Using integer programming in finding t-designs

by

Kelvin Chung

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

A t-design is a combinatorial structure consisting of a collection of blocks over a set of points satisfying certain properties. The existence of t-designs given a set of parameters can be reduced to finding nonnegative integer solutions to a given integer matrix equation. The matrix in this equation can be quite large, but by prescribing the automorphism group of the design, the matrix in the equation can be made more manageable so as to allow the equation to be solved via integer programming tools; this fact was developed by Kramer and Mesner. Algorithms to generate the matrix equation generally follow a simple template. In this thesis, a generic framework for generating the Kramer-Mesner matrix equation and solving it via integer programming is presented.

Acknowledgements

I would like to acknowledge my two supervisors, Mark Giesbrecht and Ilias Kotsireas, for guiding me through the various difficulties that I have encountered along the way. Special thanks must also be given to Soumojit Sarkar for helping me with getting me started on writing, and David Dietrich, whose experience with LaTeXin a past term project proved to be invaluable in learning the language in short order. A final thanks goes out to professor Ric Holt, who gave me the confidence that I needed to present my thesis to the public.

Dedication

This is dedicated to my parents, who go out of the way more than they should so that I can be the student that I am today. This is also dedicated to my grandparents living half a world away, who have always supported my decision to enter graduate school. This thesis is also dedicated to the memory of my great-grandmother, who I will always remember in my heart.

Table of Contents

Chapter 1

Introduction

Combinatorial designs have many applications in areas as diverse as biology, cryptography, geometry, statistics, and software testing. A key question in this field is the existence of combinatorial designs given a set of parameters, and, should they exist, to construct a combinatorial design for a given set of parameters.

The existence of a special kind of combinatorial design known as a t-design, given a set of parameters, is a well-researched problem. This question can be reduced to finding nonnegative integer solutions to matrix equations, for which there are many different approaches to solving them. Algorithms that assist in this endeavour have been extensively explored, analyzed, and implemented with the help of various computer algebra systems. However, there are very few tools that find t-designs without any dependencies on these systems.

In Chapter 2, we will explore the basic concepts behind t-designs and algorithms that assist in finding t-designs. In Chapter 3, we will look at our implementation of these algorithms as part of a generic algorithm framework implemented in C++, and we will evaluate its performance in Chapter 4. Additional work suitable for future research is placed in Chapter 5.

Chapter 2

Background

2.1 Preliminaries

2.1.1 t-designs

A **t-design** with parameters v, k, and λ, alternatively denoted t-(v, k, λ) design, is a combinatorial structure consisting of a set $X = \{1, \ldots, v\}$ of **points** and a collection B of k-subsets of X, such that every subset of X of size t (henceforth termed **t-subset**) is contained in exactly λ subsets in B; the sets in B are known as **blocks**. The t-design is said to be **simple** if all the blocks are distinct.

It is to be noted that from the definition of a t-design, a t-design is also a u-design for every $u < t$; given a t-(v, k, λ) design, it is also a u-(v, k, λ_u) design, where:

$$\lambda_u = \frac{\lambda\binom{v-u}{t-u}}{\binom{k-u}{t-u}}$$

Clearly, there exist trivial examples of t-designs.

Definition 1 (Trivial t-designs)**.** A **trivial** t-design is a t-design whose blocks consist of n copies of every k-subset in X for some integer n. The parameters of a trivial design are t, v, k, and $\lambda = n\binom{v-t}{k-t}$.

2.1.2 Existence of t-designs

Definition 2 (Incidence Matrix of a t-design)**.** The **incidence matrix** of a t-design is a $v \times |B|$ matrix where each row represents a point and each column represents a block. The

	$\{1,2,4\}$	$\{1,3,7\}$	$\{1,5,6\}$	$\{2,3,5\}$	$\{2,6,7\}$	$\{3,4,6\}$	$\{4,5,7\}$
1	1	1	1	0	0	0	0
2	1	0	0	1	1	0	0
3	0	1	0	1	0	1	0
4	1	0	0	0	0	1	1
5	0	0	1	1	0	0	1
6	0	0	1	0	1	1	0
7	0	1	0	0	1	0	1

Figure 2.1: The incidence matrix of a 2-(7,3,1) design.

(i, j)-entry in the matrix is 1 if the point associated with point i is contained in the block associated with column j, and 0 otherwise.

Incidence matrices are a good way of representing t-designs, but they do not provide for a way to determine the existence of t-designs given each of the four parameters t, v, k, and λ. A major question in this field is determining whether a t-design exists given these four parameters. Firstly, there are some necessary conditions for existence: for the definition to make sense, we must have that $t \leq k < v$.

Another condition of the existence of t-designs that is immediately apparent is known as the **divisibility condition**. From the fact that a t-design is also a u-design for $1 \leq u < t$, we have it that if there exists a t-(v, k, λ)-design, then we have that $\binom{k-u}{t-u}$ must divide $\lambda\binom{v-u}{t-u}$ for every $0 \leq u < t$. If $t = 1$, the converse also applies. Thus, there will always be a minimum value of λ for which a t-(v, k, λ) must exist, and a value of λ^* for which a t-(v, k, λ) design (not necessarily simple) must exist for every $\lambda > \lambda^*$.

