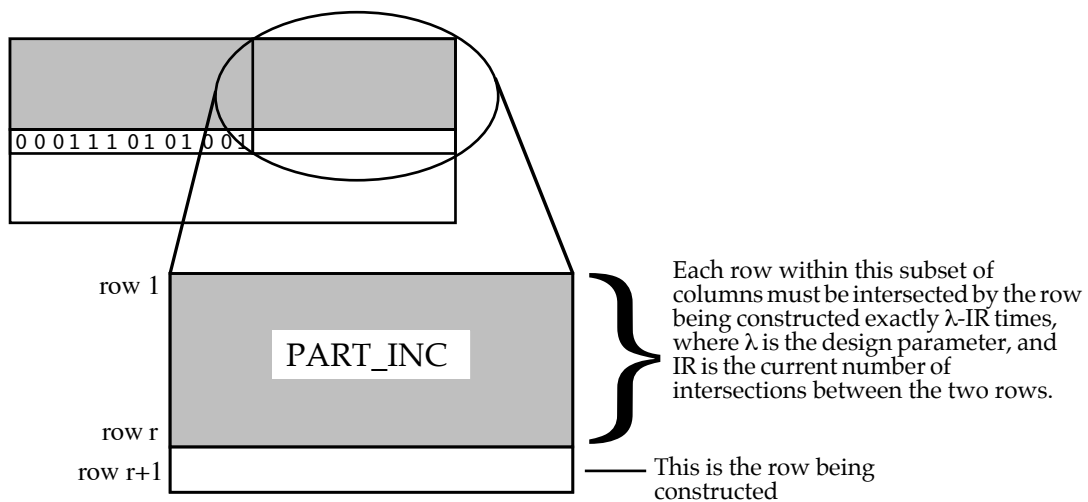
This optimisation is fairly expensive to perform, but can detect a violation at a very early stage in the construction of a row, thus eliminating a great deal of unnecessary search. Essentially this constraint performs a calculation to determine whether or not it is possible to validly complete a partially constructed row, so that the intersection between the row being constructed and all previous rows will be exactly  $\lambda$ . For instance, the diagram below illustrates a typical situation when the packing constraint would be performed:



The partial incidence matrix has been validly constructed to row  $r$ , and then construction on row  $r+1$  begins. Whenever a point is placed in column  $j$  of row  $r+1$ , a new intersection is formed between row  $r+1$  and any previous row also containing a point in column  $j$ . This reduces the number of intersections between these rows which need to be satisfied by the remaining points placed on row  $r+1$ . As soon as a point is placed on row  $r+1$  such that it is no longer possible to satisfy the remaining intersection requirements, the construction of row  $r+1$  can stop, and backtracking can begin.

#### The Calculation

The section of the incidence matrix of interest is the subset of columns for which the addition of a point on the row being constructed has not yet been considered.



If row  $r+1$  is being constructed as in the above diagram, then let PART\_INC be the section of the incidence matrix consisting of the first  $r$  rows and the subset of columns which have not yet been considered in the construction of row  $r+1$ . For each row  $i$ , for  $1 \leq i \leq r$ ,  $\lambda_i$  is the number of intersections still required between row  $i$  and the row being constructed, row  $r+1$ . The PART\_INC structure is used to derive a square matrix,  $T$ , in the following manner:

If row  $r+1$  is the row under construction, then  $T$  is a square,  $r \times r$  matrix. Let  $T_{ij}$  denote the entry in the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column of  $T$ .  $T_{ij}$  equals the number of columns of PART\_INC which contain exactly  $j$  points, including the point  $i$ .

For example, assume the incidence matrix of a particular design has been constructed validly to row 5, and row 6 is under construction. If PART\_INC contains the partial blocks:  $\{1, 2, 3\}$   $\{1, 2, 4\}$   $\{1, 3, 5\}$   $\{1, 4\}$   $\{2, 3, 5\}$   $\{2, 4, 5\}$   $\{3, 4, 5\}$

then the incidence structure of PART\_INC will look like the following:

1	1	1	1	0	0	0
1	1	0	0	1	1	0
1	0	1	0	1	0	1
0	1	0	1	0	1	1
0	0	1	0	1	1	1

In this example, the square matrix,  $T$ , would be derived as follows:

<b>T</b>		<b>j</b>				
		<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b>i</b>	<b>1</b>	0	1	3	0	0
	<b>2</b>	0	0	4	0	0
	<b>3</b>	0	0	4	0	0
	<b>4</b>	0	1	3	0	0
	<b>5</b>	0	0	4	0	0

The following vectors are then calculated for each row  $i$ , for  $1 \leq i \leq r$ :

$U_i (\leq r)$  is the smallest integer for which  $\sum_{j=1}^{U_i} T_{ij} \geq \lambda_i$

and  $V_i (\leq r)$  is the largest integer for which  $\sum_{j=V_i}^r T_{ij} \geq \lambda_i$

Then, the two rational vectors  $p_i$  and  $q_i$  are defined as follows:

$$p_i = \left( \lambda_i - \sum_{j=V_i+1}^r T_{ij} \right) / V_i + \left( \sum_{j=V_i+1}^r T_{ij} \right) / j$$

$$q_i = \left( \lambda_i - \sum_{j=1}^{U_i-1} T_{ij} \right) / U_i + \left( \sum_{j=1}^{U_i-1} T_{ij} \right) / j$$

Finally, the two rational numbers  $p$  and  $q$  are calculated:

$$p = \sum_{i=1}^r p_i \text{ and } q = \sum_{i=1}^r q_i.$$

If the partial incidence matrix, PART\_INC contains any completely empty columns, ie. corresponding to blocks containing no points, then these columns will be the extreme right columns due to the ordering imposed on the matrix. Let  $s$  denote the number of such empty columns.

If row  $r+1$ , which is under construction, still requires another  $c$  points, then the packing constraint evaluates the following inequality:

$$\lfloor p \rfloor \leq c \leq \lfloor q \rfloor + s$$

If this inequality does not hold, then it is not possible to complete the construction of row  $r+1$  and satisfy the necessary intersection requirements. As soon as this is detected, backtracking can begin on the row. Particularly if this is detected early on in the construction of a row, a considerable amount of searching can be eliminated.

For a proof of the correctness of this packing constraint, refer to Gibbons [32].

Due to the expense of this constraint, it is not efficient to perform it each time a new point is added to a row. One effective strategy is to stop performing the calculations of the constraint when a given row is nearly complete, and to not perform it at all on the last  $n$  rows of the incidence matrix, for a specified value of  $n$  depending on the particular problem.

The current implementation of the packing constraint performs floating point arithmetic to calculate the values of the rational vectors. Another version of the packing constraint was also implemented in which only integer arithmetic was performed. This approach used exact fractional representations of the rational numbers, but had a greater overhead and did not perform as well as the floating point version.

### Results

The effectiveness of the packing constraint is best analysed by examining its performance on a number of enumerations. The incidence matrix backtracking algorithm was used, with partial isomorph rejection, to generate all the designs for a number of parameter sets. Each set of designs were constructed twice - once without the packing constraint and once with the packing constraint. The results are summarised into tables which give the following information for each of these two runs:

#### # Extend Operations

*the total number of times a row is extended*

Recall that row  $r$  is only extended once the first  $r-1$  rows are validly constructed and cannot be eliminated by the partial isomorph rejection. The packing constraint only reduces the amount of backtracking performed, and has no effect on the number of row extensions. The same number of such operations are therefore performed regardless of whether or not the packing constraint is used.

#### # Backtrack Operations

*the total number of times a row is backtracked*

This is exactly what is targeted by the packing constraint. By detecting early on in the construction of a row that the row cannot be completed, a large amount of unnecessary backtracking can be eliminated.

#### # Point Placements

*the number of times an individual point is placed on a row during row construction*

Again, this should be significantly reduced by the use of the packing constraint. The early detection of a packing violation prevents a large number of unnecessary point placements.

#### # Designs

*the total number of complete, valid designs constructed*

The packing constraint reduces the amount of unnecessary backtracking performed on each row, but does not reject any validly constructed rows. Therefore, exactly the same number of completed designs are generated whether or not the packing constraint is being used.

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(6,40,20,3,8)	# Extend Operations	50	# Extend Operations	50
15 Designs	# Backtrack Operations	14837	# Backtrack Operations	3049
	# Point Placements	53152	# Point Placements	7922

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(6,80,40,3,16)	# Extend Operations	428	# Extend Operations	428
123 Designs	# Backtrack Operations	2,170,304	# Backtrack Operations	99,764
	# Point Placements	11,120,800	# Point Placements	249,613

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(7,42,18,3,6)	# Extend Operations	9006	# Extend Operations	9006
2334 Designs	# Backtrack Operations	1,159,137	# Backtrack Operations	334,912
	# Point Placements	3,548,652	# Point Placements	725,766

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(7,49,21,3,7)	# Extend Operations	35,620	# Extend Operations	35,620
8821 Designs	# Backtrack Operations	7,494,309	# Backtrack Operations	1,596,649
	# Point Placements	24,055,287	# Point Placements	3,411,920

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(7,56,24,3,8)	# Extend Operations	129,911	# Extend Operations	129,911
32038 Designs	# Backtrack Operations	42,966,251	# Backtrack Operations	6,877,343
	# Point Placements	145,431,106	# Point Placements	14,590,624

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(7,63,27,3,9)	# Extend Operations	433,118	# Extend Operations	433,118
105955 Designs	# Backtrack Operations	209,498,211	# Backtrack Operations	26,041,452
	# Point Placements	745,501,990	# Point Placements	55,059,139

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(8,28,14,4,6)	# Extend Operations	105,424	# Extend Operations	105,424
17890 Designs	# Backtrack Operations	32,094,640	# Backtrack Operations	6,532,086
	# Point Placements	62,981,338	# Point Placements	10,141,650

Design	<i>Without Packing Constraint</i>		<i>With Packing Constraint</i>	
(9,24,8,3,2)	# Extend Operations	2153	# Extend Operations	2153
344 Designs	# Backtrack Operations	31284	# Backtrack Operations	24749
	# Point Placements	69133	# Point Placements	51273

These results clearly demonstrate the effectiveness of the constraint - in the case of the 2-(6,3,16) enumeration reducing the number of backtrack operations by a factor of 20. The execution times of the enumerations are not reduced by quite the same factor, because there is a considerable overhead in the calculations. However, by maintaining the data structures efficiently throughout the search and by reusing the results of previous calculations, this constraint performs very well, particularly as the number of blocks in the designs increases, so that the corresponding incidence matrices become wider.

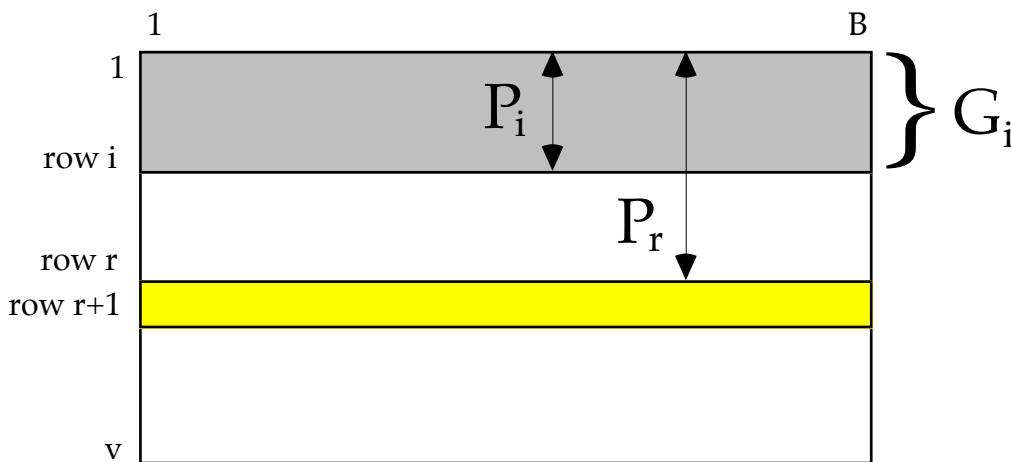
### 5.2.1.2 STRONG PARTIAL ISOMORPH REJECTION

This optimisation increases the potency of the partial isomorph rejection technique, by determining the largest amount of backtracking which can be performed in a single step following the rejection of a row.

The current partial isomorph rejection technique reduces the search space by performing column permutations derived from each automorphism of the group of a partial design on each newly constructed row. As soon as a set of column permutations is found which maps the complete row to a lexicographically smaller configuration, using the cell structures defined at each level of the incidence matrix, the row is rejected. Here, rejection simply means not generating any extension to the partial incidence matrix, but instead backtracking on the row.

The idea of strong partial isomorph rejection is to not reject the row as soon as possible, but to examine every automorphism in the group of each partial design before determining whether or not to reject a row. The reason for testing further automorphisms once it has already been decided that the row should be rejected, is so that information about the effect of each automorphism on the rejected row can be used to backtrack over a number of columns of the row in a single operation, thus eliminating large portions of the search space.

For example, consider the following partial incidence matrix below, constructed validly on the first  $r$  rows,  $P_r$ , and with a newly constructed row  $r+1$ :



$G_i$  is the automorphism group of the partial design  $P_i$ , for  $2 \leq i \leq r$ . Let  $CP_i$  represent the set of all column permutations which reorder the columns of  $P_i$  following the application of an automorphism from  $G_i$ .

For determining whether or not row  $r+1$  can be rejected, the current partial isomorph rejection technique can be summarised by the following algorithm:

```

for each  $i$  from 2 up to  $r$ 
  for each column permutation  $\emptyset \in CP_i$  do
    if  $\emptyset$  maps row  $r+1$  lexicographically backwards
      reject row  $r+1$  and begin backtracking
  
```

In comparison, an outline of the algorithm for the strong partial isomorph rejection is given below:

```

for each  $i$  from 2 up to  $r$ 
  for each column permutation  $\emptyset \in CP_i$  do
    if  $\emptyset$  maps row  $r+1$  lexicographically backwards
      calculate position LEFT_MOST on row  $r+1$ 
    if row  $r+1$  is to be rejected then
      begin backtracking from the minimum LEFT_MOST position
  
```

The strong partial isomorph rejection therefore tests every single column permutation from the group of each partial design, and if the set of permutations are able to reject the row,

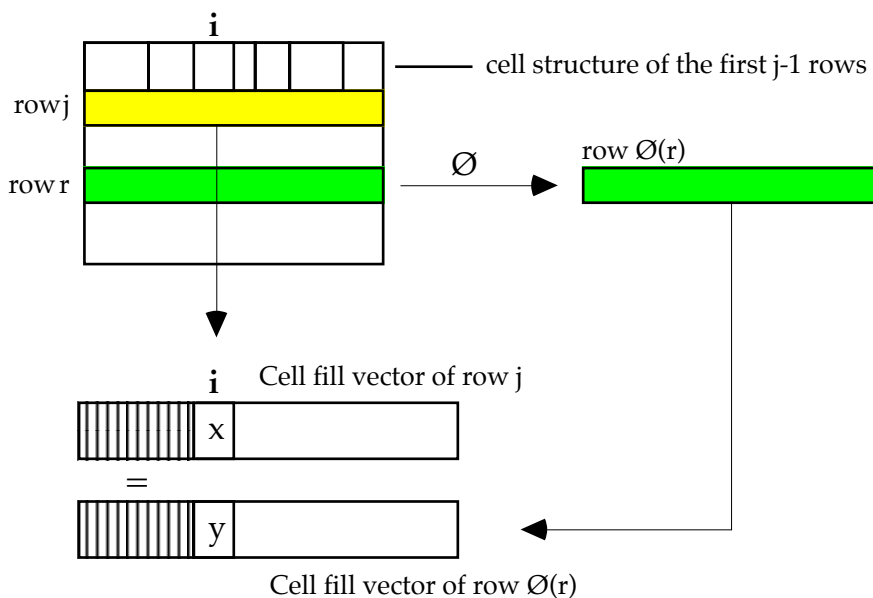
then it calculates the position LEFT\_MOST. This position specifies the leftmost column on row r+1 such that the current configuration of points from column 1 up to column LEFT\_MOST will always be rejected by the corresponding set of column permutations. In other words, for each value of LEFT\_MOST calculated, there exists a particular column permutation, say  $\emptyset$ , in one of the sets  $CP_i$ , which will always reject row r+1, regardless of the configuration of the points in the columns to the right of LEFT\_MOST. Let OVERALL\_LEFT\_MOST denote the smallest, or equivalently the leftmost, of all the LEFT\_MOST positions calculated.

If this can be calculated efficiently, then it is a very important piece of information, because it means none of the points on row r+1 to the right of column OVERALL\_LEFT\_MOST need to be repositioned. Even if these points are repositioned, there still exists a column permutation  $\emptyset$  which will map the row to a lexicographically smaller configuration. Therefore, all the points in the columns to the right of OVERALL\_LEFT\_MOST can simply be removed, and backtracking can then begin from column OVERALL\_LEFT\_MOST. The further to the left of the row that this position is calculated to be, and the more points which are placed to the right of this position, the greater the savings of this approach.

**Implementation**

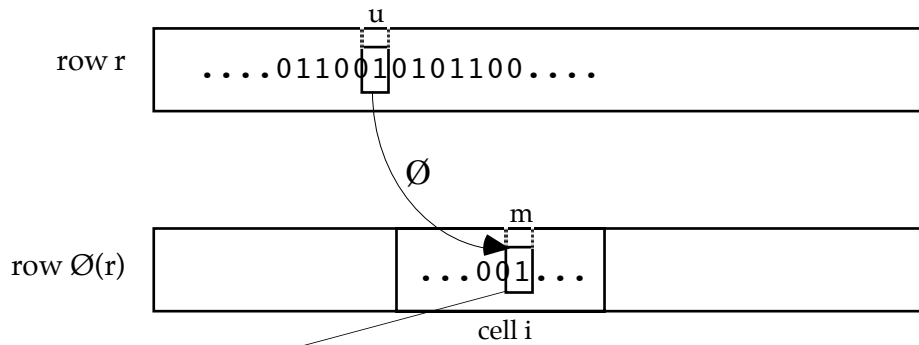
Strong partial isomorph rejection certainly has the potential to be more effective than the current scheme, but only if the extra computations required can be performed without too much overhead. This section presents an efficient implementation of the calculation of the position OVERALL\_LEFT\_MOST, which is the smallest of all the LEFT\_MOST values calculated over each set of column permutations.

To calculate the value of LEFT\_MOST for a particular row and a particular set of column permutations, it is necessary to first review how a row is rejected. Let row r be the row which is to be rejected, and let  $\emptyset$  represent the set of column permutations which correspond to an automorphism of the group of the first j-1 rows. Row r is rejected if  $\emptyset(r) < j$ , where rows  $\emptyset(r)$  and j are compared by their cell fill vectors under the cell structure of the first j-1 rows. This is illustrated in the diagram below:



Row r is rejected if the cell fill vector of row  $\emptyset(r)$  has the same initial sequence as the cell fill vector of row j, but the next entry in the cell fill vector of row  $\emptyset(r)$ , y, is greater than the next entry in the cell fill vector of row j, x. Let the position of this entry be i, as illustrated in the diagram above. This means that the permutation of row r,  $\emptyset(r)$ , has exactly the same number of points as row j in each cell c for  $1 \leq c < i$ , but a greater number of points than row j in cell i, under the cell structure imposed by the first j-1 rows of the incidence matrix. Cell i is therefore of particular significance.

It is an extra point in cell  $i$  of row  $\emptyset(r)$ , say in column  $m$ , which causes the rejection of row  $r$ . Let  $u$  represent the column of row  $r$  in which the point occurs which maps to column  $m$  of row  $\emptyset(r)$ , under the mapping of  $\emptyset$ . The diagram below schematically represents this situation:



This point in cell  $i$  of the permuted row,  $\emptyset(r)$ , is responsible for the fill of cell  $i$  being greater than the corresponding cell fill of some previous row - causing the rejection of row  $r$

For a given row  $r$ , which can be rejected by a particular set of column permutations, say  $\emptyset$ , the calculation of LEFT\_MOST requires the following two steps:

- 1) The location of column  $m$  in the mapped row,  $\emptyset(r)$ . This is the column of the mapped row in which the point exists which is explicitly responsible for the rejection of row  $r$ .
- 2) The scanning of the mapped row,  $\emptyset(r)$ , from column  $m$  backwards to column 1, and for each column in this range containing a point, examining the mapping  $\emptyset$  to calculate the column of the original row,  $r$ , from which the point came.

The rightmost position of all the points on the original row which are located in step 2 above, then becomes the position LEFT\_MOST for the corresponding column permutation,  $\emptyset$ . The very leftmost, or smallest value of LEFT\_MOST calculated over every column permutation then becomes the position OVERALL\_LEFT\_MOST for the row. When the row is rejected, instead of backtracking as usual, all the points placed in every column to the right of OVERALL\_LEFT\_MOST are simply removed, and backtracking begins on column OVERALL\_LEFT\_MOST. This guarantees that the next placement of points constructed on the row has a different configuration in the first OVERALL\_LEFT\_MOST columns. If it did not, then the row would once again be rejected by the existing column permutations.

### 5.2.1.3 RESULTS

Both optimisations presented in Section 5.2.1 reduce the amount of unnecessary backtracking performed on a given row by calculating, in their own way, the leftmost invalid position on the row. As is expected, both constraints offer greater savings as the width of the incidence matrix grows - or as the number of blocks in the corresponding design increases - because a greater amount of unnecessary search can be eliminated.

The  $2$ -( $7,3,\lambda$ ) BIBDs exist for all values of  $\lambda$ . For  $5 \leq \lambda \leq 9$ , the incidence matrix backtracking algorithm was used to exhaustively generate these designs, without performing any isomorphism testing of completed designs. The number of blocks in each design is  $7\lambda$ , so as  $\lambda$  increases, the corresponding incidence matrices become wider.

The table on the next page summarises the effectiveness of each of the optimisations, by comparing the running times of the algorithm when neither of the optimisations are used, when each is used on its own, and when they are used together.



Design	Number of complete designs constructed	Execution Time (secs)			
		Without packing constraint or strong partial isomorph rejection	With strong partial isomorph rejection only	With packing constraint only	With both packing constraint and strong partial isomorph rejection
2-(7, 3, 9)	85726	661.800	607.300	274.000	235.000
2-(7, 3, 8)	26181	144.400	131.700	75.400	71.100
2-(7, 3, 7)	7191	27.033	25.365	18.239	16.361
2-(7, 3, 6)	1929	5.516	5.083	4.450	4.083
2-(7, 3, 5)	440	1.000	0.950	0.950	0.900

As the table above exhibits, both the packing constraint and strong partial isomorph rejection perform very well - particularly when used together and as the number of blocks in the designs increases.

### 5.3 ISOMORPH CLASSIFICATION

At this stage, the incidence matrix backtracking algorithm efficiently traverses the search space of any given incidence structure, exhaustively generating valid designs, and using the partial isomorph rejection techniques to eliminate those designs which are detected in some way to be isomorphic to designs already constructed.

If every design generated was explicitly tested for isomorphism against all previously generated designs, a complete catalogue of pairwise non-isomorphic designs, containing exactly one design from each isomorphism class, would be constructed. This technique would therefore solve the classical constructive enumeration problem for any class of designs which could be encoded in the incidence matrix backtracking algorithm.

This main section describes the two methods implemented in this thesis for classifying only pairwise non-isomorphic designs:

- list building, in Section 5.3.1, and
- canonicity testing, in Section 5.3.2.

#### 5.3.1 LIST BUILDING

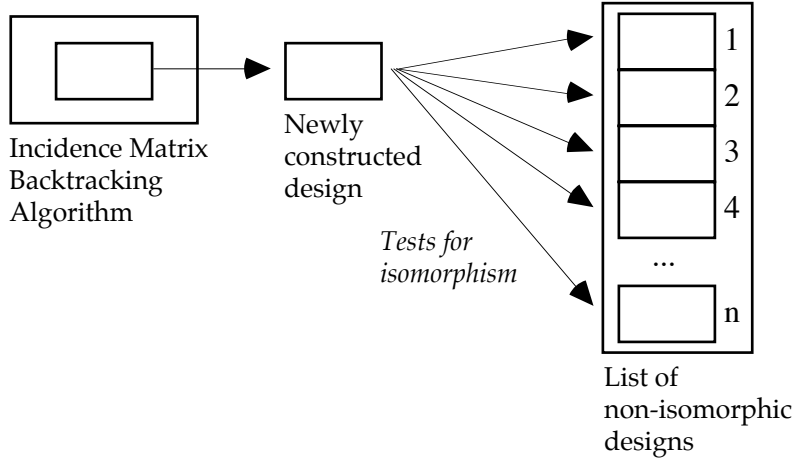
In the first approach, the non-isomorphic catalogue is constructed by storing each design in an explicit list, in memory. An algorithm for testing whether two designs are isomorphic must first be developed, and with such a tool, the implementation of this approach is straightforward.

##### 5.3.1.1 INCIDENCE STRUCTURE ISOMORPHISM TESTING

As defined in Section 4.2, two designs,  $D_1$  and  $D_2$ , on the point set  $V$ , are isomorphic if there exists a mapping  $\emptyset : V \rightarrow V$  such that for any block  $b$  of  $D_1$ ,  $\emptyset(b)$  is a block of  $D_2$ .

For any design on  $v$  points, there are exactly  $v!$  possible mappings. A simple way to test two designs,  $D_1$  and  $D_2$ , for isomorphism is therefore to apply each of the  $v!$  possible mappings to  $D_1$  and see if this maps it onto  $D_2$ . If such a mapping is found, then the designs are isomorphic, otherwise the designs are non-isomorphic. This simple test can clearly be used to extend the incidence matrix backtracking algorithm to solve the constructive enumeration problem for any given class of designs. A list of designs is explicitly constructed, with the property that every design in the list is non-isomorphic to every other design in the list. The incidence matrix backtracking algorithm produces a sequence of designs, and an attempt

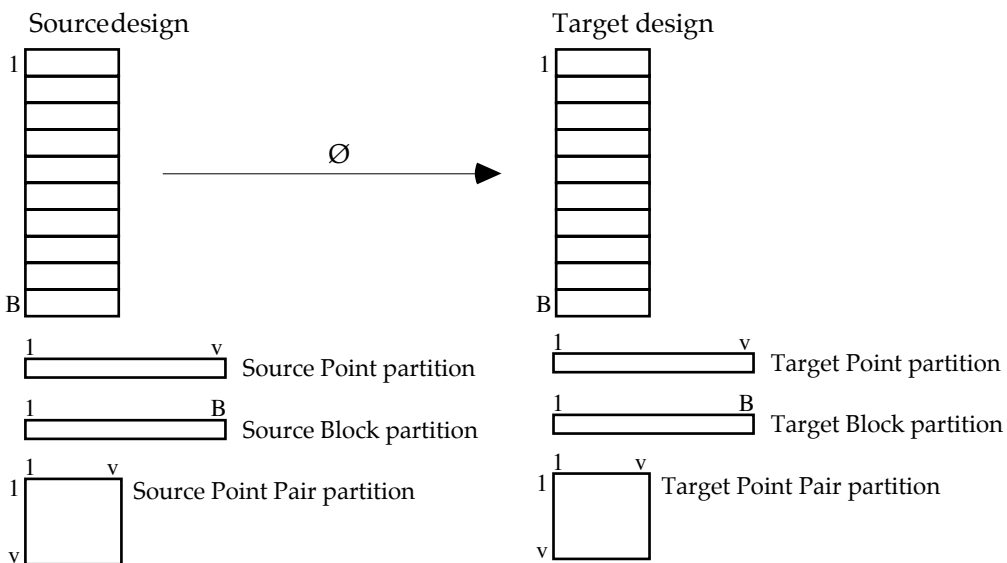
is made to add each one to the list. Certainly, the first design generated must be added to the list, because initially the list is empty. Assume that at a particular stage of the algorithm there are  $n$  designs in the list, and a new design is constructed by the incidence matrix backtracking algorithm. This design is then tested for isomorphism against each of the  $n$  designs in the list. If it is found to be isomorphic to any of the  $n$  designs, then it is rejected, and backtracking on the incidence matrix can continue. Otherwise, if it is found to be non-isomorphic to each of the  $n$  designs in the list, then it can be added to the list as the first constructed representative of its isomorphism class. The diagram below gives an outline of such an approach.



**Improving the Isomorphism Test**

For any designs on more than about 8 or 9 points, a brute force isomorphism test that examines all possible point mappings is extremely inefficient. A much faster method can be implemented as an extension of the automorphism group generation algorithm which was described in detail in Section 4.5.2. This algorithm used point, block and point pair partitions to reduce the amount of backtracking performed when searching for automorphisms. The same partitions can be used in a similar way to reduce the amount of searching performed when looking for an isomorphism between two designs.

One of the designs will be the source design, and the other will be the target design. The modified isomorphism generation algorithm will search for a mapping from the source to the target design. There will be a different set of point, block and point pair partitions for the source and target designs. The diagram below illustrates the required structures.



The partial isomorphism,  $\emptyset$ , is constructed a level at a time, where at each level  $\emptyset$  is either extended with a single new point mapping or the most recent of its constituent point mappings is backtracked. At each level of the construction,  $\emptyset$  must be incidence preserving. This means that for any subset,  $S$ , of the source treatments already mapped, if  $S$  is contained within a block of the source design, then  $\emptyset(S)$  must be contained within a block of the target design. Clearly, any valid, complete isomorphism must be incidence preserving. If a partial mapping is not incidence preserving, no extension of it can be incidence preserving, and therefore it can not lead to a valid isomorphism.

The partitions are used to reduce the number of possible mappings examined, and also to check whether each partial mapping is incidence preserving - just as in the automorphism group generation algorithm. For example, consider the point partitions. Any point,  $p$ , of the source design belonging to class  $c$  of the source point partition, only needs to be considered in a point mapping ( $p \rightarrow q$ ) of  $\emptyset$ , with those points,  $q$ , in the target design which belong to class  $c$  of the target point partition.

At the start of the search, a clique analysis of the block intersection graph of both designs is performed, and the results of this analysis are used to induce the initial block, point and point pair partitions for the source and target designs. These initial partitions significantly reduce the search space, but it is the refinement of the partitions at each level of the construction which really target the search. A clear example of such partition refinement as used in the isomorphism algorithm is given next.

For the clarity of the following example, assume that the initial point and block partitions for both the source and target designs contain no useful information, and therefore every point and block is placed into the same class. This situation is illustrated below for an example design consisting of 6 points and 10 blocks:

Initial SOURCE point partition

1	2	3	4	5	6
1	1	1	1	1	1

Initial TARGET point partition

1	2	3	4	5	6
1	1	1	1	1	1

Initial SOURCE block partition

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

Initial TARGET block partition

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

Further, assume that blocks 1 and 2 of the source design contain the point 1, and that blocks 1 and 10 of the target design contain the point 2.

Also assume that the point pair partition of the source design places the pairs (1, 2) and (1, 4) into class 1, the pairs (1, 5) and (1, 6) into class 2, and the pair (1, 3) into class 3. Let the point pair partition of the target design place the pairs (2, 1) and (2, 3) into class 1, the pairs (2, 4) and (2, 5) into class 2, and the pair (2, 6) into class 3. In other words, row 1 of the source point pair partition is:

Row 1: - 1 3 1 2 2

and row 2 of the target point pair partition is:

Row 2: 1 - 1 2 2 3

Now, let the point mapping (1→2) be the first mapping considered in the construction of the partial isomorphism,  $\emptyset$ . The point partitions would be updated by juxtaposing them with the corresponding rows of the point pair partitions:

The new *source* point partition is formed by juxtaposition with row 1 of the source point pair partition, as the source point of the current point mapping is "1":

Current source point partition:	1 1 1 1 1 1
Row 1 of source point pair partition:	- 1 3 1 2 2
New source point partition:	- 1 3 1 2 2

The new *target* point partition is formed by juxtaposition with row 2 of the target point pair partition, as the target point of the current point mapping is "2":

Current target point partition:	1 1 1 1 1 1
Row 2 of target point pair partition:	<u>1 - 1 2 2 3</u>
New target point partition:	1 - 1 2 2 3

In both cases, the juxtaposition is trivial because the initial point partitions contained only a single class. The new point partitions are in one-to-one correspondence, so the block partitions must also be refined:

The new *source* block partition is formed by juxtaposition with row 1 of the incidence matrix, which indicates the blocks of the source design to which point "1" belongs:

Old partition:	1 1 1 1 1 1 1 1 1 1
Row 1 of incidence matrix:	<u>1 1 0 0 0 0 0 0 0 0</u>
New partition:	1 1 2 2 2 2 2 2 2 2

The new *target* block partition is formed by juxtaposition with row 2 of the incidence matrix, which indicates the blocks of the target design to which point "2" belongs:

Old partition:	1 1 1 1 1 1 1 1 1 1
Row 2 of incidence matrix:	<u>1 0 0 0 0 0 0 0 0 1</u>
New partition:	1 2 2 2 2 2 2 2 2 1

The new block partitions are also in one-to-one correspondence, which indicates that the partial mapping  $\emptyset : (1 \rightarrow 2)$  is incidence preserving.

The point and block partitions divide the points and blocks of the source and target designs into equivalence classes, such that points or blocks sharing the same properties in relation to the partial isomorphism at each level of its construction belong to the same class in the partitions. For example, the new source block partition, induced above, has one class for all those blocks of the source design containing the point 1, and one class for all those not containing the point 1. The new target block partition has 1 class for all those blocks of the target design containing the point 2 and one class for all the blocks not containing the point 2. A similar argument can be presented for the point partitions, and it is clear that a point of the source design can only be mapped to a point of the target design as part of a valid isomorphism if they both belong to the same class of their respective point partitions.

Say the next point mapping examined is  $(2 \rightarrow 10)$ . Then the source block partition will be refined so that there is one class for all the blocks of the source design containing both points 1 and 2, one class for all the blocks containing point 1 but not 2, one class for all the blocks containing point 2 but not 1, and one class for all the blocks containing neither point 1 nor point 2. The target block partition will be refined so that there is one class for the blocks of the target design containing the points 2 and 10, one class for the blocks containing 2 but not 10, one class for the blocks containing 10 but not 2, and one class for the blocks containing neither 2 nor 10. For a partial mapping to be incidence preserving and thus to extend to a valid isomorphism, it is clear that these partitions must always be in one-to-one correspondence.

In general, the results of the clique analysis are used to induce strong initial partitions which place tighter conditions on the equivalence classes and further reduce the possible choices at each level of the algorithm.

### 5.3.1.2 RESULTS

A simple constructive enumeration algorithm was implemented by combining the isomorphism testing algorithm presented in the previous section with the incidence matrix backtracking algorithm. An explicit list of non-isomorphic designs is maintained, and each design constructed by the backtracking algorithm is only added to the list if it is non-isomorphic to all designs currently in the list.

The twofold triple systems have already been mentioned a number of times and were enumerated using the above approach for orders 6, 7 and 9. For the enumeration of each TTS(v), the automorphism group calculations were performed on all partial designs on the first v-3 rows, and the results of the group calculations were used to perform partial isomorphism rejection on every row of the incidence matrix.

The results below specify the total number of completed designs generated by the incidence matrix backtracking algorithm, how many of these designs were non-isomorphic and therefore added to the list, and the total number of pairwise isomorphism tests performed throughout the execution of the algorithm. If  $Des$  is the total number of designs generated by the incidence matrix backtracking algorithm, and  $Nd$  is the number of these which are non-isomorphic, then the total number of isomorphism tests performed using the current algorithm will be greater than or equal to  $1+2+3+\dots+(Nd-1) = \frac{Nd^2 - Nd}{2}$ , and less than  $Nd \times Des$ .

	Number of DesignsGenerated (Des)	Number of Non-Isomorphic Designs(Nd)	Number of Isomorphism TestsRequired	Execution Time (secs)
TTS(6)	1	<b>1</b>	0	-
TTS(7)	7	<b>4</b>	14	0.017
TTS(9)	411	<b>36</b>	7823	2.967

### 5.3.1.3 OPTIMISATIONS

The number of pairwise isomorphism tests performed by the algorithm presented in Section 5.3.1.1 to construct each list increases at an enormous rate. For the TTS(10) designs, a staggering 12,039,090 isomorphism tests are performed just to construct a list containing 960 designs. This section presents four optimisations to this algorithm, which reduce the number of required isomorphism tests and allow those necessary tests to be performed more quickly.

#### Storing the Clique Analysis Results

Most of the isomorphism tests performed detect very quickly that two designs are non-isomorphic, because the initial point and block partitions may not be in one-to-one correspondence. However, in order to compare the initial point and block partitions, they must first be induced from the results of the relatively expensive clique analysis. Currently, whenever a design from the list is tested for isomorphism with a newly constructed design, both designs must have their clique analyses performed before the initial point and block partitions can be compared. Clearly, a given list design may have this expensive operation performed a large number of times throughout the course of the algorithm. A substantial improvement in performance therefore results by storing, along with each design in the list, the results of its corresponding clique analysis. Although more space is required for storing the list, an enormous number of repeated calculations are eliminated.

With this optimisation, every constructed design has its clique analysis performed only once. The point, point pair and block partitions for a given list design are induced from these stored results for each isomorphism test performed.

Using this optimisation, the TTS(10) designs were enumerated and the results are summarised in the table below:

	Number of DesignsGenerated (Des)	Number of Non-Isomorphic Designs(Nd)	Number of Isomorphism TestsRequired	Execution Time (secs)
TTS(10)	25403	<b>960</b>	12039090	4077

The enumerations of the twofold triple systems for each of the four orders 6, 7, 9 and 10, given in Section 5.3.1.2 and above, confirm the published results for these designs. If  $v$  is the order of the TTS, and  $N_d$  is the number of non-isomorphic TTS( $v$ )'s, then these results can be summarised as below:

$v$	6	7	9	10
$N_d$	1	4	36	960

Along with each design in the list, the storage of the results of its clique analysis significantly reduced the number of clique analyses performed, and made the isomorphism testing considerably more efficient. However, the major disadvantage of the current method is still the total number of pairwise isomorphism tests required. More than 12 million were required, at a rate of nearly 3000 a second, to construct the list of 960 non-isomorphic TTS(10)'s.

As the number of constructed designs increases for larger problems, the number of required isomorphism tests becomes a severe limitation of this approach. It becomes necessary to avoid performing isomorphism tests whenever possible, and this is the idea behind the development of the next optimisation - the use of design signatures.

### DesignSignatures

The clique analysis which is performed for each design constructs a block intersection graph and then counts for each node of the graph, the number of cliques of varying sizes to which the node belongs. Currently, these results are used to induce the initial partitions of the search. Without any extra computation, each design can have a signature constructed which simply consists of a count of the total number of cliques of each size in the corresponding block intersection graph of the design. This is easily calculated by maintaining a vector `CLIQUE_TOTALS`, and incrementing element  $i$  of this vector every time a new clique of size  $i$  is constructed during the clique analysis.

This vector of clique totals is an invariant of the design, and therefore two isomorphic designs must necessarily have identical vectors. Conversely, any two designs which have different vectors cannot be isomorphic. The use of such design signatures can improve the performance of the algorithm by significantly reducing the number of isomorphism tests required.

When a design is found to be non-isomorphic to all list designs, it is stored in the list along with the results of its clique analysis and with its corresponding `CLIQUE_TOTALS` vector. This vector is called the signature of the design.

Whenever a new design is constructed by the incidence matrix backtracking algorithm, its clique analysis is performed, generating its corresponding `CLIQUE_TOTALS` signature. The design then only needs to be tested for isomorphism against those designs already in the list which have exactly the same signature.

Desirable properties for a signature are that it is easy to compute, and that it does a good job of classifying the designs. The signature described above is certainly easy to compute, because it is calculated as part of the clique analysis which is performed for every design anyway. As for its effectiveness, the twofold triple systems have been enumerated once again, this time using the signature optimisation. Of particular interest in these results, given in the table on the following page, are the number of isomorphism tests performed to build the non-isomorphic list, and the execution time:

	Number of DesignsGenerated (Des)	Number of Non-Isomorphic Designs (Nd)	Number of Isomorphism TestsRequired	Execution Time (secs)
TTS(6)	1	1	0	-
TTS(7)	7	4	3	0.017
TTS(9)	411	36	389	1.330
TTS(10)	25403	960	24969	117.300

This optimisation exhibits enormous improvement in the performance of the algorithm. With the use of the design signatures, the TTS(10) enumeration completed more than 30 times as quickly as before.

An important observation that can be made from these results is that the total number of isomorphism tests performed is less than the total number of designs generated in every case, indicating that the design signatures provide an excellent classification of the designs. In other words, every design constructed was tested for isomorphism against, on average, less than 1 other design in the list at the time.

A measure of how well the design signatures classify the designs can be determined by counting how many non-isomorphic designs in the complete catalogue of a particular class share the same signature. If each design in the catalogue has a unique signature, then the signatures are able to classify the designs perfectly - without the need for any isomorphism testing. The number of unique signatures, which were based on the clique total vectors, for the twofold triple systems are given below:

Order	Number of Non-Isomorphic Designs in List	Number of Unique Design Signatures
6	1	1
7	4	4
9	36	35
10	960	941

For orders 6 and 7, the signatures classify the designs perfectly, whereas for order 9, only two non-isomorphic designs share the same signature. For order 10, the clique signatures are highly effective, successfully classifying 98% of the designs.

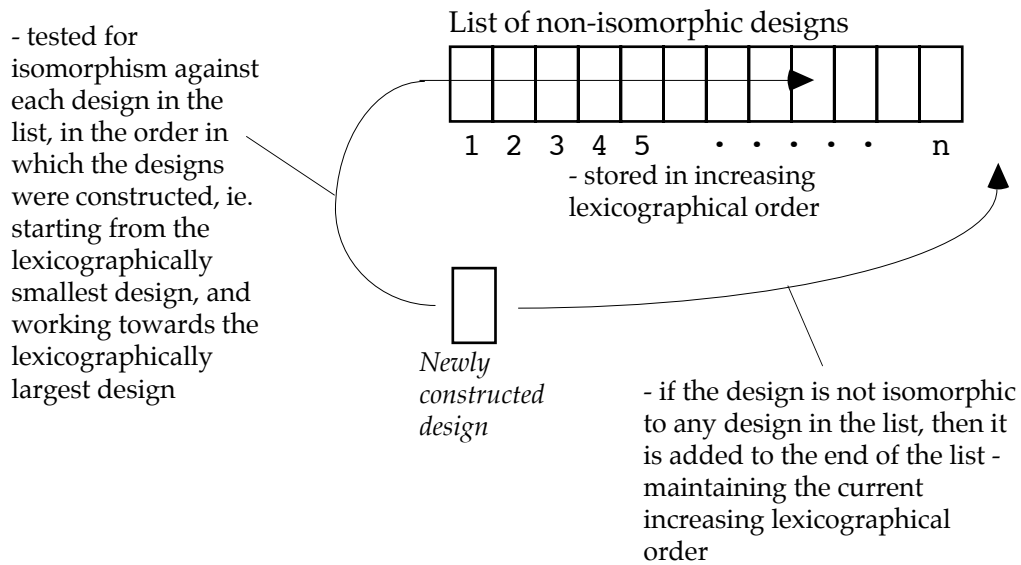
The following two optimisations focus on the maintenance of the non-isomorphic list, and are very effective for certain problems. Both are used extensively in the work on configurations, which is detailed in Case Study Seven of Chapter 6.