Less apparent is a result from Teirlinck [27] effectively stating that for any given t, there exists a simple and nontrivial t-(v, k, λ) design for some v, k, and λ.

2.1.3 Finding t-designs via recursive construction

Given an existing t-design, it is possible to construct additional t-designs. Clearly, from taking n copies of all the blocks in a t-design, a t-$(v, k, n\lambda)$-design can be constructed. Three other simple constructions can also be obtained from the definition:

Definition 3 (Complement of a t-design)**.** The **complement** of a t-design consists of the collection of blocks obtained by taking the complement under X of each block in the

t-design. The complement of a t-(v, k, λ)-design is, in fact, a t-$(v, v-k, \lambda^*)$-design, where

$$\lambda^* = \frac{\lambda\binom{v-k}{t}}{\binom{k}{t}}$$

An intermediate corollary that occurs is that since the complement of a t-design is another t-design, then we need only look for t-designs with $k \leq v/2$.

Definition 4 (Residual t-design)**.** The **residual t-design** of a t-design is obtained by removing a point from the point set as well as removing the blocks that contain the removed point from the collection of blocks. If the original t-design has parameters t, v, k, and λ, then the residual t-design is a $(t-1)$-$(v-1, k, \lambda^*)$-design, where

$$\lambda^* = \frac{\lambda(v-k)}{k-t+1}$$

Definition 5 (Derived t-design)**.** Let x be a point, and B be the collection of blocks in a t-design that contain the point x. The **derived t-design** consists of the point set $X - \{x\}$ and the blocks B', obtained from removing x from each block in B. If the original t-design has parameters t, v, k, and λ, then the derived t-design is a $(t-1)$-$(v-1, k-1, \lambda)$-design.

A number of theorems were also published from Alltop[2] that constructed t-designs from other t-designs:

Theorem 1 (Alltop's First Theorem (1))**.** *Let B be the blocks of a t-$(2k, k, \lambda)$-design, where t is even. If it is the case that the complement under X of each block in B is not also in B, then the blocks in B along with their complements form a $(t+1)$-$(2k, k, \lambda^*)$-design, where*

$$\lambda^* = \frac{2\lambda(k-t)}{2k-t}$$

Theorem 2 (Alltop's First Theorem (2))**.** *Let B be the blocks of a t-$(2k, k, \lambda)$-design, where t is even. If it is the case that the complement under X of each block in B is also in B, then B also forms a $(t+1)$-$(2k, k, \lambda^*)$-design, where*

$$\lambda^* = \frac{\lambda(k-t)}{2k-t}$$

Theorem 3 (Alltop's Second Theorem)**.** *Let B be the blocks of a t-$(2k+1, k, \lambda)$-design, where t is even. Suppose also that x is a point not in X. Then, the blocks B', formed from adding x to each block in B, and B'', formed from taking the complement under X of each block in B, together form a $(t+1)$-$(2k+2, k+1, \lambda)$-design.*

Theorem 4 (Alltop's Third Theorem). *Let B be the blocks of a t-$(2k+1, k, \lambda)$-design, where t is odd. Suppose also that x is a point not in X. If there are exactly $\frac{1}{2}\binom{2k+1}{k}$ blocks in B, then the blocks B', formed from adding x to each block in B, and B'', the collection of complements under X of each k-subset not in B, form a $(t+1)$-$(2k+2, k+1, \lambda)$-design.*

More complicated recursive constructions can be found through the concept of large sets.

Definition 6 (t-design Partition). Let $(B_1, B_2, \ldots, B_n)$ be a partition of the k-subsets of X. If all of the B_i form t-designs, then the partition is known as a t-(v, k, Λ) **partition** of the k-subsets of X, where $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$ is the sequence of λ parameters corresponding to the t-designs represented by the B_i.

A partition for which all of the λ_i are the same is said to be **uniform**. Uniform partitions thus partition the k-subsets of X into $\binom{v-t}{k-t}/\lambda$ t-(v, k, λ)-designs. Depending on the literature, these t-designs may be referred to as a **large set** of t-designs, though other sources claim that in order for a uniform partition to be a large set, λ must itself be the minimum value that satisfies the divisibility condition.

A number of results have been published that link the existence of large sets for various combinations of t, v, k, and λ, which in turn implies the existence of t-designs with the same parameters. Teirlinck in [27] shows that a t-design must exist for every t by showing the following:

Theorem 5 (Teirlinck's Proposition). *If $t \equiv v \pmod{(t+1)!^{2t+1}}$, then there exists a large set of t-$(v, t+1, (t+1)!^{2t+1})$ designs.*

2.1.4 Finding t-designs by solving matrix equations

The definitions and theorems above all depend on the existence of a pre-existing t-design with certain parameter combinations. For all other combinations of parameters, other methods must be employed.