### List Reordering

In general, a significantly larger number of designs are generated than end up in the non-isomorphic list. For any design which is constructed but does not end up in the list, it must be rejected following an isomorphism test between itself and one of the designs which is in the list. It is therefore important to try and reduce, as much as possible, the number of isomorphism tests which are performed for any design which does not end up in the list. The ideal case would be where only a single isomorphism test is performed for each constructed design which is isomorphic to some list design.

To reduce the number of such tests, one approach is to change the order in which the list designs are examined when testing each newly constructed design, rather than scanning the list in the same increasing lexicographical order in which the designs are generated and stored.

The following diagram illustrates the current situation:



Clearly, any constructed design is either non-isomorphic to every list design, or it is isomorphic to exactly one design in the list. For any constructed design which is isomorphic to a list design, the number of isomorphism tests performed which do not reject the design can be termed *wasted* tests. The list reordering optimisation attempts to reduce the number of wasted isomorphism tests by scanning the list in an order such that the designs examined earlier in the order are more likely to reject a newly constructed design than the designs examined later in the order.

The current performance of the algorithm is first examined. Recall for the TTS(9) enumeration, exactly 411 designs were constructed yielding 36 non-isomorphic triple systems. Therefore, of the 411 constructed designs, exactly 375 of them were found to be isomorphic to a design already stored in the non-isomorphic list.

Chart 5.1, on the next page, illustrates the performance of the TTS(9) enumeration, when the non-isomorphic list is scanned in increasing lexicographical order. Each of the 375 designs which were rejected during the course of the algorithm corresponds to one of the plotted data points. The value plotted for each rejected design,  $D$ , is the position in the non-isomorphic list of the design found to be isomorphic to  $D$ , and therefore responsible for its rejection.

The top line of Chart 5.1, labelled  $\mathcal{A}$ , indicates the size of the list at the particular stage of the algorithm, and the line running through the middle of the chart, labelled  $\mathcal{B}$ , indicates exactly half the size of the list. If each design constructed was isomorphic to a completely random list design, then it would be expected that the plotted series would be randomly distributed about  $\mathcal{B}$ , between the range of zero and  $\mathcal{A}$ .



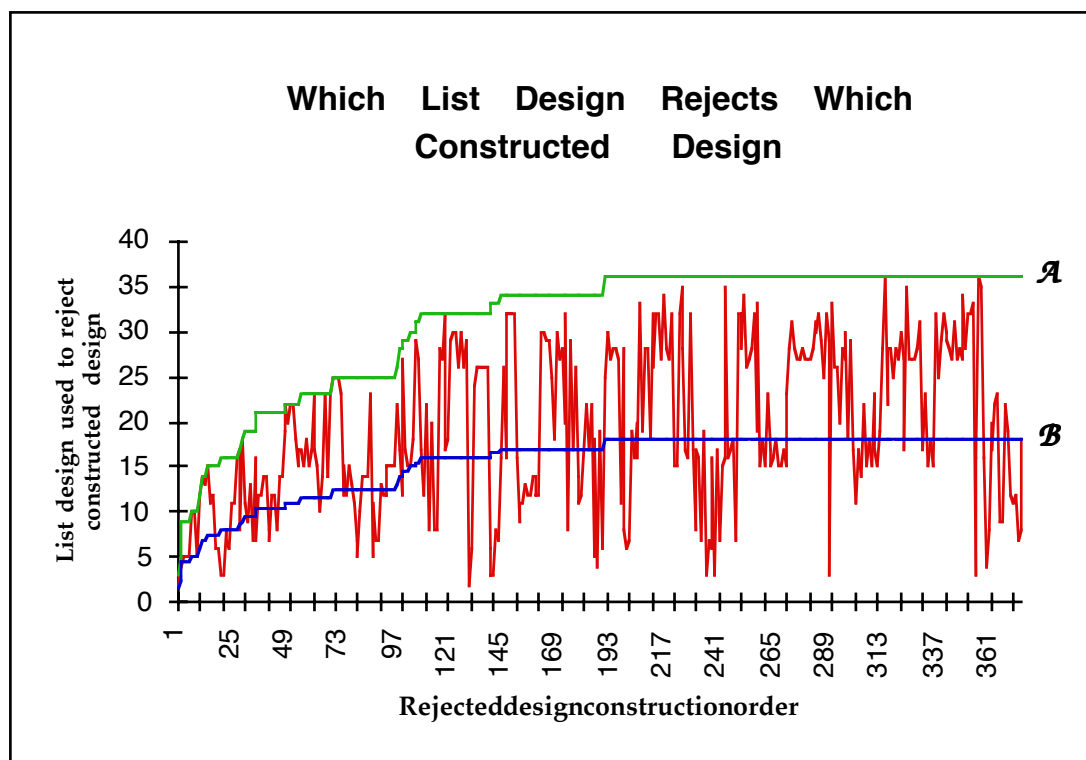


Chart 5.1

It is clear from Chart 5.1 that a larger proportion of constructed designs are isomorphic to a design in the second half of the list than in the first half of the list, which is scanned in increasing lexicographical order. In fact, of the 375 constructed designs which were rejected, 238 were isomorphic to a design in the second half of the list whereas only 137 were rejected by a design in the first half of the list.

A major reason for this observation is that once a new non-isomorphic design, *D*, is constructed and added to the end of the list, quite often a number of the subsequently generated designs turn out to be isomorphic to *D*. See for example the pattern of the first 80 or so designs in Chart 5.1, many of which are rejected by the last, and thus most recently constructed design in the list.

This observation alone indicates that it may be profitable to scan the designs in the list in reverse lexicographical order - opposite to the order in which they are constructed. The table below compares the number of isomorphism tests required to perform the TTS(*v*) enumerations, when scanning the list in increasing and decreasing lexicographical order. To produce meaningful figures, the design signatures are not used in the enumerations below, so that any reduction in the number of required isomorphism tests is solely due to the order in which the lists are scanned:

Order	Total Number of Isomorphism Tests Required	
	List scanned in increasing lexicographical order	List scanned in decreasing lexicographical order
7	14	10
9	7,823	5,455
10	12,039,090	10,326,788

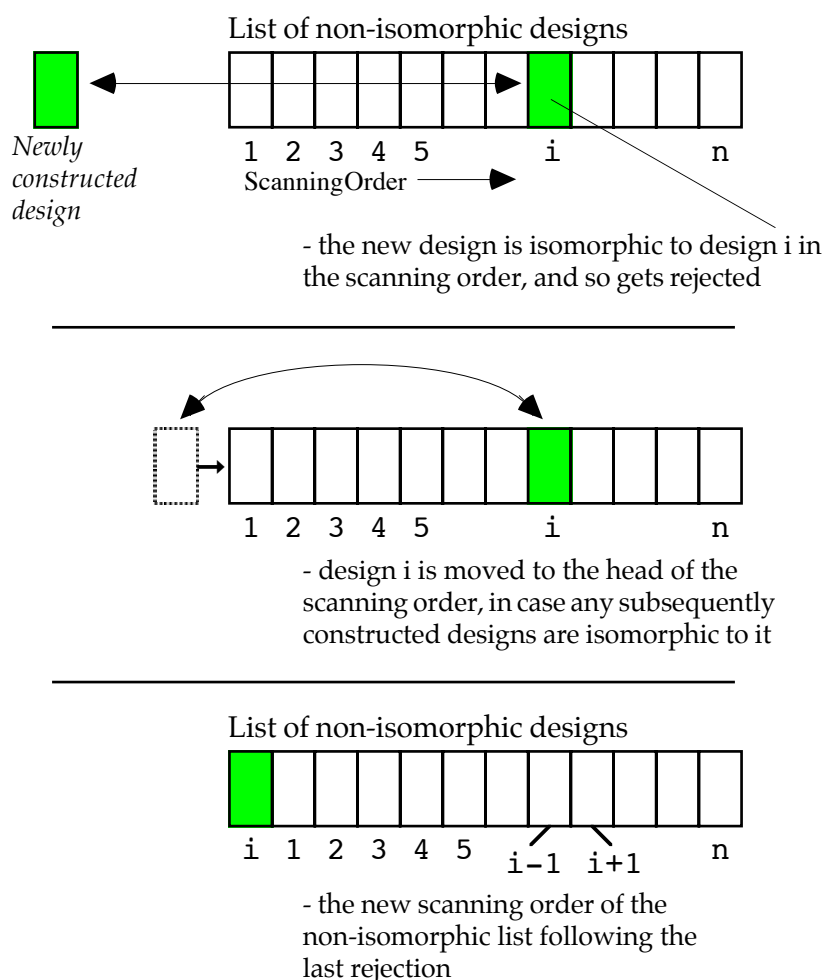
As indicated by the figures in the table above, scanning the non-isomorphic list in reverse lexicographical order results in a small improvement in the performance of the algorithm.

The pattern of rejected designs exhibited previously in Chart 5.1 suggests a further list reordering optimisation. Although the chart does fluctuate considerably, it appears to exhibit a reasonable amount of spatial locality, in this context meaning for short

consecutive sequences of constructed designs, the positions of the designs within the list which reject them tend not to be widely scattered. In fact, a given design in the list is often used to reject a consecutive sequence of constructed designs. For example, consider the TTS(9) enumeration. When the 123<sup>rd</sup> design is constructed in this enumeration, there are 25 designs in the non-isomorphic list, and it is added as the 26<sup>th</sup> list design. From this point on in the algorithm, design number 26 in the list rejects exactly 14 of the remaining 288 designs constructed. The precise order in which these 14 designs are constructed is given below:

157, 160, 165, 166, 167, 168, 169, 170, 179, 211, 248, 289, 328, 329

If after rejecting the 165<sup>th</sup> design, list design 26 was moved to the head of the scanning order, then the next 5 designs constructed would be rejected by the very first isomorphism test performed. In general, this self-modification of the list scanning order is a very effective technique. The diagram below illustrates the idea:



The TTS enumerations were performed once again, this time building a list with the self-modifying scanning order presented above. To summarise the complete process, let design  $D$  be the most recently constructed design by the incidence matrix backtracking algorithm. If  $D$  is non-isomorphic to the existing list designs and thus added to the list, it is also moved to the head of the scanning order, so is the first list design to be examined when the next design is constructed. Otherwise, the list design to which  $D$  is isomorphic, and is thus responsible for rejecting  $D$ , is moved to the head of the scanning order.

In the following table, the results of the TTS( $v$ ) enumerations using lists with self-modifying scanning orders are compared to the increasing and decreasing lexicographical order scanning strategies, and as before the design signature optimisation is not used:

Order	Total Number of Isomorphism Tests Required		
	List scanned in increasing lexicographical order	List scanned in decreasing lexicographical order	List Generated with Self-Modifying Scanning Order
7	14	10	11
9	7,823	5,455	4,413
10	12,039,090	10,326,788	9,079,349

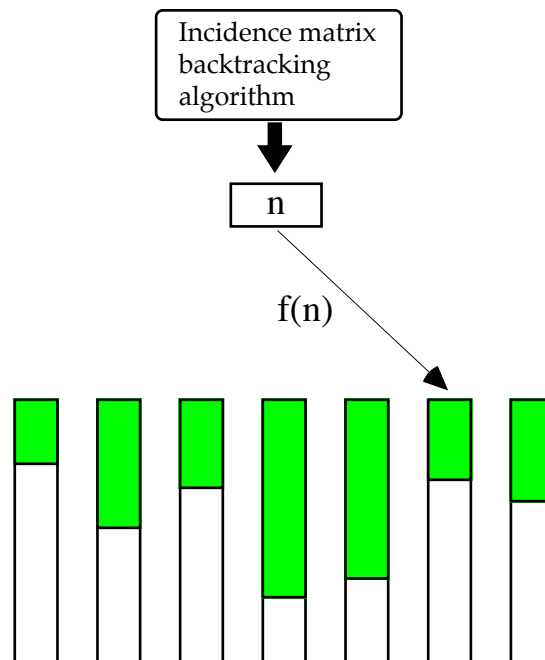
Disregarding the enumerations of the order 7 designs in which very few isomorphism tests are performed, the results clearly provide evidence that building a list with a self-modifying scanning order is preferable to the other two simpler strategies.

**List Distributing**

There is still a major drawback to the explicit construction of a non-isomorphic list, which is not addressed by the list reordering schemes. This drawback is the size of the constructed list. For many large enumerations, storing the complete catalogue of designs is simply not feasible. It may not be possible to store the entire list in primary memory, and using a secondary storage device such as a disc is impractical due to the considerably longer access times. Even for lists which can be stored in memory, without a very effective design signature a large number of isomorphism tests may need to be performed for each constructed design. In addition, there may be considerably more designs constructed than actually belong in the non-isomorphic list, and the total number of isomorphism tests required may be prohibitively large.

This section presents a simple optimisation for avoiding some of the problems mentioned above. The idea behind the list distribution optimisation is that instead of building a single large list, a number of smaller lists are constructed instead. The only condition which must be satisfied for such a scheme to be successful is that no isomorphism tests need to be performed between designs in separate lists.

The diagram below illustrates the behaviour of this algorithm:



A new design, say  $n$ , is constructed by the incidence matrix backtracking algorithm, and this is mapped by means of some function, say  $f$ , to one of the non-isomorphic lists being constructed. This can be viewed as a generalisation of the construction of a single list, in which case the function  $f$  maps every design to the same list. The most desirable properties for such a function,  $f$ , are that it can be calculated quickly, and that it maps constructed

designs evenly across all the lists. The necessary property for such a function,  $f$ , is that any two isomorphic designs must be mapped to exactly the same list. In other words, only those designs which map to the same list need to be tested for isomorphism, as it is guaranteed that any two designs which are mapped to different lists cannot be isomorphic.

This distribution of lists offers two advantages. Firstly, it overcomes the memory problem. It is only necessary to store one of the distributed lists in memory at one time, because any constructed designs which do not map to this list do not need to be processed by it, and these designs can be processed at another time when their corresponding list is being built. If the function is defined appropriately, it may be possible to divide the list into almost an arbitrary number of pieces depending on the amount of memory available.

Secondly, the lists do not need to be built sequentially. This is the primary advantage for it means each list can be constructed on a different processor, and therefore a number of isomorphism tests can be performed concurrently.

### Mapping Functions

The desirable and necessary properties of a mapping function have already been mentioned, and two possible functions are presented in this subsection - both based on the signature of the designs, which is a vector of clique totals calculated from the clique analysis of the block intersection graph of each design.

The first possibility is simply to assign a range of distinct signature types to each list, so that each list processes only those designs which have signatures lying in the corresponding range. The specific ranges may be defined in a number of ways. For example, each element in the vector of clique totals could be summed, so that for each design it is known the total number of cliques, of all sizes processed, in the corresponding block intersection graph of the design. A given list may then process all those designs for which the sum of the clique totals falls between some specified range. This guarantees that any isomorphic designs will be processed by the same list, because two isomorphic designs must have exactly the same signature, and hence exactly the same clique totals sum. However, it does not satisfy the desirable properties very well, because specifying ranges for the summed vectors which produce fairly even sized lists is not straightforward, and would certainly require at least some initial analysis of the designs.

The second possibility simply hashes the sum of the elements of the vector of clique totals using a modulo function. This has the advantage that no knowledge of the clique structure of the designs is required to produce evenly distributed lists. Certainly, two isomorphic designs which possess the same clique signature and hence the same summed clique signature, will also possess the same hashed, summed clique signature. Hashing also provides an easy way of scaling the list distribution to suit the number of available processors. If  $n$  processors are available, they are assigned unique identifiers  $0, 1, 2, \dots, n-1$ . Any constructed design, with summed clique signature  $S$ , is processed by the list on the processor with identifier  $S \bmod n$ .

This second mapping function performs very well, and results of its use on the enumeration of configurations are given in Case Study Seven of the next chapter. To analyse its performance on the twofold triple systems, they are enumerated again here for orders 9 and 10. In each case, the enumeration is distributed amongst 4 lists, and run on 4 separate processors identified as 0, 1, 2 and 3. Each processor performs the exhaustive design construction, but any generated design  $D$ , with summed clique signature  $S$ , is processed only by the list built on processor  $S \bmod 4$ . The processors are completely independent, and do not need to communicate.

For the results summarised on the following page, the design signatures are only used to determine the list on which each design is processed, but are not used to reduce the number of isomorphism tests performed. The lists are built using the self-modifying scanning order.

*The TTS(9) Enumeration*

Total number of designs constructed: 411

	Single list	List 0	List 1	List 2	List 3
Size of constructed list	36	10	13	5	8
Total number of isomorphism tests performed	4,413	291	664	142	289
Execution Time (secs)	2.08	1.00	1.10	0.983	1.02

The combined number of isomorphism tests performed in the distributed enumeration is considerably less than when the single list is constructed. The observed running time of the distributed enumeration is approximately half that of the single list enumeration, although the combined running time is approximately twice as long. The reason for this is that the design generation is performed on each processor, and is therefore an overhead of the list construction in each case. A more efficient method would be to use a number of communicating processors, with one or more being responsible for the design generation, and the others responsible for the list construction. The advantage of the approach presented in this section is that no communication is necessary as the lists are built independently of one another.

*The TTS(10) Enumeration*

Total number of designs constructed: 25,403

	Single list	List 0	List 1	List 2	List 3
Size of constructed list	960	256	238	236	230
Total number of isomorphism tests performed	9,079,349	665,870	569,890	524,154	525,910
Execution Time (secs)	3051.9	306.6	272.8	258.3	259.6

Several observations can be made from the above result. Firstly, all 960 non-isomorphic TTS(10)'s are indeed generated in the 4 distributed lists. The combined number of isomorphism tests performed for the distributed lists is 75% less than the number performed when constructing the single list. This is a direct result of the fact that the list generation is  $O(n^2)$ , and the distributed method generates smaller lists. Similarly, the combined running time of the 4 distributed list constructions is 2.8 times faster than the single list construction, and the observed running time is 10 times as fast. Even if the enumeration wasn't parallelised, but each of the four pieces were run sequentially, the performance would be significantly better than if a single list was constructed.

As a final observation, the simple hash function has performed well at distributing the enumeration into even pieces, both space-wise and time-wise.

**Results**

Overall, the optimisations discussed in the previous sections make the list generation enumeration method feasible for many classes of designs. For example, there are 1508 non-isomorphic 2-(7,3,7) designs. Generating the automorphism group of partial designs on the first 5 rows, and performing partial isomorph rejection on every completed row produces exactly 8821 designs in the construction process. These can each be tested for isomorphism, to build a complete list of the 1508 non-isomorphic designs. Given at the top of the next page is the direct output of the implemented algorithm which uses the design signature and self-modifying scanning order optimisations to constructively enumerate these designs:

```

DESIGNENUMERATOR
=====

Enumeration of 2-(7,3,7) design (v=7, B=49, r=21, k=3 and Lb=7):

(Automorphism groups calculated only on sub-designs down to row 5.)
(Partial isomorph rejection performed to row 7.)

<working>

The enumeration is complete.

The number of complete 2-(7,3,7) designs generated was 8821.
Exactly 1508 of these are non-isomorphic, and stored in the list.

The process took 100.963 seconds to complete.

In total, 21349 isomorphism tests were performed on the list designs.
Total number of unique signatures: 775.

```

This complete cataloguing took a little over a minute and a half. A total of 21349 isomorphism tests were performed, and of the 1508 non-isomorphic designs, there were exactly 775 unique signature types. The signatures were based on the results of a (4,1)-clique analysis of the block intersection graph.

### 5.3.2 CANONICITY TESTING

The list building algorithm, in addition to the optimisations presented Section 5.3.1.3, is an effective method for the constructive enumeration of many small classes of designs. However, despite the optimisations, the construction of a list of non-isomorphic designs using standard pairwise isomorphism testing has several fundamental disadvantages:

- Although the construction of the list can be effectively parallelised to reduce the number of required comparisons and to reduce memory requirements, every constructed design still needs to be tested for isomorphism against a subset of the designs currently stored.
- The incidence matrix backtracking algorithm is used to exhaustively generate all feasible designs for every list being constructed, so there is a great deal of repeated computation. For many design classes, this may be a very large search, and it would be desirable if the design generation could be parallelised as well as the list generation. Although parallelisation of the design generation is certainly possible (see Case Study Two of Chapter 6), combining the parallelised design generation with the parallelised list generation is very complicated and requires support for communication between the processors.
- Even more restrictively, for very large classes, it simply may not be feasible to store every non-isomorphic design due to space limitations. In these cases, the algorithm fails completely because it is not possible to determine whether each newly constructed design is isomorphic to some design in the non-isomorphic list.

This section presents the technique of canonicity testing, which provides a means for determining whether or not a given design constructed at any stage of the search belongs in a catalogue of non-isomorphic designs. The primary advantage of this approach is that in order to answer this question for a particular design, it is not necessary to examine any previously classified designs and therefore the catalogue does not even need to be stored - overcoming both of the main drawbacks of the explicit list building approach.

The key to canonicity testing is that the designs are generated in increasing lexicographical order. As in the list building approach, a design is only classified as belonging to the non-isomorphic catalogue if it is not isomorphic to any design currently in the catalogue. In

order to implement such a scheme it is necessary to answer, for any design D constructed at any stage of the algorithm, the question: "Is design D isomorphic to any design which has already been classified?" If the answer is yes, then D should be rejected, otherwise D should be classified. A straightforward way of answering this question is to generate all the designs isomorphic to D and compare them, lexicographically, to D. If any one of these designs is lexicographically less than D, then it must have already been generated in the search and would have already been classified. If all of the designs are lexicographically greater than D, then none of them would have been generated earlier in the search and so D should be classified as a new non-isomorphic design.

More formally, the isomorphism class of a design, D, is the set of all designs to which D is isomorphic. For any isomorphism class, the unique lexicographically smallest design within the class is said to be the canonical representative of the isomorphism class, or is simply said to be in canonical form. The incidence matrix backtracking algorithm, with the partial isomorph rejection techniques, is guaranteed to generate at least the canonical representative of each isomorphism class. Applying this to the constructive enumeration problem, a design D constructed at some point in the algorithm is classified as belonging to the non-isomorphic catalogue if and only if D is in canonical form. The process of determining this for each design is termed a *canonicity test*, and the focus of this section is the efficient implementation of such a test.

The following example demonstrates the uniqueness of the canonical representative of a given isomorphism class. There is a single TTS(6) up to isomorphism, and the incidence matrix of this design, in canonical form, is given below:

```
1111100000
1100011100
1010010011
0101001011
0010101110
0001110101
```

The block list of this design is given below:

1 2 3	1 3 5	1 5 6	2 4 5	3 4 5
1 2 4	1 4 6	2 3 6	2 5 6	3 4 6

Any other distinct TTS(6) design, which can be produced by relabelling the points of the above design, must have a corresponding incidence matrix which is lexicographically greater than the matrix given above. For example, consider the point relabelling: (1→1) (2→2) (3→3) (4→4) (5→6) (6→5)

This interchanges points 5 and 6 of the design above. The block list of the relabelled design, which is isomorphic to the original, is given below:

1 2 3	1 3 6	1 5 6	2 4 6	3 4 5
1 2 4	1 4 5	2 3 5	2 5 6	3 4 6

The incidence matrix of this design, displayed below, is lexicographically larger than the previous incidence matrix, and is therefore clearly not in canonical form.

```
1111100000
1100011100
1010010011
0101001011
0001110110
0010101101
```

The two incidence matrices differ at row 5, as the following table illustrates. The underlined point on row 5 of each matrix is the first point at which the matrices differ when read from left to right, top to bottom. The matrix on the left, in canonical form, has a "1" in this position - indicating it is lexicographically smaller than the relabelled design:

<i>Original incidence matrix of the TTS(6) design in canonical form</i>	<i>Lexicographically larger incidence matrix of the relabelled TTS(6) design</i>
1111100000	1111100000
1100011100	1100011100
1010010011	1010010011
0101001011	0101001011
00 <u>1</u> 0101110	00 <u>0</u> 1110110
0001110101	0010101101

The previous example also demonstrates how two complete incidence matrices can be compared lexicographically. This is an important operation, and is used extensively in the various implementations of the canonicity test, such as the brute force implementation presented in the next section.

### 5.3.2.1 BRUTE FORCE IMPLEMENTATION

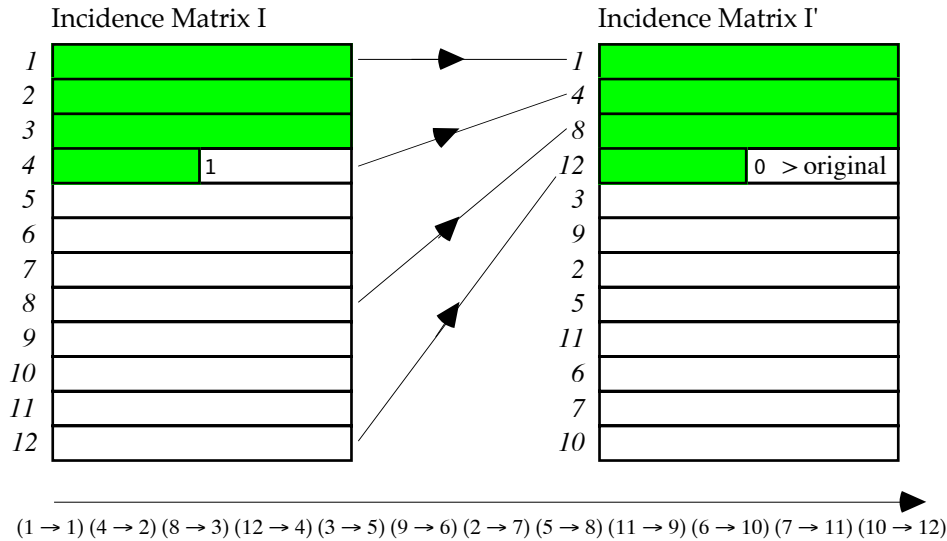
The canonicity test must determine whether or not a given design is in canonical form, which essentially means determining whether or not it is isomorphic to any design lexicographically smaller than itself. One simple way of implementing this for a given design  $D$ , of order  $v$ , is to apply all possible  $v!$  point relabellings to  $D$  and examine the resulting  $v!$  relabelled designs. Although some of these designs may be duplicates of one another, exactly  $v!/|G|$  of them will be distinct, where  $|G|$  is the size of the automorphism group of  $D$ . Let  $D_s$  be the set of these  $v!/|G|$  distinct designs. If the lexicographically smallest design in the set  $D_s$  is in fact  $D$  itself, then  $D$  is in canonical form. Otherwise  $D$  is not in canonical form, and the canonical representative of the isomorphism class to which  $D$  belongs is the lexicographically smallest design in the set  $D_s$ .

The drawback of the above approach is that  $v!$  relabellings must be examined, and as previously mentioned, algorithms exhibiting factorial behaviour are impractical for even relatively small problems. However, by making some simple observations about the process of relabelling and lexicographically comparing a given design, only a tiny fraction of the  $v!$  relabellings explicitly need to be examined. For example, consider a design  $D$  on 12 points, with corresponding incidence matrix  $I$ , and let  $\emptyset$  be the following point relabelling of design  $D$ :

$$\emptyset = \begin{array}{cccccccccccc} (1 & 4 & 8 & 12 & 3 & 9 & 2 & 5 & 11 & 6 & 7 & 10) \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ (1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12) \end{array}$$

The effect that this mapping has on design  $D$  can be represented by the following permutations of the rows of the incidence matrix:





Once the columns of the permuted incidence matrix  $I'$  are ordered, which essentially requires sorting the columns into increasing lexicographical order, it can be compared lexicographically to incidence matrix  $I$  simply by comparing the rows working left to right and top to bottom.

As illustrated in the diagram above, assume the first 3 rows of the incidence matrices are exactly the same, as are the initial portions of row 4, but let the discrepancy between the matrices at row 4 indicate that the permuted incidence matrix  $I'$  is lexicographically greater than incidence matrix  $I$ . In terms of the ordering imposed by the generation of the matrices this would imply that in the first column of row 4 in which the rows are different, matrix  $I$  contains a 1 whereas matrix  $I'$  contains a 0.

Now, using a brute force canonicity test which examines all  $12!$  point relabellings in increasing lexicographical order, then next relabelling to be considered would be:

$$\emptyset = \begin{matrix} (1 & 4 & 8 & 12 & 3 & 9 & 2 & 5 & 11 & 6 & 10 & 7) \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ (1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12) \end{matrix}$$

This simply interchanges the target point mappings for source points 7 and 10. Obviously, from the diagram above, this point relabelling will again map the incidence matrix of design  $D$  to a lexicographically larger incidence matrix, because the initial 4 point mappings of the relabelling are unchanged. Once again the first four rows of the permuted incidence matrix will be  $(1, 4, 8, 12)$  and, once ordered, this will be larger than the original matrix.

In fact, any point relabelling which contains the initial point mappings:  
 $(1 \rightarrow 1) (4 \rightarrow 2) (8 \rightarrow 3) (12 \rightarrow 4)$

will permute the incidence matrix of design  $D$  to a lexicographically larger matrix. As the canonicity test is a search for a point relabelling which maps  $D$  to a lexicographically smaller design, none of these relabellings need to be explicitly considered. This observation is the basis behind one of the optimisations discussed in the next section.

### 5.3.2.2 EFFICIENT CANONICITY TESTING

The most obvious optimisation which can be applied to the brute force canonicity test of the previous section is that a design,  $D$ , which is not in canonical form can be rejected as soon as the first point relabelling is found which maps  $D$  to a lexicographically smaller design. As soon as this is detected, it follows immediately that  $D$  can not be in canonical form, without the need to examine any further point relabellings.

The last observation made in the previous section was particularly important, because it suggests an idea for eliminating large sets of point relabellings from the search during the canonicity test of a design  $D$ . To take advantage of this observation, instead of generating complete point relabellings, the effect on design  $D$  of relabelling just a small subset of the point set is examined. If the subset of points is chosen carefully, then the effect of the relabelling on design  $D$  can be analysed so that often a very large number of complete relabellings can be eliminated from consideration. This is the underlying idea for the efficient canonicity test described below, which systematically generates partial point relabellings using a backtracking approach.

#### Implementation

Let design  $D$ , on the point set  $X = \{1, 2, 3, \dots, v\}$ , be the design which is to be tested for canonicity. Essentially, a point mapping that maps design  $D$  to a lexicographically smaller design is searched for. If such a mapping is not found, then  $D$  is in canonical form and is classified as belonging to the non-isomorphic catalogue. If such a mapping is found, then  $D$  is not in canonical form, and is rejected.

Let  $\emptyset_v = [p_1, p_2, p_3, \dots, p_v]$  represent the complete point relabelling:  
 $(p_1 \rightarrow 1) (p_2 \rightarrow 2) (p_3 \rightarrow 3) \dots (p_v \rightarrow v)$ .

Any relabelling,  $\emptyset_v$ , of this form which maps design  $D$  onto itself is an automorphism of the design.

Let  $\emptyset_k = [p_1, p_2, p_3, \dots, p_k]$  represent the partial point relabelling:  
 $(p_1 \rightarrow 1) (p_2 \rightarrow 2) (p_3 \rightarrow 3) \dots (p_k \rightarrow k)$ .

Such a partial relabelling does not specify any mapping for points  $\{k+1, k+2, \dots, v\}$ .

The canonicity test constructs a series of partial point relabellings by backtracking in strictly increasing lexicographical order. This ensures that for any target point  $i$ , all point mappings of the form  $(p_i \rightarrow i)$  will be considered in strictly increasing order of the source point  $p_i$ . For example, the point mapping  $(1 \rightarrow 5)$  will be considered before the mapping  $(2 \rightarrow 5)$ .

Given a partial relabelling,  $\emptyset_k$ , and the design  $D$ , a relabelled partial design  $D'$  on the point set  $\{1, 2, 3, \dots, k\}$  can be constructed, which represents the effect of the partial mapping  $\emptyset_k$  on the design  $D$ .  $D$  can be compared lexicographically to  $D'$  by considering the first  $k$  rows of the incidence matrix of both designs. There are 3 possible outcomes to the lexicographical comparison of the initial  $k$  rows of the incidence matrices of these 2 designs:

- $D$  is lexicographically less than  $D'$ . This means the partial mapping  $\emptyset_k$  maps  $D$  to a lexicographically larger design, and therefore  $D$  may be in canonical form. However, any extension to the current mapping  $\emptyset_k$  by adding a point mapping of the form  $(p_{k+1} \rightarrow k+1)$  will also map design  $D$  to a lexicographically larger design, because the initial  $k$  points will still be mapped in the same way. Therefore, no extensions to  $\emptyset_k$  need to be considered, and backtracking is performed on level  $k$  by attempting to map  $(p_k \rightarrow k)$  for some larger point  $p_k$ . If  $k$  is small, backtracking at this level will enable an enormous number of complete point mappings to be pruned from the search space.
- $D$  is equal to  $D'$ . This means the partial mapping  $\emptyset_k$  maps design  $D$  onto itself. Therefore  $\emptyset_k$  may define the mapping of the initial  $k$  points of an automorphism of  $D$ . It is not possible to determine whether or not any extension to  $\emptyset_k$  may lead to a complete

mapping which maps  $D$  to a lexicographically smaller design, and so  $\emptyset_k$  must be extended by appending the point mapping  $(p_{k+1} \rightarrow k+1)$  for the smallest possible value of  $p_{k+1}$ .

- $D$  is lexicographically greater than  $D'$ . This means that the partial mapping  $\emptyset_k$  maps  $D$  to a lexicographically smaller design. In other words the first  $k$  rows of the incidence matrix of design  $D'$  are lexicographically less than the first  $k$  rows of the incidence matrix of design  $D$ . Any extension to the partial mapping will merely define the permutation of the remaining rows of the incidence matrix of  $D'$ , but have no effect on the initial  $k$  rows. Therefore, any extension to  $\emptyset_k$  will lead to a complete point relabelling which maps design  $D$  to a lexicographically smaller design - and so  $D$  can not possibly be in canonical form. At this stage, no further searching needs to be performed and design  $D$  can be rejected.

The following two important observations can be made:

- 1) Let  $\emptyset_k$  be a partial mapping specifying the relabelling of  $k$  points of the design  $D$ . Let  $D'$  be the design resulting from applying  $\emptyset_k$  to  $D$ . Let  $I_k$  and  $I_k'$  be the first  $k$  rows of the incidence matrices of designs  $D$  and  $D'$  respectively. Only if  $I_k$  is exactly equal to  $I_k'$  does the partial mapping  $\emptyset_k$  need to be extended. In all other cases, either backtracking will be performed at level  $k$  of  $\emptyset_k$  or it can be determined that design  $D$  is not in canonical form.
- 2) Let  $G$  represent the complete automorphism group of design  $D$ . Let  $\emptyset_v$  be any complete point relabelling of design  $D$  corresponding to one of the automorphisms belonging to  $G$ . Clearly the application of  $\emptyset_v$  to design  $D$  will map design  $D$  exactly onto itself. Moreover, any initial subset of the point mappings of  $\emptyset_v$  which define the mappings of the target points  $\{1,2,3,\dots,k\}$ , say  $\emptyset_k$ , will map the first  $k$  rows of the incidence matrix of design  $D$  exactly onto themselves. Every such partial mapping,  $\emptyset_k$ , which leaves the first  $k$  rows of the incidence matrix of design  $D$  unchanged, will be extended during the backtracking process, and therefore a unique point relabelling corresponding to each of the automorphisms in  $G$  will be explicitly constructed. In other words, for any design  $D$  in canonical form, a side effect of the above canonicity test is that the complete automorphism group of the design will be constructed.

### Effectiveness

The efficient canonicity test described in Section 5.3.2.2 was implemented. The effectiveness of the optimisations can be measured by comparing the performance of this efficient canonicity test against the performance of the earlier brute force approach of Section 5.3.2.1.

For any design,  $D^c$ , which is in canonical form, the brute force approach will test all  $v!$  point relabellings, whereas the efficient approach will examine a smaller number of partial relabellings, but will at least examine all the relabellings which map  $D^c$  onto itself. For a design,  $D^n$ , which is not in canonical form, both the brute force and the efficient canonicity check terminate as soon as the first mapping is found which maps  $D^n$  to a lexicographically smaller design. To sort the columns of a given row permuted incidence matrix so that the original and permuted designs can be compared lexicographically, a quick radix sort was implemented. This makes  $v$  passes of the incidence matrix, one for each row working from row  $v$  up to row 1, and on the  $i^{\text{th}}$  pass, all the columns containing a "1" in row  $i$  are sorted to the left hand side of the incidence matrix, and all those columns containing a "0" are sorted to the right hand side. The relative ordering of the columns is maintained after each pass, and once the top row has been sorted, the columns of the incidence matrix will be in lexicographical order.

In the following tables, the brute force and efficient canonicity testing approaches are compared by enumerating 3 classes of BIBDs - of orders 7, 8 and 9. The total number of designs generated by the incidence matrix backtracking algorithm is given, along with the number of non-isomorphic designs. The total number and average number of point

relabellings (whether partial or complete) which are examined for both canonical and non-canonical designs are also given.

*Brute Force Canonicity Test Results:*

2-(7, 35, 15, 3, 5): Nd = 109

Total number of designs constructed = 798

	Total Number of Designs Generated	Number of Permutations Tested in Total	Average Number of Permutations Tested Per Design
Canonical	109	549360	5040
Non-Canonical	689	458761	665.8

2-(8, 14, 7, 4, 3): Nd = 4

Total number of designs constructed = 7

	Total Number of Designs Generated	Number of Permutations Tested in Total	Average Number of Permutations Tested Per Design
Canonical	4	161280	40320
Non-Canonical	3	10663	3554.3

2-(9, 18, 8, 4, 3): Nd = 11

Total number of designs constructed = 43

	Total Number of Designs Generated	Number of Permutations Tested in Total	Average Number of Permutations Tested Per Design
Canonical	11	3991680	362880
Non-Canonical	32	2954366	92323.9

Obviously, for every canonical design, exactly  $v!$  complete point relabellings were examined by this brute force strategy. Even for the non-canonical designs, the average number of relabellings considered grows very quickly as the order of the designs increases.

*Efficient Canonicity Test Results:*

2-(7, 35, 15, 3, 5): Nd = 109

Total number of designs constructed = 798

	Total Number of Designs Generated	Number of Permutations Tested in Total	Average Number of Permutations Tested Per Design
Canonical	109	37774	346.6
Non-Canonical	689	35630	51.7

2-(8, 14, 7, 4, 3): Nd = 4

Total number of designs constructed = 7

	Total Number of Designs Generated	Number of Permutations Tested in Total	Average Number of Permutations Tested Per Design
Canonical	4	9249	2312.3
Non-Canonical	3	339	113

2-(9, 18, 8, 4, 3): Nd = 11

Total number of designs constructed = 43

	Total Number of Designs Generated	Number of Permutations Tested in Total	Average Number of Permutations Tested Per Design
Canonical	11	17065	1551.4
Non-Canonical	32	10183	318.2

The improvements here over the brute force algorithm are enormous. The average number of relabellings tested for canonical designs grows very slowly, and in fact in the particular case of the above designs of orders 8 and 9, fewer relabellings are tested on average for the order 9 designs. In each case, only a tiny proportion of the symmetric group of  $v!$  permutations needs to be examined, making this efficient canonicity test practical for much larger designs.

Consider, for example, the twofold triple systems of order 12 - the 2-(12, 44, 11, 3, 2) designs. The average number of permutations required to be tested for each completed design in canonical form (taken over the first 300 non-isomorphic designs) is only 4661. This is only twice as many permutations as are examined for the order 8 design above, and is a mere 0.00097% of the 12! permutations which would require testing in the brute force approach.

In addition to being a very effective way of testing quite large designs for canonicity, this efficient test is completely general - not requiring any information about the structure of the designs. For example, exactly the same algorithm was used to test the triangle-free symmetric  $20_3$  configurations for canonicity. These structures are described, and their enumeration is covered in detail, in Case Study Seven of Chapter 6. The incidence matrix of one of these designs, on 20 points, is given to the right:

1:	11100000000000000000
2:	10011000000000000000
3:	10000110000000000000
4:	01000001100000000000
5:	01000000110000000000
6:	00100000001100000000
7:	00100000000011000000
8:	00010001000100000000
9:	00010000010001000000
10:	00001000100010000000
11:	00001000001000010000
12:	00000100100000010000
13:	00000100010000000100
14:	00000010001000000010
15:	00000010000001000001
16:	00000001000000110000
17:	00000000000100001010
18:	00000000000010000101
19:	0000000000000100110
20:	00000000000000011001

A canonicity test for this design, which shows that it is in canonical form, took 1.49 seconds and required only 45892 relabellings to be tested.

1:	11100000000000000000
2:	10011000000000000000
3:	10000110000000000000
4:	01000001100000000000
5:	01000000110000000000
6:	00100000001100000000
7:	00100000000011000000
8:	00010001000100000000
9:	00010000010001000000
10:	00001000100010000000
11:	00001000001000010000
12:	00000101000000100000
13:	000001000000000011000
14:	000000101000000000100
15:	000000100000000100010
16:	00000000100000000011
17:	000000000010000010100
18:	000000000001000010010
19:	000000000000100001001
20:	000000000000010101000
21:	000000000000001000101

To the left is the incidence matrix of a triangle-free symmetric configuration of order 21, also in canonical form.

In this case, the total number of relabellings tested to determine canonicity was only 51210, and this took just 1.75 seconds.

### 5.3.2.3 RESULTS

The incidence matrix backtracking algorithm, augmented with the efficient canonicity testing procedure, has been used to enumerate a number of design classes. A selection of the results are given in this section to illustrate its effectiveness. The algorithms were implemented in C, and run on a DEC ALPHA 800 5/400.

For the enumeration of BIBDs, ie. 2-(v,k,λ) designs, of particular interest is the calculation of the number of non-isomorphic designs when v and k are held constant, but λ varies. The algorithm has been used in this way to enumerate the 2-(6,3,λ) and 2-(7,3,λ) designs for all admissible values of λ ≤ 16. The results of these enumerations, including accurate execution times, are summarised below.

#### The 2-(6,3,λ) Designs

The number of non-isomorphic 2-(6,3,λ) designs are well known, and in fact can be calculated exactly by the formulas of Mathon [43] and of Engel and Gronau [26]. These designs exist for all even values of λ, and the program developed in this thesis has confirmed these numbers for λ ≤ 16. The results are summarised in the table on the next page, giving the total number of designs generated, the number of non-isomorphic designs as determined by the efficient canonicity test, and the required execution time for each enumeration:

Design	Number of Mutually Non-Isomorphic Designs	Number of Designs Generated in Total	Time (secs)
2-(6, 10, 5, 3, 2)	1	1	0.000
2-(6, 20, 10, 3, 4)	4	4	0.017
2-(6, 30, 15, 3, 6)	6	7	0.017
2-(6, 40, 20, 3, 8)	13	15	0.100
2-(6, 50, 25, 3, 10)	19	26	0.183
2-(6, 60, 30, 3, 12)	34	47	0.450
2-(6, 70, 35, 3, 14)	48	75	0.867
2-(6, 80, 40, 3, 16)	76	123	1.867

The combined running time for all the  $2-(6,3,\lambda)$  enumerations performed above is only 3.5 seconds. Note, the running times quoted in each of the cases include the time taken to generate the automorphism group of each of the non-isomorphic designs, which is calculated as part of the canonicity test. For example, below is the complete output produced by the program for the  $(6,3,16)$  enumeration, from which the execution time above was taken:

```

DESIGNENUMERATOR
=====

Ready to enumerate the 2-(6, 80, 40, 3, 16) designs.

Press <Return> to continue...

The enumeration is complete.

The number of complete 2-(6,3,16) designs generated was 123.
The total number of mutually non-isomorphic designs = 76.

Group Sizes:   Frequency:
-----
720             1.
60              4.
36              1.
24              8.
12             12.
8               2.
6              11.
4               3.
3              17.
2              12.
1               5.
-----
              76.

The process took 1.867 seconds to complete.

```

### The $2-(7,3,\lambda)$ Designs

The  $2-(7,3,\lambda)$  designs exist for all positive, integer, values of  $\lambda$ . A considerable amount of effort has gone into studying the  $2-(7,3,\lambda)$  designs in this thesis, and Case Study Three of Chapter 6 is devoted to this work.

The results of the  $2-(7,3,\lambda)$  enumerations are given over the page, for  $1 \leq \lambda \leq 16$ :

Design	Number of Mutually Non-Isomorphic Designs	Number of Designs Generated in Total	Time (secs)
2-(7,7,3,3,1)	1	1	0
2-(7,14,6,3,2)	4	7	0
2-(7,21,9,3,3)	10	20	0.1
2-(7,28,12,3,4)	35	98	0.5
2-(7,35,15,3,5)	109	440	2.0
2-(7,42,18,3,6)	418	1929	8.5
2-(7,49,21,3,7)	1508	7191	32.5
2-(7,56,24,3,8)	5413	26181	134.5
2-(7,63,27,3,9)	17785	85726	457.9
2-(7,70,30,3,10)	54613	255337	1643.1
2-(7,77,33,3,11)	155118	709953	5113.7
2-(7,84,36,3,12)	412991	1863037	13684.5
2-(7,91,39,3,13)	1033129	4565870	40396.5
2-(7,98,42,3,14)	2449592	10639833	100378.9
2-(7,105,45,3,15)	5528257	23684481	258076.8
2-(7,112,48,3,16)	11944627	50594093	590782.2

The program output of the 2-(7,49,21,3,7) enumeration is given below:

```

DESIGNENUMERATOR
=====

Ready to enumerate the 2-(7, 49, 21, 3, 7) designs.

Press <Return> to continue...

The enumeration is complete.

The number of complete 2-(7,3,7) designs generated was 7191.
The total number of mutually non-isomorphic designs = 1508.

Group Sizes:   Frequency:
-----
168             2.
 48             1.
 42             1.
 24             10.
 21             5.
 16             2.
 12             8.
 8              10.
 6              30.
 4              37.
 3             103.
 2             214.
 1            1085.
-----
                1508.

The process took 32.465 seconds to complete.

```

The complete set of all 1508 pairwise non-isomorphic 2-(7,3,7) designs can be constructed, and have their automorphism group sizes calculated, in a little over half a minute. This equates to building, testing for canonicity, and calculating the automorphism group of almost 50 non-isomorphic designs a second. In fact the first 100 non-isomorphic designs are constructed and classified in 1.1 seconds.

In [47], McKay mentions "We have an implementation that will generate small designs efficiently. For example, the set of all 1508 2-(7,3,7) designs is made in about 2.5 minutes". In addition, McKay confirmed the  $\lambda = 16$  result in a CPU time of 832 hours [51].

The running time of the 2-(7,3,16) enumeration above is equivalent to 164 hours, and the 2-(7,3,7) enumeration was completed in around half a minute. Although different architectures were used to perform the enumerations, it is likely that the algorithm developed in this thesis performs at least as well as McKay's implementation on this class of designs.

The final results given in this section are for the enumeration of the twofold triple systems, which have been used frequently as example designs up to this point.

Design	Number of Mutually Non-Isomorphic Designs	Number of Designs Generated in Total	Time (secs)
TTS(6)	1	1	0.000
TTS(7)	4	7	0.017
TTS(9)	36	294	1.950
TTS(10)	960	11316	118.100

The group sizes of these designs are given in the table below. The column headed  $|G|$  gives the automorphism group sizes of the designs, and the column headed  $N_d$  gives the number of non-isomorphic TTSs with the corresponding group size:

TTS(6)		TTS(7)		TTS(9)		TTS(10)	
$ G $	$N_d$	$ G $	$N_d$	$ G $	$N_d$	$ G $	$N_d$
60	1	168	1	432	1	108	2
		48	1	108	1	60	1
		42	1	80	1	28	1
		24	1	32	1	24	2
				24	3	21	1
				18	1	18	1
				8	4	12	5
				6	5	9	2
				4	5	8	1
				3	1	6	8
				2	8	5	1
				1	5	4	15
						3	36
						2	117
						1	767

## 5.4 CONCLUSION

This chapter has explored two ways in which the efficient incidence matrix backtracking algorithm developed in Chapters 3 and 4 can be used to constructively enumerate classes of designs. An efficient list building algorithm was developed, and has been used in Case Study Seven of Chapter 6 to enumerate configurations. An efficient canonicity test was also developed, and was then used to enumerate some classes of small designs for which the number of non-isomorphic structures were known. The resulting enumeration tool appears to be very efficient, and probably as good as other general enumeration algorithms.

The most important optimisation to the enumeration algorithm - that of parallelism - has only been touched on at this stage. Any enumeration problem tackled with the algorithm using the canonicity test can easily be divided into an arbitrary number of pieces, such that each piece can be computed on a separate, independent processor. The results simply need to be collated and added once each piece of the search has completed. Parallelisation with the canonicity test is covered in detail in Case Study Two of Chapter 6, and has been vitally important to the calculation of a number of new results in the field of combinatorial design theory. These new results are covered in their respective case studies in Chapter 6, and summarised in Chapter 7.





# Chapter 6

## Case Studies

### 6.1 CASE STUDY ONE: WEAKLY UNION FREE TWOFOLD TRIPLE SYSTEMS

Recall from Section 2.3.6 that a  $TTS(v)$  is said to be weakly union free, and denoted  $wuf-TTS(v)$ , if whenever 4 distinct blocks  $B_1, B_2, B_3$  and  $B_4$  of the design are chosen, it is not the case that  $B_1 \cup B_2 = B_3 \cup B_4$ .

This case study continues on from the problem studied in Section 2.3.6.2, in which a hill-climbing algorithm was unsuccessful in constructing a  $wuf-TTS(12)$ . Chee, Colbourn and Ling [11], state the following theorem:

**Theorem:**

A  $wuf-TTS(n)$  exists whenever  $n \equiv 0,1 \pmod{3}$  except when  $n \in \{3,4,6,7,9,10\}$  and possibly when  $n \in \{12,15,18,22\}$ .

The aim of this case study is to settle the existence question for the smallest unknown order, 12.

This problem is also particularly interesting, because it provides an excellent example of the effectiveness of the developed isomorph rejection techniques. In order to determine existence or non-existence, the basic incidence matrix backtracking algorithm was extended to include the weakly union free constraint. This case study explores the problem in two stages: firstly without partial isomorph rejection, and secondly with it. In the first case, a Monte-Carlo algorithm is described and implemented to estimate the size of the search space, and the feasibility of the approach. The quality of the estimations are evaluated and then compared to the results achieved with the use of isomorph rejection.

#### 6.1.1 THE SEARCH TREE

The search space for this problem can be represented as a tree, on 12 levels, where each node at level  $n$  in the tree corresponds to a particular valid configuration of the first  $n$  rows of the incidence matrix. The aim is therefore to find any node of the tree at level 12, as this would correspond to a valid  $wuf-TTS(12)$ .

This section examines the search tree of the design. Initially, no partial isomorph rejection as presented in Section 4.4.3 is performed, however the pair counting and point counting constraints are used, and no point is moved outside of the columns containing its covering point during backtracking. When isomorph rejection is introduced, so are all of the optimisations related to it, and its effect on the search tree can be more accurately determined.

The complete parameter set of a  $wuf-TTS(12)$  design is  $t(v,b,r,k,\lambda) = 2-(12,44,11,3,2)$ , thus the incidence matrix is 12 rows deep and 44 columns wide, and contains 11 points on each row.

The first row of the incidence matrix is unique:

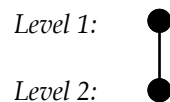
Row 1 1111111111100

The 1's must be in the 11 leftmost columns, for any other placement of points on the first row could be rearranged to give the above configuration simply by swapping columns.

The next row of the incidence matrix is also forced. It must intersect with row 1 in exactly two places, the leftmost two columns, and the remaining 9 1's will be placed in the leftmost columns which do not contain the point 1.

Row 2 1111111111100  
 Row 1 100000000011111111100000000000000000000000000000000

The search tree corresponding to these first two rows is therefore trivial:



Row 3 is the first row for which there is any choice over the placement of points. There are only 3 distinct ways row 3 can be configured:

Row 3 1111111111100  
 Row 2 100000000011111111100000000000000000000000000000000  
 Row 3 1100000000000000000011111111100000000000000000000000  
 OR  
 Row 3 1010000000010000000011111111000000000000000000000000  
 OR  
 Row 3 0011000000011000000011111110000000000000000000000000

The first configuration of row 3 above forms two copies of the block {1,2,3}, the second configuration forms just one copy of the block {1,2,3}, and the third configuration forms no complete blocks. Due to the lexicographical row ordering imposed, no extension to the third configuration can lead to a complete solution because the first two blocks can not be completed.

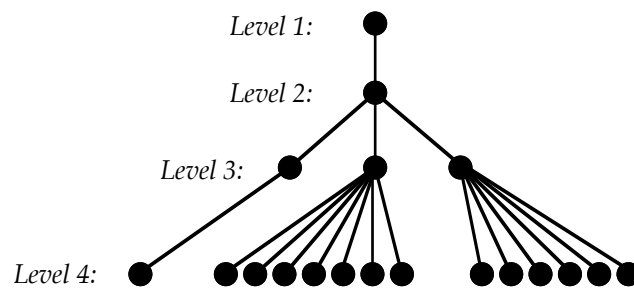
The final row to be examined in detail is row 4. There are exactly 14 possible configurations for row 4. One of these extends the first configuration of row 3 above, 7 extend the second configuration of row 3, and 6 extend the third configuration.

Possible extension of the first configuration of row 3:  
 Row 4 001100000001100000001100000001111100000000000000000

Possible extensions of the second configuration of row 3:  
 Row 4 011000000001000000010000000111111000000000000000000  
 Row 4 010100000001000000010000000111111000000000000000000  
 Row 4 010100000001000000011000000111111000000000000000000  
 Row 4 00110000000110000000000000000111111000000000000000000  
 Row 4 001100000001100000010000000111111000000000000000000  
 Row 4 00011000000110000000100000001111110000000000000000000  
 Row 4 0001100000001100000011000000111110000000000000000000

Possible extensions of the third configuration of row 3:  
 Row 4 001100000000110000000000000111111000000000000000000  
 Row 4 0010100000010100000000000000111111000000000000000000  
 Row 4 0010100000001100000100000011111100000000000000000000  
 Row 4 00001100000110000000000000001111110000000000000000000  
 Row 4 00001100000101000000100000011111100000000000000000000  
 Row 4 0000110000000110000011000001111100000000000000000000

The search tree, down to level 4, now becomes:



## 6.1.2 WEAKLY UNION FREE CONSTRAINT

For each row, certain point placements are invalid due to the weakly union free constraint. Just as in Chapter 1, a forbidden blocks structure is used containing those blocks which cannot be added to the design without causing a weakly union free violation. An important optimisation regarding the maintenance of this structure is the fact that the backtracking algorithm removes blocks in the opposite order to which they are constructed. For example, if row  $n$  is being constructed, and a point is added to column  $c$  containing  $k-1$  points, then this will generate a new block containing the point " $n$ ". Any other blocks which are completed and added to the partial design after this block, must be removed from the partial design before this block can be removed. This is a direct result of the systematic nature of backtracking, and is a useful property for maintaining the forbidden blocks structure.

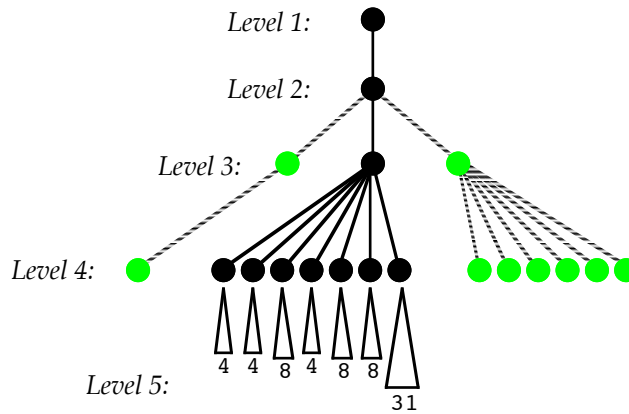
Whenever a new block is added to the partial solution, the set of forbidden blocks it induces is generated. This is done by examining all pairs of blocks currently in the partial solution along with the new block, and then generating the set of all blocks which when combined with these 3 blocks create a weakly union free violation. Obviously, once the new block is added to the current partial solution, none of the constructed forbidden blocks can also be added, without the violation of the weakly union free property.

Let  $b_n$  be a new block which is just created and added to the current partial solution. Every forbidden block induced by  $b_n$ , which was not already forbidden before  $b_n$  was constructed, is inserted into a list of newly created forbidden blocks and attached to  $b_n$  in the current partial solution. This information can then be used to reject subsequently generated blocks. When the backtracking of the incidence matrix reaches block  $b_n$  again, it is removed from the current partial solution. At this point, it is known that every block which was generated after  $b_n$  has also been removed, and every block which was generated before  $b_n$  still exists in the current partial solution. Therefore, the complete set of forbidden blocks must be reset to exactly what it was before the addition of the block  $b_n$ . This is achieved by simply removing from the complete set of forbidden blocks, each block belonging to the list of forbidden blocks attached to  $b_n$ .

## 6.1.3 ESTIMATION OF THE SIZE OF THE SEARCH TREE

To approximate how long it will take to perform the exhaustive search, the total number of nodes in the search tree can be estimated. If a good approximation can be made of the rate at which these nodes are constructed, then an estimate of the total time required for the search can be computed.

For simplicity, the analysis performed in this section will be restricted to the middle branch at level 3 of the search tree, containing 7 extensions to row 4. The number of valid extensions at the next level, level 5, of this middle branch have been examined. These are illustrated overleaf:



There are 67 valid row 5's which can be extended from the 7 valid row 4's which can be extended from the second of the 3 valid row 3's.

As deeper levels of the tree are examined, the exponential growth rate of the search space becomes apparent. A very rough estimate of the number of nodes in the search tree can be calculated by computing the average number of nodes at each level of the tree, taken from a number of samples.

For each of the 67 nodes at level 5, the average number of valid rows that these extend to at level 6 is approximately 40. Each valid row 6 then extends to approximately 100 valid row 7's, each of which extend to roughly 35 valid row 8's. Each valid row 8 yields around 500 valid row 9's, which in turn extend to roughly 100 valid row 10's. The constraints of the search at levels 11 and 12 are very tight, so there are very few valid rows at either of these levels. The table below summarises this very crude analysis:

Level of the search tree	Approximation to the average number of nodes at the specified level which are extended to from each node at the previous level
6	40
7	100
8	35
9	500
10	100
11	few
12	few

These figures can be used to estimate the number of nodes in the search tree. As each figure was calculated by averaging the density of branches in the tree at fixed sample points, the calculated estimate relies on the assumption that the search tree is highly symmetric. Although this is not the case, the purpose of this first estimate is to give a very rough indication of the size of the search tree, and this can be computed by taking the product of the average densities at each level of the tree. This yields a very rough estimate of 500,000,000,000 nodes in the search tree lying below the middle branch at level 3. Although imprecise, this figure gives some idea of the magnitude of the search.

### 6.1.3.1 MONTE-CARLO ESTIMATION

A much more accurate estimate of the search space can be calculated using a Monte-Carlo algorithm, which makes a number of random walks down the tree, examining at each step the density of the branches at that level. The greater the number of random walks made, the more accurate the approximation.

A random walk starts at the root of the search tree, and steps down from level to level until a node of the tree is reached which is either a final solution, or has no valid extensions.

The walk traverses the search tree from node to node, and in the case of the wuf-TTS search space, a node at level  $L$  corresponds to a valid partial incidence matrix constructed on the first  $L$  rows. At a particular level,  $L$ , of the random walk, all the possible extensions from the current node to level  $L+1$  are generated. If there are  $x_L$  of these possible extensions, one of them is chosen at random with probability  $1/x_L$ , and this extension then becomes the node at level  $L+1$  of the random walk. In this way, a partial incidence matrix is built up row by row as each new node of the tree is traversed during the walk, and the random walk ends when a level is reached where  $x_L = 0$ .

At the end of a random walk, the following formula is calculated:

$$f = 1 + x_1 + x_1 x_2 + x_1 x_2 x_3 + x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_4 x_5 + \dots$$

The expected value of  $f$ ,  $E(f)$ , is precisely the number of nodes in the tree, which corresponds to the number of cases which will be examined by the backtrack algorithm. A proof of the calculation of  $E(f)$  is given in [57].

If an accurate way of estimating the time taken to generate each case is developed, then the results of the Monte-Carlo algorithm can be used to estimate the running time of the complete search.

The C-pseudocode for the main loop of the Monte Carlo algorithm is as follows:

```

Overall_Product=0;
Overall_Node_Total=0;
for (i=1; i<=num_runs; i++) {
  Node_Total = 1; Product = 1; current_level = 1;
  do {
    current_level++;
    Generate_All_Extensions(current_level);
    branch_density = Total Number of Extensions
    if (branch_density > 0) {
      rand_branch = Choose one of the branch_density extensions at random
      Use rand_branch to extend partial solution
      Product = Product * branch_density;
      Node_Total = Node_Total + Product;
    }
  } while ((branch_density > 0) && (current_level < ENUMERATE_LEVEL));

  Overall_Product = Overall_Product + Product/num_runs;
  Overall_Node_Total = Overall_Node_Total + Node_Total/num_runs;
}

```

The variable `ENUMERATE_LEVEL` specifies the level of the search tree at which the Monte-Carlo random walks should stop. The calculated result of the algorithm will therefore be an estimate of the total number of nodes in the search tree down to `ENUMERATE_LEVEL`. The reason for allowing such a partial calculation is so that the accuracy of the estimated numbers can be evaluated. For example, it will always be possible to perform a complete enumeration of the search space down to some relatively shallow level of the search tree, say  $L$ , in a reasonable amount of time. Once this has been performed, the number of nodes in the search tree down to level  $L$  will be known precisely. To evaluate the accuracy of the Monte-Carlo algorithm, the random walks can be performed down to level  $L$  of the search tree, and the estimated number of nodes can then be compared with the exact number of nodes. This will give a qualitative measure of the accuracy of the algorithm, which is important for evaluating the accuracy of an estimate of the total number of nodes in the entire search space.

### 6.1.3.2 RESULTS

Recall that the analysis of the search space is performed using only the second of the valid extensions to row 3, meaning there is a single node at level 3, and 7 nodes in total at level 4. A complete enumeration of the wuf-TTS search space without isomorph rejection was possible down to level 8 of the search tree. This took a little over 5 hours of CPU time, and the results are given below:

Level	Total Number of nodes in the search space down to this level	Number of nodes at this level of the search space only	Execution Time (secs)
3	1	1	0.00
4	8	7	0.00
5	75	67	0.01
6	2,660	2,585	0.09
7	401,733	399,073	23.59
8	108,573,273	108,171,540	19176.00

The exponential growth rate of the search space is well illustrated in the table above, both in the number of nodes and the execution time required for the enumerations. The Monte-Carlo algorithm was run on the same search tree, estimating the number of nodes down to each level from 4 to 12. At levels 1 and 2 there is only a single node, and at level 3, only the second of the valid extensions to row 3 is considered.

The Monte-Carlo experiments for each level of the search tree were performed twice, once using 1,000 random walks and once using 10,000 random walks. The results, given below, are then compared to the actual figures.

The number of runs, or the number of random walks made down the tree in the results below is 1,000:

Level	Estimate of the total number of nodes in the tree down to this level:	Estimate of the total number of nodes at this level only:
4	8.00	7.00
5	75.68	67.68
6	2,704.36	2,630.57
7	440,068.99	437,093.71
8	82,848,500.76	82,524,126.75
9	2,946,597,085.52	2,848,747,084.13
10	3,732,798,141.80	3,244,361,205.30
11	3,841,927,537.25	3,648,600,204.61
12	4,892,276,200.99	4,579,886,718.62

The same experiment was performed again, this time using 10,000 random walks:

Level	Estimate of the total number of nodes in the tree down to this level:	Estimate of the total number of nodes at this level only:
4	8.00	7.00
5	75.43	67.43
6	2,705.61	2,629.84
7	400,315.40	397,662.22
8	112,624,073.53	112,212,662.46
9	4,118,632,626.71	4,007,472,737.92
10	3,175,715,595.52	2,831,455,433.96
11	4,310,931,212.42	3,960,991,928.12
12	4,516,930,874.02	4,137,773,521.89

For the shallow levels of the search space, the Monte-Carlo algorithm performs very accurately when compared to the exact figures. Deeper than level 8, the results are likely to be of the correct order of magnitude and are certainly useful approximations. The second

Monte-Carlo experiment, using 10,000 random walks, produced more accurate figures than the first, estimating 112,000,000 nodes, compared to the actual 108,000,000 at level 8.

The reason why the values for levels 10, 11 and 12 of the Monte Carlo output are so similar is that there are very few valid extensions to rows on these levels, as the constraints of the problem become very restrictive. The nodes produced by the random walk represent valid partial designs, and are quite rare in the deep levels of the tree. Any node at level 12 of the search tree would correspond to a valid wuf-TTS(12) design, but none were constructed during the Monte-Carlo experiments.

To produce an accurate estimation of the execution time required to perform a complete enumeration of the search space, the rate at which valid rows of the search tree are constructed must be determined.

#### 6.1.4 ESTIMATION OF EXHAUSTIVE SEARCH RUNNING TIME

The time taken to generate 10,000 valid, extendable configurations of the incidence matrix, depends on the level to which the enumeration is being performed. For an enumeration down to each level  $L$ , for  $7 \leq L \leq 12$ , of the search space, this time is given below:

Level of the SearchTree	Average time taken to construct 10,000 valid, extendable configurations(secs)
7	0.65
8	2.30
9	25.00
10	197.00
11	175.00
12	170.00

Firstly, these results will be used to estimate how long a complete enumeration of the search space would take down to level 7 and down to level 8. Both of these results are known, and so the accuracy of the method being used in this section for estimating the running time can be evaluated. If the prediction appears to be reliable, then a similar estimation can be made for the running time of a complete enumeration.

Given that, when enumerating down to level  $L$ , it takes an average of  $t$  seconds to generate 10,000 valid configurations of the search space, a complete enumeration down to level  $L$  will take an estimated  $(N_L t) / 10,000$  seconds, where  $N_L$  is the total number of nodes in the search space down to level  $L$ .

For example, down to level 7 of the search tree, there are exactly  $N_L = 401733$  nodes. From the table above, it takes approximately  $t = 0.65$  seconds to examine 10,000 nodes of the search tree. The estimated time to completely enumerate down to level 7 is therefore:

$$\begin{aligned} &= (N_L t) / 10000 \\ &= (401733 \times 0.65) / 10000 \\ &= 26 \text{ seconds.} \end{aligned}$$

The actual CPU time for this enumeration was 23.59 seconds, so the estimate appears to be quite accurate.

The same analysis can be performed to level 8 of the search tree. There are exactly 108,573,273 nodes down to this level, and from the sample enumeration it took 2.3 seconds to examine 10,000 nodes. The estimated running time for a complete enumeration of the first 8 levels is therefore:

$$\begin{aligned} &= (N_L t) / 10000 \\ &= (108573273 \times 2.3) / 10000 \\ &= 24971 \text{ seconds.} \end{aligned}$$



The actual CPU time for this enumeration was 19176.3 seconds, so again the estimate is fairly accurate.

Using the same calculations, an estimate of the execution time for a complete enumeration down to level 12 can be computed.

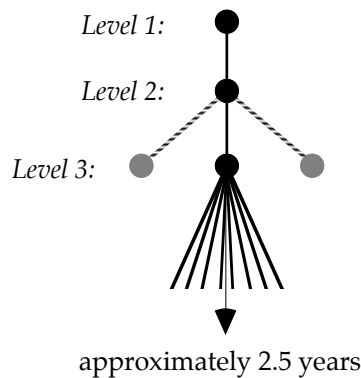
The estimated number of nodes in the complete search tree, taken from the Monte-Carlo experiment using 10,000 random walks, was approximately 4,500,000,000. From the sample searches performed to this level, it took approximately 170 seconds to examine 10,000 valid nodes. The reason for the slow rate of valid node generation is that for searches deep in the tree, a considerable amount of time is spent backtracking on the rows of the incidence matrix between the construction of each valid row.

An estimate for the total time required to search the entire tree is therefore:

$$\begin{aligned}
 &= (N_{\text{I,xt}}) / 10000 \\
 &= (4,500,000,000 \times 170) / 10000 \\
 &= 76,500,000 \text{ seconds.}
 \end{aligned}$$

This is equivalent to nearly 900 days of computation time - approximately 2 and a half years.

Recall that this analysis was performed for only the middle branch of the search tree at level 3. This figure is therefore an estimate of the time required to search only extensions to this middle branch, as the diagram below illustrates:

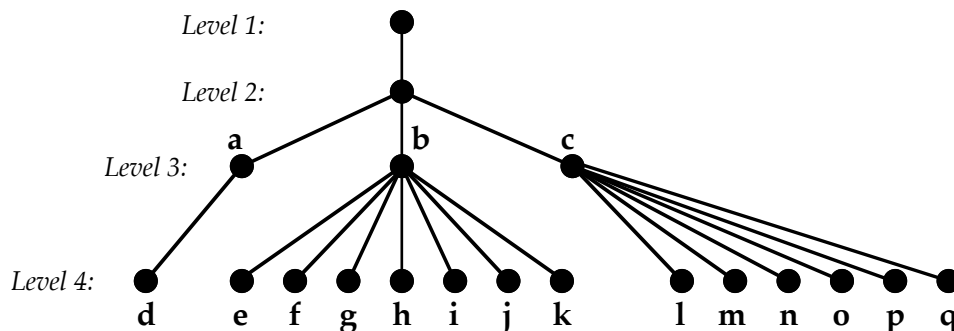


Extensions to the other two branches at level 3 would also need to be exhaustively searched.

The general conclusion of this analysis is that such a search is infeasible, without some further optimisations. The partial isomorph rejection techniques presented in Chapter 4 are ideal.

### 6.1.5 THE EFFECT OF ISOMORPH REJECTION

Firstly, recall the search tree structure at levels 3 and 4, labelled as below:



The nodes are labelled such that the lexicographical order of the labels represents the construction order of the corresponding partial incidence matrices across level 3 and across level 4. For example, the partial incidence matrix represented by node e is lexicographically larger than the partial incidence matrix represented by node d, and was hence produced later in the search.

Using the technique of partial isomorph rejection presented in Section 4.4.3, a number of the partial incidence matrices at level 4 turn out to be isomorphic, and so their corresponding nodes can be eliminated from the search tree. The real significance of such rejection is that not only are the individual nodes at level 4 rejected, but so are all the nodes in the search tree formed as extensions to them.

None of the 3 nodes at level 3 are isomorphic, and so no rejection can take place at this level.

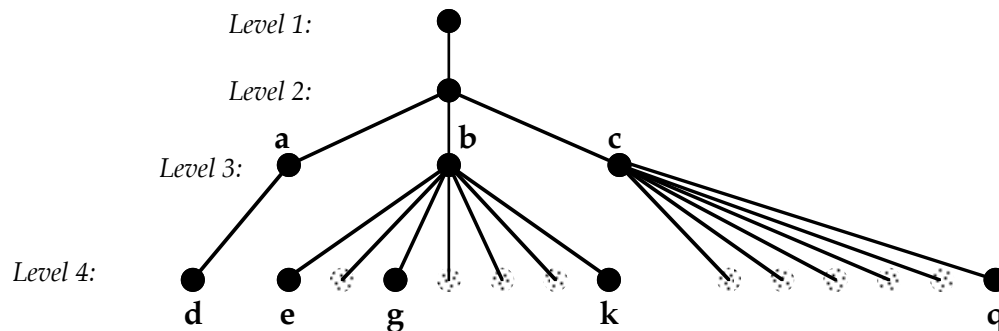
Recall that for node b, its 7 extensions to row 4, e-k, themselves extended to either 4, 8 or 31 nodes at level 5. The table below summarises this:

Node	e	f	g	h	i	j	k
Number of valid nodes at level 5 this node extends to	4	4	8	4	8	8	31

As it turns out for these 7 nodes, nodes f and h are isomorphic to node e, and nodes i and j are isomorphic to node g. Nodes e, g and k are the only representative nodes required as extensions to b.

For the 6 extensions to node c, which are the lexicographically largest configurations at this level, the rejection is even greater. Nodes l, m, n, o and p are all isomorphic to previous nodes at level 4, and can be rejected. Only node q is non-isomorphic to the other representative nodes at level 4, and cannot be discarded.

When pruned with partial isomorph rejection, the new search tree therefore becomes:



Instead of there being 14 nodes to extend at level 4, there are now only 5 - representing a reduction in the number of level 4 nodes by a factor of nearly 3. If a similar proportion of nodes were rejected at each of the next  $n$  levels of the search tree, the overall size of the search tree could be reduced by a factor of roughly  $3^{n+1}$ .

The amount of rejection has been examined in detail for all extensions down to level 8 of the middle node at level 3 - ie. node b. Partial isomorph rejection is performed by generating the complete automorphism group of all partial designs constructed in each of the searches. In addition, explicit row ordering is enforced, and the first point placed on each row is never backtracked - both consequences of the isomorph rejection techniques.

The results are summarised in the table below, which gives the total number of nodes in the tree with and without partial isomorph rejection:

Level of the search tree	Total number of nodes in the tree down to this level, without partial isomorph rejection.	Total number of nodes in the tree down to this level, with partial isomorph rejection.
3	1	1
4	8	3
5	75	9
6	2,660	72
7	401,733	1,547
8	108,573,273	73,932
<i>Execution Time for level 8 (secs)</i>	19128.0	10.3

The amount of rejection is considerable, both in terms of the number of nodes rejected from the tree, and the running time of the algorithm. Without isomorph rejection, the enumeration to level 8 took a little over 5 hours. Although performing the partial isomorph rejection adds a certain overhead to the search, such as the construction of the automorphism groups of the partial designs, the running time of the complete search to level 8 was completed in around 10 seconds.

The number of nodes in the search tree is in fact a good predictor for the running time of the algorithm. Using the number of nodes in the tree with and without partial isomorph rejection, and the execution time of the algorithm without partial isomorph rejection, the estimated time to perform the search with partial isomorph rejection can be calculated as:

$$\left(\frac{73932}{108573273}\right) \times 19128 = 13.0$$

which is very close to the actual running time of the algorithm, using partial isomorph rejection.

## 6.1.6 SETTLING THE WUF-TTS(12) EXISTENCE QUESTION

With the algorithm performing so efficiently for this problem, the existence question was settled in less than a minute and a half.

Automorphism groups were calculated for all partial incidence matrices down to level 7 of the 12 row incidence matrix, and the column permutations derived from these groups were used to reject rows down to level 9 of the incidence matrix. The screen output of the program performing the enumeration is given below:

The number of wuf-TTS(12) designs generated was 0.  
 The total number of mutually non-isomorphic designs = 0.  
 The process took 81.930 seconds to complete.

### Result 6.1.1:

There are no weakly union free twofold triple systems of order 12.

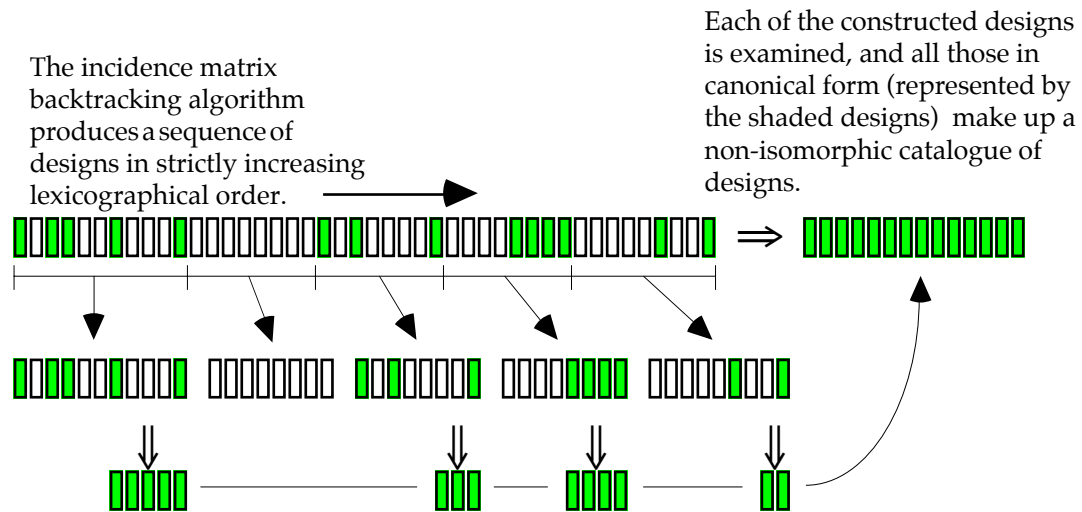
## 6.1.7 ENUMERATION OF THE WUF-TTS(13) DESIGNS

The smallest order for which weakly union free twofold triple systems are known to exist is 13. With result 6.1.1 above, order 13 is therefore the smallest order for which wuf-TTS designs exist. In fact only one such design is known, having an automorphism group of size 156, and its construction was given by Frankl and Furedi in [29].



design independently, without the need to explicitly examine any other design constructed at another point of the search.

Parallelisation of the algorithm is therefore relatively straightforward, and can be achieved by splitting the generation of the designs by the incidence matrix backtracking algorithm into a number of pieces. The designs within each piece of the search can then be tested for isomorphism independently using the canonicity test. This is illustrated below:



If the design generation is split up into arbitrary pieces, then all the designs within a given piece which are in canonical form can be used to form a set of non-isomorphic designs for the particular piece.

These sets can simply be combined to give the complete set of non-isomorphic representatives.

Two different techniques of partitioning the design generation have been explored. The first method splits the design generation by counting the number of times a particular row is backtracked, and enumerating all the designs which lie between some given starting and ending backtrack operation. This first approach is the easier to implement, but has certain drawbacks - mainly the fact that it is difficult to create partitions of even size.

The second method splits the design generation in a more natural way, by producing all possible extensions to some subset of partial designs. This approach has a number of advantages, including considerably more even partitioning of the search space, and the fact that the canonicity test can be used to prune all isomorphs from the initial set of starting partial designs, resulting in an even greater search space reduction.

## 6.2.2 SPLITTING ON BACKTRACK CALLS

This is the simplest technique by which the search can be partitioned. Some arbitrary row of the incidence matrix can be selected as the level at which to partition the design generation. The selected level should be shallow enough to allow a complete enumeration of the search tree down to that level in a reasonable amount of time. Let  $L$  represent the selected level, and let  $B_L$  denote the total number of times row  $L$  of the incidence matrix is backtracked during a complete enumeration of the search space.  $B_L$  can be easily calculated by enumerating completely down to level  $L$ , but no deeper.

The design generation is then split into pieces on the number of backtrack calls to row  $L$ , by specifying a range of backtrack calls between which the search tree will be completely examined, and all possible designs will be generated. Outside this range, the search will never extend past level  $L$ , so no designs will be constructed. As long as the specified ranges

for each piece of the search are non-overlapping, and together cover all possible values, the entire search space will be exhaustively explored and all valid designs generated.

In practice, this technique is not very useful, because it is very difficult to specify ranges which partition the search into roughly equal pieces. The next section outlines a much more useful technique.

### 6.2.3 SPLITTING ON STARTER CONFIGURATIONS

Once again, a level,  $L$ , of the search space is chosen such that a complete enumeration of the search space down to level  $L$  can be performed fairly quickly.

Let  $N_L$  denote the total number of partial designs, consisting of the initial  $L$  rows of the incidence matrix, constructed following a complete enumeration to level  $L$ , and call each partial design a *starter configuration*. An exhaustive enumeration of the entire search space can be performed by producing all possible extensions to each one of the starter configurations. Therefore, the enumeration can be split into an arbitrary number of pieces, where a given piece generates all possible extensions to a particular subset of the starter configurations.

The particularly attractive feature of this approach is that the canonicity test can be performed on each starter configuration down to level  $L$ , and only those in canonical form need to be extended. Any starter configuration,  $C_n$ , which is not in canonical form, is isomorphic to the canonical representative,  $C_c$ , of its isomorphism class. Both  $C_n$  and  $C_c$  have the same structure, and so any extension of  $C_n$  will be isomorphic to, and lexicographically greater than, a corresponding extension of  $C_c$ . Therefore, only the canonical starter configurations need to be considered. Extending only canonical starter configurations is a considerable optimisation, because not only does it eliminate the unnecessary searching performed following the extension of a non-canonical starter configuration, but it also eliminates all the unnecessary canonicity tests which would be performed on any constructed designs.

If the canonicity test is performed on every row of the incidence matrix during the construction process, only a single representative of each isomorphism class of each partial design will be constructed, which is very much like a true orderly algorithm as presented by Read in [56]. However, as the size of the partial designs increases, performing the canonicity tests becomes more expensive. In practice, it is therefore more efficient to perform the complete canonicity test on only a certain number of initial rows, and let the partial isomorph rejection techniques work on the remaining rows. There is complete freedom over selecting the level of the incidence matrix past which canonicity testing should no longer be performed, and the optimal level depends very much on the class of designs being enumerated.

There still remains the choice of how to distribute the starter configurations amongst the available processors. A very simple way is to produce all possible extensions to a specified range of consecutive starter configurations. For example, each processor has a starting index, *START*, and an ending index, *END*, specified, and if all the canonical starter configurations are numbered in the lexicographical order in which are constructed, then only those lying between the range of *START* and *END* are extended.

For example, assume a class of designs, of order  $v$ , are to be enumerated. Further, assume that there are exactly 10 canonical, or pairwise non-isomorphic, starter configurations constructed down to some level  $L < v$ . These starter configurations are illustrated in the diagram below, and are labelled  $CS_1, CS_2, \dots, CS_{10}$  in the lexicographical order of construction.



Given 2 processors, the enumeration could be split by producing all extensions to starter configurations  $CS_1, CS_2, \dots, CS_5$  on one processor, and all the extensions to  $CS_6, CS_7, \dots, CS_{10}$  on the other. There will be a small overhead to this approach however, because in order to independently generate  $CS_6, CS_7, \dots, CS_{10}$ , the second processor must also build the initial 5 starter configurations, repeating the work of the first processor. However, if the level  $L$  is chosen so that the complete set of starter configurations can be constructed very quickly, the overhead of this repeated computation will be negligible compared to the CPU time required for the complete search.

This approach has one drawback however - some starter configurations will take longer to completely extend than others. For example, the lexicographically smaller starter configurations will generally take longer to extend, because they tend to generate the bulk of completed designs which are in canonical form, and the partial isomorph rejection is less effective towards the early part of the search. The extensions of the lexicographically larger starter configurations are rejected to a greater degree by the partial isomorph rejection, and a larger proportion of completed designs are not in canonical form so are quickly rejected.

Therefore, a more effective way of parallelising the search is to distribute lexicographically consecutive starter configurations evenly amongst the available processors. The simplest way of doing this is with a modulo function. If the search is to be split amongst  $P$  processors, labelled 0 to  $P-1$ , and there are  $N$  starter configurations labelled in lexicographical order from 1 to  $N$ , then processor  $i$  generates the extensions of all starter configurations labelled  $j$ , such that  $j \bmod P = i$ .

Returning to the example above, if the 10 starter configuration  $CS_1, CS_2, \dots, CS_{10}$  are to split again amongst 2 processors,  $P_0$  and  $P_1$ , then  $P_0$  will produce all extensions to  $CS_2, CS_4, CS_6, CS_8$  and  $CS_{10}$ , and  $P_1$  will produce all extensions to  $CS_1, CS_3, CS_5, CS_7$  and  $CS_9$ . Clearly the entire search tree will be examined, and this method of distributing the search produces much more even partitions which require more similar amounts of execution time.

## 6.2.4 RESULTS

Each of the methods of parallelisation discussed in this case study have been implemented and have had their performance analysed. Certainly, the use of canonical starter configurations is far superior to splitting the search on the number of backtrack calls, and using the modulo function to split the search produces the best overall results.

Consider the 1508 non-isomorphic 2-(7,3,7) designs. In Section 5.3.2.3 of the previous chapter, these were constructed in 32.5 seconds. This section illustrates how this search can be split into two even pieces, using the techniques discussed in Section 6.2.3.

Firstly, using the approach of extending lexicographically adjacent starter configurations, this search was parallelised into two independent pieces. There are exactly 702 canonical starter configurations on the first 5 rows of the 7 row incidence matrix. If the search space is split into the first 351 and the second 351 starter configurations, then the first piece takes considerably longer than the second piece to enumerate. To achieve a much more even split of the search space, the first 250 of these can be enumerated on one processor, and the remaining 452 enumerated on another processor. The results are given on the facing page:

```

DESIGNENUMERATOR
=====

Ready to enumerate the 2-(7, 49, 21, 3, 7) designs.

Produce starter configurations to what level?      5
Begin extending at starter configuration: 1
Stop extending at starter configuration:           250

Press <Return> to continue...

The enumeration is complete.

The number of complete 2-(7,3,7) designs generated was 3061.
The total number of mutually non-isomorphic designs = 1144.

The group sizes:
Group Sizes:  Frequency:
-----
168          1.
24           7.
21           3.
16           2.
12           4.
8            8.
6           18.
4           19.
3           69.
2          155.
1           858.

The process took 16.249 seconds to complete.

```

```

DESIGNENUMERATOR
=====

Ready to enumerate the 2-(7, 49, 21, 3, 7) designs.

Produce starter configurations to what level?      5
Begin extending at starter configuration: 250
Stop extending at starter configuration:           end

Press <Return> to continue...

The enumeration is complete.

The number of complete 2-(7,3,7) designs generated was 4130.
The total number of mutually non-isomorphic designs = 364.

The group sizes:
Group Sizes:  Frequency:
-----
168          1.
48           1.
42           1.
24           3.
21           2.
12           4.
8            2.
6           12.
4           18.
3           34.
2           59.
1           227.

The process took 16.783 seconds to complete.

```



The combined number of valid designs generated for both enumerations above, 7191, is the same as the number generated by the single enumeration, which is as expected.

The first piece generated the first 1144 non-isomorphic designs in 16.25 seconds, and the second piece generated the remaining 364 non-isomorphic designs in 16.8 seconds - thus the observed running time of the algorithm was halved.

The combined running time of the two searches is approximately 33 seconds, which is only slightly longer than the enumeration performed on a single processor. The extra overhead comes from the repeated generation of the starter configurations, and contributes only a very small percentage to the total running time.

The same enumeration was performed using the modulo function to split the design generation. The first piece of the search extended all the even starter configurations and the second piece extended all the odd starter configurations. The results are given below:

Enumeration Piece	Valid designs generated	Non-Isomorphic designs	Execution time (secs)
First Piece (even starter configurations)	3514	707	15.89
Second Piece (odd starter configurations)	3677	801	17.11

This technique results in a very even partitioning of the search space, without any extra effort being required to calculate the divisions. The parallelisation techniques discussed in this section have been vital to many of the large families of designs enumerated in this thesis.

## **6.3 CASE STUDY THREE: THE 2-(7,3, $\lambda$ ) DESIGN FAMILY**

### **6.3.1 BACKGROUND**

The 2-(7,3, $\lambda$ ) designs are the class of BIBDs of block size 3, containing 7 points. The constructive enumeration algorithm developed in this thesis, in addition to the parallelisation techniques discussed in the previous case study, have been used to enumerate these designs for  $1 \leq \lambda \leq 16$ . The table on the next page gives the number of non-isomorphic and the number of distinct designs for each value of  $\lambda$ .

The three largest enumerations of this class have been performed for the first time here. Prior to this work, Mathon and Pietsch [44] had enumerated these designs for  $\lambda \leq 13$

Design	Number of Mutually Non-Isomorphic Designs	Number of Distinct Designs
2-(7,7,3,3,1)	1	30
2-(7,14,6,3,2)	4	465
2-(7,21,9,3,3)	10	5045
2-(7,28,12,3,4)	35	41265
2-(7,35,15,3,5)	109	265792
2-(7,42,18,3,6)	418	1402140
2-(7,49,21,3,7)	1508	6266415
2-(7,56,24,3,8)	5413	24389705
2-(7,63,27,3,9)	17785	84512555
2-(7,70,30,3,10)	54613	265328404
2-(7,77,33,3,11)	155118	765418845
2-(7,84,36,3,12)	412991	2052124690
2-(7,91,39,3,13)	1033129	5160956540
2-(7,98,42,3,14)	2449592	12268945580
2-(7,105,45,3,15)	5528257	27746605182
2-(7,112,48,3,16)	11944627	60016818765

The complete group spectrums for these designs are given in Chapter 7.

This case study gives a brief description of the enumeration of this family, but focuses on the  $\lambda = 15$  designs, for which the computed result was in doubt for a short period of time. The reasons for the uncertainty, the techniques used to check the result and gain confidence in the calculation, and the resolution of the discrepancy are discussed in detail.

## 6.3.2 COUNTING FORMULAS

For a small number of classes of combinatorial designs, counting formulas have been derived which are able to compute, for any arbitrary sized instance of the class, the number of distinct or the number of non-isomorphic structures. A good example of this is the application of Polya's general enumeration theory to compute the number of non-isomorphic graphs for arbitrary orders. Harary and Palmer give a detailed explanation of Polya's theorem, and its many applications, in [36].

### 6.3.2.1 ORDER 6 TRIPLE SYSTEMS

Similar counting formulas have been derived for small classes of block designs. For example, the following polynomial of degree 5 computes the number of distinct 2-(6,3,2 $\lambda$ ) designs:

$$D(2\lambda) = (\lambda + 1)(3\lambda^4 + 12\lambda^3 + 23\lambda^2 + 22\lambda + 12) / 12$$

Mathon [43] developed the above formula, along with a series of formulas for computing the number of non-isomorphic 2-(6,3,2 $\lambda$ ) designs, prior to 1984, yet these were not published until 1996.

In the meantime, Engel and Gronau [26] independently derived the same formula for the number of distinct 2-(6,3,2 $\lambda$ ) designs, using interpolation. They proved that the counting formula must be a polynomial of degree  $\binom{v}{k} - \binom{v}{t} = \binom{6}{3} - \binom{6}{2} = 5$ , which could thus be uniquely determined from the number of distinct designs for the first 6 values of 2 $\lambda$ , which were easily computed.

To illustrate the use of the above formula, it has been evaluated for  $2\lambda=54$ :

$$D(54) = 4,311,748$$

The constructive enumeration of the 2-(6, 270, 135, 3, 54) BIBDs has been performed using the algorithm developed in this thesis. There are exactly 7458 non-isomorphic 2-(6,3,54) designs, and their group sizes are given below, along with the number of induced distinct designs:

$ G $	$Nd$	$\frac{v!}{ G } \times Nd$
60	14	168
36	5	100
12	182	10920
6	177	21240
4	44	7920
3	942	226080
2	951	342360
1	5143	3702960
<b>TOTALS:</b>	<b>7458</b>	<b>4311748</b>

The number of distinct designs calculated agrees exactly with the general formula.

### 6.3.2.2 ORDER 7 TRIPLE SYSTEMS

Given a  $t$ -( $v,k,\lambda$ ) design,  $(X,B)$ , let  $X_h$  ( $0 < h < v$ ) be the set of all  $h$ -subsets of  $X$ , and let  $X_t = \{x_1, x_2, \dots, x_m\}$  and  $X_k = \{y_1, y_2, \dots, y_n\}$ , where  $m = \binom{v}{t}$  and  $n = \binom{v}{k}$ .

The  $A_{tk}$ -matrix of the design,  $A_{tk} = (a_{ij})$ , is the  $m \times n$  0-1 matrix where  $a_{ij} = 1$  if and only if  $x_i \in y_j$ . Such matrices are useful, because a  $t$ -( $v,k,\lambda$ ) design exists if and only if there is a solution,  $z$ , to the matrix equation  $A_{tk} \cdot z = \lambda \cdot J$  where  $z$  is an  $n$ -dimensional vector with nonnegative integer entries, and  $J$  is the  $m$ -dimensional all-ones vector.

Using the technique of interpolation adopted by Gronau and Engel, Mathon [46] suggested that the number of distinct  $2$ -( $7,3,\lambda$ ) designs may be computed by a polynomial of degree  $n$ , where  $n$  is the difference between the number of rows and the number of columns of the  $A_{tk}$ -matrix. For the  $2$ -( $7,3,\lambda$ ) designs, this equals  $\binom{v}{k} - \binom{v}{t}$ :

$$\begin{aligned}
 &= \binom{7}{3} - \binom{7}{2} \\
 &= 35 - 21 \\
 &= 14.
 \end{aligned}$$

The proposition was therefore that the number of distinct  $2$ -( $7,3,\lambda$ ) designs could be calculated by a polynomial of degree 14.

In order to calculate the coefficients of this degree 14 polynomial, 15 data points were required, which could simply be the number of distinct  $2$ -( $7,3,\lambda$ ) designs for  $\lambda = 0, 1, \dots, 14$ . In addition, just as the formula for the number of distinct  $2$ -( $6,3,2\lambda$ ) designs has a root at  $-1$ , it was also proposed that the  $2$ -( $7,3,\lambda$ ) formula would have a root at  $-1$ . With this assumption, only the number of distinct  $2$ -( $7,3,\lambda$ ) designs for  $\lambda = 0, 1, \dots, 13$  were required to derive the coefficients of the formula. These values, induced from the results of the constructive enumerations, are tabulated on the following page. Note that  $D(0)$  is always 1, because when  $\lambda=0$ , there is precisely one solution to the equation  $A_{tk} \cdot z = \lambda \cdot J$ .

$\lambda$	Number of Distinct Designs
0	1
1	30
2	465
3	5045
4	41265
5	265792
6	1402140
7	6266415
8	24389705
9	84512555
10	265328404
11	765418845
12	2052124690
13	5160956540

### 6.3.2.3 PROPOSED FORMULA

Interpolating a degree 14 polynomial from the above data points, with a root at -1, gives the following formula for the number of distinct 2-(7,3, $\lambda$ ) designs:

$$D(\lambda) = \frac{1}{c_0}(\lambda + 1)(c_{13}\lambda^{13} + c_{12}\lambda^{12} + c_{11}\lambda^{11} + \dots + c_2\lambda^2 + c_1\lambda + c_0)$$

where the coefficients are summarised in the table below.

Coefficient	Value
$c_0$	43589145600
$c_1$	98967778560
$c_2$	280683736992
$c_3$	24143028888
$c_4$	176667961492
$c_5$	623482838
$c_6$	25589663121
$c_7$	2080068969
$c_8$	1305056346
$c_9$	157314564
$c_{10}$	27370937
$c_{11}$	2416093
$c_{12}$	155112
$c_{13}$	4488

To test the derived polynomial, the 2-(7,3,14) enumeration was performed. Exactly 2,449,592 non-isomorphic 2-(7,3,14) designs were constructed, inducing a total of 12,268,945,580 distinct designs. The formula was then evaluated for  $\lambda = 14$ :

$$D(14) = 12,268,945,580$$

This agreed exactly with the calculated value, and supported the assumption that the formula contained a root at -1. In fact, the initial proposition dictated that the formula must be of degree 14, and so the result of this enumeration confirmed the coefficients.

The following table gives the evaluation of the formula for the number of distinct 2-(7,3, $\lambda$ ) designs, for  $\lambda = 1, 2, \dots, 15$ . Note the calculated number of distinct 2-(7,3,15) designs, given in

bold. The constructive enumeration algorithm induced exactly 27,746,605,182 distinct 2-(7,3,15) designs, which differs from the calculated number by a mere 120.

$\lambda$	Number of Distinct Designs Predicted by the Formula
0	1
1	30
2	465
3	5045
4	41265
5	265792
6	1402140
7	6266415
8	24389705
9	84512555
10	265328404
11	765418845
12	2052124690
13	5160956540
14	12268945580
<b>15</b>	<b>27746605062</b>

### 6.3.2.4 DISCREPANCY

The evaluation of the proposed general formula for  $\lambda = 15$  gave 120 less distinct 2-(7,3,15) designs than were actually induced following the corresponding constructive enumeration and automorphism group calculations. Either the proposed formula was incorrect, or there was a very subtle error with the constructive enumeration.

The results of the constructive enumeration of the 2-(7,3,15) designs are given below. Summarised are the number of non-isomorphic designs calculated, the automorphism group sizes of the designs, and the number of distinct designs induced which is calculated with Burnside's Lemma:

G	Nd	$v!/ G  \times Nd$
5040	1	1
168	3	90
144	1	35
60	3	252
48	1	105
42	2	240
36	3	420
24	46	9660
21	21	5040
20	4	1008
16	9	2835
12	83	34860
10	4	2016
8	134	84420
6	422	354480
4	1266	1595160
3	4866	8174880
2	36292	91455840
1	5485096	27644883840
<b>TOTALS:</b>	<b>5528257</b>	<b>27746605182</b>

Examining the group sizes above indicated that the two designs with group size 42 should be checked. Each of these designs induces exactly 120 distinct designs, and so if these were

isomorphic, it would completely account for the discrepancy. However, these two designs are certainly non-isomorphic, and in fact a calculation of their clique signatures indicates that they have different invariants. In addition, a brute force isomorphism testing algorithm, examining all 5040 possible point mappings, was implemented and no isomorphism was found.

In order to locate the discrepancy, a lot of redundancy was built into the algorithm. In particular, the enumeration was performed a number of times, the automorphism group sizes of the designs were calculated using 3 independent algorithms and the efficient canonicity test was replaced by a brute force method which examined all 5040 point mappings. In each case, however, the same result was calculated.

Finally it was decided that all 5528257 non-isomorphic 2-(7,3,15) designs should be constructed, stored on disc, and processed from the disc. This would have the effect of eliminating the incidence matrix backtracking algorithm from being a possible source of error, and the processing of the constructed designs could be focussed on.

Each 2-(7,3,15) design consists of 105 blocks containing 3 points from the set  $\{1, 2, 3, \dots, 7\}$ . If the complete block list of each design was explicitly stored on disc, storing each point as a 32-bit integer, the entire catalogue would require 6,965,603,820 bytes - well over 6 GBytes. This enormous space requirement motivated the development of an efficient storage method for design catalogues, which is described in detail in the next section. Using the developed techniques, the entire catalogue of non-isomorphic 2-(7,3,15) designs was stored in just 80 MBytes - just 15 bytes per design.

Section 6.3.4 returns to the 2-(7,3,15) discrepancy and overviews the different approaches used to check the result.

### 6.3.3 STORAGE OF LARGE CATALOGUES

In some cases it is desirable to store all non-isomorphic designs of a certain class in a catalogue, which can then be processed in various ways. Except for a small number of cases, space efficiency is an important consideration for the format with which the designs are to be represented in the catalogue.

The running example of this section is the storage of the TTS(12) designs, which have the defining parameters:  $(v, b, r, k, \lambda) = (12, 44, 11, 3, 2)$ . Each design to be stored in the catalogue therefore contains 44 blocks consisting of 3 points from the set  $\{1, 2, \dots, 12\}$ .

To examine the possible optimisations to produce an efficient method of storage, the first five TTS(12)'s which are generated in increasing lexicographical order by the algorithm are given below, represented as block lists:

*Design Number One:*

1 2 3	1 6 7	1 9 10	2 4 6	2 7 9	3 4 10	3 7 9	3 10 11	4 8 11	5 8 10	6 8 10
1 2 3	1 6 8	1 11 12	2 5 9	2 8 11	3 4 12	3 7 11	4 7 10	4 9 12	5 8 12	6 9 11
1 4 5	1 7 8	1 11 12	2 5 11	2 10 12	3 5 6	3 8 9	4 7 11	5 7 10	5 9 11	6 9 12
1 4 5	1 9 10	2 4 6	2 7 8	2 10 12	3 5 6	3 8 12	4 8 9	5 7 12	6 7 12	6 10 11

*Design Number Two:*

1 2 3	1 6 7	1 9 10	2 4 6	2 7 9	3 4 10	3 7 9	3 10 11	4 8 12	5 8 10	6 8 10
1 2 3	1 6 8	1 11 12	2 5 9	2 8 11	3 4 12	3 7 11	4 7 10	4 9 11	5 8 11	6 9 11
1 4 5	1 7 8	1 11 12	2 5 11	2 10 12	3 5 6	3 8 9	4 7 11	5 7 10	5 9 12	6 9 12
1 4 5	1 9 10	2 4 6	2 7 8	2 10 12	3 5 6	3 8 12	4 8 9	5 7 12	6 7 12	6 10 11

*Design Number Three:*

1 2 3	1 6 7	1 9 10	2 4 6	2 7 9	3 4 10	3 7 9	3 10 11	4 8 11	5 8 10	6 8 10
1 2 3	1 6 8	1 11 12	2 5 9	2 8 11	3 4 12	3 7 11	4 7 10	4 9 11	5 8 12	6 9 11
1 4 5	1 7 8	1 11 12	2 5 11	2 10 12	3 5 6	3 8 9	4 7 12	5 7 10	5 9 12	6 9 12
1 4 5	1 9 10	2 4 6	2 7 8	2 10 12	3 5 6	3 8 12	4 8 9	5 7 11	6 7 12	6 10 11

*Design Number Four:*

1 2 3	1 6 7	1 9 10	2 4 6	2 7 9	3 4 10	3 7 9	3 10 11	4 8 11	5 8 10	6 8 10
1 2 3	1 6 8	1 11 12	2 5 9	2 8 11	3 4 12	3 7 11	4 7 10	4 9 11	5 8 12	6 9 12
1 4 5	1 7 8	1 11 12	2 5 11	2 10 12	3 5 6	3 8 9	4 7 12	5 7 10	5 9 11	6 9 12
1 4 5	1 9 10	2 4 6	2 7 8	2 10 12	3 5 6	3 8 12	4 8 9	5 7 12	6 7 11	6 10 11

*Design Number Five:*

1 2 3	1 6 7	1 9 10	2 4 6	2 7 9	3 4 10	3 7 9	3 10 11	4 8 12	5 8 10	6 8 10
1 2 3	1 6 8	1 11 12	2 5 9	2 8 11	3 4 12	3 7 12	4 7 10	4 9 11	5 8 12	6 9 11
1 4 5	1 7 8	1 11 12	2 5 11	2 10 12	3 5 6	3 8 9	4 7 11	5 7 10	5 9 12	6 9 12
1 4 5	1 9 10	2 4 6	2 7 8	2 10 12	3 5 6	3 8 11	4 8 9	5 7 11	6 7 12	6 10 11

**6.3.3.1 DESIGN REPRESENTATIONS**

Storing the entire incidence matrix or block list of each TTS(12) design can be very wasteful if they are not encoded efficiently. It is ideal to store the minimum amount of data which allows each corresponding design to be reconstructed. In general, a good approach is to encode the blocks of the design in some way.

Assume each design is to be stored as its block list, where each point of each block is stored as a 32-bit integer.

Each encoded block list would therefore require:

$$\begin{aligned}
 & B \times k \\
 = & 44 \times 3 \\
 = & 132 \text{ integers} \\
 = & 528 \text{ bytes}
 \end{aligned}$$

Using such a scheme, just under 2000 designs can be stored in each MByte.

There is still much room for improvement here, because the basic data type being used for storage is much larger than necessary. For the TTS(12) designs, every point within each block lies between the value of 1 and 12, and thus can be encoded in just 4 bits. Still assuming that the basic storage type is the 32-bit integer, using this encoding scheme exactly 8 points of the design can be stored in each integer. Therefore each TTS(12) can be stored in:

$$\begin{aligned}
 & B \times k / 8 \\
 = & 44 \times 3 / 8 \\
 = & 16.5 \text{ integers} \\
 = & 66 \text{ bytes.}
 \end{aligned}$$

Thus, 15887 designs can be stored in each MByte.

This method of encoding is still very wasteful though, because every block is explicitly stored as each of its 3 constituent points. A superior scheme is to encode the blocks so that each possible block type is represented by a single identifier.

For example, each block of a TTS(12) consists of 3 points chosen from the set {1, 2, 3, ..., 12}. Therefore there are only  $\binom{12}{3} = 220$  distinct block types. If each block is assigned a value from 1 to 220, called its block identifier (or block I.D.), then a single block can be encoded as its identifier in just 8 bits, or 1 Byte. Therefore a single integer can encode 4 complete blocks, and so only 11 integers are required to store the design. This equates to just 44 Bytes, and so 23831 designs can be stored in each MByte.

This is about as well as a simple approach can do, in which each design is encoded independently. However, the incidence matrix backtracking algorithm produces the designs in lexicographical order, and any given design usually differs only slightly from the previous design. A more sophisticated encoding scheme therefore completely stores the first design generated, but thereafter only stores the changes required to reconstruct the subsequent designs.

The block lists for the first two TTS(12)'s are listed below, and all the blocks in the second design which are different from the blocks in the first design are highlighted:

*Design Number One:*

1 2 3	1 6 7	1 9 10	2 4 6	2 7 9	3 4 10	3 7 9	3 10 11	4 8 11	5 8 10	6 8 10
1 2 3	1 6 8	1 11 12	2 5 9	2 8 11	3 4 12	3 7 11	4 7 10	4 9 12	5 8 12	6 9 11
1 4 5	1 7 8	1 11 12	2 5 11	2 10 12	3 5 6	3 8 9	4 7 11	5 7 10	5 9 11	6 9 12
1 4 5	1 9 10	2 4 6	2 7 8	2 10 12	3 5 6	3 8 12	4 8 9	5 7 12	6 7 12	6 10 11

*Design Number Two:*

1 2 3	1 6 7	1 9 10	2 4 6	2 7 9	3 4 10	3 7 9	3 10 11	<u>4 8 12</u>	5 8 10	6 8 10
1 2 3	1 6 8	1 11 12	2 5 9	2 8 11	3 4 12	3 7 11	4 7 10	<u>4 9 11</u>	<u>5 8 11</u>	6 9 11
1 4 5	1 7 8	1 11 12	2 5 11	2 10 12	3 5 6	3 8 9	4 7 11	5 7 10	<u>5 9 12</u>	6 9 12
1 4 5	1 9 10	2 4 6	2 7 8	2 10 12	3 5 6	3 8 12	4 8 9	5 7 12	6 7 12	6 10 11

There are only four blocks which have changed. To encode this change, the number of blocks which have changed, the positions of the blocks which have changed, and the values of the new blocks must be stored.

If the number of blocks which have changed and the positions of the changed blocks are stored in 1 byte (the same magnitude as the block identifiers themselves), then the above change could be represented in  $(1 + 4 + 4) = 9$  bytes. This method will clearly perform better than if each design is encoded in full, as long as less than half the blocks change.

To estimate the number of TTS(12) designs which could be stored using this encoding, a sample of the first 100,000 non-isomorphic designs was examined. On average, 6.68 blocks changed between successive designs in the sample. If these changes were encoded using the technique presented above, 73020 designs could be stored per MByte.

### 6.3.3.2 OPTIMISATIONS

A number of further optimisations to this approach are possible. For example, not all of the block changes need to be explicitly encoded, because the last block in any design is always forced by the structure of the other blocks, and so can be derived rather than being stored. In fact, often more than one block may be forced, so provided a given design can be reconstructed uniquely, only a subset of the block changes actually need to be stored.

However, calculating the largest subset of blocks which do not need to be stored but which still allow the partial design to be uniquely completed is not trivial, and unless storage is very limited, such optimisations are not necessary.

### 6.3.3.3 IMPLEMENTATION

The efficient cataloguing technique presented in Section 6.3.3.1, was used to store all non-isomorphic 2-(7,3,15) designs on disc. Each block was encoded as a single identifier, and each design was represented as a sequence of block changes. This section briefly discusses the implementation of this cataloguing procedure.

In general, there are  $\binom{v}{k}$  distinct block types for a given block design on  $v$  points with block size  $k$ . To implement the design encoding efficiently, a lookup table is used which specifies the block identifier for each distinct block type.

For example, there are 35 distinct block types for the 2-(7,3, $\lambda$ ) designs, as given below:

{1,2,3}	{1,3,4}	{1,4,6}	{2,3,4}	{2,4,6}	{3,4,5}	{3,6,7}
{1,2,4}	{1,3,5}	{1,4,7}	{2,3,5}	{2,4,7}	{3,4,6}	{4,5,6}
{1,2,5}	{1,3,6}	{1,5,6}	{2,3,6}	{2,5,6}	{3,4,7}	{4,5,7}
{1,2,6}	{1,3,7}	{1,5,7}	{2,3,7}	{2,5,7}	{3,5,6}	{4,6,7}
{1,2,7}	{1,4,5}	{1,6,7}	{2,4,5}	{2,6,7}	{3,5,7}	{5,6,7}



Each block in lexicographical order is given a unique identifier, and stored in a table such as the one below:

Block identifiers

1	2	3	4	5	6	7	8	9	.....	.....	34	35
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	3	3	3	4	4	4	4
3	4	5	6	7	4	5	6	7	5	6	7	6

Any given block can be encoded by finding the position within the table of its entry, which corresponds to its unique identifier. Similarly, any identifier can be decoded to give its corresponding block, using the same table, simply by locating the given index of the table.

Whether the search is performed in one piece and a single list is constructed, or whether it is parallelised and a number of lists are built, only the first design in each list has every one of its blocks encoded and stored. All other designs in the list have only their block changes encoded.

As an example, assume that design N has been generated and is to be stored on disc, and that it differs from design N-1, the last design stored, by exactly 4 blocks. Let  $p_1$ ,  $p_2$ ,  $p_3$  and  $p_4$  denote the positions of the blocks which have changed, and let  $b_1$ ,  $b_2$ ,  $b_3$  and  $b_4$  respectively denote the new block identifiers which should be stored in each position. Design N is then encoded by storing the number of blocks that have changed, the positions of each changed block, and the new block identifiers, as represented by the following sequence:

4,  $p_1$ ,  $b_1$ ,  $p_2$ ,  $b_2$ ,  $p_3$ ,  $b_3$ ,  $p_4$ ,  $b_4$ .

Each of these values occupies a single byte, thus design N can be stored in just 9 bytes.

### 6.3.3.4 BUFFERING

Writing each design directly to disc as soon as it is encoded will result in poor performance, as disc access tends to be slow. It becomes necessary to store the designs in memory as they are encoded, and store them more permanently to disc later.

If there are only a small number of designs, they can all be stored in memory as the program is running, and then all written to disc once the program finishes and the catalogue is complete. However, if there are a large number of designs, then a great deal of memory will be necessary for such an approach. To overcome these potential drawbacks, a buffer is used which stores a specified number of designs in memory, and flushes them out to disc each time the buffer becomes full. Depending on the designs being catalogued, a suitable buffer size can be selected which minimises both memory requirements and the number of disc operations.

### 6.3.3.5 FILE PROCESSING

Any catalogue of designs constructed using this method can be processed by first reading the initial design off the disc and decoding each of its block identifiers. Every subsequent design is then reconstructed by reading its sequence of block changes from the catalogue, and updating the current design accordingly.

Again, buffering the file processing is an advantage as it reduces the number of disc operations required. The catalogue is read into a memory buffer a piece at a time, and is processed quickly from the memory buffer. When the buffer becomes empty, it is refreshed by reading the next piece of the catalogue from the disc.

### 6.3.4 DISCREPANCY RESOLUTION

This section describes the techniques used to resolve the discrepancy between the number of designs computed by the proposed formula, and the number of designs induced from the result of the constructive enumeration. In order to eliminate the incidence matrix backtracking algorithm from being a possible source of error, all 5,528,257 non-isomorphic 2-(7,3,15) designs were constructed and stored on disc, using the cataloguing techniques discussed in the previous section. The complete catalogue was constructed in 36 pieces, and occupied exactly 83,671,141 bytes. On average, each design was stored in approximately 15 bytes.

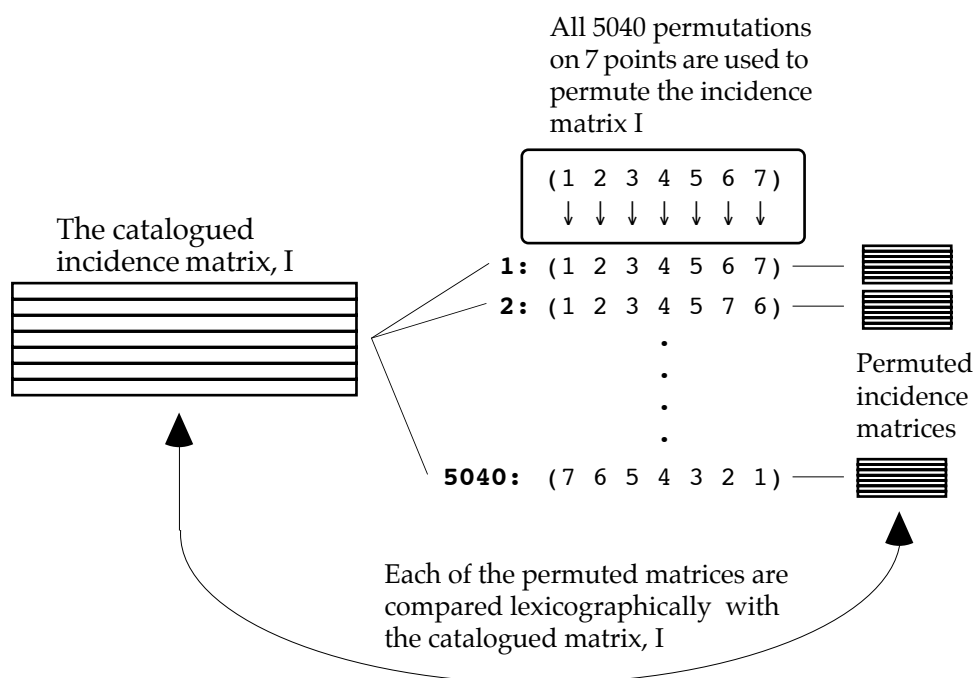
The designs were then processed off the catalogue, and again exactly 27746605182 distinct designs were induced - 120 more than the 27746605062 predicted by the formula.

If the number of distinct designs computed by the formula was larger than the number induced by the results of the constructive enumeration, this would indicate that a design may be missing from the catalogue. However, because this is not the case, if an error exists in the catalogue, it must belong to one of the following 4 categories, and should therefore be straightforward to locate:

- 1) There is a design in the catalogue which is not a valid 2-(7,3,15) design.
- 2) There is more than one copy of exactly the same design in the catalogue - in other words the catalogued designs are not all distinct.
- 3) There is a design in the catalogue which is not in canonical form, and therefore is isomorphic to some other design in the catalogue.
- 4) The automorphism group size of a design has been calculated incorrectly.

Each of these conditions were thoroughly checked using the techniques described below:

- 1) Every design processed off the catalogue was explicitly checked for validity. It is required that the incidence matrix of each design must contain exactly 3 points in every column and 45 points in every row, and the intersection between every pair of rows must be exactly 15. This test was successful for every design in the catalogue.
- 2) Every design processed off the catalogue was compared lexicographically to the previous design processed. The purpose of this check was to demonstrate that no two designs in the catalogue were equal, by verifying that every design in the catalogue was strictly lexicographically greater than the previous design. This test was also successful, and along with the result of the first test, this confirmed that the catalogue contained 5,528,257 valid, distinct 2-(7,3,15) designs.
- 3) The efficient canonicity test was replaced by a brute force permutation tester. All 5040 permutations on 7 points were used to permute each catalogued incidence matrix,  $I$ , and all 5040 permuted matrices were then compared lexicographically to  $I$ . The diagram on the next page gives a schematic overview of this test:



In every case, the incidence matrices in the catalogue were lexicographically less than or equal to each of the permuted designs. Therefore, each catalogued design must be the lexicographically smallest design in its isomorphism class, or equivalently, in canonical form.

- 4) The final condition to be checked was the calculation of the automorphism group of each design. The brute force permutation tester, which was implemented to perform check 3 above, examines all possible permutations, and is therefore guaranteed to encounter all those which map the catalogued design onto itself. The set of all such permutations is the automorphism group of the design, and the group size is calculated simply by counting these permutations.

In addition, two other independent methods of automorphism group generation were performed. Both methods explicitly constructed all automorphisms of the design using a backtracking approach. The first method calculated the group using the centraliser subgroup technique, as described in Section 4.5.3, and extended each subgroup every time a new generator automorphism was found. The other method performed an exhaustive search and generated each automorphism by backtracking, using partitions of the blocks, points and point pairs to reduce the search.

For every catalogued design, all three techniques agreed perfectly. In addition, the resulting group sizes computed by processing the catalogue off disc were exactly the same as computed before, when the incidence matrix backtracking algorithm was used, and thus exactly the same number of distinct designs were induced.

In summary, the 5,528,257 2-(7,3,15) designs generated by the constructive enumeration algorithm were stored in a catalogue on disc. A series of tests were performed, which verified that every one of the 5,528,257 catalogued designs:

- were valid 2-(7,3,15) designs
- were distinct designs, with no repetitions
- were in canonical form
- had their automorphism group sizes calculated correctly

### 6.3.5 RESULTS

The results of the catalogue processing agreed exactly with the results of the enumerations, which had been performed a number of times. In addition, Brendan McKay [51] was able to independently confirm that there are indeed 27,746,605,182 distinct 2-(7,3,15) designs.

In conclusion, this work showed that the proposed formula for computing the number of distinct 2-(7,3, $\lambda$ ) designs was incorrect, and thus these numbers are not predicted by a polynomial of degree 15.

The algorithm developed in this thesis was also used to enumerate all the 2-(7,3,16) designs, and this result was again confirmed by McKay [51].

## 6.4 CASE STUDY FOUR: THE 2-(7,3, $\lambda$ ) $\rightarrow$ 3-(8,4, $\lambda$ ) DESIGN EXTENSION

### 6.4.1 BACKGROUND

In general, constructive enumeration is limited by the magnitude of the class of designs which are to be generated. For example, in [45], a lower bound on the number of non-isomorphic 2-(133, 399, 36, 12, 3) designs is given as  $10^{208}$ . The constructive enumeration of any class of designs of such magnitude is clearly not possible.

However, it is often possible to define techniques for extending smaller designs to larger designs. These approaches may lead to the construction of previously unknown designs, or in some cases even to a complete enumeration of the larger class. A good example of this is the work by McKay and Radziszowski [49], in which they prove the non-existence of 4-(12,6,6) designs, which was the last open existence question in design theory for  $v \leq 12$ . Their proof makes use of their enumeration of the smaller 3-(10,4,3) and 4-(11,5,3) designs, without repeated blocks.

This case study examines one such extension, from the 2-(7,3, $\lambda$ ) designs to the 3-(8,4, $\lambda$ ) designs, and an efficient method of isomorphism testing for the larger designs is presented. It is shown that the number of distinct 3-(8,4, $\lambda$ ) designs equals the number of distinct 2-(7,3, $\lambda$ ) designs, for every value of  $\lambda$ , and this result is verified for  $1 \leq \lambda \leq 14$  using a modification to the constructive enumeration algorithm.

The 3-(8,4, $\lambda$ ) designs can in fact be enumerated independently using a simple variation of the current enumeration algorithm (see Case Study Eight), however it can be performed significantly faster using the extension techniques discussed in this case study rather than via a direct enumeration approach.

In any case, the techniques developed in this case study can be applied to other classes of designs for which direct enumeration may not be possible. In [2], Alltop showed that if  $t$  is even then any  $t$ -( $2k+1$ ,  $k$ ,  $\lambda$ ) design extends to a  $(t+1)$ -( $2k+2$ ,  $k+1$ ,  $\lambda$ ) design, using exactly the extension technique detailed in Section 6.4.2.

This case study is divided into three main sections:

- 6.4.2 Generating 3-(8,4, $\lambda$ ) Designs From 2-(7,3, $\lambda$ ) Designs
- 6.4.3 Testing the Constructed 3-(8,4, $\lambda$ ) Designs for Isomorphism
- 6.4.4 Results

Section 6.4.2 describes in detail the process of extension from the 2-(7,3, $\lambda$ ) designs to the 3-(8,4, $\lambda$ ) designs. It is proven that the extension indeed generates valid 3-(8,4, $\lambda$ ) designs, that every distinct 2-(7,3, $\lambda$ ) design extends to a distinct 3-(8,4, $\lambda$ ) design and that two isomorphic 2-(7,3, $\lambda$ ) designs produce isomorphic extensions.

Section 6.4.3 outlines an efficient method of isomorphism testing for the constructed 3-(8,4,λ) designs, which makes use of a special canonicity test. This test is not straightforward because the extensions are not generated in lexicographical order, and the lexicographically smallest design in a given isomorphism class may not even be generated as an extension.

The final section, 6.4.4, discusses the results of the case study.

### 6.4.2 GENERATING 3-(8,4,λ) DESIGNS FROM 2-(7,3,λ) DESIGNS

This section describes in detail the method of extension from a given 2-(7,3,λ) design to a 3-(8,4,λ) design.

The table below compares the two designs, with their parameters expressed in terms of λ:

Parameter	2-(7,3,λ)	3-(8,4,λ)
v	7	8
b	7λ	14λ
r	3λ	7λ
k	3	4
λ	λ	λ

From the table it can be seen that there are exactly twice as many blocks in a 3-(8,4,λ) design as there are in a 2-(7,3,λ) design. The extension takes place in two distinct stages, each of which produces exactly half of the blocks of the 3-(8,4,λ) design. The first stage generates the complement of the 2-(7,3,λ) design, and the second stage appends a new point to the design. The details of each stage are described below.

Following the descriptions of these two stages, three important observations are presented, from which it is concluded that the extensions to the set of all non-isomorphic 2-(7,3,λ) designs will contain at least one representative from each isomorphism class of the corresponding 3-(8,4,λ) designs. It then remains to test such a set of extensions for isomorphism, which is the subject of Section 6.4.3.

#### Stage One: Complementary Design Construction

This stage defines a *complementary design*, and describes the straightforward process of generating one. It shows that the complement of each 2-(7,3,λ) design is a 2-(7,4,2λ) design, and that there are exactly the same number of distinct and non-isomorphic 2-(7,3,λ) and 2-(7,4,2λ) designs for a given value of λ.

For any collection, B, of k-element subsets of a v-element set X, the *complement* of B, or COMP(B) is defined as:

$$\text{COMP}(B) = \{X \setminus b : b \in B\}$$

For example, the complement of the block {1,2,3} on the point set {1,2,...,7} is the block {4,5,6,7}.

Clearly, the complementary design of a 2-(7,3,λ) design will contain blocks of size 4, from the point set {1,2,...,7}. In fact, the complementary design of a 2-(7,3,λ) design is a 2-(7,4,2λ) design, as given by the following theorem:

Theorem 3.6 of [40] states:

If (X,B) is a t-(v,k,λ) design and S is any s-element subset of X with 0 ≤ s ≤ t, then the number of blocks that do not contain any point of S is:

$$b(s) = |\{b \in B : b \cap S = \emptyset\}| = \lambda \binom{v-s}{k} / \binom{v-t}{k-t}$$

In particular, (X, COMP(B)) is a t-(v, v-k, b(t)) design.

For the  $2-(7,3,\lambda)$  designs,  $b(t) = 2\lambda$ , and so the complementary design is a  $2-(7,4,2\lambda)$  design.

It is obvious that there are the same number of distinct  $2-(7,3,\lambda)$  and  $2-(7,4,2\lambda)$  designs for a given value of  $\lambda$ , as every block of a  $2-(7,3,\lambda)$  design maps to a unique block of a  $2-(7,4,2\lambda)$  design under the process of complementation.

Any automorphism of a  $2-(7,3,\lambda)$  design must also be an automorphism of its corresponding complementary  $2-(7,4,2\lambda)$  design, and so there must be exactly the same number of isomorphism classes of each type of design. In other words, there are the same number of non-isomorphic  $2-(7,3,\lambda)$  and  $2-(7,4,2\lambda)$  designs for a given value of  $\lambda$ . In addition, the complementary designs of any two isomorphic  $2-(7,3,\lambda)$  designs must themselves be isomorphic.

### Stage Two: Adding the New Point

This stage outlines the construction process of the other half of the blocks of a  $3-(8,4,\lambda)$  design, in which the new point is added to the point set. It also proves that this other half is in fact precisely the blocks of the original  $2-(7,3,\lambda)$  design with the new point appended.

A  $2-(7,4,2\lambda)$  design, which is complementary to a given  $2-(7,3,\lambda)$  design, contains exactly  $7\lambda$  blocks, and so another  $7\lambda$  blocks are required to form a  $3-(8,4,\lambda)$  design. The  $2-(7,4,2\lambda)$  design has block size 4, and so each block contains exactly 4 unordered triples. For example, the block  $\{a, b, c, d\}$  contains the unordered triples:

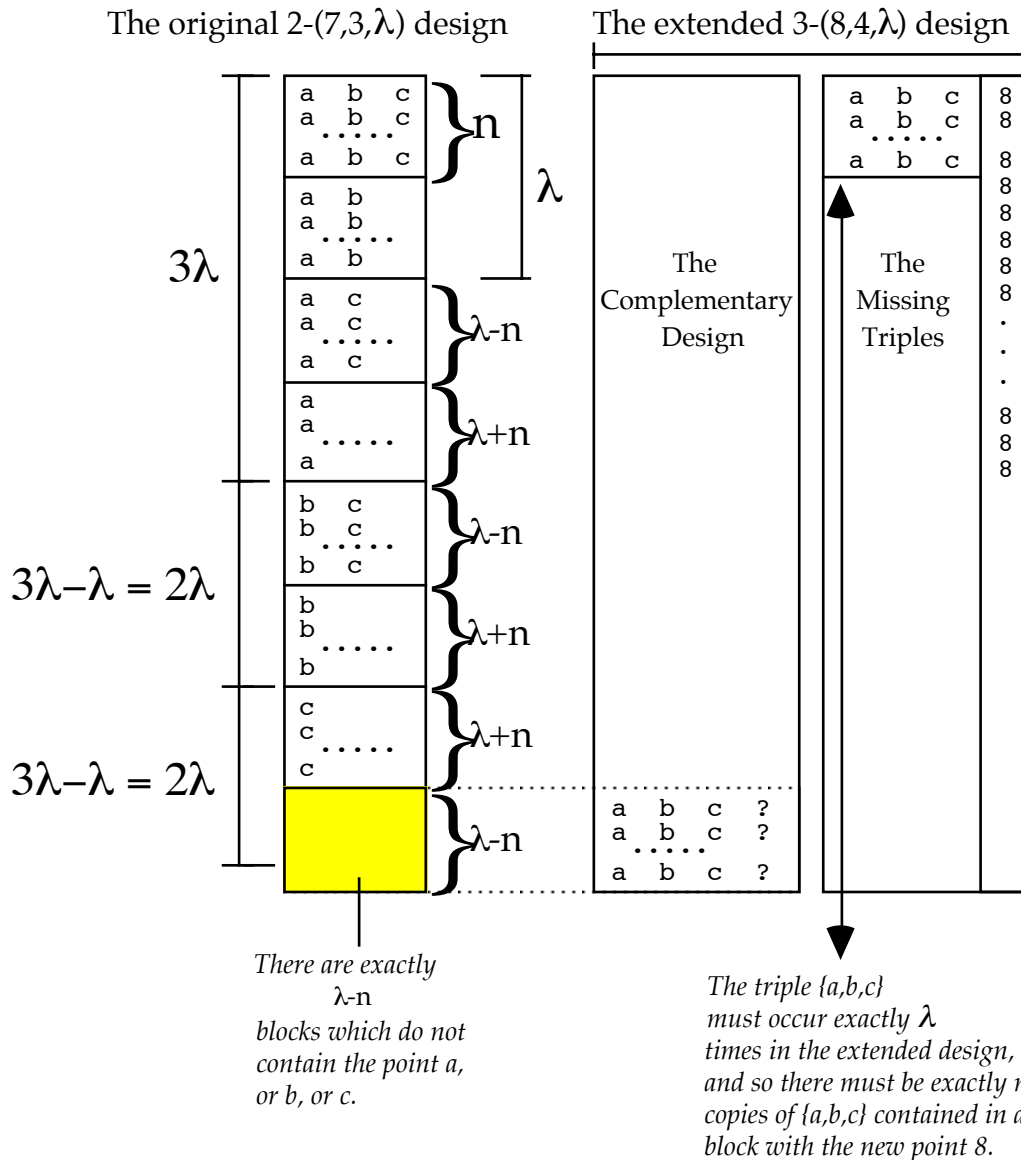
$$\{a, b, c\}, \{a, b, d\}, \{a, c, d\} \text{ and } \{b, c, d\}.$$

Each  $2-(7,4,2\lambda)$  design therefore covers exactly  $28\lambda$  triples. However, there are  $\binom{7}{3} = 35$  possible unordered triples on 7 points, and each one must occur  $\lambda$  times in a valid  $3-(8,4,\lambda)$  design. In other words, there are  $7\lambda$  missing triples which must be covered before a valid  $3-(8,4,\lambda)$  design can be constructed. The replication number of a  $3-(8,4,\lambda)$  design is  $7\lambda$ , and so if each of these  $7\lambda$  triples have the point "8" added to them, not only will the missing triples be covered, but the point "8" will occur in the design the correct number of times.

To summarise the method of extension, the  $2-(7,3,\lambda)$  design has each block complemented to generate a  $2-(7,4,2\lambda)$  design. To this design, the  $7\lambda$  uncovered triples with the new point 8 appended, are added. The resulting design has  $14\lambda$  blocks, and every triple is covered exactly  $\lambda$  times.

For every  $3-(8,4,\lambda)$  design constructed in this way, the blocks containing the point "8" have a special relationship to the original  $2-(7,3,\lambda)$  design. In fact, if each of these  $7\lambda$  blocks have the point "8" removed, the resulting design is exactly the original  $2-(7,3,\lambda)$  design.

This can be proven by showing how any arbitrary block,  $\{a,b,c\}$ , occurring exactly  $n$  times in a given  $2-(7,3,\lambda)$  design, must also occur exactly  $n$  times as a triple in those blocks containing the point 8 of the extended  $3-(8,4,\lambda)$  design. The diagram on the following page illustrates this situation:



As the diagram illustrates, if the block {a,b,c} occurs exactly n times in a 2-(7,3,λ) design, D, then there will be exactly λ-n blocks in D which do not contain the points a, or b, or c. Once the complementary design is constructed, there will be exactly λ-n blocks which contain the triple {a,b,c}. Every triple must be covered exactly λ times, and so the 3-(8,4,λ) design must contain another n copies of the triple {a,b,c} - which will come from those blocks containing the point "8".

Therefore, those triples which are formed by removing point "8" from every block of the constructed 3-(8,4,λ) design which contains this point, are exactly the blocks of the original 2-(7,3,λ) design.

The extension process for a given 2-(7,3,λ) design has now been covered in detail. Next, three important observations are presented which provide a means for enumerating the 3-(8,4,λ) designs.

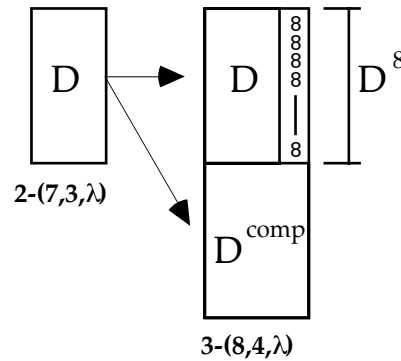
**ObservationOne**

It is observed that any two distinct 2-(7,3,λ) designs will yield distinct 3-(8,4,λ) designs as a result of the extension process.

Having shown that all the blocks in the extended 3-(8,4,λ) design containing the point "8" are exactly the blocks of the original 2-(7,3,λ) design, the method of extension can be simplified to the following steps.

- Let  $D$  be the  $2-(7,3,\lambda)$  design which is to be extended.
- Generate  $D^{\text{comp}}$  by complementing each block of  $D$  with respect to the point set  $\{1,2,3,\dots,7\}$
- Generate  $D^8$  by adding the point "8" to every block of  $D$
- $D^{\text{comp}} \cup D^8$  is the extended  $3-(8,4,\lambda)$  design

This extension process is summarised in the diagram below:



A simple, yet important, observation can now be made. Every distinct  $2-(7,3,\lambda)$  design will produce a distinct  $3-(8,4,\lambda)$  design. Consider two  $2-(7,3,\lambda)$  designs, say  $A$  and  $B$ , which are distinct (but not necessarily non-isomorphic). Consider their extensions,  $A'$  and  $B'$  respectively. All the blocks in the  $3-(8,4,\lambda)$  design  $A'$  which contain the point "8" will certainly be different from all the blocks in the design  $B'$  which also contain the point "8", because these block come directly from the designs  $A$  and  $B$ , and therefore the designs  $A'$  and  $B'$  must also be distinct.

Observation Two, which is presented shortly, explains how every  $3-(8,4,\lambda)$  design can be decomposed to give a unique  $2-(7,3,\lambda)$  design. This result, together with the result of Observation One, implies the number of distinct  $2-(7,3,\lambda)$  designs equals the number of distinct  $3-(8,4,\lambda)$  designs.

An example of the extension process is given in the table below. On the left is a  $2-(7,3,2)$  design, with an automorphism group of size 168, and on the right is the  $3-(8,4,2)$  design generated as the extension of this design, with a group of size 1344. It is clear how the blocks of the extended design have been constructed:

<u>2-(7,3,2)</u>  G  = 168	<u>3-(8,4,2)</u>  G  = 1344
1 2 3	1 2 3 8
1 2 3	1 2 3 8
1 4 5	1 4 5 8
1 4 5	1 4 5 8
1 6 7	1 6 7 8
1 6 7	1 6 7 8
2 4 6	2 4 6 8
2 4 6	2 4 6 8
2 5 7	2 5 7 8
2 5 7	2 5 7 8
3 4 7	3 4 7 8
3 4 7	3 4 7 8
3 5 6	3 5 6 8
3 5 6	3 5 6 8
	4 5 6 7
	4 5 6 7
	2 3 6 7
	2 3 6 7
	2 3 4 5
	2 3 4 5
	1 3 5 7
	1 3 5 7
	1 3 4 6
	1 3 4 6
	1 2 5 6
	1 2 5 6
	1 2 4 7
	1 2 4 7



Note that the number of distinct designs admitted by each of the above structures is exactly 30 in both cases. The 2-(7,3,2) design induces  $7!/168=30$  distinct designs and the 3-(8,4,2) design also induces  $8!/1344=30$  distinct designs.

**Observation Two**

It is observed that every 3-(8,4, $\lambda$ ) design decomposes in a unique way, by the reverse of the extension process, to give a 2-(7,3, $\lambda$ ) design. This result guarantees that all 3-(8,4, $\lambda$ ) designs can be generated by the extension process.

Take any arbitrary 3-(8,4, $\lambda$ ) which has not necessarily been constructed by the described method of extension. The following properties hold:

- Every unordered triple occurs in exactly  $\lambda$  blocks.
- Every block containing points a and b, say {a,b,c,d} contribute exactly two triples containing the pair {a,b}, namely {a,b,c} and {a,b,d}.
- Any pair {a,b} is contained in 6 distinct triples of the form {a,b,x} for  $x \in \{1,2,3,\dots,8\}$ ,  $x \neq a,b$ , of which the design contains  $\lambda$  copies.

Therefore, every distinct pair, {a,b} occurs in exactly  $6\lambda/2=3\lambda$  blocks of the design.

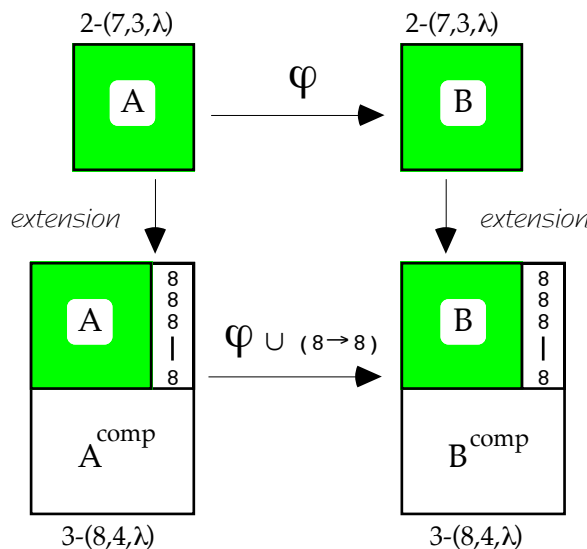
Every triple occurs in  $\lambda$  blocks, and so every triple of the form {a,b,8}, where  $a,b \in \{1,2,\dots,7\}$ ,  $a \neq b$ , must also occur in  $\lambda$  blocks. In other words, every unordered pair, {a,b}, for  $a,b \in \{1,2,3,\dots,7\}$ , occurs exactly  $\lambda$  times in the blocks which contain the point "8". Therefore, the set of all blocks containing point "8", but with point "8" removed, cover every unordered pair exactly  $\lambda$  times, and so form a 2-(7,3, $\lambda$ ) design.

The described method of extension therefore creates a 1-1 mapping from the set of all distinct 2-(7,3, $\lambda$ ) designs to the set of all distinct 3-(8,4, $\lambda$ ) designs. Any arbitrary 2-(7,3, $\lambda$ ) design has a unique extension to a 3-(8,4, $\lambda$ ) design, and conversely, any arbitrary 3-(8,4, $\lambda$ ) design has a unique decomposition to a 2-(7,3, $\lambda$ ) design by the reverse process.

**Observation Three**

Finally, it is observed that any two isomorphic 2-(7,3, $\lambda$ ) designs extend to produce two isomorphic 3-(8,4, $\lambda$ ) designs. This is an important result, for it shows that in order to enumerate all non-isomorphic 3-(8,4, $\lambda$ ) designs, it is not necessary to extend every distinct 2-(7,3, $\lambda$ ) design and test the extensions for isomorphism. It is sufficient to extend a set of 2-(7,3, $\lambda$ ) designs which only contain a single representative from each isomorphism class. Such a set can be efficiently constructed by the enumeration algorithm developed in this thesis.

Consider the following diagram, in which A and B are isomorphic 2-(7,3, $\lambda$ ) designs which are extended to give two 3-(8,4, $\lambda$ ) designs:



Let  $\varphi$  be the isomorphism between designs A and B. In other words,  $\varphi$  maps all the blocks of design A onto the blocks of design B.

Now consider the effect of the mapping  $\varphi \cup (8 \rightarrow 8)$  on the extended 3-(8,4, $\lambda$ ) designs. All the blocks of A's extension which contain the point "8" will certainly map onto the blocks of B's extension which contain the point "8", because these blocks consist of the blocks of the original designs, and point "8" is fixed by the isomorphism. It has already been shown, in the description of Stage One of the construction process, that if A is isomorphic to B then  $A^{\text{comp}}$  must be isomorphic to  $B^{\text{comp}}$ , and so the mapping  $\varphi$  will map  $A^{\text{comp}}$  onto  $B^{\text{comp}}$ . This completes the proof.

Therefore, because isomorphic 2-(7,3, $\lambda$ ) designs generate isomorphic extensions, only non-isomorphic 2-(7,3, $\lambda$ ) designs need to be considered during the extension process. It may be the case that two non-isomorphic 2-(7,3, $\lambda$ ) designs generate isomorphic extensions however, and so each extension must still be tested for isomorphism. The canonicity test developed for this purpose is described in the next section.

### 6.4.3 TESTING THE CONSTRUCTED 3-(8,4, $\lambda$ ) DESIGNS FOR ISOMORPHISM

The previous section detailed how each non-isomorphic 2-(7,3, $\lambda$ ) design can be extended to a unique 3-(8,4, $\lambda$ ) design and stated that the number of distinct 2-(7,3, $\lambda$ ) designs is equal to the number of distinct 3-(8,4, $\lambda$ ) designs for all  $\lambda$ .

In general however, there are fewer non-isomorphic 3-(8,4, $\lambda$ ) designs than there are non-isomorphic 2-(7,3, $\lambda$ ) designs. The fact that the number of distinct designs are equal indicates that the 3-(8,4, $\lambda$ ) designs tend to induce larger numbers of distinct designs, which is to be expected as they admit a greater number of possible point relabellings.

The constructive enumeration algorithm can easily be modified to produce the unique extension for each non-isomorphic 2-(7,3, $\lambda$ ) design. To classify only the non-isomorphic 3-(8,4, $\lambda$ ) designs however, some facility is required for testing all extended designs for isomorphism.

A list building approach will work correctly, but becomes very inefficient when the number of designs is large, and is difficult to parallelise. A better approach is therefore to define some canonical form and use a canonicity test in which each constructed design can be tested independently of the other designs.

A straightforward canonicity test such as the one used to classify the 2-(7,3, $\lambda$ ) designs which are then extended, does not work for obvious reasons. The 2-(7,3, $\lambda$ ) designs are generated in strictly increasing lexicographical order, and the canonicity test classifies only those designs which are the lexicographically smallest in their isomorphism classes, or in other words which are in canonical form. Unfortunately, the extended 3-(8,4, $\lambda$ ) designs are not constructed in lexicographical order. As a result, for a particular isomorphism class of the extended designs, the canonical representative of the class may not even be constructed at all. For example, it may be the case that a 3-(8,4, $\lambda$ ) design is constructed which is the lexicographically largest design in its isomorphism class, and if no other extension is constructed during the enumeration which is isomorphic to it, this design must be classified.

Clearly, a new ordering must be defined for the extensions, from which their order of construction can be easily determined. With such an ordering, it can be calculated for each extension whether or not it is isomorphic to any other extension generated at some point earlier in the construction. Only those extensions which are the first ones generated in their isomorphism class are classified, and all other extensions in the same class are rejected. In fact, this is similar to the current canonicity test because when the design construction is

ordered, the first design constructed in each isomorphism class is in fact the lexicographically smallest one in the class.

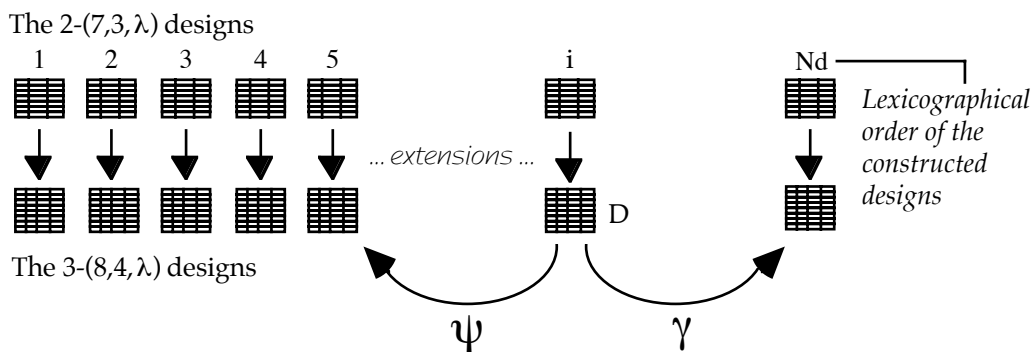
Given any two  $3-(8,4,\lambda)$  designs, say A and B, the relative order in which they are constructed by the extension algorithm must be determined. This can be answered by pulling out the derived  $2-(7,3,\lambda)$  designs from A and B, and comparing these lexicographically. Let A' and B' be the  $2-(7,3,\lambda)$  designs which are derived from A and B respectively, consisting of the set of blocks containing the point "8", but with this point removed. These derived designs are constructed using the reverse of the extension process, and the derived design formed in this way for any  $3-(8,4,\lambda)$  design is the unique  $2-(7,3,\lambda)$  design from which it was extended.

The  $2-(7,3,\lambda)$  designs are generated in strictly increasing lexicographical order, so if A' is lexicographically less than B' then the  $3-(8,4,\lambda)$  design A must have been generated before the  $3-(8,4,\lambda)$  design B. Therefore, lexicographical comparison of the derived designs imposes an order on the extended  $3-(8,4,\lambda)$  designs.

A brute force canonicity test for the  $3-(8,4,\lambda)$  designs is therefore as follows:

- Let D be the  $3-(8,4,\lambda)$  design which is to be tested for canonicity. It will be in canonical form if it is the first design generated in the extension algorithm belonging to its isomorphism class.
- Generate the  $2-(7,3,\lambda)$  design D', from which design D was extended, by the reverse of the extension process. It consists of all the blocks of D which contain the point 8 - but with point 8 removed.
- Perform all  $8! = 40,320$  point mappings to design D to generate the set  $D^{iso}$  of all permuted incidence matrices representing the  $3-(8,4,\lambda)$  designs to which D is isomorphic.
- For every permuted incidence matrix in the set  $D^{iso}$ , the derived  $2-(7,3,\lambda)$  design from which it would have been extended is constructed. If any of these derived designs are lexicographically less than D', then design D is isomorphic to a  $3-(8,4,\lambda)$  design with a lexicographically smaller derived design, which would have been constructed earlier in the enumeration. Design D is therefore not the first design generated in its isomorphism class and is rejected. Otherwise, if none of the derived designs are lexicographically less than D', then design D is classified.

For example, the diagram below illustrates the constructive enumeration, and subsequent extension, of the non-isomorphic  $2-(7,3,\lambda)$  designs. These designs are represented on the top row of the diagram, and are labelled in the order of construction from 1 to Nd.



Consider the  $2-(7,3,\lambda)$  design  $i$ , which extends to the  $3-(8,4,\lambda)$  design  $D$ , labelled in the diagram. The lexicographical ordering of the  $3-(8,4,\lambda)$  designs themselves is unimportant. Design  $D$  is constructed earlier than all other  $3-(8,4,\lambda)$  designs which have lexicographically larger derived designs, and is constructed after all those  $3-(8,4,\lambda)$  designs which have lexicographically smaller derived designs. Therefore, if design  $D$  is

isomorphic to any design which has a lexicographically smaller derived design, it is not classified. In the diagram, the mapping  $\psi$  represents such an isomorphism.

Otherwise,  $D$  must be isomorphic to only those  $3-(8,4,\lambda)$  designs which have lexicographically larger derived designs. Isomorphisms of this type are represented in the diagram as  $\gamma$ .  $D$  is therefore the first design to be constructed in its isomorphism class and so must be classified.

### Optimising the Canonicity Test

Just as with the ordinary canonicity test, which was used to classify the constructed  $2-(7,3,\lambda)$  designs, not all  $v!$  point permutations need to be examined.

In order to implement the optimisation, an initial adjustment must be made regarding the process for constructing the derived designs. Rather than removing point "8" from the permuted  $3-(8,4,\lambda)$  designs, the derived designs are constructed by removing the point "1".

This is a perfectly valid modification, and as before a  $3-(8,4,\lambda)$  design,  $D$ , is classified if and only if its derived design is lexicographically less than the derived designs of all isomorphs of  $D$ . Following are reasons for the validity and purpose of this modification.

Let  $D$  be the design being tested for canonicity. Its derived design,  $D'$ , is currently constructed by removing point "8" from each block containing this point. Let  $\emptyset$  be an isomorphism which maps design  $D$  to another  $3-(8,4,\lambda)$  design, say  $P$ , which has its derived design  $P'$  constructed by removing point "8" from each block of  $P$  containing it. If  $P'$  is lexicographically less than  $D'$ , then  $P$  must have been generated prior to  $D$ , and so  $D$  must be rejected. Now if point "1" is to be removed from the permuted designs rather than point "8", the test will behave in exactly the same way. Let  $\gamma$  be the point mapping constructed by taking the mapping  $\emptyset$  and swapping the sources of points "1" and "8". As an example, if  $\emptyset$  was the mapping:

a	b	c	d	e	f	g	h
↓	↓	↓	↓	↓	↓	↓	↓
1	2	3	4	5	6	7	8

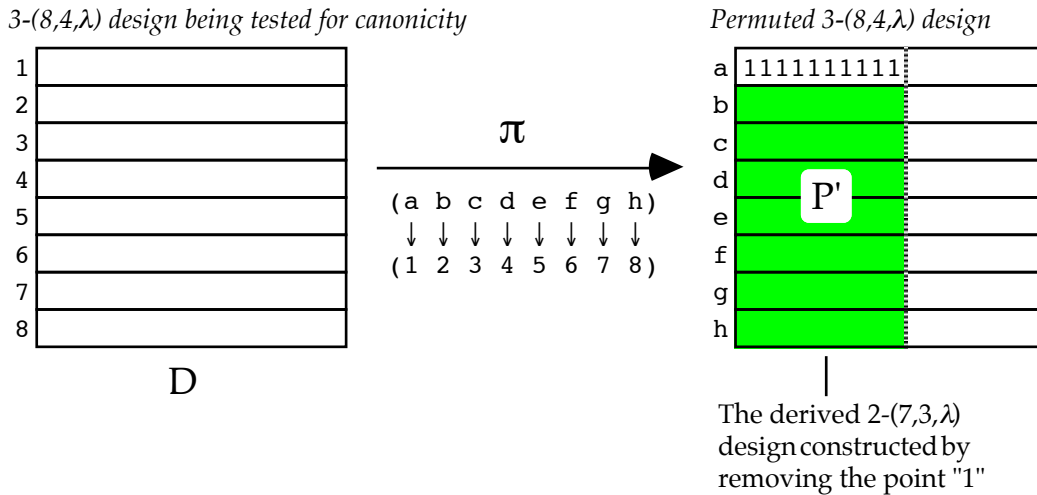
then  $\gamma$  becomes the mapping:

h	b	c	d	e	f	g	a
↓	↓	↓	↓	↓	↓	↓	↓
1	2	3	4	5	6	7	8

Then  $\gamma$  will map design  $D$  to a new design, say  $Q$ , and the derived design of  $Q$  formed by removing point "1" will be exactly the same as the derived design of  $P$  formed by removing point "8".

As long as all possible permutations are tested, the point which is removed from the permuted designs to form the derived designs is not important. All the corresponding derived designs will still be computed correctly, it is only the order in which they are examined that will change.

If point 1 is selected to be removed from the permuted  $3-(8,4,\lambda)$  designs to form the derived designs, the optimisation to the canonicity test becomes straightforward. Every permuted design has its columns ordered and this ordering sorts the 1's to the left hand side and the 0's to the right hand side of the first row. Therefore, once a given permuted design is ordered, the leftmost  $r$  columns (where  $r$  is the replication number of the design) will all contain the point 1. The incidence matrix of the derived design, formed by removing point 1, is therefore precisely the first  $r$  columns and rows 2, 3, ..., 8 of the permuted incidence matrix. The diagram on the next page illustrates this point, for a given  $3-(8,4,\lambda)$  design.



In the diagram above, design D is the design which is being tested for canonicity. Let D' be the derived design of D, which is formed by removing the point 8.

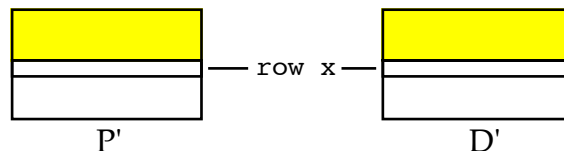
Design D is permuted by the mapping π, and the permuted incidence matrix has its columns and its derived design, P', constructed by removing the point "1". Now P' is compared lexicographically to D'. There are exactly 3 possibilities:

*P' is lexicographically less than D'*

In this case, design D is isomorphic to another 3-(8,4,λ) design which has a smaller derived design, and hence must have been generated before design D. Design D is therefore rejected as soon as this is detected.

*P' is lexicographically greater than D'*

In this case, design D is isomorphic to another 3-(8,4,λ) design which was generated after design D, so it cannot be rejected at this stage and further permutations must be tested. However, a large number of permutations can be eliminated from consideration.



Let x denote the first row at which the incidence matrices of P' and D' differ. Therefore, P' and D' are identical on the first x-1 rows, and row x of P' is larger than row x of D'. Any isomorphism which maintains the current point mappings of the first x points will map design D to a new permuted design, say Q, and the derived design of Q, say Q', will have an incidence matrix which is identical to P' on the first x rows. Q' will again be lexicographically larger than D'.

Therefore, no further permutations of design D need to be examined in which the first x point mappings do not change. Especially if x is quite small, this optimisation results in considerable improvement.

*P' and D' are equal*

In this case, design D is isomorphic to another 3-(8,4,λ) design which has an identical derived design. D can not be rejected at this stage, and further mappings must be considered. This is the only case in which the lexicographically next permutation in sequence must be examined.

## 6.4.4 RESULTS

The constructive enumeration algorithm was used to generate all non-isomorphic  $2-(7,3,\lambda)$  designs for  $1 \leq \lambda \leq 14$ , which were then extended to  $3-(8,4,\lambda)$  designs. The optimised canonicity test for these extensions, as described above, was used to classify all the non-isomorphic  $3-(8,4,\lambda)$  designs. The automorphism group size of each non-isomorphic  $3-(8,4,\lambda)$  design was generated using the algorithm described in Section 4.5.3, making use of centraliser subgroups and the partition refinement techniques.

As mentioned in Observation One of Section 6.4.2, the number of distinct  $2-(7,3,\lambda)$  designs must equal the number of distinct  $3-(8,4,\lambda)$  designs for all values of  $\lambda$ . This has been verified in each of the enumerations performed.

Chapter 7 gives a complete transcript of the results. This includes the total number of non-isomorphic and distinct  $2-(7,3,\lambda)$  and  $3-(8,4,\lambda)$  designs for  $1 \leq \lambda \leq 14$ , and the number of non-isomorphic and distinct designs of each automorphism group size. The totals are summarised below. The number of non-isomorphic  $2-(7,3,\lambda)$  and  $3-(8,4,\lambda)$  designs is given, along with the number of induced distinct designs, for  $1 \leq \lambda \leq 14$ .

Lambda ( $\lambda$ )	Number of non-isomorphic $2-(7,3,\lambda)$ designs	Number of Distinct $2-(7,3,\lambda)$ designs	Number of non-isomorphic $3-(8,4,\lambda)$ designs	Number of Distinct $3-(8,4,\lambda)$ designs
1	1	30	1	30
2	4	465	4	465
3	10	5045	10	5045
4	35	41265	31	41265
5	109	265792	82	265792
6	418	1402140	240	1402140
7	1508	6266415	650	6266415
8	5413	24389705	1803	24389705
9	17785	84512555	4763	84512555
10	54613	265328404	12369	265328404
11	155118	765418845	30780	765418845
12	412991	2052124690	74257	2052124690
13	1033129	5160956540	172046	5160956540
14	2449592	12268945580	385073	12268945580

## 6.5 CASE STUDY FIVE: THE MENDELSON TRIPLE SYSTEMS

### 6.5.1 BACKGROUND

Mendelsohn triple systems (MTSs) differ from standard triple systems in that the blocks are not unordered triples, but are instead cyclic triples. Any set of three numbers can be made into a cyclic triple, by orienting it in one of two possible directions. For example, the triple  $\{1,2,3\}$  can be oriented in the following two ways:

- 1)  $(1,2,3)$  which cyclically contains the ordered pairs  $(1,2)$   $(2,3)$  and  $(3,1)$
- 2)  $(1,3,2)$  which cyclically contains the ordered pairs  $(1,3)$   $(3,2)$  and  $(2,1)$ .

In general, the  $k$ -tuple  $(x_1, x_2, x_3, \dots, x_k)$  cyclically contains the  $k$  ordered pairs:  $(x_1, x_2)$   $(x_2, x_3)$  ...  $(x_k, x_1)$ .

**Definition:**

A  $(v,k,\lambda)$ -Mendelsohn design is a set  $V$ , together with a collection  $B$  of ordered  $k$ -tuples of distinct elements of  $V$ , called blocks, such that each ordered pair  $(x,y)$ , with  $x \neq y$ , is contained in exactly  $\lambda$  blocks. A  $(v,3,\lambda)$ -Mendelsohn design is a Mendelsohn triple system, denoted  $MTS(v,\lambda)$ .

The *converse* of an  $MTS(v,\lambda)$  is the design obtained by reversing the order of its blocks. If the converse of a given MTS is isomorphic to itself, the design is said to be *self-converse*.

The *underlying design* of an  $MTS(v,\lambda)$  is the  $2-(v,k,2\lambda)$  BIBD which results from ignoring the ordering structure of the blocks.

A  $(v,k,2\lambda)$  BIBD is *orientable* if and only if a cyclic order can be placed on its blocks to form a valid  $MTS(v,\lambda)$ .

### 6.5.2 CURRENT RESULTS

Two MTSs are *equivalent* if they are isomorphic or if one is isomorphic to the converse of the other. Prior to the work carried out in this case study, the number,  $M(v)$ , of inequivalent  $MTS(v,1)$  had been computed [30, 31] for orders  $v \leq 10$ , as summarised in the table below:

$v$	3	4	6	7	9	10
$M(v)$	1	1	0	3	18	144

No MTSs exist for order 11, because this is not an admissible order. This case study was able to produce two new results. The first is a correction to the number of  $MTS(10,1)$ 's given in the table above, and the second is the calculation of the number of inequivalent  $MTS(12,1)$ 's.

### 6.5.3 ENUMERATION APPROACH

Each  $MTS(v,1)$  design has an underlying twofold triple system. The number of non-isomorphic TTS designs, and the number of these which can and cannot be oriented to give a valid MTS are summarised in the table below:

Order, $v$ , and corresponding BIBD	Number of Non-Isomorphic TTS( $v$ )'s	Number orientable	Number not orientable
$v=3$ : 2-(3,3,2)	1	1	0
$v=4$ : 2-(4,3,2)	1	1	0
$v=6$ : 2-(6,3,2)	1	0	1
$v=7$ : 2-(7,3,2)	4	3	1
$v=9$ : 2-(9,3,2)	36	16	20
$v=10$ : 2-(10,3,2)	960	134	826

For example, there is exactly one non-isomorphic TTS(6), but this cannot be oriented to give an  $MTS(6,1)$ .

Two MTSs which have non-isomorphic underlying twofold triple systems must also be non-isomorphic, for if there is no point mapping from one underlying design to the other underlying design, there can certainly be no mapping between the corresponding oriented designs. Therefore, the number of non-isomorphic, orientable, twofold triple systems is a lower bound on the number of inequivalent MTSs of the same order. It is only a lower bound, for it may be possible to orient a given underlying TTS in a number of inequivalent ways.

If all of the non-isomorphic TTS designs of order  $v$  are enumerated, then all of the non-equivalent MTSs of order  $v$  can be derived from them. This can be achieved by taking each

non-isomorphic TTS in turn and orienting it in all possible ways, classifying each orientation.

Let  $S$  denote the set of all possible orientations for a given underlying TTS( $v$ ). The oriented designs belonging to set  $S$  need to be partitioned into equivalence classes, such that any two designs within a particular equivalence class are either directly isomorphic or the converse of one is isomorphic to the other. The number of equivalence classes of a given set  $S$  will be the number of inequivalent MTS( $v$ ) designs admitted by the corresponding TTS( $v$ ). Likewise, the total number of equivalence classes for all sets  $S$  generated for all underlying non-isomorphic TTS( $v$ ) designs will be exactly the total number of inequivalent MTS( $v,1$ ) designs,  $M(v)$ . A catalogue of all inequivalent MTS( $v,1$ )s can be generated by selecting a single representative from each equivalence class of each set  $S$ .

The important observation here is that the inequivalent MTSs can be calculated by examining each TTS independently. The problem can therefore be tackled in two distinct stages:

- 1) generation of all underlying TTS designs
- 2) generation of all inequivalent orientations of the underlying designs

#### 6.5.4 GENERATION OF ALL UNDERLYING TTS DESIGNS

All non-isomorphic TTS designs of a given order can be generated using the incidence matrix backtracking algorithm, along with the canonicity test to eliminate isomorphs. For the purposes of constructing MTSs, only those TTSs which can be oriented in at least one way, ie. the orientable TTS designs, need to be considered. A simple algorithm for testing whether or not a given TTS design can be oriented to give an MTS design is outlined in [16], and summarised below. It exhibits linear running time in the number of blocks in the design.

Two important observations can be made about the general process of orienting a design. Firstly, once the direction of a single pair  $\{x,y\}$  in a triple  $\{x,y,z\}$  is chosen, the cyclic ordering of the triple is then uniquely determined. Secondly, once the cyclic ordering of any block containing the pair  $\{x,y\}$  is chosen, the cyclic ordering of the other block in the design containing the second copy of the pair  $\{x,y\}$  is forced.

The algorithm proceeds by attempting to produce a single orientation of the TTS design. An arbitrary block of the design is chosen, and it is cyclically ordered in one of the two possible ways. The selection of this ordering is unimportant, because any valid orientation of the design can be reversed to produce the converse design, one of which must contain the selected block orientation.

Once this initial block is oriented, all the other blocks in the design which contain the same pairs as the initial block will have their orientations forced. These forced blocks are then oriented, which in turn may force the orientation of other blocks in the design. This process is repeated, until one of the following three situations arises:

- 1) the design becomes completely oriented, forming a valid MTS
- 2) a contradiction may occur, where a single block is forced to be oriented in two different ways
- 3) all of the forced block orientations are satisfied, but there still remain some unoriented blocks

In either of the first two situations, the algorithm terminates. Case 1 gives a valid orientation, so the TTS design is certainly orientable. Case 2 indicates that the design is not orientable, because only one block in the design had its cyclic ordering selected arbitrarily, which would be valid if an orientation existed. Case 3 arises when a portion of the blocks of the design have been oriented, and all forced cyclic orderings are satisfied. The ordering of the triples already oriented in no way affects the ordering of the triples yet to be



oriented, because there are no requirements which must be satisfied. At this stage, any one of the blocks which is not yet oriented can be given an arbitrary cyclic ordering, and the process then continues until either the entire design is oriented, or a contradiction is encountered.

Given this efficient method for determining orientability of TTS designs, the first part of the  $MTS(v,1)$  enumeration problem can be solved. The complete set of non-isomorphic  $TTS(v)$  designs can be constructed by the incidence matrix backtracking algorithm, and the orientability test just described can be used to remove from this set all the designs which cannot be oriented and so cannot admit a valid  $MTS(v,1)$ .

The next section presents an optimisation to this method in which the orientability condition is embedded into the design construction, and only orientable TTS designs are constructed, eliminating the need for a post-check.

### 6.5.5 EMBEDDED ORIENTABILITY CONSTRAINT

The basis of the optimisation for the construction of orientable TTS designs is that if a partial TTS design is not orientable, then no extension to it can be oriented either. A partial TTS, constructed down to a particular row of the incidence matrix, is not orientable if the completed blocks of the partial design cannot be oriented. Any extension to the partial design, formed by adding extra rows to the incidence matrix, can only add new blocks to the design, but cannot alter any of the existing blocks. Therefore, if the set of completed blocks of a partial design,  $P$ , cannot be oriented, any valid design generated as an extension to  $P$  will also be non-orientable.

This observation allows the orientability constraint to be embedded within the backtrack search, just as the weakly union free constraint was embedded within the search for wuf-TTS designs, in Case Study One of this chapter. No partial incidence matrix is extended if its completed blocks cannot be oriented. The overhead in embedding this constraint is very small, because orientability can be determined quickly.

For example, consider the construction of all orientable  $TTS(10)$ 's, from which the  $MTS(10,1)$ 's are to be built. All 960 non-isomorphic  $TTS(10)$  designs can be constructed in just under two minutes (see Section 5.3.2.3 at the end of Chapter 5). Only those designs which are orientable are of interest, so the orientability test would need to be performed on each of these designs. It turns out that exactly 134 of the 960 designs are orientable.

Performing the  $TTS(10)$  enumeration with the embedded orientability constraint means only orientable designs are constructed, because all non-orientable designs are pruned from some level of the search tree. Using this approach, all 134 non-isomorphic orientable TTS designs can be constructed in approximately 16 seconds - a speed up by a factor of more than 7.

The aim of this case study was to confirm the current enumeration results for MTSs, and to enumerate all the inequivalent MTSs of order 12. The underlying designs of the  $MTS(12,1)$ 's are the  $TTS(12)$ 's, which is the same class of designs for which it was shown in the first case study of this chapter that no weakly union free designs exist. This previous enumeration also used an embedded constraint - in which only weakly union free partial designs were extended - with great success.

### 6.5.6 GENERATION OF ALL INEQUIVALENT ORIENTATIONS OF THE UNDERLYING DESIGNS

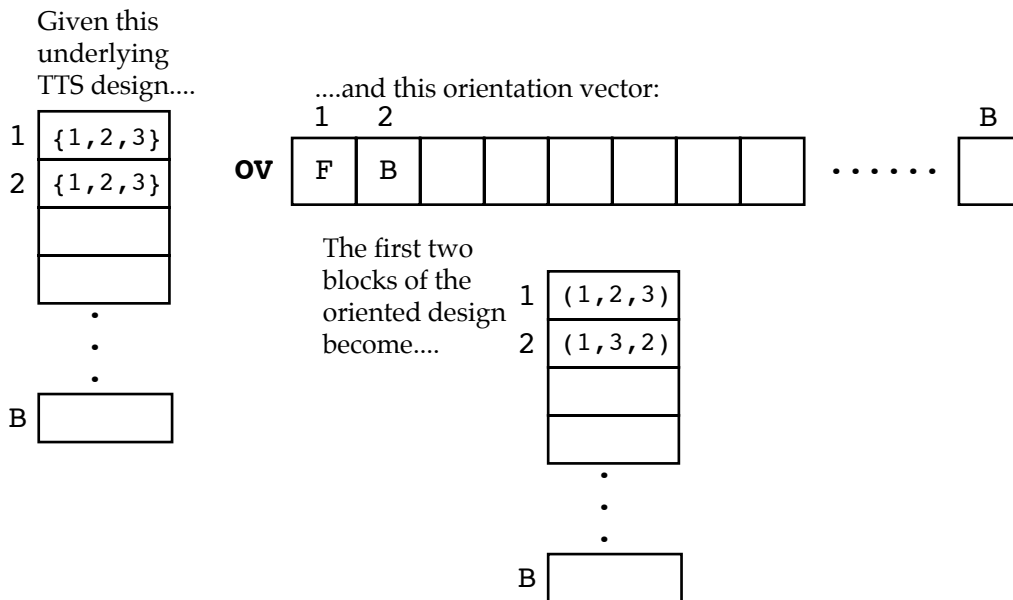
With the use of the embedded orientability constraint, the non-isomorphic, orientable TTS designs can be quickly constructed, but it remains to generate all of their inequivalent orientations. This process is tackled in two stages - firstly an exhaustive method for

producing all distinct orientations is presented, followed by ways in which this search can be pruned to produce only non-equivalent orientations.

### 6.5.6.1 PRODUCING ALL DISTINCT ORIENTATIONS

For a given TTS design, all distinct orientations can be produced by a simple exhaustive backtracking algorithm.

Assume that the TTS to be oriented contains B blocks, and consider an unoriented block {a, b, c} of the design. This block can be cyclically ordered in only two distinct ways: either (a,b,c) or (a,c,b). Let a be the smallest point in the block, and note that the elements of any cyclically ordered block can be cycled so that the smallest point, a, can be written in the leftmost position. For any cyclically ordered block (a,b,c) where a is the smallest point, if  $b < c$  then let the cyclic ordering of the block be called a *forward orientation*, and if  $b > c$  then let it be called a *backward orientation*. For example, given the block {1,2,3} of a TTS, this block can be given either a forward orientation: (1,2,3) or a backward orientation: (1,3,2). Let OV denote a boolean valued orientation vector, containing B entries. The orientation vector, OV, stores for each block in the design, whether it has a forward (F) or backward (B) orientation. The diagram below illustrates this using a design with a repeated first block:



To produce all distinct orientations then, exhaustive backtracking is performed on the orientation vector. Each element of the orientation vector gives the corresponding block of the design either a forward or a backward orientation. Potentially there are an enormous number of orientations,  $2^B$  in fact, but the search space is considerably reduced by systematically constructing a series of partial orientations and pruning the search with a feasibility function which detects a conflict if any ordered pair is covered more than once. In fact, conflicts of this type occur very early on in the construction of the partial orientation vectors, allowing very large sections of the potential search space to be pruned. With such an approach, all the distinct orientations can be produced very quickly.

One point which should be mentioned is that to avoid producing identical orientations, care must be taken when orienting any design containing duplicate blocks. Any block which is duplicated is only ever given one of the orientation types, as this forces the opposite orientation on the other block in the duplicate. If each block of the duplicate is given the opposite orientation, the resulting oriented design will be identical to the original, even though they will be represented by distinct orientation vectors.

### 6.5.6.2 PRODUCING ONLY INEQUIVALENT ORIENTATIONS

In order to produce all MTSs of a given order,  $v$ , all inequivalent orientations must be produced for each underlying orientable TTS( $v$ ). In other words, if two orientations are constructed such that they are isomorphic or one is isomorphic to the converse of the other, then only one of them must be classified. The following sections present a way of modifying the procedure of the previous section to only produce inequivalent orientations, thus providing a technique for enumerating Mendelsohn Triple Systems.

#### Optimising The Orientation Generation

Firstly, a very simple optimisation can be made to the orientation backtracking algorithm due to the fact that only inequivalent orientations are required.

Every orientation of a given TTS design has a converse, which is produced by swapping the cyclic ordering of every block. The converse of an MTS may or may not be isomorphic to itself - if it is, the design is termed self-converse. However, the definition of equivalence states that two MTSs are equivalent if they are isomorphic, or if one is isomorphic to the converse of the other - therefore, every MTS is equivalent to its converse. It is therefore possible to reduce the number of distinct orientations constructed by the algorithm, simply by not generating those orientations which are known to be converses of orientations already considered. The following argument describes a simple modification to the backtracking algorithm to achieve this.

Every underlying TTS design which is in canonical form will necessarily contain the block  $\{1,2,3\}$ , as this constraint is forced by the incidence matrix backtracking algorithm. Therefore, the first element of the orientation vector,  $OV$ , will always denote the cyclic ordering given to the block  $\{1,2,3\}$  in the orientation of the TTS design. Initially, the block  $\{1,2,3\}$  will be given a forward orientation, so that its cyclic ordering is  $(1,2,3)$ . From this point on, the remaining blocks of the design are systematically oriented in all possible ways, until backtracking returns to the first block. At this point, the first block is then given a backward orientation, ie. it becomes  $(1,3,2)$  and again all possible orientations which extend this are constructed. The algorithm terminates when backtracking returns to the first block once more, because it has already been oriented in both possible ways.

If it can be shown that every orientation produced by the algorithm in which the first block has a backward orientation, ie.  $(1,3,2)$ , is equivalent to one of the orientations constructed when the first block had a forward orientation, ie.  $(1,2,3)$ , then the first block never needs to be given a backward orientation, which will halve the number of orientations constructed.

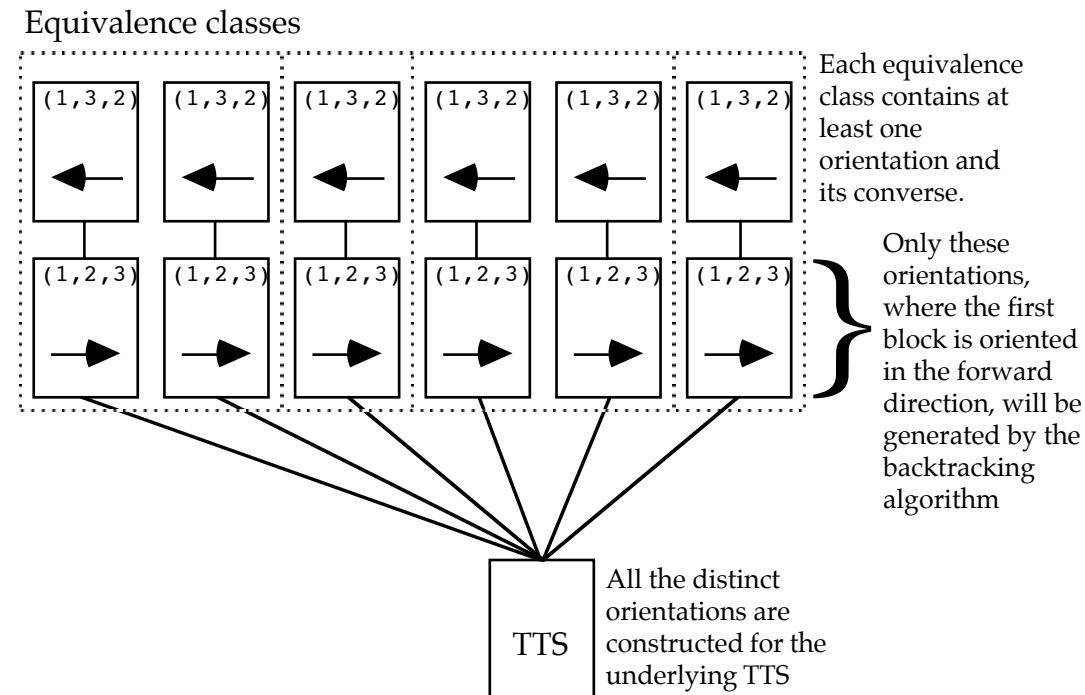
Firstly consider the case where the underlying TTS design contains a duplicate of the block  $\{1,2,3\}$ . As soon as the first block of the design is oriented in the forward direction,  $(1,2,3)$ , the second block of the design must be oriented in the backward direction,  $(1,3,2)$ . The algorithm will then proceed to orient the remaining blocks of the design. Once backtracking reaches the first element of the orientation vector again, if the first block is then given a backward orientation the second block of the design must be given a forward orientation, but this is equivalent to the previous situation, so every extension to this partial orientation vector will be exactly the same as before.

Now consider the case where the underlying design does not have a repeated first block. In other words, the second block of the design in lexicographical order is not  $\{1,2,3\}$ . Every orientation produced by the backtracking algorithm in which the first block is oriented in the forward direction, ie.  $(1,2,3)$  must have a converse in which the first block is oriented in the backward direction, ie.  $(1,3,2)$ . Therefore, any extension of the orientation vector in which the first block is oriented in the backward direction must be the converse of some orientation produced in which the first block was given a forward orientation.

In conclusion, during the backtracking construction of all orientations of a given underlying TTS design, the first block,  $\{1,2,3\}$  only needs to be considered with the forward orientation  $(1,2,3)$ . This simple optimisation halves the amount of backtracking performed in generating the orientations.

### Classifying Only Inequivalent Orientations

Even with the above optimisation, the orientations constructed will in general not all be inequivalent, so an explicit test for equivalence must still be performed. The definition of equivalence - that two MTSs are equivalent if they are isomorphic or if one is isomorphic to the converse of the other - partitions the constructed orientations into a number of equivalence classes. Every orientation and its converse must belong to the same equivalence class. The diagram below represents the general situation for a given underlying TTS:



To classify only the inequivalent orientations for a particular underlying TTS design, exactly one representative must be selected from each equivalence class. A simple approach to this is to build an explicit list of inequivalent orientations. This would be implemented by explicitly testing each generated orientation,  $O$ , for equivalence with the other orientations in the list, and only adding  $O$  to the list if it is inequivalent to them all. To test any two orientations,  $O_1$  and  $O_2$ , for equivalence, it is sufficient to perform an isomorphism test between  $O_1$  and  $O_2$ , and between  $O_1$  and the converse of  $O_2$ . There is no need to explicitly test the converse of  $O_1$  and  $O_2$  because of the following theorem:

#### Theorem:

For any two orientations,  $O_1$  and  $O_2$ , of a given underlying TTS, if  $O_1$  is isomorphic to the converse of  $O_2$  then the converse of  $O_1$  is isomorphic to  $O_2$ .

A simple proof of the above theorem follows:

Let  $O^c$  denote the converse of the orientation,  $O$ . Firstly, it must be shown that, for a given isomorphism,  $\phi$ , and oriented design,  $A$ ,  $(\phi(A))^c = \phi(A^c)$  is true. This is evident by examining the effect of  $\phi$  on an arbitrary block of the design  $A$ .

Say  $\phi$  maps the block  $(x y z)$  onto the block  $(a b c)$ . Then  $\phi(x y z) = (a b c)$ . Taking the converse of both sides:  $(\phi(x y z))^c = (a b c)^c$ , and,  $(a b c)^c = (a c b) = \phi(x z y) = \phi((x y z)^c)$ .

In addition,  $A^c = B \Rightarrow A = B^c$  also holds. If  $A^c = B$ , then  $(A^c)^c = B^c$ , and  $(A^c)^c = A$ . Therefore, if  $A^c = B$  then  $A = B^c$ .

Now, say there exists an isomorphism,  $\phi$ , mapping  $A^c$  to  $B$ . Then:

$$\begin{aligned}\phi(A^c) &= B \\ (\phi(A))^c &= B \\ \phi(A) &= B^c,\end{aligned}$$

so the same mapping,  $\phi$ , must map  $A$  to  $B^c$ , and this concludes the proof.

Therefore, to test whether two orientations,  $O_1$  and  $O_2$  are equivalent, it is necessary to test only the following two conditions:

- 1) Is  $O_1$  isomorphic to  $O_2$ ?
- 2) Is  $O_1$  isomorphic to  $O_2^c$ ?

If the answer to both of these questions is no, then the designs are inequivalent, otherwise they are equivalent.

The list building method of isomorphism testing developed in Section 5.3.1 is a perfectly valid way of classifying the inequivalent orientations, and in fact is very efficient because the number of orientations to be considered for a given underlying TTS design is usually very small. However, to be consistent with the isomorphism testing of the underlying TTS designs, a canonicity test has been developed to classify the orientations for equivalence.

In order to develop the canonicity test for orientations, an ordering must be defined on them which represents the order in which they are constructed and allows two orientations to be compared lexicographically. Only one orientation must be classified from each equivalence class, and with the defined ordering, this is chosen to be the lexicographically smallest one.

### Lexicographical Ordering Of Orientations

Whenever two orientations are to be compared lexicographically, they must first be sorted into a *totally ordered* form. Any orientation may be totally ordered, by first cycling the blocks of the design so that the lexicographically smallest element in each block is in the leftmost position of the block. Once this is done, the positions of the other two elements within the block is uniquely determined from the blocks current orientation. Finally, the blocks of the oriented design are sorted into increasing lexicographical order. Any two designs in totally ordered form can then be compared lexicographically simply by comparing the elements of each block from left to right, and examining the blocks in order from the lexicographically smallest to the lexicographically largest.

For example, consider the two cyclically ordered blocks  $A=(2,9,6)$  and  $B=(8,9,2)$ , which are to be compared lexicographically. The blocks must first be converted into totally ordered form, and so they are cycled until the smallest element is in the leftmost position. Block  $A$  remains  $(2,9,6)$ , but block  $B$  becomes  $(2,8,9)$ . Comparing the elements from left to right, it is clear the block  $B$  is lexicographically less than block  $A$ .

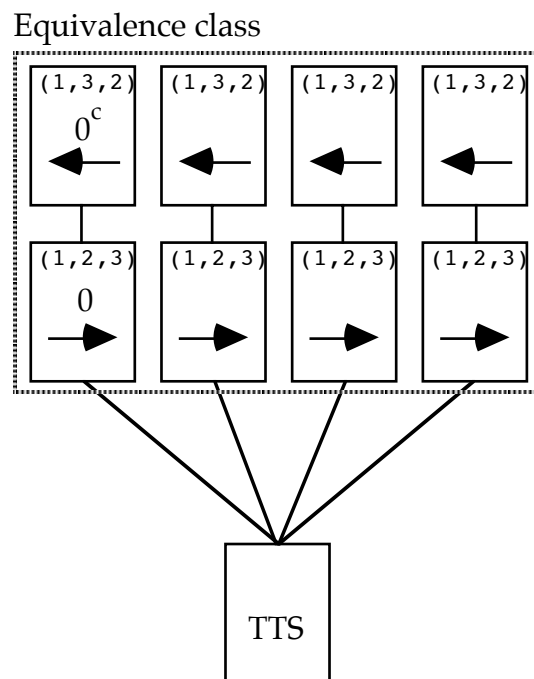
### Canonical Form For Orientations

One final important observation must be made. All orientations of a given underlying TTS design, once totally ordered, are produced in strictly increasing lexicographical order by the backtracking algorithm. This is obvious from the description of the backtracking algorithm already given. For any block  $\{x,y,z\}$  in the underlying TTS design, the forward orientation of this block will always be lexicographically smaller than the backward orientation of the block. When the design is oriented, every block is always given a forward orientation first, and is only given a backward orientation once it is backtracked. In addition, the blocks of the underlying TTS design are always in increasing lexicographical order, as this is enforced by the incidence matrix backtracking algorithm. Therefore, all orientations of a given underlying TTS design are constructed in increasing lexicographical order, as the backtracking of the orientation vector proceeds.

A given orientation,  $O$ , is therefore in canonical form if and only if it is the lexicographically smallest orientation in its equivalence class. In other words, if all other orientations to which  $O$  is equivalent are constructed,  $O$  must be lexicographically smaller than them all. Note that  $O$  is equivalent to all orientations isomorphic to  $O$ , and all orientations isomorphic to  $O^c$ , the converse of  $O$ .

Therefore, if a mapping,  $\emptyset$ , exists which maps either  $O$  or  $O^c$  to another orientation in the same equivalence class which is lexicographically smaller than  $O$ , then  $O$  is not in canonical form. If no such mapping exists, then  $O$  is the canonical representative of its equivalence class, and is classified by the algorithm.

As the above description indicates, the only mappings which need to be considered are those which map  $O$  and  $O^c$  to other orientations within the same equivalence class. Consider the diagram below:



All orientations, such as  $O$  and  $O^c$  above, within a given equivalence class necessarily have exactly the same underlying TTS design and therefore must contain exactly the same blocks as this design, although the blocks are oriented. Any orientation which does not contain exactly the same blocks as  $O$  or  $O^c$  when the block orientations are ignored, clearly has a different underlying TTS design. Therefore, the only mappings which can possibly map a given design,  $O$ , or its converse,  $O^c$ , onto another design within the same equivalence class, are those mappings which preserve the unoriented blocks of these designs. The complete set of these mappings is exactly the automorphism group of the underlying TTS design, which can be calculated quickly as a side effect of its canonicity test.

The efficient canonicity test for orientations can now be described:

Let  $O$  be the new orientation constructed by the orientation vector backtracking algorithm, which is to be tested for canonicity. Let  $O^c$  be the converse of  $O$ , and let  $G$  be the automorphism group of the underlying TTS design of  $O$ .

- Apply all mappings  $\emptyset \in G$  to  $O$ , to construct the set of  $|G|$  permuted orientations  $P$ .
- Apply all mappings  $\emptyset \in G$  to  $O^c$ , to construct the set of  $|G|$  permuted orientations  $P^c$ .
- If any totally ordered permuted orientation from the set  $P$  or from the set  $P^c$  is lexicographically less than the original orientation  $O$ , then  $O$  is not in canonical form.

- Otherwise, if  $O$  is lexicographically less than or equal to every totally ordered permuted orientation from the sets  $P$  and  $P^c$ , then  $O$  is in canonical form, and should be classified.

Less formally, any given orientation,  $O$ , is in canonical form if and only if there does not exist any mapping from the automorphism group of the underlying TTS design of  $O$  which maps either  $O$  or  $O^c$  to a new orientation which is lexicographically smaller than  $O$ . If such a mapping exists, then  $O$  should not be classified as it is equivalent to another orientation which is lexicographically smaller than itself.

## 6.5.7 RESULTS

For a given order,  $v$ , the incidence matrix backtracking algorithm and general canonicity test were used to generate all non-isomorphic TTS( $v$ ) designs. For each non-isomorphic TTS( $v$ ), the orientation vector backtracking algorithm was used to generate a set of orientations, and the canonicity test described above was used to classify only those orientations which were inequivalent.

The current published results, given in Section 6.5.2, were confirmed for all orders  $v \leq 9$ , and summarised in the table below where  $M(v)$  is the number of inequivalent MTS( $v,1$ ) designs:

$v$	3	4	6	7	9
$M(v)$	1	1	0	3	18

The total CPU time required to calculate all 5 results above using the algorithm presented in this case study was just 1.4 seconds.

### The MTS(10,1) Designs

Prior to the work in this thesis on this problem, the current published result for  $v=10$  was  $M(10)=144$ , computed by Ganter, Mathon and Rosa [30, 31].

The number of inequivalent MTS(10,1) designs calculated using the method described in this case study was 143. The discrepancy here indicated that either a design was missed in the calculation of this thesis, or an extra equivalent copy of a design was calculated in the catalogue of Ganter et al.

After examination of the catalogue published in [30], designs 3(A) and 3(B) were found to be isomorphic, and therefore equivalent. These designs are given on the following page, along with an isomorphism mapping 3(B) onto 3(A):

MTS 3(A)		MTS 3(B)	
Block 1:	( 1 2 3 )	Block 1:	( 1 2 3 )
Block 2:	( 1 3 4 )	Block 2:	( 1 3 4 )
Block 3:	( 1 4 2 )	Block 3:	( 1 4 2 )
Block 4:	( 1 5 6 )	Block 4:	( 1 5 6 )
Block 5:	( 1 6 8 )	Block 5:	( 1 6 8 )
Block 6:	( 1 7 5 )	Block 6:	( 1 7 5 )
Block 7:	( 1 8 10 )	Block 7:	( 1 8 10 )
Block 8:	( 1 9 7 )	Block 8:	( 1 9 7 )
Block 9:	( 1 10 9 )	Block 9:	( 1 10 9 )
Block 10:	( 2 4 3 )	Block 10:	( 2 4 3 )
Block 11:	( 2 5 10 )	Block 11:	( 2 5 10 )
Block 12:	( 2 6 5 )	Block 12:	( 2 6 5 )
Block 13:	( 2 7 9 )	Block 13:	( 2 7 9 )
Block 14:	( 2 8 7 )	Block 14:	( 2 8 7 )
Block 15:	( 2 9 6 )	Block 15:	( 2 9 6 )
Block 16:	( 2 10 8 )	Block 16:	( 2 10 8 )
Block 17:	( 3 5 7 )	Block 17:	( 3 5 7 )
Block 18:	( 3 6 9 )	Block 18:	( 3 6 9 )
Block 19:	( 3 7 8 )	Block 19:	( 3 7 8 )
Block 20:	( 3 8 6 )	Block 20:	( 3 8 6 )
Block 21:	( 3 9 10 )	Block 21:	( 3 9 10 )
Block 22:	( 3 10 5 )	Block 22:	( 3 10 5 )
Block 23:	( 4 5 8 )	Block 23:	( 4 5 9 )
Block 24:	( 4 6 7 )	Block 24:	( 4 6 7 )
Block 25:	( 4 7 10 )	Block 25:	( 4 7 10 )
Block 26:	( 4 8 9 )	Block 26:	( 4 8 5 )
Block 27:	( 4 9 5 )	Block 27:	( 4 9 8 )
Block 28:	( 4 10 6 )	Block 28:	( 4 10 6 )
Block 29:	( 5 9 8 )	Block 29:	( 5 8 9 )
Block 30:	( 6 10 7 )	Block 30:	( 6 10 7 )

The isomorphism:

(6→1) (10→2) (7→3) (4→4) (5→5) (2→6) (1→7) (9→8) (8→9) (3→10)

maps design 3(B) onto design 3(A). This discovery leads to the following new result:

**Result 6.5.1:**

There are exactly 143 inequivalent MTS(10,1) designs.

The implementation is very fast. All 134 non-isomorphic underlying TTS(10)'s are constructed, have their automorphism group sizes calculated, and have all orientations generated and tested for canonicity to produce the 143 inequivalent MTS(10,1)'s in just 23 seconds. A transcript of the program output for this enumeration is included below:

```

DESIGN ENUMERATOR
=====
Enumerating the orientable TTS(10)'s, and the inequivalent MTS(10,1)'s.

(Automorphism groups calculated only on sub-designs down to row 7).
(Partial isomorph rejection performed to row 10).

262 canonical starter configurations to row 7 were constructed.
The number of complete 2-(10,3,2) designs generated was 1409.
The total number of mutually non-isomorphic designs = 134.
The total number of inequivalent MTS's admitted = 143.

Group Sizes:      Frequency:      MTS's Admitted:
-----
    108             2             4
     24             1             2
     21             1             1
     18             1             1
     12             2             3
      8             1             1
      6             5             5
      4             7             9
      3            11            11
      2            21            23
      1            82            83
-----
                134                143

The process took 23.932 seconds to complete.

```



**The MTS(12,1) Designs**

The algorithm described in this case study was also used to enumerate the Mendelsohn Triple Systems of the next admissible order, 12.

Firstly, the orientable, underlying TTS(12) design were enumerated. There are exactly 4,692,239 non-isomorphic orientable TTS(12)'s.

Each of these designs were oriented in all inequivalent ways, to produce exactly 4,905,693 inequivalent Mendelsohn Triple Systems of order 12. The results are summarised in the table below with the following column headings:

IGI	<i>The automorphism group size of the corresponding designs.</i>
Nd	<i>The number of non-isomorphic orientable TTS(12) designs, which admit at least one MTS(12,1) design, by group size.</i>
MTS	<i>The number of inequivalent Mendelsohn Triple Systems admitted by the underlying TTS(12) designs of the corresponding group size</i>
Distinct	<i>The number of distinct orientable TTS(12) designs.</i>

IGI	Nd	MTS	Distinct
1536	1	3	311850
576	1	2	831600
432	1	7	1108800
192	3	4	7484400
144	1	4	3326400
128	2	5	7484400
72	2	6	13305600
64	2	6	14968800
54	1	1	8870400
48	6	12	59875200
36	1	2	13305600
32	9	15	134719200
24	7	15	139708800
18	2	3	53222400
16	24	40	718502400
12	15	38	598752000
9	1	1	53222400
8	77	173	4610390400
6	106	196	8462361600
4	482	867	57719692800
3	531	647	84783283200
2	11011	15003	2637143308800
1	4679953	4888643	2241704974924800
<b>Totals:</b>	<b>4692239</b>	<b>4905693</b>	<b>2244499522961850</b>

**Result 6.5.2:**

There are exactly 4,905,693 inequivalent MTS(12,1) designs.

**6.5.8 CONCLUSION**

This case study has produced two new results for the Mendelsohn Triple Systems, for orders 10 and 12. This information can be summarised in the following table which represents the most up to date knowledge of the number of inequivalent MTS(v,1) designs:

v	3	4	6	7	9	10	12
MTS(v)	1	1	0	3	18	143	4905693

The number of inequivalent  $MTS(v,1)$  designs have also been enumerated by automorphism group size, for all admissible orders  $v \leq 12$ . These tables are listed, along with the new results computed here, in Chapter 7.

## 6.6 CASE STUDY SIX: THE 2-(10,4,4), 3-(11,5,4), 4-(12,6,4) DESIGN FAMILIES

### 6.6.1 BACKGROUND

#### Definition:

Given a collection,  $B$ , of  $k$ -element subsets of a  $v$ -element set  $X$ , and a point  $x \in X$ , the collection of blocks:

$DER_x(B) = \{ b \setminus \{x\} : x \in b \in B \}$  is called the *derivation* of  $B$  with respect to  $x$ .

For any  $t$ -( $v,k,\lambda$ ) design  $(X, B)$ , given a point  $x \in X$ , the *derived design* with respect to  $x$ , or  $(X \setminus \{x\}, DER_x(B))$ , is a  $(t-1)$ -( $v-1,k-1,\lambda$ ) design.

Conversely, a  $(t+1)$ -( $v+1,k+1,\lambda$ ) design  $(Y, A)$  is an *extension* of a  $t$ -( $v,k,\lambda$ ) design  $(X,B)$  if  $Y = X \cup \{\rho\}$  and  $B = DER_\rho(A)$ , where  $\rho$  is a new point not in  $X$ .

Case Study Four, which examined the  $2$ -( $7,3,\lambda$ )  $\rightarrow$   $3$ -( $8,4,\lambda$ ) extension, introduced the idea of derived designs. For a given  $3$ -( $8,4,\lambda$ ) design,  $D$ , the derived design was generated by building a set of all the blocks of  $D$  which contained a particular point, but with that point removed, and each derivation was a  $2$ -( $7,3,\lambda$ ) design. In addition, every  $2$ -( $7,3,\lambda$ ) design could be uniquely extended to and hence uniquely derived from a  $3$ -( $8,4,\lambda$ ) design.

This case study continues with this idea, and examines 3 families of designs, the  $2$ -( $10,4,4$ ),  $3$ -( $11,5,4$ ) and  $4$ -( $12,6,4$ ) families. Given any  $4$ -( $12,6,4$ ) design and point  $x \in \{1,2, \dots, 12\}$ , the set of blocks containing  $x$ , but with  $x$  removed, form a  $3$ -( $11,5,4$ ) design. Similarly, any  $3$ -( $11,5,4$ ) design can have its derivative taken with respect to any point of the design to form a  $2$ -( $10,4,4$ ) design. However, only a small subset of the  $2$ -( $10,4,4$ ) designs can be derived from the  $3$ -( $11,5,4$ ) designs, and only a small subset of the  $3$ -( $11,5,4$ ) designs can be derived from the  $4$ -( $12,6,4$ ) designs.

This case study describes the methods used for determining the number of non-isomorphic designs for each family. Prior to the work done in this thesis, none of these design families had been completely classified. In addition, the number of designs which can be derived from, or equivalently extended to, the larger designs is also determined.

This case study is divided into the following sections.

- Section 6.6.2 presents the results of the enumeration of the  $2$ -( $10,4,4$ ) designs, which is performed here for the first time.
- Section 6.6.3 discusses the motivation for, and enumeration of, the  $3$ -( $11,5,4$ ) designs.
- Section 6.6.4 summarises the results of the work carried out in collaboration with Mathon [21] in which the catalogue of constructed  $3$ -( $11,5,4$ ) designs was used to produce all non-isomorphic  $4$ -( $12,6,4$ ) designs.
- Section 6.6.5 discusses the analysis of the derived designs, in particular, how all derived  $2$ -( $10,4,4$ ) designs were constructed from the catalogue of non-isomorphic  $3$ -( $11,5,4$ ) designs. The derived  $3$ -( $11,5,4$ ) designs were determined directly from the method of construction of the  $4$ -( $12,6,4$ ) designs.
- Finally, Section 6.6.6 gives a summary of the interesting results of the case study.

## 6.6.2 THE 2-(10,4,4) DESIGNS

In [45], Mathon and Rosa summarise the current state of knowledge regarding the enumeration of BIBDs. Their table explicitly lists all admissible parameter sets  $(v,b,r,k,\lambda)$ , for  $r \leq 41$  and  $3 \leq k \leq v/2$ , and for each parameter set gives the number of non-isomorphic designs or the best known lower bound.

Design #71 on this list is the 2-(10,30,12,4,4) design - the smallest design in the list on 10 points for which the exact number of non-isomorphic designs was previously unknown. The published lower bound for this design was 1,759,613, and calculated in [44].

The algorithm developed in this thesis was used to constructively enumerate these designs. In addition, the number of simple designs (ie. without repeated blocks) and the number of non-simple designs (ie. with repeated blocks) were counted. The enumeration was performed several times using different variations of the algorithm, with the results agreeing each time. For example, the incidence matrix backtracking algorithm was modified to generate only simple designs, and the number of constructed designs agreed exactly with the number of simple designs counted from the complete list of 2-(10,4,4) designs. The results are stated below, and tabulated by group size.

### Result 6.6.1:

There are exactly 13,769,944 non-isomorphic 2-(10,4,4) designs.  
10,081,743 of these are simple, and 3,688,201 are non-simple.

The group size spectrum of the non-isomorphic 2-(10,4,4) designs is given below:

Group Sizes	Number of non-isomorphic 2-(10,4,4) designs		
	<i>Simple Only</i>	<i>Non-Simple Only</i>	<i>All Designs</i>
1440	1	0	1
720	0	1	1
192	0	1	1
96	0	2	2
64	0	2	2
48	1	2	3
36	0	1	1
32	5	7	12
24	2	8	10
20	0	1	1
18	2	2	4
16	4	12	16
12	6	23	29
10	8	1	9
9	6	0	6
8	20	84	104
6	52	61	113
5	13	0	13
4	94	568	662
3	525	516	1041
2	10318	11795	22113
1	10070686	3675114	13745800
<b>Totals:</b>	10081743	3688201	13769944

From the table above, and Burnside's Lemma, the number of distinct designs can be calculated. There are exactly 49,922,888,066,100 distinct 2-(10,4,4) designs, as indicated in the following table:

Number of Distinct 2-(10,4,4) Designs	
Simple Only	36,564,006,003,840
Non-simple Only	13,358,882,062,260
All Designs	49,922,888,066,100

### 6.6.3 THE 3-(11,5,4) DESIGNS

A modified version of the constructive enumeration algorithm used to enumerate the 2-(10,4,4) designs in the previous section was also used to enumerate the 3-(11,5,4) designs. For a discussion of the modifications refer to Section 6.8.1 of Case Study Eight. Section 6.6.3.1 describes the motivation for studying this class of designs, and Section 6.6.3.2 presents the results of the enumeration.

#### 6.6.3.1 MOTIVATION

The complete classification of all 3-(11,5,4) and 4-(12,6,4) designs had not previously been accomplished, although some work had gone into the latter problem prior to this thesis. Breach, Sharry and Street attempted to construct all non-isomorphic 4-(12,6,4) designs, in a series of papers [6, 7, 8], by examining the block types and the block intersection types of these designs. A series of partial skeleton designs were constructed from which all 4-(12,6,4) designs were to be extended. The conclusion of the series of papers was that there were 10 non-isomorphic 4-(12,6,4) designs, but this result was later shown to be incorrect by Mathon [46] when he constructed a 4-(12,6,4) design by probabilistic techniques which was not one of the designs classified by Breach et al.

One possible technique of enumerating the 4-(12,6,4) designs, is to start with a set,  $S$ , of derived designs, and produce all non-isomorphic extensions of this set. As already mentioned, taking the derivative with respect to any point of a 4-(12,6,4) design must give a 3-(11,5,4) design. Therefore if the set  $S$  is the set of all non-isomorphic 3-(11,5,4) designs, then the set of all non-isomorphic 4-(12,6,4) designs can be constructed by producing all extensions to  $S$ .

In a similar fashion, any 4-(12,6,4) design, if derived once to give a 3-(11,5,4) design can then be derived again to give a 2-(10,4,4) design. All 4-(12,6,4) designs can be twice derived in this way to form 2-(10,4,4) designs, and so all 4-(12,6,4) designs could be enumerated by producing all non-isomorphic extensions to the complete set of non-isomorphic 2-(10,4,4) designs.

However, producing all non-isomorphic extensions of a set of derived designs is not trivial. Each derived design forms a number of partial blocks, and these blocks must be extended using some exhaustive strategy. The larger the number of partial blocks, the less searching will need to be performed to produce the extensions, because the constraints on the search will be tighter. This makes the 3-(11,5,4) designs more desirable than the 2-(10,4,4) designs because they contain twice as many blocks, and one extra point. Also, it is desirable to have only a small number of designs in the set to be extended, so that fewer extensions need to be considered. Once again, the 3-(11,5,4) designs are desirable to the 2-(10,4,4) designs, because there are many orders of magnitude fewer 3-(11,5,4) designs.

In conclusion, a catalogue of all non-isomorphic 3-(11,5,4) designs was required in order to enumerate the 4-(12,6,4) designs by extension, and the efficient extension algorithm of Mathon [21] was used for this purpose.

### 6.6.3.2 RESULTS

The constructive enumeration algorithm was modified to enumerate 3-designs, as described in Section 6.8.1.

**Result 6.6.2:**

There are exactly 1749 non-isomorphic 3-(11,5,4) designs. All 3-(11,5,4) designs are simple (ie. have no repeated blocks)

The group order spectrum of these designs is given in the table below:

Group Size:	Number of Non-Isomorphic Designs	Number of Distinct Designs
7920	1	5040
288	1	138600
144	1	277200
72	1	554400
48	1	831600
24	1	1663200
20	1	1995840
18	1	2217600
16	6	14968800
12	1	3326400
11	1	3628800
10	1	3991680
8	34	169646400
6	8	53222400
5	5	39916800
4	34	339292800
3	8	106444800
2	119	2375049600
1	1524	60833203200
<b>TOTALS:</b>	<b>1749</b>	<b>63950375160</b>

### 6.6.4 THE 4-(12,6,4) DESIGNS

Mathon [21] was able to extend each of the 1749 non-isomorphic 3-(11,5,4) designs to produce the 4-(12,6,4) designs. There are exactly 11 non-isomorphic 4-(12,6,4) designs. For details of the extension process and statistics of the results, please refer to [21].

### 6.6.5 THE DERIVED DESIGNS

The 3 design families classified in this case study are all related in the sense that the derivation of the larger designs give the smaller designs, although it is not the case that every one of the smaller designs can be derived from a larger design. This property raises two very interesting questions:

- How many of the 13,769,944 non-isomorphic 2-(10,4,4) designs are derived from the 3-(11,5,4) designs?
- How many of the 1749 non-isomorphic 3-(11,5,4) designs are derived from the 4-(12,6,4) designs?

The second question can be answered immediately. This is because the non-isomorphic 4-(12,6,4) designs were constructed by explicitly extending each of the 3-(11,5,4) designs. Exactly 13 of the 3-(11,5,4) designs can be extended to give a valid 4-(12,6,4) design, and thus are derived. It is interesting to note that these 13 3-(11,5,4) designs are in fact the 13

lexicographically largest designs, and so were the last 13 designs produced by the constructive enumeration algorithm. The table below summarises the group sizes of these designs.

Group Size	Number of non-isomorphic derived 3-(11,5,4) designs
7920	1
144	1
72	1
24	1
16	1
8	3
5	1
3	1
2	1
1	2
<i>TOTAL:</i>	13

The first question is slightly more complicated. It could be answered by attempting to extend every one of the non-isomorphic 2-(10,4,4) designs, as only those which can extend to a valid 3-(11,5,4) design can be derived. Unfortunately, this would be a very expensive operation, due to the large number of 2-(10,4,4) designs and the difficulty of performing the extensions.

A much better approach is to examine each 3-(11,5,4) design, and generate all 2-(10,4,4) designs which can be derived from it. Testing each generated derived design for isomorphism, the complete set of non-isomorphic derived 2-(10,4,4) designs can be constructed, thus answering the first question.

Recall that a derived design is formed by selecting a point,  $x$ , in the original design, and returning all those blocks containing the point  $x$ , but with  $x$  removed. The selected point  $x$  can be any one of the  $v$  points in the design, yet two derivations of the same design may be isomorphic. Therefore, any  $t$ -( $v,k,\lambda$ ) design admits at least one non-isomorphic derived design, and at most  $v$  non-isomorphic derived designs. If all  $v$  derivations are isomorphic, the design is said to be *homogeneous*, and if all the derivations are non-isomorphic the design is said to be *heterogeneous*.

In the case of each 3-(11,5,4) design, each point from the point set  $X = \{1,2,3,\dots,11\}$  is removed systematically, and the corresponding derived designs are constructed. To classify the derived designs isomorphically, either an explicit non-isomorphic list can be built, or the constructed derived designs can be canonically labelled. This second method, described in Section 6.6.5.1 below, is more desirable than the first, as it is much more practical for large design families and is in fact an extremely useful procedure which can be applied to many combinatorial design problems.

### 6.6.5.1 CANONICAL LABELLING

The canonical labelling of a given design,  $D$ , is the canonical representative of  $D$ 's isomorphism class, ie. the lexicographically smallest design to which  $D$  is isomorphic. All designs belonging to a particular isomorphism class clearly have the same canonical labelling, which can therefore be used to test designs for isomorphism. Two designs are isomorphic if and only if they have identical canonical labellings.

In relation to the current problem, each of the 11 distinct derived designs constructed from a given 3-(11,5,4) design, say  $D$ , is given its canonical labelling. Each of the relabelled designs can then be inserted into a list, which maintains a single copy of each unique design. The number of designs in the list is then exactly the number of non-isomorphic derived designs admitted by  $D$ , and in addition they are all stored in canonical form.

To generate all non-isomorphic derived 2-(10,4,4) designs, the process above is repeated for all 3-(11,5,4) designs, and a single list is maintained. The resulting list will contain all non-isomorphic derived designs in canonical form.

Therefore, some facility for computing the canonical labelling of a given design is required. Obviously, the brute force method would construct all possible relabellings of  $D$  and return the lexicographically smallest, but this can be significantly optimised.

A more efficient approach is to make repeated canonicity tests. For example, say design  $D$  is to be given its canonical labelling. An efficient canonicity test, as described in Section 5.3.2.2, is performed for  $D$ , generating the sequence of ordered relabellings of  $D$  until a new design,  $D'$ , is constructed which is lexicographically less than, but clearly isomorphic to  $D$ . If no such design is constructed, then  $D$  is already in canonical form, and is therefore canonically labelled. Otherwise, design  $D$  is rejected, and  $D'$  is accepted as the current design. The process is then repeated for  $D'$ . Thus, a sequence of isomorphic designs will be produced - each one lexicographically smaller than the last - until the canonical labelling of  $D$  is constructed. Only one complete canonicity test will be performed - that corresponding to the final design, whereas all the intermediate canonicity tests will complete as soon as the next design in the sequence is found.

### 6.6.5.2 IMPLEMENTATION

The algorithm presented in Section 6.6.5.1 above for generating the canonical labelling of a given incidence structure was implemented. A series of canonicity tests are performed which generate a sequence of lexicographically decreasing designs, until the canonical labelling is produced.

For example, a random TTS(9) was constructed, and represented in the following row and column ordered incidence matrix:

```

11111111000000000000000000
11000000111111000000000000
110000000000001111110000
001100001100001100001100
001100000011000011000011
000011000010101000101010
000010101000010010010110
000000110101000000111001
000001010000110101000101

```

This design then had its canonical labelling constructed, which is given below:

```

11111111000000000000000000
11000000111111000000000000
110000000000001111110000
001100001100001100001100
001100000011000011000011
000011001010000000111010
000010100100100010100101
000001010000111001000110
000000110001010100011001

```

The automorphism group size of this design is 2, as calculated by the final canonicity test. Exactly 6 other relabelled designs were constructed in the sequence. In total, the relabelling and automorphism group calculation took 0.018 seconds to complete.

The canonical relabelling procedure was then used to determine which of the 2-(10,4,4) designs are derived from the 3-(11,5,4) designs. All of the 1749 3-(11,5,4) designs had their corresponding derived designs constructed by removing each of the 11 points in turn. Every derived design was then canonically labelled and stored in a list of unique designs. After every design was processed, the resulting list was a complete catalogue of the non-isomorphic derived 2-(10,4,4) designs. The results of this are given in the table at the top of the next page:

Group Size	Number of non-isomorphic derived 2-(10,4,4) designs
1440	1
32	4
18	1
16	3
10	2
9	1
8	5
6	7
5	2
4	14
3	17
2	288
1	8633
<b>TOTAL:</b>	<b>8978</b>

Exactly 8,978 of the 13,769,944 non-isomorphic 2-(10,4,4) designs are derived from a 3-(11,5,4) design.

In addition, the 3-(11,5,4) designs were processed to determine how many non-isomorphic derived designs they each admit. Each design must admit at least one, yet no more than 11 non-isomorphic derived designs. These results are summarised below:

Number of non-isomorphic derived designs admitted	Frequency
11	1503
10	18
9	1
8	79
7	17
6	27
5	34
4	26
3	33
2	9
1	2
<b>TOTAL:</b>	<b>1749</b>

In conclusion, most 3-(11,5,4) designs are heterogeneous - ie. admit 11 non-isomorphic 2-(10,4,4) derived designs, and only 2 are homogeneous.

### 6.6.6 RESULTS SUMMARY

A brief overview of the main results of this section are given below.

*Number of non-isomorphic designs:*

Design	2-(10,4,4)	3-(11,5,4)	4-(12,6,4)
Non-isomorphic designs	13,769,944	1,749	11

*Number of non-isomorphic derived designs:*

Design	2-(10,4,4)	3-(11,5,4)
Non-isomorphic derived designs	8,978	13



## 6.7 CASE STUDY SEVEN: CONFIGURATIONS

### 6.7.1 INTRODUCTION TO CONFIGURATIONS

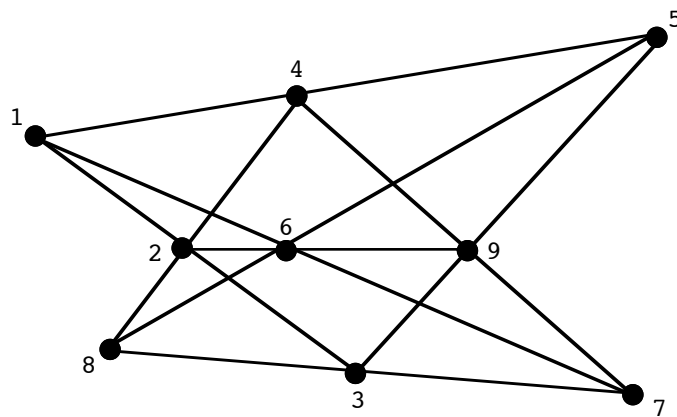
**Definition:**

A *configuration*  $(v_r, b_k)$  is an incidence structure of  $v$  points and  $b$  lines such that:

- 1) each line contains  $k$  points
- 2) each point lies on  $r$  lines
- 3) two different points are connected by at most one line

The *configuration graph* of a configuration  $(v_r, b_k)$ , is a graph in which the vertices are the  $v$  points of the configuration, and two vertices are joined by an edge if and only if the corresponding points are not collinear in the configuration. The configuration graph is  $d$ -regular, where  $d = v - r(k-1) - 1$ , and is called the deficiency of the configuration.

If  $v=b$  and hence  $r=k$ , then the configuration is said to be *symmetric* and denoted by  $v_k$ . For example, below is a diagram of a  $9_3$  configuration. There are 9 points and 9 lines. Every line passes through exactly 3 points, every point lies on exactly 3 lines, and no two points are connected by more than one line:



### 6.7.2 CONSTRUCTION OF CONFIGURATIONS

Any configuration can be represented as an incidence matrix, where the rows and columns of the incidence matrix correspond to the points and lines of the configuration respectively. For example, the incidence matrix of the above  $9_3$  configuration is given below:

Columns:	
<u>123456789</u>	
1:	111000000
2:	100110000
3:	100001100
4:	010100010
5:	010001001
6:	001010001
7:	001000110
8:	000100101
9:	000011010

Clearly, from the definition, every row of the incidence matrix of a symmetric  $v_3$  configuration must contain 3 points, every column must contain 3 points, and the intersection or cross product between any two rows of the matrix must be either 0 or 1.

The automorphism group size of the  $9_3$  configuration drawn in Section 6.7.1 above is 108. There are in fact only two other non-isomorphic  $9_3$  configurations, with group sizes of 9 and 12.

The incidence matrix backtracking algorithm can easily be modified to generate  $v_3$  configurations, by allowing the intersection between any two rows of the matrix to be variable, and enforcing that this intersection be either 0 or 1. Isomorphisms between configurations are defined in exactly the same way as before, and so the constructive enumeration tools developed in this thesis can be directly applied.

### 6.7.3 CONFIGURATION ENUMERATION BY LIST BUILDING

The incidence matrix backtracking algorithm, with the modification to generate  $v_3$  configurations, and the explicit list building algorithm discussed in Section 5.3.1 of Chapter 5, have been used to enumerate the non-isomorphic, symmetric  $v_3$  configurations.

The results are summarised in the following table, where Nd gives the number of non-isomorphic  $v_3$  configurations. These numbers confirm the known results for these designs:

v	7	8	9	10	11	12	13	14
Nd	1	1	3	10	31	229	2036	21399

The list building algorithm used the optimisation of design signatures, discussed in Section 5.3.1.3, to reduce the number of required isomorphism tests. The signatures were derived from the clique totals of the block intersection graph of each configuration. In addition, the list reordering optimisation of Section 5.3.1.3 was also used to modify the order in which the list was scanned. For the  $13_3$  enumeration, the execution time of the algorithm was improved by 15% simply by using a self modifying list rather than a list built and scanned in lexicographical order.

As the order of the configurations increases, the problems related to building the non-isomorphic list become the main difficulty to performing the enumerations. To make an enumeration of the  $15_3$  configurations feasible using explicit list generation, two problems need to be addressed. Firstly, the number of isomorphism tests required to build the list must be reduced even further, and secondly the storage of the non-isomorphic configurations must be made more space efficient.

The current implementation stores, for each design in the list, the signature of the design, the block list of the design, and the results of the clique analysis which are used along with the design itself to induce point, point pair and block partitions. To reduce the space requirements of the complete list, the block list and clique analysis results of each configuration are encoded into the smallest possible number of bits. To reduce the number of isomorphism tests performed, a stronger signature is introduced, and this is detailed next.

#### Reducing Isomorphism Testing-Stronger Signature

The most effective method for reducing the number of isomorphism tests performed is to produce a stronger signature for the configurations. The signature needs to be some characteristic of the design which is invariant under isomorphism, so that any two isomorphic configurations necessarily share the same signature.

The current signature being used is derived from the clique totals of the block intersection graph of the configuration. In this section, a new signature termed a Line-Intersection invariant is introduced, which turns out to perform remarkably well when used in conjunction with the clique totals signature.

The Line-Intersection invariant is calculated by examining all possible  $n$ -subsets of the point set, and counting how many lines in total are intersected by all the points in the subset. Any 1-subset of the point set, ie. a single point, must intersect with exactly 3 lines. Similarly, because the configuration graph of any configuration is  $d$ -regular, where  $d$  is the

deficiency of the configuration, the total number of intersections of all pairs of points with the lines of the configuration is constant for a given parameter set. Therefore, the smallest sized point subsets examined by the line intersection invariant are the 3-subsets, as 1 and 2 point subsets offer no useful information.

For any  $15_3$  configuration, the number of lines intersected by any subset of the point set of size  $\geq 3$  must lie between 6 and 15. Clearly no more than 15 lines can be intersected because this is the total number of lines in the configuration, and no less than 6 lines can be intersected by any 3 points.

Let  $S_n$  be the set of all possible  $n$ -subsets of the point set  $X = \{1, 2, 3, \dots, v\}$ , and let  $\text{Int}(s)$  be the total number of lines intersected by a given point subset  $s \in S_n$ . The Line-Intersection invariant counts, for all  $\binom{v}{n}$  point subsets  $s \in S_n$ , how often each possible value of  $\text{Int}(s)$  occurs.

### Effectiveness Of New Signature

To evaluate the effectiveness of the new signature, statistics were generated for several sample enumerations. Let CS represent the signature derived from the clique analysis results, and let LS represent the signature derived from the line intersection analysis. Using only the CS signature, the construction of the complete list of 2036 non-isomorphic  $13_3$  configurations, from the 38,234 generated configurations, required 1,384,264 isomorphism tests.

The number of isomorphism tests performed is reduced considerably when the LS signature is used in conjunction with the CS signature. The effectiveness of the LS signature is clearly dependent on which  $n$ -subsets of the point set are chosen for its calculation. The table below summarises several possible point subset selections, and the total number of isomorphism tests required to construct the list of 2036 non-isomorphic  $13_3$  configurations:

Point Subsets Examined	Total number of isomorphism tests required
<i>No subsets examined - only CS signature used</i>	1,384,264
<i>All 3 and 6 point subsets</i>	366,546
<i>All 5 point subsets</i>	95,994
<i>All 3, 4 and 5 point subsets</i>	66,989

There is a small overhead in calculating the LS signature, as there are  $\binom{v}{n}$   $n$ -subsets of a point set containing  $v$  points. However, the savings offered by this method are very substantial, and for large searches the overhead of computing the Line-Intersection signature is insignificant compared to the reduction in expensive isomorphism tests it offers. As exhibited in the table above, there is a 95% reduction in the number of isomorphism tests required for the  $13_3$  enumeration when using the CS signature and an LS signature derived from examining 3, 4 and 5 point subsets, rather than just using the CS signature.

### Results

The  $15_3$  enumeration was performed, with the signature of each configuration being derived from the results of a (6,1)-clique analysis and a Line-Intersection analysis examining all 3, 4 and 5 point subsets.

There are exactly 245,342 non-isomorphic  $15_3$  configurations. Their automorphism group sizes are given in the following table:

Group Sizes:	Number of non-isomorphic configurations
8064	1
720	1
192	2
128	1
72	1
48	6
32	1
30	2
24	2
20	2
18	1
16	10
15	2
12	11
10	3
8	34
6	59
5	5
4	180
3	69
2	3709
1	241240
<i>Total</i>	245342

This result confirms the recently published result of A. Betten and D. Betten in [3].

#### Distributed Enumeration

Section 5.3.1.3 presented a method for distributing the non-isomorphic list construction for a particular design class which, in this case study, has been applied to the enumeration of configurations. This technique splits the list generation up into  $n$  pieces, such that each piece classifies only those configurations for which the sum of the elements of the clique signature hash to a given value mod  $n$ . Any two configurations which are hashed into different list classes must be non-isomorphic and therefore do not need to be explicitly tested. A separate list is constructed for each list class, and the union of all such lists represents the complete catalogue of all non-isomorphic configurations.

This distributed approach was used to enumerate the  $15_3$  configurations once again, this time with the construction divided into 8 list classes. The partitioning was very even, and the number of non-isomorphic designs generated in each list class is given below:

List Class	0	1	2	3	4	5	6	7
No. of non-isomorphic designs in the class	28135	25878	26655	29830	33293	35243	34187	32121

The combined CPU time of this parallel  $15_3$  enumeration was only 45% as long as the single enumeration, simply because the number of signature comparisons performed is significantly reduced by building shorter lists.

The observed CPU time of the parallelised enumeration, which is equivalent to the longest time taken to build the list of any of the list classes, was 15.8 times faster than the single enumeration. This is a better-than-linear speed up, due to the basic complexity of the list building algorithm. This example illustrates the practical effectiveness of the list building optimisations presented in Section 5.3.1.3.

### 6.7.4 CONFIGURATION ENUMERATION BY CANONICITY TESTING

Enumeration of the  $16_3$  configurations using a list building approach would have been very difficult. As expected, there turn out to be more than 3,000,000 non-isomorphic  $16_3$  designs, and even with the list building optimisations, managing this number of designs becomes quite complicated. A much better approach involves the canonicity test developed in Section 5.3.2 of Chapter 5. This canonicity test was designed to be general purpose, and so can be applied without modification to any incidence structure, such as the  $16_3$  configurations.

The major advantage here is that none of the configurations need to actually be stored in memory or on disc, but can simply be counted. The methods of parallelisation discussed in Case Study Two of this chapter were used to reduce the running time of the search.

#### Results

All symmetric  $v_3$  configurations, for  $v \leq 15$ , were enumerated again using the canonicity test and the results agreed with the results of the list building enumerations. In addition 3,004,881 non-isomorphic  $16_3$  configurations were enumerated. Their automorphism group sizes are summarised below:

GROUP SIZES:	Number of non-isomorphic configurations
18144	1
4608	1
2016	1
1512	1
96	2
48	5
32	5
24	7
16	24
12	19
8	93
6	88
4	635
3	320
2	17119
1	2986560
<i>Total:</i>	3004881

This result confirms the recent result of Betten, Brinkmann and Pisanski in [4].

In Chapter 7, a complete summary of the enumeration results for the symmetric  $v_3$  configurations computed in this thesis are given. In particular, the number of non-isomorphic configurations of each group size are given for all admissible orders  $v \leq 16$ .

### 6.7.5 OTHER SYMMETRIC CONFIGURATIONS

The  $v_4$  symmetric configurations have also been enumerated for all admissible orders,  $v \leq 17$ . The required modification to the incidence matrix backtracking algorithm was trivial - the number of points in each column and the number of points on each row must be 4. The results are given on the next page:

v	13	14	15	16	17
No. of non-isomorphic $v_4$ configurations	1	1	4	19	1972

The automorphism group sizes of these designs are tabulated below:

Group Sizes:	Order of the symmetric $v_4$ configurations				
	13	14	15	16	17
5616	1				
1152				1	
360			1		
336		1			
72					1
36					2
32				1	
30			1		
24			1		
18				1	1
17					2
16				1	
15			1		
12				1	2
8					5
6				2	13
4				1	27
3				3	24
2				6	134
1				2	1761
<b>TOTALS:</b>	<b>1</b>	<b>1</b>	<b>4</b>	<b>19</b>	<b>1972</b>

## 6.7.6 NON-SYMMETRIC CONFIGURATIONS

A non-symmetric configuration is simply one in which the number of points is not equal to the number of lines. Again, the required modification to the incidence matrix backtracking algorithm to enumerate non-symmetric configurations is trivial. To generate  $(v_r, b_k)$  configurations, the number of points placed on each row must be  $r$  and the number of points in each column of the matrix must be  $k$ . As before, the intersection between any two rows of the matrix must be 0 or 1.

In [35], Gropp has published the number of non-isomorphic non-symmetric configurations for the following parameter sets:

$(v_r, b_k)$	$(12_4, 16_3)$	$(12_5, 20_3)$	$(14_6, 28_3)$
Number of non-isomorphic configurations	574	5	787

The extent of the work on non-symmetric configurations in this thesis was to confirm each of the above numbers, although there are still many interesting problems. For example, Betten and Betten [3] list the following 4 open enumeration problems for  $v \leq 16$ :  $(15_6, 30_3)$ ,  $(15_5, 25_3)$ ,  $(15_4, 20_3)$  and  $(16_6, 32_3)$ .

The algorithm presented in this case study could be used directly to compute solutions to some of these open problems within a reasonable amount of time, particularly with the use of the parallelisation techniques presented in Case Study Two.

### 6.7.7 BLOCKING SET FREE CONFIGURATIONS

#### Definition:

A subset of the point set of a configuration is a *blocking set* if it intersects all lines in at least 1 and at most  $k-1$  points. A configuration is *blocking set free* if it does not contain a blocking set.

In [34] and [35], Gropp gives the following summary of the current state of knowledge concerning the existence of blocking set free configurations, and provides references to the work which has gone towards solving this problem:

- There is no blocking set free configuration  $v_3$  for  $v=8, 9, 10, 11, 12, 14$ .
- There is a blocking set free configuration  $v_3$  for  $v=7, 13, 19, 21, 22, 25$  and for all  $v \geq 27$ .
- The existence is open for  $v = 15, 16, 17, 18, 20, 23, 24$  and 26.

The work of this section solves the existence question for the 3 smallest orders of the above spectrum.

#### Disconnected Configurations

Blocking set free configurations may not consist of one or more disconnected components. For example, one of the non-isomorphic  $14_3$  configurations which was enumerated earlier consists of two copies of the unique  $7_3$  configuration. This configuration has group size 56,448, and is given in the incidence matrix below:

```

1: 11100000000000
2: 10011000000000
3: 10000110000000
4: 01010100000000
5: 01001010000000
6: 00110010000000
7: 00101100000000
8: 00000001110000
9: 00000001001100
10: 00000001000011
11: 00000000101010
12: 00000000100101
13: 00000000011001
14: 00000000010110

```

As each of the component  $7_3$  configurations are blocking set free, the configuration above must also be blocking set free. However, only fully connected configurations, which do not consist of one or more copies of smaller configurations, are considered in the search for blocking set free designs.

#### Method

To show that a constructed configuration is blocking set free, all possible point subsets must be considered, none of which should intersect all the lines of the configuration between 1 and  $k-1$  times. It is difficult to imbed this constraint early within the backtracking construction of the incidence matrix, because it is not until very late in the construction that all the columns of the incidence matrix contain even one point, and no blocking set can possibly exist in a partial incidence matrix in which some of the lines are not incident with any points. For this reason, only completed configurations are examined for blocking sets, and in most cases such sets are found very quickly, rejecting the corresponding configuration.

For example, it is known that a blocking set free  $19_3$  configuration exists. The algorithm described above was used to construct the  $19_3$  configuration given at the top of the facing page, which is blocking set free:

```

1: 11100000000000000000
2: 10011000000000000000
3: 10000110000000000000
4: 01010100000000000000
5: 01001010000000000000
6: 00110010000000000000
7: 00100001100000000000
8: 00001000011000000000
9: 00000100000110000000
10: 00000001010100000000
11: 00000001001010000000
12: 00000000110010000000
13: 00000000100001100000
14: 00000000010000110000
15: 0000000000010000011000
16: 00000000000000101010
17: 000000000000000100101
18: 000000000000000011001
19: 000000000000000010110

```

### Results

The above method was used to exhaustively search for blocking set free  $15_3$ ,  $16_3$  and  $17_3$  configurations, however none were constructed.

#### Result 6.7.1:

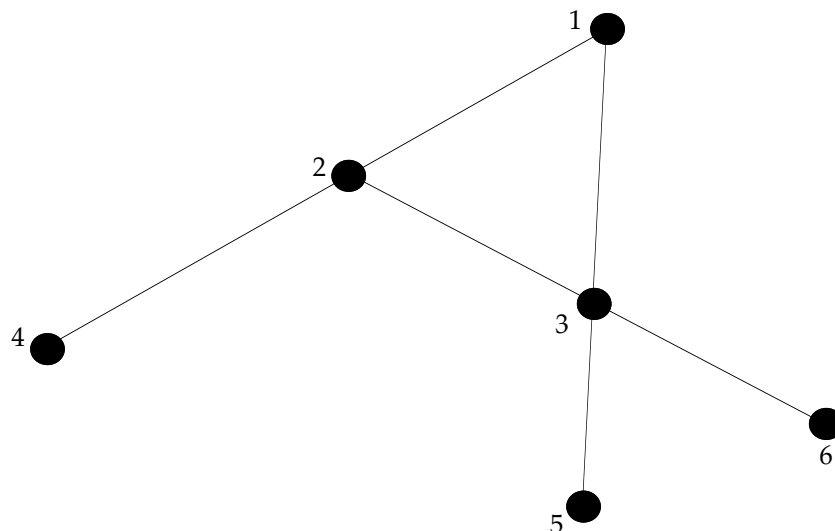
No blocking set free  $v_3$  configurations exist for orders 15, 16 or 17.

The results above regarding blocking set free configurations were performed at the same time as, and independently of, the enumerations for the paper of Betten et al. [4], in which the blocking set free existence problem is settled for orders 15, 16, 17 and 18. There are also no blocking set free  $18_3$  configurations.

## 6.7.8 TRIANGLE FREE CONFIGURATIONS

Another interesting problem is that of constructing those configurations which contain no triangles. A *triangle* in a configuration is defined as a set of 3 distinct points which are connected by three distinct edges.

For example, if the following 3 blocks are contained within a particular configuration:  $\{1,2,4\}$ ,  $\{1,3,5\}$ ,  $\{2,3,6\}$ , then this will represent a triangle, because the points  $\{1,2\}$ ,  $\{1,3\}$  and  $\{2,3\}$  are all joined by distinct lines. A graphical representation of this is given below:



The triangle connecting the points  $\{1,2,3\}$  is clearly visible in the above diagram.



**Method**

The added constraint of not allowing triangles to be formed can be integrated very nicely into the incidence matrix backtracking algorithm.

Consider the problem of constructing a triangle free  $v_3$  configuration. The replication number of a  $v_3$  configuration is 3, because each point must be incident with exactly 3 lines. In this implementation, checking for triangles is performed each time a new valid row is completed, the construction of which is very fast because only a small amount of backtracking needs to be performed.

A partial incidence matrix, valid up to row  $r$ , will only be extended to row  $r+1$  if there are no triangles in the corresponding partial configuration consisting of points  $1, 2, \dots, r$ . If a triangle does exist, then the partial configuration can be rejected immediately because the same triangle will be contained in any extension of it.

To check whether adding a newly constructed row,  $r+1$ , to a triangle free partial incidence matrix consisting of rows  $1, 2, \dots, r$ , will form a triangle, it is necessary to consider each pair of distinct rows  $\{a, b\}$  (for  $1 \leq a, b \leq r$ ). The intersection between any two rows can only be 0 or 1. If the intersection between rows  $\{a, r+1\}$  is 1, the intersection between rows  $\{b, r+1\}$  is 1, and the intersection between rows  $\{a, b\}$  is 1, and if  $a, b$  and  $r+1$  do not all belong to the same line, then a triangle has been formed. In this case, row  $r+1$  is rejected, and backtracking can begin immediately on the incidence matrix.

**Results**

It turns out that for small orders, every  $v_3$  configuration contains a triangle. More precisely, there are no triangle free symmetric  $v_3$  configurations for orders 7, 8, 9, ..., 14.

The first order for which a triangle free configuration exists is 15. Recall that there are 245,342 non-isomorphic  $15_3$  configurations. Exactly one of these is triangle free. The incidence matrix of this unique configuration is given below:

```

1: 111000000000000
2: 100110000000000
3: 100001100000000
4: 010000011000000
5: 010000000110000
6: 001000000001100
7: 001000000000011
8: 000100010001000
9: 000100000100010
10: 000010001000100
11: 000010000010001
12: 000001010000001
13: 000001000100100
14: 000000101000010
15: 000000100011000
    
```

The complete results for the other orders enumerated using the described approach are summarised in the table below:

Order, $v$	$15_3$	$16_3$	$17_3$	$18_3$	$19_3$	$20_3$
Number of non-isomorphic, triangle free, $v_3$ configurations	1	0	1	4	14	162

The incidence matrices of all triangle free  $v_3$  configurations, for  $v \leq 18$ , are given in canonical form in Chapter 7.

The triangle free configuration enumerations of this case study were performed independently to the enumerations of Betten et al. [4], and confirm their results. In addition to the numbers above, Betten et al. enumerate the 4713 non-isomorphic triangle free  $21_3$  configurations.

## 6.8 CASE STUDY EIGHT: OTHER COMBINATORIAL DESIGNS

This final case study of Chapter 6 presents several further applications of the constructive enumeration algorithm developed in this thesis to other incidence structure problems. The work of this case study has led to a number of new results, in particular the correction of two currently published, yet incorrect, BIBD enumerations - that of the  $2-(8,4,9)$  and the  $2-(16,4,2)$  designs.

In total, four problems are discussed in detail, and each is covered in one of the sections listed below:

- 6.8.1  $t$ -designs
- 6.8.2 cubic multigraphs
- 6.8.3 symmetric designs
- 6.8.4  $n$ -simple designs

### 6.8.1 $t$ -DESIGNS

A  $t-(v,k,\lambda)$  design based on a set,  $S$ , of  $v$  points is a collection of subsets, each of size  $k$ , called blocks, such that each  $t$ -subset of  $S$  appears in exactly  $\lambda$  blocks.

#### 2-Designs

The balanced incomplete block designs, or BIBDs, are the  $2-(v,k,\lambda)$  designs and have been mentioned frequently throughout this thesis.

The constructive enumeration algorithm has been used to confirm a large number of published results for the BIBDs, primarily from the list given in [45] which is an excellent summary of the current state of knowledge for these designs. Several new results have also been generated, such as the enumeration of the  $2-(10,4,4)$  designs, covered in Case Study Six. The enumeration of the  $2-(8,42,21,4,9)$  designs also led to a new result. It was discovered that the currently published number of non-isomorphic  $2-(8,4,9)$  designs, given in [45], was incorrect. The correct number is in fact 8,360,901, from which exactly 336,634,114,975 distinct designs are induced. Spence [61] has subsequently confirmed this result.

The group sizes of these designs are summarised in the table on the following page:

Group Size:	Number of non-isomorphic 2-(8,4,9) designs:	Number of distinct 2-(8,4,9) designs:
1344	1	30
1152	1	35
336	1	120
192	1	210
168	1	240
96	1	420
72	1	560
64	2	1260
48	4	3360
32	5	6300
24	9	15120
21	10	19200
16	19	47880
14	2	5760
12	28	94080
8	109	549360
7	6	34560
6	144	967680
4	625	6300000
3	1897	25495680
2	19611	395357760
1	8338423	336205215360
Totals:	8360901	336634114975

**Result 6.8.1:**

There are 8,360,901 non-isomorphic 2-(8,4,9) designs.

**3-Designs**

Section 6.6.3 examined the 3-(11,5,4) designs, and indicated that a description of the modifications required to the algorithm to enumerate 3-designs would be given in this case study.

The required modification is very simple. For BIBDs, the constraint on the rows of the incidence matrix is that the intersection between any pair of rows must be exactly  $\lambda$ . For 3-designs, the intersection between any 3-set of rows must be exactly  $\lambda$ . In general, for  $t$ -designs, the intersection between any  $t$ -set of rows must be  $\lambda$ .

Consider the incidence matrix backtracking construction of a 3-design. As row  $r$  of the incidence matrix is being built, every time a point is added to the row, it should be checked that it does not intersect with any other pair of rows,  $r_i$  and  $r_j$  ( $1 \leq i < j < r$ ), more than  $\lambda$  times. It should also be checked that every newly completed row,  $r$ , intersects with every previous pair of rows  $r_i$  and  $r_j$  ( $1 \leq i < j < r$ ) exactly  $\lambda$  times.

An important property of  $t$ -designs which can be incorporated into the algorithm, is that every  $t$ -design is also a  $(t-1)$ -design, with a different value of  $\lambda$ .

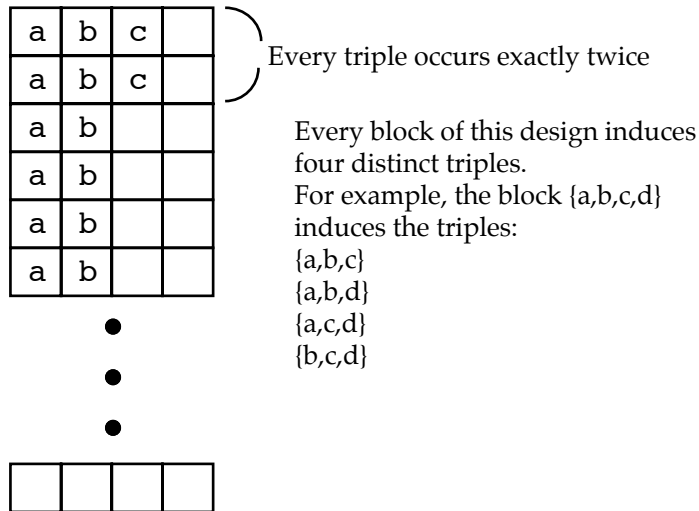
Formally, If  $(X, B)$  is a  $t$ -( $v, k, \lambda$ ) design and  $S$  is any  $s$ -element subset of  $X$ , with  $0 \leq s \leq t$ , then the number of blocks containing  $S$  is:

$$b_s = |\{b \in B: S \subseteq b\}| = \lambda \binom{v-s}{t-s} / \binom{k-s}{t-s}$$

In particular,  $b_s$  is an integer and  $(X, B)$  is a  $s$ -( $t, k, b_s$ ) design for all  $0 \leq s \leq t$ . Therefore, every 3-( $v, k, \lambda_3$ ) design is also a 2-( $v, k, \lambda_2$ ) design where:

$$\lambda_2 = \lambda_3(v-2) / (k-2)$$

For example, the 3-design given by parameters  $(v,b,r,k,\lambda) = (8,28,14,4,2)$  is also a 2-design with parameters  $(8,28,14,4,6)$ . This can be derived by examining the blocks of the 3-(8,4,2) design:



Consider a block containing the points a and b. The pair {a,b} exists in exactly  $k-2 = 2$  distinct triples in this block. As all the triples must be represented  $\lambda$  times, the pair {a,b} must occur with all other  $(v-2) = 6$  treatments exactly  $3\lambda$  times.

Let D be a 3-( $v,k,\lambda_3$ ) design which is to be enumerated, and let the corresponding 2-design have parameters 2-( $v,k,\lambda_2$ ), where  $\lambda_2$  is calculated from the formula above.

The search space can be pruned significantly by not only enforcing that the intersection of every 3-set of rows be exactly  $\lambda_3$ , but also enforcing that the intersection of every pair of rows be  $\lambda_2$ .

For example, consider the first 6 blocks of a partially complete incidence matrix for a 3-design with  $\lambda_3 = 2$ , and with  $\lambda_2 = 6$ , given below.

1	1	1	1	1	1	.....
1	1	1	1	1	0	.....
1	1	0	0	0	0	.....

If only the intersection between triples of rows were being checked, then after the construction of row 3 is completed above, the intersection between the first triple of rows would be exactly 2, which satisfies the constraint of  $\lambda_3$ . However, the corresponding 2 design must have every pair of treatments occurring exactly 6 times, yet the intersection between rows 1 and 2 above is only 5. By including the 2-design constraint as well as the 3-design constraint, row 2 would not be accepted as above, and it could be backtracked - avoiding a potentially large amount of fruitless search which would eventually find no valid extensions to the above partial design.

In addition to checking the row intersections of the corresponding 2-design, the packing constraint for BIBDs detailed in Section 5.2.1.1 can also be used to prune the search tree of partially constructed rows which can not be validly completed to satisfy the 2-design constraints. In fact, the packing constraint on pairs can be reformulated for 3-designs to give an explicit packing constraint on triples, however Gibbons [32] found the extra overhead in these calculations on triples too large to make the stronger constraint beneficial.

A sample of small 3-designs have been explicitly enumerated with the described algorithm and the results are given overleaf. The full parameter set of each design is given, including

$\lambda_2$ , and the column Nd gives the number of non-isomorphic designs, including both simple and non-simple varieties.

The Design	Parameters						Nd
	v	b	r	k	$\lambda_3$	$\lambda_2$	
3-(8,4,1)	8	14	7	4	1	3	1
3-(8,4,2)	8	28	14	4	2	6	4
3-(8,4,3)	8	42	21	4	3	9	10
3-(8,4,4)	8	56	28	4	4	12	31
3-(8,4,5)	8	70	35	4	5	15	82
3-(8,4,6)	8	84	42	4	6	18	240
3-(8,4,7)	8	98	49	4	7	21	650
3-(8,4,8)	8	112	56	4	8	24	1803
3-(8,4,9)	8	126	63	4	9	27	4763
3-(10,4,1)	10	30	12	4	1	4	1
3-(10,4,2)	10	60	24	4	2	8	132
3-(10,5,3)	10	36	18	5	3	8	7
3-(11,5,2)	11	33	15	5	2	6	0
3-(11,5,4)	11	66	30	5	4	12	1749

**4-Designs**

Just as the enumeration algorithm for 2-designs was extended to enumerate 3-designs, the 3-design enumerator can be extended in a similar fashion to enumerate 4-designs.

A 4-(v,k, $\lambda_4$ ) design must be both a 3-(v,k, $\lambda_3$ ) design and a 2-(v,k, $\lambda_2$ ) design, where:

$$\lambda_3 = \lambda_4(v-3) / (k-3) \text{ and } \lambda_2 = \lambda_3(v-2) / (k-2)$$

Thus, every pair of rows must intersect  $\lambda_2$  times, every 3-set of rows must intersect  $\lambda_3$  times, and every 4-set of rows must intersect  $\lambda_4$  times.

For example, a 4-(11,5,1) design consists of 66 blocks of size 5, has a replication number of 30, and has  $\lambda_3 = 4$  and  $\lambda_2 = 12$ . The 4-design enumeration algorithm was used to enumerate the designs with this parameter set, and in fact the 4-(11,5,1) design is unique, and has automorphism group size of 7,920.

Group Size	Number of non-isomorphic 4-(11,5,1) designs
7920	1

All 4-(11,5,2) designs were also enumerated:

Group Size	Number of non-isomorphic 4-(11,5,2) designs
7920	1
1440	1
288	1
144	1
110	1
48	1
32	1
20	1
16	2
12	1
10	1
<b>Total:</b>	<b>12</b>

## 6.8.2 CUBIC MULTIGRAPHS

### Definition:

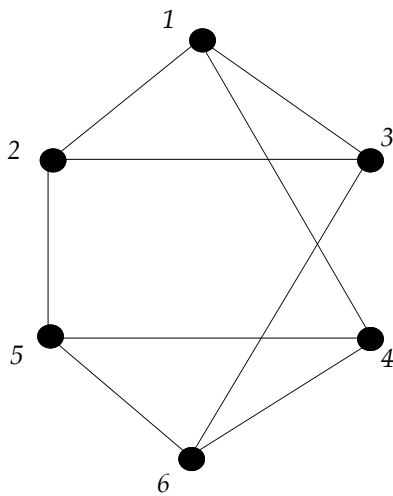
A graph,  $G$ , is a pair  $(V(G), E(G))$  where  $V(G)$  is a finite set called the *vertices* of  $G$  and  $E(G)$  is a set of unordered pairs of distinct vertices of  $G$ , called the *edges* of  $G$ .

If two vertices,  $v$  and  $w$ , are connected by an edge, ie. if  $\{v,w\} \in E(G)$ , then  $v$  and  $w$  are said to be *adjacent*. A graph consisting of  $v$  vertices is often represented as a square  $v \times v$  adjacency matrix, with the rows and columns of the matrix representing the vertices of the graph. The entry in row  $r$  and column  $c$  of the adjacency matrix is 1 if and only if vertices  $r$  and  $c$  are adjacent in the graph, otherwise the entry is 0.

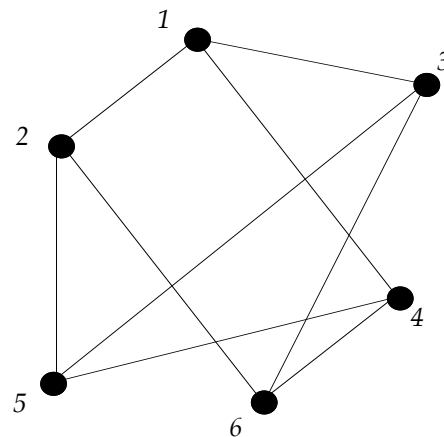
Equivalently, the graph can be represented by an incidence matrix, where the rows and columns of the incidence matrix represent the vertices and edges of the graph respectively. The incidence matrix backtracking and canonicity testing algorithms can then be applied in the same fashion as before, to generate all non-isomorphic graphs. In addition, graphs with special properties can be enumerated by having the properties encoded into the incidence matrix construction.

For example, the *degree* of a vertex,  $v$ , is the number of edges incident to  $v$ . A graph is *regular* if all vertices have the same degree, and a *cubic graph* is a regular graph where all vertices have degree 3. A graph is said to be *connected* if for any two vertices, one vertex may be reached from the other by following a sequence of edges connecting adjacent vertices.

There are 2 non-isomorphic, connected, cubic graphs on 6 vertices, as represented below:



Group size = 12



Group size = 72

Both graphs above are represented in canonical form - ie. the vertices are labelled such that the incidence matrix representation of each graph is lexicographically smaller than all other labellings.

For instance, the incidence matrix of the graph with group size 72 is given below:

```

1: 111000000
2: 100110000
3: 010001100
4: 001000011
5: 000101010
6: 000010101

```

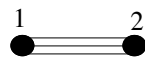
Here, each vertex is incident with exactly 3 edges, and each edge in the graph connects exactly 2 vertices. Clearly, there must be exactly two 1's in each column, representing the two vertices joined by the corresponding edge, and there must be exactly three 1's on each row, as that is the degree of the regular graph. The intersection between any pair of rows

must be either 0 or 1, indicating whether or not the corresponding pair of points are connected.

The enumeration of such regular graphs can be performed using the same incidence matrix backtracking algorithm as before, with the new constraints described above. The general purpose canonicity test which was developed in Section 5.3.2 can then be used to classify the non-isomorphic graphs. The construction of regular graphs has undergone much research, and two very efficient regular graph enumeration algorithms are given by Brinkmann [9] and Meringer [53].

In a similar way, the algorithm can be extended to handle multigraphs. A *multigraph*  $G$  is a pair  $(V(G), E(G))$  where  $V(G)$  is a finite set, the vertices of  $G$ , and  $E(G)$  is a multiset of unordered pairs of distinct vertices, called the edges of  $G$ .

Multigraphs differ from ordinary graphs in that each pair of vertices may be joined but more than one edge. Just as for ordinary graphs, if every vertex has the same degree, the graph is regular, and a multigraph for which every vertex has degree 3 is a *cubic multigraph*. Any cubic multigraph must have an even number of vertices. Below is a representation of the unique cubic multigraph on two vertices:



The unique multigraph on two vertices with 3 edges

The incidence matrix representing this design is:

Row 1: 111  
Row 2: 111

There are 2 connected cubic multigraphs on 4 vertices:



The two connected cubic multigraphs on 4 vertices  
Their corresponding incidence matrices:

Row 1: 111000	Row 1: 111000
Row 2: 100110	Row 2: 110100
Row 3: 010101	Row 3: 001011
Row 4: 001011	Row 4: 000111

In order to enumerate the cubic multigraphs, the intersection between any two rows of the incidence matrix may be any value from the set  $\{0, 1, 2, \dots, E\}$  where  $E$  is the number of edges in the graph. In addition, repeated edges in the graph are represented in the incidence matrix by two or more adjacent, identical columns.

In [58], Royle gives the following table of the number,  $N_c$ , of non-isomorphic, connected, cubic multigraphs on  $n$  vertices for  $2 \leq n \leq 16$ :

$n$	2	4	6	8	10	12	14	16
$N_c$	1	2	6	20	91	509	3608	31856

The modified algorithm described above was used to confirm each of these numbers, and also used to enumerate the cubic multigraphs on 18 vertices, of which there are 340,416, with the following group order spectrum:

Group Size	Number of non-isomorphic cubic multigraphs
384	2
288	1
256	6
216	1
128	35
96	6
72	1
64	267
48	18
36	2
32	1439
24	5
18	4
16	5605
12	61
8	17216
6	56
4	44754
3	4
2	103353
1	167580
<i>Total:</i>	340416

### 6.8.3 SYMMETRIC DESIGNS

A balanced incomplete block design, or BIBD, for which the number of points,  $v$ , equals the number of blocks,  $b$ , is a symmetric  $(v,k,\lambda)$  design, or simply a symmetric design. The trivial necessary condition for the existence of a BIBD, ie.  $vb=rk$  must hold, and so symmetric designs have  $v=b$  and  $r=k$ , ie. the block size is the same as the replication number.

If  $D = (V, B)$  is any BIBD, then the dual of  $D$  is  $D' = (B, V)$ , where  $b \in B$  is contained in  $v \in V$  if and only if  $v$  is contained in  $b$ , in  $D$ . The dual,  $D'$ , of a design,  $D$ , is a valid design if and only if  $D$  is symmetric. In this case, the parameters of  $D'$  are identical to the parameters of  $D$ , yet  $D$  and  $D'$  are not necessarily isomorphic.

#### Symmetric Constraint

Any valid symmetric design has a dual design, which is also a valid symmetric design. If  $D$  is a symmetric design with incidence matrix  $I$ , the incidence matrix,  $I'$ , of the dual design,  $D'$ , is formed by transposing  $I$ . This swaps the rows and columns of the incidence matrix, which equivalently interchanges the blocks and points of the design.

Clearly, for any symmetric design, not only must every pair of rows of the incidence matrix intersect in exactly  $\lambda$  points, but every pair of columns must also intersect in exactly  $\lambda$  points, or else the dual design would not be a valid symmetric design. This is a very strong constraint, and is straightforward to incorporate into the backtrack search when generating symmetric designs. When constructing a row, say  $r$ , of the incidence matrix, a point can only be placed in column  $c$  if the intersection between row  $r$  and all other rows containing points in column  $c$  is less than  $\lambda$ , and if the intersection between column  $c$  and all other columns containing a point on row  $r$  is less than  $\lambda$ . This new symmetric intersection constraint on the columns is illustrated overleaf.



	x	y	z	c	
				1	
				1	
				1	
	1	1	1	?	row r

Row r has been partially constructed up to column c, and it must be determined whether or not a point can be placed in this column. The only other points currently placed on row r are in columns x, y and z. If the intersection between column c and any of these 3 columns is already equal to  $\lambda$ , then no point can be placed in column c. If appropriate data structures are maintained, this constraint can be quickly checked.

For example, consider the construction of row 5 of the incidence matrix of a symmetric 2-(19,19,9,9,4) design, displayed on the right. The intersection between every pair of rows and every pair of columns must be exactly  $\lambda = 4$ .

```

1: 1111111110000000000
2: 1111000001111100000
3: 1110100001000011110
4: 1100011000110011001
5: 1000000000000000000
6: 0000000000000000000
7: 0000000000000000000
8: 0000000000000000000
9: 0000000000000000000
10: 0000000000000000000
11: 0000000000000000000
12: 0000000000000000000
13: 0000000000000000000
14: 0000000000000000000
15: 0000000000000000000
16: 0000000000000000000
17: 0000000000000000000
18: 0000000000000000000
19: 0000000000000000000
    
```

The first point has just been placed in the first column of row 5. Without the symmetric constraint, a bit will be placed in the second column, the row will then be completed, and the partial design will be extended to deeper rows of the incidence matrix. However, all this searching is pointless, for as long as there are 2 points placed in the first two columns of row 5, no symmetric design can be validly constructed. This is because the intersection between the first two columns is greater than 4, and so the dual design would not be valid.

Although the symmetric constraint is not necessary, it is able to prune very large portions of the search space very cheaply. This constraint was implemented, and incorporated with the constructive enumeration algorithm to examine some classes of symmetric designs. A summary of the results follows.

**Results**

The number of non-isomorphic symmetric designs was confirmed for several parameter sets. For example, the 6 non-isomorphic 2-(19,19,9,9,4) designs which were enumerated by Gibbons in [32] were exhaustively constructed, tested for canonicity and had their group sizes calculated in just 10 seconds. The group sizes are given below:

Group Size	Number of non-isomorphic 2-(19,19,9,9,4) designs
171	1
72	1
24	1
9	1
8	1
6	1

In [45] and [59], the number of non-isomorphic 2-(23,23,11,11,5) symmetric designs, is given as 1102, and in [60] it is given as 1103. In fact, the correct number of designs, 1106, was calculated by Spence [59, 61]. The enumeration algorithm developed in this thesis along with the symmetric constraint discussed in this section was used to enumerate these designs, and this number was confirmed. The group orders are tabulated on the next page:

Group Size:	Number of non-isomorphic 2-(23,23,11,11,5) designs:
660	1
253	1
60	2
55	2
12	4
11	1
6	14
5	10
4	17
3	35
2	110
1	909
Total:	1106

A selection of other enumerated symmetric designs are the 2-(15,15,7,7,3), 2-(16,16,6,6,2) and 2-(25,25,9,9,3) designs. In each case, the results confirm the previously published results in [45], and are summarised below:

Group Size	Number of non-isomorphic 2-(15,7,3) designs
20160	1
576	1
168	2
96	1
Total:	5

Group Size	Number of non-isomorphic 2-(16,6,2) designs
11520	1
768	1
384	1
Total:	3

Group Size	Number of non-isomorphic 2-(25,9,3) designs
24	2
12	4
8	2
6	2
4	4
3	16
2	10
1	38
Total:	78

#### 6.8.4 N-SIMPLE DESIGNS

For any  $t$ -( $v,k,\lambda$ ) design,  $D$ , if any two blocks,  $a$  and  $b$ , of the design intersect in  $k$  points then  $a$  and  $b$  must be the same block and  $D$  is said to contain repeated blocks. Often it is desirable to examine only simple designs, which are designs that do not contain repeated blocks. A  $t$ -( $v,k,\lambda$ ) design is *simple* if no two blocks intersect in  $k$  points. This case study presents a generalisation to the idea of constraining the block intersection types of designs.

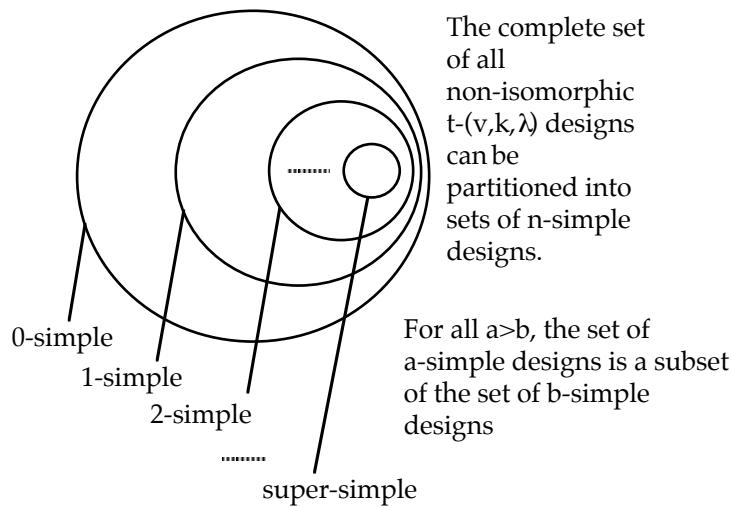
**Definition:**

A  $t$ -( $v,k,\lambda$ ) design is *n-simple* if no two blocks intersect in more than  $k-n$  points. Clearly, the set of all  $k+1$ -simple designs is a subset of the set of all  $k$ -simple designs.

For example, a  $t$ -( $v,k,\lambda$ ) design is 0-simple if no two blocks of the design intersect in more than  $k-0 = k$  points. Obviously, no two blocks can intersect in more than  $k$  points, as the size of each block is  $k$ . Therefore, the set of all 0-simple designs for a particular parameter set consists of all non-isomorphic designs with the given parameters, as there are no constraints on the allowed intersection types between the blocks.

A design is 1-simple if no two blocks intersect in more than  $k-1$  points - in other words if no two blocks intersect in  $k$  points. The set of all 1-simple designs for a given parameter set, consists of only those designs which are simple in the ordinary sense - ie. contain no repeated blocks. The 1-simple designs are a subset of the 0-simple designs.

The set of all  $n$ -simple designs is contained within the set of all  $n-1$  simple designs as illustrated below:



As illustrated in the diagram above, the number of  $n$ -simple designs diminishes as  $n$  increases, because the constraints on the allowed block intersection types become tighter. For any given parameter set, there is a value  $s$ , such that the set of  $s$ -simple designs is non-empty, yet there are no  $(s+1)$ -simple designs. In a sense, this is the set of all designs which are as "simple" as possible. In this case, the  $s$ -simple designs are called *super-simple*.

### Implementation

The constraint on block intersection types can be incorporated into the incidence matrix backtracking algorithm, and so the set of  $(n+1)$ -simple designs can be generated more rapidly than the set of  $n$ -simple designs.

The implementation maintains a structure which stores the column intersections as well as the row intersections for a given incidence matrix. For the construction of the  $n$ -simple  $t-(v,k,\lambda)$  designs, no two columns of the incidence matrix are permitted to intersect in more than  $k-n$  points, and this constraint is checked each time a new point is added to a block of the design.

Using this implementation, several classes of  $n$ -simple BIBDs were enumerated, and a summary of the results follows.

### Results

#### *n*-Simple Design Classifications

For each parameter set given in the following tables, the enumerated designs range from 0-simple to super-simple. The set of 0-simple designs corresponds to the complete set of non-isomorphic designs, with no block intersection constraints. For each parameter set, the  $m$ -simple designs, where  $m$  is the largest value listed, are in fact the super-simple designs. In other words, for that parameter set, no  $(m+1)$ -simple designs exist.

The n-simple 2-(12,22,11,6,5) designs:

Group Sizes	0-simple	1-simple	2-simple	3-simple
7920	1	1	1	1
120	1	1		
60	2	2	2	
55	1	1		
18	1	1	1	
16	2	2	2	
12	4	4	4	
11	2	2	2	
10	2	2		
8	1	1	1	
6	15	15	14	
5	15	15	6	
4	13	13	6	
3	60	60	57	
2	368	368	178	
1	11115	11115	8476	
Total:	11603	11603	8750	1

The n-simple 2-(11,22,10,5,4) designs:

Group Sizes	0-simple	1-simple	2-simple
660	1		
120	3		
110	1	1	1
24	4		
20	2	2	
12	5		
11	1	1	1
10	5	3	2
8	9		
6	20	1	
5	5	2	1
4	67	36	2
3	27	10	1
2	430	280	27
1	3813	3401	414
Total:	4393	3737	449

The n-simple 2-(13,26,8,4,2) designs:

Group Sizes	0-simple	1-simple	2-simple
5616	1		
576	1		
144	3		
108	1		
78	1	1	1
48	1		
39	1	1	1
36	1		
32	3		
24	5	2	1
18	1		
16	6		
12	5	1	
8	20	5	
6	37	19	6
4	84	34	2
3	47	38	17
2	470	196	8
1	1773	1279	119
Total:	2461	1576	155

The n-simple 2-(9,36,12,3,3) designs:

Group Size	0-simple	1-simple
1296	1	
432	1	
54	2	1
48	1	
36	1	
24	4	1
18	3	1
16	3	1
12	8	
9	4	1
8	4	
6	39	5
4	37	4
3	85	6
2	794	37
1	21534	275
Total:	22521	332

#### *New Lower Bound on the Number of 2-(16,4,2) Designs*

In [45], the number of non-isomorphic 2-(16,4,2) designs is given as 22859. The constructive enumeration algorithm developed in this thesis was used to examine these designs, and it was quickly discovered that there are in fact many more than this number.

Colbourn [18] has indicated that the super-simple 2-(16,4,2) designs may have interesting applications, and so the algorithm described above was used to investigate this class. Super-simple 2-(16,4,2) designs are such that no two blocks intersect in 3 or 4 points. As a side effect of this analysis, a lower bound was generated for the number of non-isomorphic 2-(16,4,2) designs.

Let  $N^{(16,4,2)}$  denote the set of all non-isomorphic 2-(16,4,2) designs. The set of all non-isomorphic super-simple 2-(16,4,2) designs is a subset of  $N^{(16,4,2)}$ , and its size is therefore a lower bound on the size of  $N^{(16,4,2)}$  - although this bound is not necessarily tight.

A very small section of the search space of super-simple 2-(16,4,2) designs was enumerated, yielding more than 2.2 million non-isomorphic designs. This number is therefore a weak lower bound on the true number of super-simple designs.

The actual number of non-isomorphic 2-(16,4,2) designs, which is the size of the set  $N^{(16,4,2)}$ , will be many orders of magnitude greater than the number of super simple 2-(16,4,2) designs, which in turn will be many orders of magnitude greater than 2.2 million. As a result, the following statement can be made regarding the set  $N^{(16,4,2)}$ , which updates the current state of knowledge for the 2-(16,4,2) designs:

**Result 6.8.2:**

The number of non-isomorphic 2-(16,4,2) designs is at least, and in fact many orders of magnitude greater than,  $2.2 \times 10^6$ .



# Chapter 7

## Summary of Results

This chapter summarises a selection of the results computed in this thesis. A large proportion of the results presented in this chapter are new, in the sense that either they had not been calculated prior to this thesis, or they correct a result currently published in the literature. A number of the tables have already been given in previous chapters, alongside their relevant sections, and are repeated here for convenience. In each case, a reference is given to the corresponding section of work from which the result was calculated.

The results in this chapter are organised into the following main sections:

- 7.1 Block Designs
- 7.2 Weakly Union Free Twofold Triple Systems
- 7.3 Mendelsohn Triple Systems
- 7.4 Design Families
- 7.5 Configurations
- 7.6 Miscellaneous

### **7.1 BLOCK DESIGNS**

#### **7.1.1 2-DESIGNS (BIBDS)**

This section summarises the results of several BIBD enumerations performed with the constructive enumeration algorithm developed in this thesis [Chapter 5]. In particular, the results of the 2-(10,4,4) and 2-(8,4,9) enumerations are given in the tables on the next page which display the number of non-isomorphic designs by automorphism group size, and the number of induced distinct designs.



**The 2-(10,30,12,4,4) Designs** [Section 6.6.2]**Result 6.6.1:**

There are exactly 13,769,944 non-isomorphic 2-(10,4,4) designs. 10,081,743 of these are simple, and 3,688,201 are non-simple.

Group Size:	Number of non-isomorphic 2-(10,4,4) designs:	Number of distinct 2-(10,4,4) designs:
1440	1	2520
720	1	5040
192	1	18900
96	2	75600
64	2	113400
48	3	226800
36	1	100800
32	12	1360800
24	10	1512000
20	1	181440
18	4	806400
16	16	3628800
12	29	8769600
10	9	3265920
9	6	2419200
8	104	47174400
6	113	68342400
5	13	9434880
4	662	600566400
3	1041	1259193600
2	22113	40121827200
1	13745800	49880759040000
Totals:	13769944	49922888066100

**The 2-(8,42,21,4,9) Designs** [Section 6.8.1]**Result 6.8.1:**

There are 8,360,901 non-isomorphic 2-(8,4,9) designs.

Group Size:	Number of non-isomorphic 2-(8,4,9) designs:	Number of distinct 2-(8,4,9) designs:
1344	1	30
1152	1	35
336	1	120
192	1	210
168	1	240
96	1	420
72	1	560
64	2	1260
48	4	3360
32	5	6300
24	9	15120
21	10	19200
16	19	47880
14	2	5760
12	28	94080
8	109	549360
7	6	34560
6	144	967680
4	625	6300000
3	1897	25495680
2	19611	395357760
1	8338423	336205215360
Totals:	8360901	336634114975

**The 2-(16,40,10,4,2) Designs** [Section 6.8.4]

The currently published result [45] for the number of non-isomorphic 2-(16,40,10,4,2) designs is 22,859, which is incorrect.

**Result 6.8.2:**

The number of non-isomorphic 2-(16,4,2) designs is at least, and in fact many orders of magnitude greater than,  $2.2 \times 10^6$ .

**7.1.2 3-DESIGNS**

A sample of small 3-designs have been explicitly enumerated [Section 6.8.1] and the results are given below. The full parameter set of each design is given, including  $\lambda_2$ . The column Nd gives the total number of valid non-isomorphic designs.

The Design	Parameters						Nd
	v	B	r	k	$\lambda_3$	$\lambda_2$	
3-(8,4,1)	8	14	7	4	1	3	1
3-(8,4,2)	8	28	14	4	2	6	4
3-(8,4,3)	8	42	21	4	3	9	10
3-(8,4,4)	8	56	28	4	4	12	31
3-(8,4,5)	8	70	35	4	5	15	82
3-(8,4,6)	8	84	42	4	6	18	240
3-(8,4,7)	8	98	49	4	7	21	650
3-(8,4,8)	8	112	56	4	8	24	1803
3-(8,4,9)	8	126	63	4	9	27	4763
3-(10,4,1)	10	30	12	4	1	4	1
3-(10,4,2)	10	60	24	4	2	8	132
3-(10,5,3)	10	36	18	5	3	8	7
3-(11,5,2)	11	33	15	5	2	6	0
3-(11,5,4)	11	66	30	5	4	12	1749

**7.1.3 4-DESIGNS**

The 4-design enumeration algorithm [Section 6.8.1] was used to enumerate all 4-(11,5,1) and 4-(11,5,2) designs, and the number of non-isomorphic designs of each type are given below. Also given below is the number of 4-(12,6,4) designs which was calculated in collaboration with Rudi Mathon [21], as described in Case Study Six of Chapter 6.

The Design	Parameters						Nd	
	v	b	r	k	$\lambda_4$	$\lambda_3$		$\lambda_2$
4-(11,5,1)	11	66	30	5	1	4	12	1
4-(11,5,2)	11	132	60	5	2	8	24	12
4-(12,6,4)	12	132	66	6	4	12	30	11

**7.2 WEAKLY UNION FREE TWOFOLD TRIPLE SYSTEMS****7.2.1 WUF-TTS(12)** [Section 6.1.6]**Result 6.1.1:**

There are no weakly union free twofold triple systems of order 12.

**7.2.2 WUF-TTS(13)** [Section 6.1.7]**Result 6.1.2:**

There are exactly two non-isomorphic weakly union free twofold triple systems of order 13, with automorphism group sizes of 156 and 39.



## 7.3 MENDELSON TRIPLE SYSTEMS

### 7.3.1 THE MTS(10,1) DESIGNS [Section 6.5.7]

**Result 6.5.1:**

There are exactly 143 inequivalent MTS(10,1) designs.

### 7.3.2 THE MTS(12,1) DESIGNS [Section 6.5.7]

**Result 6.5.2:**

There are exactly 4,905,693 inequivalent MTS(12,1) designs.

The number of inequivalent MTS( $v$ ,1) designs, for all admissible orders  $v$ , where  $v \leq 12$ :

$v$	3	4	6	7	9	10	12
MTS( $v$ )	1	1	0	3	18	143	4905693

The number of non-isomorphic, orientable, TTS(12) designs, the number of distinct designs induced, and the number of inequivalent MTS(12,1) designs admitted, are given in the following table. The results are tabulated in the following four columns:

G	<i>The automorphism group size of the corresponding designs.</i>
Nd	<i>The number of non-isomorphic orientable TTS(12) designs, which admit at least one MTS(12) design, by group size.</i>
MTS	<i>The number of inequivalent Mendelsohn Triple Systems admitted by the underlying TTS(12) designs of the corresponding group size</i>
Distinct	<i>The number of distinct orientable TTS(12) designs.</i>

### MTS(12)

IGI	Nd	MTS	Distinct
1536	1	3	311850
576	1	2	831600
432	1	7	1108800
192	3	4	7484400
144	1	4	3326400
128	2	5	7484400
72	2	6	13305600
64	2	6	14968800
54	1	1	8870400
48	6	12	59875200
36	1	2	13305600
32	9	15	134719200
24	7	15	139708800
18	2	3	53222400
16	24	40	718502400
12	15	38	598752000
9	1	1	53222400
8	77	173	4610390400
6	106	196	8462361600
4	482	867	57719692800
3	531	647	84783283200
2	11011	15003	2637143308800
1	4679953	4888643	2241704974924800
<b>Totals:</b>	<b>4692239</b>	<b>4905693</b>	<b>2244499522961850</b>

### 7.3.3 THE MTS(v,1) DESIGNS, $v < 12$ [Section 6.5.7]

The tables below give the number of non-isomorphic, orientable, TTS(v) designs, the number of distinct designs induced, and the number of inequivalent MTS(v,1) designs admitted, for all admissible orders,  $v < 12$ . The tables are organised into the same column headings as the table on the previous page for the MTS(12,1) designs.

#### MTS(10)

IGI	Nd	MTS	Distinct
108	2	4	67200
24	1	2	151200
21	1	1	172800
18	1	1	201600
12	2	3	604800
8	1	1	453600
6	5	5	3024000
4	7	9	6350400
3	11	11	13305600
2	21	23	38102400
1	82	83	297561600
<b>Totals:</b>	<b>134</b>	<b>143</b>	<b>359995200</b>

#### MTS(9)

IGI	Nd	MTS	Distinct
432	1	1	840
108	1	1	3360
80	1	1	4536
32	1	1	11340
24	2	3	30240
8	2	2	90720
6	3	4	181440
4	2	2	181440
2	3	3	544320
<b>Totals:</b>	<b>16</b>	<b>18</b>	<b>1048236</b>

#### MTS(7)

IGI	Nd	MTS	Distinct
168	1	1	30
48	1	1	105
42	1	1	120
<b>Totals:</b>	<b>3</b>	<b>3</b>	<b>255</b>

#### MTS(6)

IGI	Nd	MTS	Distinct
	0	0	0
<b>Totals:</b>	<b>0</b>	<b>0</b>	<b>0</b>

#### MTS(4)

IGI	Nd	MTS	Distinct
24	1	1	1
<b>Totals:</b>	<b>1</b>	<b>1</b>	<b>1</b>

#### MTS(3)

IGI	Nd	MTS	Distinct
6	1	1	1
<b>Totals:</b>	<b>1</b>	<b>1</b>	<b>1</b>

## 7.4 DESIGN FAMILIES

### 7.4.1 THE 2-(7,3, $\lambda$ ) DESIGN FAMILY

The tables below give the number of non-isomorphic, and the number of distinct, 2-(7,3, $\lambda$ ) designs by automorphism group size, for  $1 \leq \lambda \leq 16$  [Section 6.3.5]:

$\lambda = 1$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	1	30
TOTAL	1	30

$\lambda = 2$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	1	30
48	1	105
42	1	120
24	1	210
TOTAL	4	465

$\lambda = 3$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	1	30
144	1	35
42	1	120
24	1	210
21	1	240
12	1	420
8	1	630
6	2	1680
3	1	1680
TOTAL	10	5045

$\lambda = 4$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	2	60
48	1	105
42	1	120
24	4	840
21	1	240
16	2	630
12	3	1260
8	1	630
6	4	3360
4	3	3780
3	6	10080
2	6	15120
1	1	5040
TOTAL	35	41265

$\lambda = 5$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
5040	1	1
168	1	30
60	1	84
24	7	1470
21	3	720
20	1	252
16	1	315
12	3	1260
8	4	2520
6	8	6720
4	5	6300
3	19	31920
2	25	63000
1	30	151200
TOTAL	109	265792

$\lambda = 6$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	2	60
144	1	35
72	1	70
48	1	105
42	2	240
24	8	1680
21	3	720
16	2	630
12	7	2940
8	8	5040
6	22	18480
4	19	23940
3	45	75600
2	89	224280
1	208	1048320
TOTAL	418	1402140

$\lambda = 7$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	2	60
48	1	105
42	1	120
24	10	2100
21	5	1200
16	2	630
12	8	3360
8	10	6300
6	30	25200
4	37	46620
3	103	173040
2	214	539280
1	1085	5468400
TOTAL	1508	6266415

$\lambda = 8$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	3	90
144	1	35
48	2	210
42	2	240
24	15	3150
21	5	1200
16	4	1260
12	17	7140
8	24	15120
6	49	41160
4	79	99540
3	191	320880
2	558	1406160
1	4463	22493520
TOTAL	5413	24389705

$\lambda = 9$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	3	90
144	1	35
72	1	70
48	1	105
42	2	240
36	1	140
24	17	3570
21	7	1680
16	3	945
12	22	9240
8	26	16380
6	69	57960
4	113	142380
3	353	593040
2	1123	2829960
1	16043	80856720
TOTAL	17785	84512555

$\lambda = 10$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
5040	1	1
168	2	60
60	2	168
48	1	105
42	1	120
24	23	4830
21	10	2400
20	3	756
16	6	1890
12	25	10500
10	1	504
8	45	28350
6	110	92400
4	212	267120
3	598	1004640
2	2418	6093360
1	51155	257821200
TOTAL	54613	265328404



$\lambda = 11$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	3	90
144	1	35
72	1	70
48	1	105
42	2	240
24	23	4830
21	11	2640
16	5	1575
12	32	13440
8	48	30240
6	143	120120
4	295	371700
3	971	1631280
2	4290	10810800
1	149292	752431680
TOTAL	155118	765418845

$\lambda = 12$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	4	120
144	2	70
72	1	70
48	2	210
42	3	360
36	1	140
24	32	6720
21	12	2880
16	8	2520
12	54	22680
8	82	51660
6	206	173040
4	505	636300
3	1511	2538480
2	8164	20573280
1	402404	2028116160
TOTAL	412991	2052124690

$\lambda = 13$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	4	120
144	1	35
48	2	210
42	2	240
24	35	7350
21	15	3600
16	7	2205
12	57	23940
8	90	56700
6	249	209160
4	651	820260
3	2305	3872400
2	13406	33783120
1	1016305	5122177200
TOTAL	1033129	5160956540

$\lambda = 14$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	4	120
144	1	35
72	1	70
48	2	210
42	3	360
36	1	140
24	40	8400
21	17	4080
16	11	3465
12	71	29820
8	128	80640
6	353	296520
4	1016	1280160
3	3360	5644800
2	23455	59106600
1	2421129	12202490160
TOTAL	2449592	12268945580

$\lambda = 15$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
5040	1	1
168	3	90
144	1	35
60	3	252
48	1	105
42	2	240
36	3	420
24	46	9660
21	21	5040
20	4	1008
16	9	2835
12	83	34860
10	4	2016
8	134	84420
6	422	354480
4	1266	1595160
3	4866	8174880
2	36292	91455840
1	5485096	27644883840
TOTAL	5528257	27746605182

$\lambda = 16$		
<i>Group Size</i>	<i>Nd</i>	<i>Distinct</i>
168	5	150
144	1	35
72	1	70
48	3	315
42	3	360
24	57	11970
21	22	5280
16	15	4725
12	107	44940
8	200	126000
6	554	465360
4	1938	2441880
3	6815	11449200
2	59388	149657760
1	11875518	59852610720
TOTAL	11944627	60016818765

### 7.4.2 THE 2-(7,3, $\lambda$ ) $\rightarrow$ 3-(8,4, $\lambda$ ) EXTENSION

The tables below give the number of non-isomorphic and distinct 2-(7,3, $\lambda$ ) and 3-(8,4, $\lambda$ ) designs by group size, for  $1 \leq \lambda \leq 14$  [Section 6.4.4]. Note that the number of distinct 2-(7,3, $\lambda$ ) designs equals the number of distinct 3-(8,4, $\lambda$ ) designs, for all values of  $\lambda$ .

The results for each class of designs are tabulated in three columns under the headings of:

- |G|               - automorphism group size
- Nd               - number of non-isomorphic designs
- Distinct       - number of distinct designs

$\lambda = 1$					
The 2-(7,3,1) Designs			The 3-(8,4,1) Designs		
G	Nd	Distinct	G	Nd	Distinct
168	1	30	1344	1	30
TOTAL	1	30	TOTAL	1	30

$\lambda = 2$					
The 2-(7,3,2) Designs			The 3-(8,4,2) Designs		
G	Nd	Distinct	G	Nd	Distinct
168	1	30	1344	1	30
48	1	105	384	1	105
42	1	120	336	1	120
24	1	210	192	1	210
TOTAL	4	465	TOTAL	4	465

$\lambda = 3$					
The 2-(7,3,3) Designs			The 3-(8,4,3) Designs		
G	Nd	Distinct	G	Nd	Distinct
168	1	30	1344	1	30
144	1	35	1152	1	35
42	1	120	336	1	120
24	1	210	192	1	210
21	1	240	168	1	240
12	1	420	96	1	420
8	1	630	64	1	630
6	2	1680	48	2	1680
3	1	1680	24	1	1680
TOTAL	10	5045	TOTAL	10	5045

$\lambda = 4$					
The 2-(7,3,4) Designs			The 3-(8,4,4) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	2	60	1344	2	60
48	1	105	384	1	105
42	1	120	336	1	120
24	4	840	192	4	840
21	1	240	168	1	240
16	2	630	128	2	630
12	3	1260	96	1	420
8	1	630	64	1	630
6	4	3360	48	3	2520
4	3	3780	32	1	1260
3	6	10080	24	8	13440
2	6	15120	16	3	7560
1	1	5040	12	1	3360
			8	2	10080
TOTAL	35	41265	TOTAL	31	41265

$\lambda = 5$					
The 2-(7,3,5) Designs			The 3-(8,4,5) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
5040	1	1	40320	1	1
168	1	30	1344	1	30
60	1	84	192	6	1260
24	7	1470	168	3	720
21	3	720	128	1	315
20	1	252	120	1	336
16	1	315	96	3	1260
12	3	1260	64	3	1890
8	4	2520	48	7	5880
6	8	6720	32	3	3780
4	5	6300	24	13	21840
3	19	31920	16	12	30240
2	25	63000	12	2	6720
1	30	151200	8	12	60480
			6	6	40320
			4	7	70560
			2	1	20160
TOTAL	109	265792	TOTAL	82	265792

$\lambda = 6$					
The 2-(7,3,6) Designs			The 3-(8,4,6) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	2	60	1344	2	60
144	1	35	1152	1	35
72	1	70	384	1	105
48	1	105	336	2	240
42	2	240	192	7	1470
24	8	1680	168	3	720
21	3	720	144	1	280
16	2	630	128	2	630
12	7	2940	96	4	1680
8	8	5040	64	6	3780
6	22	18480	48	12	10080
4	19	23940	32	9	11340
3	45	75600	24	31	52080
2	89	224280	16	27	68040
1	208	1048320	12	10	33600
			8	43	216720
			6	17	114240
			4	36	362880
			2	26	524160
TOTAL	418	1402140	TOTAL	240	1402140

$\lambda = 7$					
The 2-(7,3,7) Designs			The 3-(8,4,7) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	2	60	1344	2	60
48	1	105	384	1	105
42	1	120	336	1	120
24	10	2100	192	9	1890
21	5	1200	168	5	1200
16	2	630	128	2	630
12	8	3360	96	6	2520
8	10	6300	64	9	5670
6	30	25200	48	20	16800
4	37	46620	32	15	18900
3	103	173040	24	54	90720
2	214	539280	16	57	143640
1	1085	5468400	12	11	36960
			8	116	584640
			6	51	342720
			4	128	1290240
			2	141	2842560
			1	22	887040
TOTAL	1508	6266415	TOTAL	650	6266415

$\lambda = 8$					
The 2-(7,3,8) Designs			The 3-(8,4,8) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	3	90	1344	3	90
144	1	35	1152	1	35
48	2	210	384	2	210
42	2	240	336	2	240
24	15	3150	192	13	2730
21	5	1200	168	5	1200
16	4	1260	128	4	1260
12	17	7140	96	9	3780
8	24	15120	64	16	10080
6	49	41160	48	27	22680
4	79	99540	32	28	35280
3	191	320880	24	91	152880
2	558	1406160	16	111	279720
1	4463	22493520	12	24	80640
			8	266	1340640
			6	108	725760
			4	374	3769920
			2	547	11027520
			1	172	6935040
TOTAL	5413	24389705	TOTAL	1803	24389705

$\lambda = 9$					
The 2-(7,3,9) Designs			The 3-(8,4,9) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	3	90	1344	3	90
144	1	35	1152	1	35
72	1	70	384	1	105
48	1	105	336	2	240
42	2	240	192	15	3150
36	1	140	168	7	1680
24	17	3570	144	1	280
21	7	1680	128	3	945
16	3	945	96	13	5460
12	22	9240	72	1	560
8	26	16380	64	21	13230
6	69	57960	48	37	31080
4	113	142380	32	43	54180
3	353	593040	24	139	233520
2	1123	2829960	16	188	473760
1	16043	80856720	12	33	110880
			8	540	2721600
			6	222	1491840
			4	894	9011520
			2	1708	34433280
			1	891	35925120
TOTAL	17785	84512555	TOTAL	4763	84512555

$\lambda = 10$					
The 2-(7,3,10) Designs			The 3-(8,4,10) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
5040	1	1	40320	1	1
168	2	60	1344	2	60
60	2	168	384	1	105
48	1	105	336	1	120
42	1	120	192	21	4410
24	23	4830	168	10	2400
21	10	2400	128	6	1890
20	3	756	120	2	672
16	6	1890	96	16	6720
12	25	10500	64	29	18270
10	1	504	48	54	45360
8	45	28350	32	71	89460
6	110	92400	24	203	341040
4	212	267120	20	1	2016
3	598	1004640	16	323	813960
2	2418	6093360	12	58	194880
1	51155	257821200	8	1010	5090400
			6	404	2714880
			4	2019	20351520
			2	4585	92433600
			1	3552	143216640
TOTAL	54613	265328404	TOTAL	12369	265328404

$\lambda = 11$					
The 2-(7,3,11) Designs			The 3-(8,4,11) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	3	90	1344	3	90
144	1	35	1152	1	35
72	1	70	384	1	105
48	1	105	336	2	240
42	2	240	192	21	4410
24	23	4830	168	11	2640
21	11	2640	144	1	280
16	5	1575	128	5	1575
12	32	13440	96	21	8820
8	48	30240	64	37	23310
6	143	120120	48	68	57120
4	295	371700	32	97	122220
3	971	1631280	24	292	490560
2	4290	10810800	16	498	1254960
1	149292	752431680	12	76	255360
			8	1780	8971200
			6	690	4636800
			4	4030	40622400
			2	11125	224280000
			1	12021	484686720
TOTAL	155118	765418845	TOTAL	30780	765418845

$\lambda = 12$					
The 2-(7,3,12) Designs			The 3-(8,4,12) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	4	120	1344	4	120
144	2	70	1152	2	70
72	1	70	384	2	210
48	2	210	336	3	360
42	3	360	192	28	5880
36	1	140	168	12	2880
24	32	6720	144	1	280
21	12	2880	128	8	2520
16	8	2520	96	28	11760
12	54	22680	72	1	560
8	82	51660	64	55	34650
6	206	173040	48	87	73080
4	505	636300	32	146	183960
3	1511	2538480	24	407	683760
2	8164	20573280	16	759	1912680
1	402404	2028116160	12	121	406560
			8	2983	15034320
			6	1128	7580160
			4	7797	78593760
			2	24763	499222080
			1	35922	1448375040
TOTAL	412991	2052124690	TOTAL	74257	2052124690

$\lambda = 13$					
The 2-(7,3,13) Designs			The 3-(8,4,13) Designs		
$ G $	$Nd$	<i>Distinct</i>	$ G $	$Nd$	<i>Distinct</i>
168	4	120	1344	4	120
144	1	35	1152	1	35
48	2	210	384	2	210
42	2	240	336	2	240
24	35	7350	192	31	6510
21	15	3600	168	15	3600
16	7	2205	128	7	2205
12	57	23940	96	35	14700
8	90	56700	64	64	40320
6	249	209160	48	110	92400
4	651	820260	32	192	241920
3	2305	3872400	24	544	913920
2	13406	33783120	16	1089	2744280
1	1016305	5122177200	12	143	480480
			8	4789	24136560
			6	1783	11981760
			4	13936	140474880
			2	51583	1039913280
			1	97716	3939909120
TOTAL	1033129	5160956540	TOTAL	172046	5160956540



$\lambda = 14$					
The 2-(7,3,14) Designs			The 3-(8,4,14) Designs		
G	Nd	Distinct	G	Nd	Distinct
168	4	120	1344	4	120
144	1	35	1152	1	35
72	1	70	384	2	210
48	2	210	336	3	360
42	3	360	192	37	7770
36	1	140	168	17	4080
24	40	8400	144	1	280
21	17	4080	128	11	3465
16	11	3465	96	41	17220
12	71	29820	72	1	560
8	128	80640	64	82	51660
6	353	296520	48	138	115920
4	1016	1280160	32	269	338940
3	3360	5644800	24	718	1206240
2	23455	59106600	16	1592	4011840
1	2421129	12202490160	12	217	729120
			8	7406	37326240
			6	2671	17949120
			4	24344	245387520
			2	101693	2050130880
			1	245825	9911664000
TOTAL	2449592	12268945580	TOTAL	385073	12268945580

### 7.4.3 THE 2-(10,4,4), 3-(11,5,4), 4-(12,6,4) DESIGNS

Number of non-isomorphic designs [Section 6.6.2, 6.6.3, 6.6.4]:

Design	2-(10,4,4)	3-(11,5,4)	4-(12,6,4)
Non-isomorphic designs	13,769,944	1,749	11

Number of non-isomorphic derived designs [Section 6.6.5]:

Design	2-(10,4,4)	3-(11,5,4)
Non-isomorphic derived designs	8,978	13

The 2-(10,4,4) Designs [Section 6.6.2]

Group Sizes	Number of non-isomorphic 2-(10,4,4) designs		
	<i>Simple Only</i>	<i>Non-Simple Only</i>	<i>All Designs</i>
1440	1	0	1
720	0	1	1
192	0	1	1
96	0	2	2
64	0	2	2
48	1	2	3
36	0	1	1
32	5	7	12
24	2	8	10
20	0	1	1
18	2	2	4
16	4	12	16
12	6	23	29
10	8	1	9
9	6	0	6
8	20	84	104
6	52	61	113
5	13	0	13
4	94	568	662
3	525	516	1041
2	10318	11795	22113
1	10070686	3675114	13745800
<b>Totals:</b>	10081743	3688201	13769944

The 3-(11,5,4) Designs [Section 6.6.3]

Group Size:	All 3-(11,5,4) designs are simple	
	<i>Number of Non-Isomorphic Designs</i>	<i>Number of Distinct Designs</i>
7920	1	5040
288	1	138600
144	1	277200
72	1	554400
48	1	831600
24	1	1663200
20	1	1995840
18	1	2217600
16	6	14968800
12	1	3326400
11	1	3628800
10	1	3991680
8	34	169646400
6	8	53222400
5	5	39916800
4	34	339292800
3	8	106444800
2	119	2375049600
1	1524	60833203200
<b>TOTALS:</b>	1749	63950375160

**The non-isomorphic derived 2-(10,4,4) designs [Section 6.6.5]**

Group Size	Number of non-isomorphic derived 2-(10,4,4) designs
1440	1
32	4
18	1
16	3
10	2
9	1
8	5
6	7
5	2
4	14
3	17
2	288
1	8633
<i>TOTAL:</i>	8978

**The non-isomorphic derived 3-(11,5,4) designs [Section 6.6.5]**

Group Size	Number of non-isomorphic derived 3-(11,5,4) designs
7920	1
144	1
72	1
24	1
16	1
8	3
5	1
3	1
2	1
1	2
<i>TOTAL:</i>	13

**The non-isomorphic 2-(10,4,4) designs admitted by each 3-(11,5,4) design [Section 6.6.5]**

Number of non-isomorphic derived designs admitted	Frequency
11	1503
10	18
9	1
8	79
7	17
6	27
5	34
4	26
3	33
2	9
1	2
<i>TOTAL:</i>	1749

## 7.5 CONFIGURATIONS

### 7.5.1 SYMMETRIC $v_3$ CONFIGURATIONS

The enumeration results for the symmetric  $v_3$  configurations are given by automorphism group size in the table below [Section 6.7.4]:

G	Order, $v$									
	7	8	9	10	11	12	13	14	15	16
56448								1		
18144										1
8064									1	
4608										1
2016										1
1512										1
720									1	
192									2	
168	1									
128								1	1	
120				1						
108			1							
96							1			2
72						1			1	
48		1							6	5
39							1			
36						1				
32						1			1	5
30									2	
24				1		1		2	2	7
20									2	
18						1			1	
16								3	10	24
15									2	
14								3		
13							1			
12			1	1		3	4	7	11	19
11					1					
10				1					3	
9			1							
8					1	1	3	15	34	93
7								1		
6				1	3	8	16	12	59	88
5									5	
4				2	2	3	30	91	180	635
3				2	1	3	20	19	69	320
2				1	13	60	190	916	3709	17119
1					10	146	1770	20328	241240	2986560
Totals:	1	1	3	10	31	229	2036	21399	245342	3004881

### 7.5.2 BLOCKING SET FREE CONFIGURATIONS

**Result 6.7.1:** [Section 6.7.7]

No blocking set free  $v_3$  configurations exist for orders 15, 16 or 17.

### 7.5.3 TRIANGLE FREE CONFIGURATIONS

The number of non-isomorphic, triangle-free,  $v_3$  configurations for  $v \leq 20$  [Section 6.7.8]:

Order, $v$	$15_3$	$16_3$	$17_3$	$18_3$	$19_3$	$20_3$
Number of triangle free, $v_3$ configurations	1	0	1	4	14	162

The incidence matrices of all triangle free  $v_3$  configurations, in canonical form, for  $v \leq 18$  are given below [Section 6.7.8]:

The unique  $15_3$  triangle free configuration:

```

111000000000000
100110000000000
100001100000000
010000011000000
010000000110000
001000000001100
001000000000011
000100010001000
000100000100010
000010001000100
000010000010001
000001010000001
000001000100100
000000101000010
000000100011000
    
```

The unique  $17_3$  triangle free configuration:

```

11100000000000000
10011000000000000
10000110000000000
01000001100000000
01000000011000000
00100000000110000
00100000000001100
00010001000100000
00010000010001000
00001000100000100
00001000001000010
00000100100010000
00000100000001010
00000010001000000
000000100000000101
000000010000000011
00000000010010001
    
```

The four non-isomorphic  $18_3$  triangle free configurations:

*Configuration #1:*

```

111000000000000000
100110000000000000
100001100000000000
010000011000000000
010000000110000000
001000000001100000
001000000000011000
000100010001000000
000100000100010000
000010001000100000
000010000010000100
000001000000000100
000000100000000001
000000010000000001
000000000100000001
000000000001000001
000000000000010001
    
```

*Configuration #2:*

```

111000000000000000
100110000000000000
100001100000000000
010000011000000000
010000000110000000
001000000001100000
001000000000011000
000100010001000000
000100000100010000
000010001000100000
000010000010000100
000001000000000100
000000100000000001
000000010000000001
000000000100000001
000000000001000001
000000000000010001
    
```

*Configuration #3:*

```

111000000000000000
100110000000000000
100001100000000000
010000011000000000
010000000110000000
001000000001100000
001000000000011000
000100010001000000
000100000100010000
000010001000000100
000010000010000100
000001000000000100
000000100000000001
000000010000000001
000000000100000001
000000000001000001
000000000000010001
    
```

*Configuration #4:*

```

111000000000000000
100110000000000000
100001100000000000
010000011000000000
010000000110000000
001000000001100000
001000000000011000
000100010001000000
000100000100010000
000010001000000100
000010000010000100
000001000000000100
000000100000000001
000000010000000001
000000000100000001
000000000001000001
000000000000010001
    
```

## 7.6 MISCELLANEOUS

### 7.6.1 CUBIC MULTIGRAPHS

The table below gives the number of non-isomorphic, connected, cubic multigraphs for all admissible orders,  $v \leq 18$  [Section 6.8.2]. The leftmost column specifies the automorphism group sizes, and each other column gives the number of non-isomorphic graphs for the given order.

G	Number of vertices in connected cubic multigraph graph (order)								
	2	4	6	8	10	12	14	16	18
384								1	2
336							1		
288									1
256									6
216									1
128							2	7	35
120					1				
96							2	1	6
72			1						1
64						1	4	38	267
48				1	2	3	1	12	18
36						1			2
32					1	5	34	216	1439
28							2		
24		1				3	2	6	5
20					2				
18						1			4
16				2	4	32	153	882	5605
14							2		
12			1	2	1	8	11	14	61
10					1				
8			1	5	19	80	421	2481	17216
6			1		4	2	7	28	56
4		1	2	7	33	146	834	5581	44754
3								4	4
2	1			3	18	157	1186	10214	103353
1					5	70	946	12371	167580
TOTALS	1	2	6	20	91	509	3608	31856	340416



# Chapter 8

## Conclusions and Future Work

### 8.1 CONCLUSIONS

This thesis has examined both non-exhaustive and exhaustive approaches to combinatorial design construction. The non-exhaustive techniques investigated were based on probabilistic hill-climbing and simulated annealing strategies and were applied to various construction problems. A number of search heuristics were developed and refined following analysis of the performance of the algorithms. The development of an incidence matrix backtracking algorithm was the main focus of the investigation of exhaustive construction techniques. A number of "lookahead" and isomorph rejection optimisations were examined, and this work led to the development of a constructive enumeration algorithm for general incidence structures with which a number of new results were established.

### 8.2 FUTURE WORK

There are a number of observations stemming from the results of this thesis which indicate directions for future research, and several of these are presented in this section.

Although probabilistic techniques can successfully generate many small designs, they are currently not effective at constructing designs with more complicated structure. In [43], Mathon mentions that hill-climbing approaches do not perform well for constructing  $t$ -( $v,k,\lambda$ ) designs with  $t > 2$  or  $k > 3$ , and suggests a possible reason for such failure being that almost all partial designs created during the search contain a large number of subconfigurations which are "forbidden" in the final design. This is a similar conclusion to the one drawn in this thesis following the work on weakly derived Steiner triple systems in Section 2.3.5. The probabilistic algorithm developed for this purpose was able to come close to generating a solution, but was unable to remove the final few conflicts. An investigation of the behaviour of the algorithm using a reverse engineering technique utilised in Section 2.3.5.4, indicated that the random construction of the partial designs introduced forbidden configurations early in the generation which prohibited the valid construction of completed designs. If such forbidden subconfigurations could be detected, their introduction to the partial solutions could be avoided, possibly leading to the development of successful probabilistic algorithms for the construction of random  $t$ -( $v,k,\lambda$ ) designs with  $t > 2$  or  $k > 3$ , such as the Steiner quadruple systems.

Regarding the constructive enumeration algorithm developed in Chapter 5, there are also a number of directions for future work. The isomorph rejection techniques discussed in Section 4.4 require the calculation of the automorphism groups of partial designs, which is currently performed using the group generation algorithm developed in Section 4.5.3. Currently the group of each partial design is calculated independently from previous group calculations. However this could be significantly improved by using the fact that the group of the partial design on the first  $n+1$  rows has the automorphism group of the design on the first  $n$  rows as the stabiliser of the  $n+1^{\text{st}}$  row.



The "lookahead" constraints and optimisations such as those presented in Section 3.4.3.7 and the packing constraint and strong isomorph rejection techniques detailed in Section 5.2.1 can detect early on in the construction of a row whether or not it can be validly completed. The formulation of such constraints, which can enormously reduce the computation required, is extremely valuable and is another direction for future research.

The constructive enumeration algorithm developed in this thesis generates all non-isomorphic designs satisfying a given parameter set, and does not assume any group acting on the blocks of the design. The generation of designs with prescribed group structures was introduced in Section 2.3.4 in which Steiner triple systems containing cyclic and 3-cyclic automorphisms were formed by the probabilistic construction of a set of base blocks. A corresponding extension to the constructive enumeration algorithm can be implemented by restricting the generation of designs to those with prescribed underlying automorphism groups. A group action induces a tactical decomposition on the incidence matrix which significantly reduces the amount of computation required. Such an extension would be extremely useful, for considerably larger designs could be examined and all designs with a particular group structure could be enumerated or perhaps shown to not exist.

In conclusion, the investigation of the search and enumeration techniques carried out in this thesis have led to a number of interesting discoveries. A wide variety of problems have been studied, illustrating the practical uses of the algorithms and highlighting the advantages and drawbacks of the different approaches. In addition, a number of directions for future research have been presented, with the hope that progress in these directions will lead to more exciting results.

# References

- [1] E. Aarts and J. Korst, "Simulated Annealing and Boltzmann Machines - A stochastic approach to combinatorial optimisation and neural computing" (Wiley, New York, 1989).
- [2] W.O. Alltop, "Extending t-designs", *J. Combin Theory A* **18**, 177-186.
- [3] A. Betten and D. Betten, "Regular Linear Spaces", *Contributions to Algebra and Geometry*, **38** (1997), No. 1, 111-124.
- [4] A. Betten, G. Brinkmann and T. Pisanski, "Counting Symmetric Configurations  $v_3$ ", *preprint*.
- [5] R.C. Bose, E.T. Parker and S. Shrikhande, "Further results on the construction of mutually orthogonal Latin squares and the falsity of Euler's conjecture", *Canad. J. Math.* **12** (1960) 189-203.
- [6] D. R. Breach, "In search of 4-(12,6,4) designs; Part II", *Australasian Journal of Combinatorics* **7**(1993), pp. 247-258.
- [7] D.R. Breach, M.J. Sharry and A.P. Street, "In search of 4-(12,6,4) designs; Part I", *Australasian Journal of Combinatorics* **7**(1993), pp. 237-245.
- [8] D.R. Breach and A.P. Street, "In search of 4-(12,6,4) designs; Part III", *Australasian Journal of Combinatorics* **7**(1993), pp. 259-273.
- [9] G. Brinkmann, "Fast Generation of Cubic Graphs", *Journal of Graph Theory* **23**, No. 2, 139-149 (1996).
- [10] V. Cerny, "Thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm", *J. Optimization Theory and Applications* **45** (1985), 41-51.
- [11] Y.M. Chee, C.J. Colbourn and A.C.H. Ling, "Weakly Union-Free Twofold Triple Systems", *Annals of Combinatorics*, to appear (acc. July 97).
- [12] C.J. Colbourn, "Embedding partial Steiner triple systems is NP-complete", *Journal of Combinatorial Theory* **A35** (1983) 100-105.
- [13] C.J. Colbourn and J.H. Dinitz, "Making the MOLS table", *Computational and Constructive Design Theory*, W.D. Wallis (ed.), Kluwer Academic, 1996, 67-134.
- [14] C.J. Colbourn and R. Mathon, "Steiner Systems", in "The CRC Handbook of Combinatorial Designs", C.J. Colbourn and J.H. Dinitz eds., CRC Press, 1996.
- [15] C.J. Colbourn and P.C. van Oorschot, "Applications of combinatorial designs in computer science", *ACM Computing Surveys* **21** (1989) 223-250.
- [16] C. J. Colbourn and A. Rosa, "Directed and Mendelsohn Triple Systems", in "Contemporary Design Theory, A Collection of Surveys", Edited by J. H. Dinitz and D. R. Stinson.
- [17] C.J. Colbourn and A. Rosa, "Triple Systems", Oxford University Press, *to appear*.
- [18] C.J. Colbourn, *private communication*.
- [19] J. Denes and A.D. Keedwell, "Latin Squares and Their Applications", English Universities Press, London, 1974.

- [20] J. Denes and A.D. Keedwell, "Latin Squares: New Developments in the Theory and Applications", North-Holland, Amsterdam, 1991.
- [21] P.C. Denny and R. Mathon, "A census of  $t$ - $(t+8, t+2, 4)$  designs,  $2 \leq t \leq 4$ ", in *preparation*.
- [22] J.H. Dinitz, D.K. Garnick and B.D. McKay, "There are 526,915,620 nonisomorphic one-factorisations of  $K_{12}$ ", *J. Combin. Des.* **2** (1994), 273-285.
- [23] J.H. Dinitz and D.R. Stinson, "A hill-climbing algorithm for the construction of one-factorizations and room squares", *Siam J. Alg. Disc. Meth.* **8**, No. 3, July 1987.
- [24] J. Doyen and A. Rosa, "An updated bibliography and survey of Steiner Systems", *Annals of Discrete Math.* **7** (1980), 317-349.
- [25] J.R. Elliott and P.B. Gibbons, "The Construction of Subsquare Free Latin Squares by Simulated Annealing", *Australasian Journal of Combinatorics* **5** (1992), pp. 209-228.
- [26] K. Engel and H.-D.O. F. Gronau, "On  $2$ - $(6, 3, \lambda)$  designs", *Rostock. Math. Kolloq.* **34**, pp. 37-48 (1988).
- [27] L. Euler, "Recherches sur une nouvelle espèce des quarres magiques", *Verh. Zeeuwisch Genoot. Wetenschappen* **9** (1782) 85-239.
- [28] I.A. Faradzhev, "Constructive enumeration of combinatorial objects", *Problemes Combinatoires et Theorie des Graphes Colloque Internat. CNRS 260. CNRS Paris* (1978), 131-135.
- [29] P. Frankl and Z. Furedi, "A new extremal property of Steiner triple systems", *Discrete Math.* **48** (1984), 205-212.
- [30] B. Ganter, R. Mathon, A. Rosa, "A complete census of  $(10, 3, 2)$ -block designs and of Mendelsohn Triple Systems of order ten. I. Mendelsohn Triple Systems without repeated blocks", in: *Proc. Seventh Manitoba Conf. Numerical Math. and Computing*, 1977, pp. 383-398.
- [31] B. Ganter, R. Mathon, A. Rosa, "A complete census of  $(10, 3, 2)$ -block designs and of Mendelsohn Triple Systems of order ten. II. Mendelsohn Triple Systems with repeated blocks", in: *Proc. Eighth Manitoba Conf. Numerical Math. and Computing*, 1978, pp. 181-204.
- [32] P.B. Gibbons, "Computing techniques for the construction and analysis of block designs", *PhD thesis*, University of Toronto, 1976.
- [33] R.L. Graham, M. Grötschel, L. Lovász, "Handbook of Combinatorics", Volume II, Part IV, 1995, North-Holland.
- [34] H. Gropp, "Blocking set free configurations and their relations to digraphs and hypergraphs", *Discrete Mathematics*, **165/166**, (1997), 359-370.
- [35] H. Gropp, "Configurations", in "CRC Handbook of Combinatorial Designs", ed. C.J. Colbourn and J.H. Dinitz, CRC Press, 1996, pp. 253-255.
- [36] F. Harary and E.M. Palmer, "Graphical Enumeration", Academic Press, New York, 1973.
- [37] O. Kempthorne, "The Design and Analysis of Experiments", (Wiley International, 1952).
- [38] T.P. Kirkman, "On a problem in combinations", *Cambridge and Dublin Mathematical Journal* (1847), 191-204.
- [39] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, "Optimisation by simulated annealing", *Science* **220** (1983), 671-680.
- [40] D.L. Kreher, " $t$ -Designs,  $t \geq 3$ ", in "CRC Handbook of Combinatorial Designs", ed. C.J. Colbourn and J.H. Dinitz, CRC Press, 1996, pp. 47-66.

- [41] P.J.M. van Laarhoven, "Theoretical and computational aspects of simulated annealing", Erasmus University, Rotterdam, *PhD thesis* (available as a CWI Tract, 1988).
- [42] C.W.H. Lam, L. Thiel and S. Swiercz, "The Non-Existence of Finite Projective Planes of Order 10", *Canad. J. Math.* **41** (1989) 1117-1123.
- [43] R. Mathon, "Computational Methods in Design Theory", in "Computational and Constructive Design Theory", 29-48, W.D. Wallis (ed.), 1996, Kluwer Academic.
- [44] R. Mathon and C. Pietsch, "On the family of  $2-(7,3,\lambda)$  designs", *preprint*.
- [45] R. Mathon and A. Rosa, " $2-(v,k,\lambda)$  Designs of Small Order", in "CRC Handbook of Combinatorial Designs", ed. C.J. Colbourn and J.H. Dinitz, CRC Press, 1996, pp. 3-41.
- [46] R. Mathon, *private communication*.
- [47] B.D. McKay, "Isomorph-free exhaustive generation", *to appear*.
- [48] B.D. McKay, "nauty users guide (version 1.5)", Technical Report TR-CS-90-02, Department of Computer Science, Australian National University, 1990.
- [49] B.D. McKay and S.P. Radziszowski, "The non-existence of  $4-(12,6,6)$  Designs", in "Computational and Constructive Design Theory", W.D. Wallis (ed.), pp. 177-188, Kluwer, 1996.
- [50] B.D. McKay and S.P. Radziszowski, "Towards deciding the existence of  $2-(22,8,4)$  designs", *J. Combin. Math and Combin. Computing*, **22** (1996) 211-222.
- [51] B.D. McKay, *private communication*.
- [52] Eric Mendelsohn, *private communication*.
- [53] M. Meringer, "Fast Generation of Regular Graphs and Construction of Cages", *to appear*.
- [54] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller, "Equation of state calculations by fast computing machines", *J. Chem. Physics* **21** (1953), 1087-1092.
- [55] E.T. Parker, "Computer Investigations of Orthogonal Latin Squares of order 10", *Proc. Sympos. Appl. Math* **15** (1963), 73-81.
- [56] R.C. Read, "Every One a Winner", *Annals of Discrete Math.* **2** (1978) 107-120.
- [57] E.M. Reingold, J. Nievergelt and N. Deo, "Combinatorial Algorithms - Theory and Practice", Prentice-Hall, 1977, pp. 112-114.
- [58] G.F. Royle, "Graphs and Multigraphs", in "CRC Handbook of Combinatorial Designs", ed. C.J. Colbourn and J.H. Dinitz, CRC Press, 1996, pp. 644-653.
- [59] E. Spence, "Classification of Hadamard matrices of order 24 and 28", *Discrete Math.* **140** (1995) 185-243.
- [60] E. Spence, "Construction and Classification of Combinatorial Designs", in *Surveys in Combinatorics 1995*, (ed. P. Rowlinson), L.M.S Lecture Notes 218, Cambridge University Press, London, 191-213.
- [61] E. Spence, *private communication*.
- [62] D.R. Stinson, "Combinatorial Designs and Cryptography", in *Surveys in Combinatorics 1993*, (ed. K. Walker), L.M.S Lecture Notes 187, Cambridge University Press, London, 257-287.
- [63] D.R. Stinson, "Hill-Climbing Algorithms for the Construction of Combinatorial Designs", *Annals of Discrete Mathematics* **26** (1985) 321-334.
- [64] J.D. Swift, "Isomorph Rejection in exhaustive search techniques" *Proc. A.M.S. Symp. Appl. Math.* **10** (1958), 195-200.