A straightforward constructive method of determining the existence of a t-design is provided by the following theorem:

Theorem 6. *A t-(v, k, λ) design exists if and only if the matrix equation*

$$\mathbf{A}_{t,k}\mathbf{x} = \begin{bmatrix} \lambda \\ \vdots \\ \lambda \end{bmatrix}$$

has a nonnegative integer solution. The matrix $\mathbf{A}_{t,k}$ is a $\binom{v}{t} \times \binom{v}{k}$ matrix where:

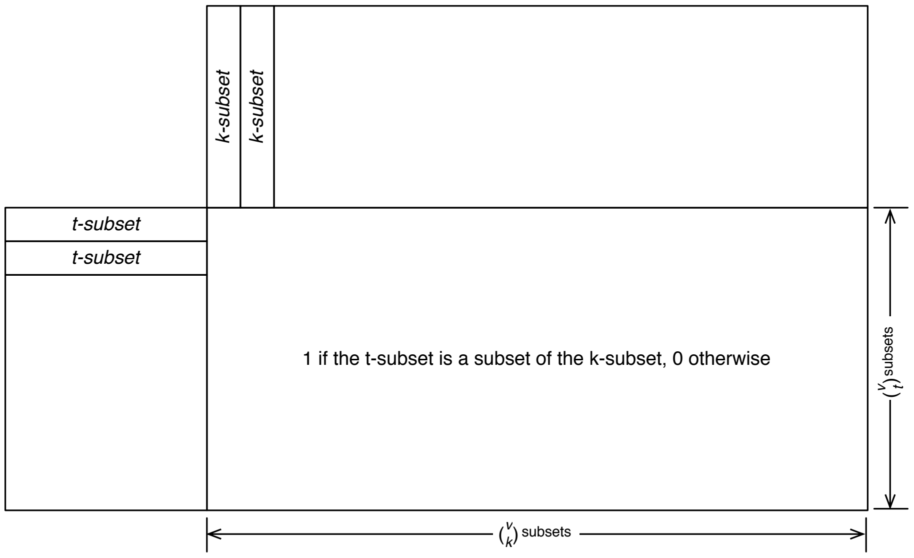


Figure 2.2: A visualization of the matrix in Theorem 6.

- *Each row of* $\mathbf{A}_{t,k}$ *is associated with a* t*-subset of* X
- *Each column of* $\mathbf{A}_{t,k}$ *is associated with a* k*-subset of* X
- *The* (i, j)*-entry of* $\mathbf{A}_{t,k}$ *is* 1 *if the* t*-subset for row* i *is a subset of the* k*-subset corresponding to column* j*, and* 0 *otherwise.*

The solution vector $\mathbf{x}$ *details the* k*-subsets that form the* t*-design and the number of repetitions therein.*

Though straightforward, solving the equation above suffers from serious drawbacks: the finding of a nonnegative integer solution is itself not an easy problem, and it is only exacerbated by the fact that the dimensions of $\mathbf{A}_{t,k}$ grows exponentially with larger t and k. A common way of reducing the difficulty of the problem is to reduce the dimensions of the matrix. This involves the following:

Definition 7 (Tactical Decomposition of a Matrix)**.** Let A be any matrix. A **tactical decomposition** of A is a partition of its rows and a partition of its columns such that every submatrix formed from taking one set of rows and one set of columns all have constant row and column sums.

For every matrix, there is at least one tactical decomposition: by taking every row and every column individually, we form a set of 1×1 submatrices, which by definition have constant row and column sums. For the incidence matrix of a t-design, there is also a second trivial tactical decomposition, that being the whole of the matrix: by definition, each column sums to k, and as every t-design is also a 1-design, each row must also sum to a constant.

A tactical decomposition of the matrix from the theorem is of particular interest; however, before it can be explained, a few other concepts from [17] and [25] are needed.

Definition 8 (t-design Automorphism)**.** An **automorphism** of a t-design is a bijection f over X such that if B is a block in the design, then $f(B)$ is another block in the design.

As t-design automorphisms are permutations over X, the automorphisms of a t-design form a subgroup of S_X, the symmetric group over X; this group is known as the **automorphism group** of the t-design, and is denoted by $\text{Aut}(B)$.

Definition 9 (Group Actions)**.** Suppose G is a group and X is a set of points. A **group action** is a function $G \times X \to X$ such that the group operation is preserved. We say that X is a **G-set**, and that G **acts on** X.

Definition 10 (Orbit of a Point)**.** The **orbit** of a point $x \in X$, denoted Gx is the set of points that results from applying every element of G on it.

Definition 11 (Stabilizer Subgroup)**.** The set of elements in G that fix a point x is known as the **stabilizer** of x, denoted G_x. The stabilizer of x is a subgroup of G, and thus it may also be referred to as the **stabilizer subgroup** of x.

The definition of orbits and stabilizers can be expanded so that it encompasses sequences of points or sets of points by having G act on each element therein separately.

Definition 12 (Quotient of a Set of Points)**.** Orbits of points in X under G form a partition of X, known as the **quotient** of X under G, and denoted X/G.

Note the similarity of notation between the quotient of a set of points X and the set of all left cosets of a subgroup of G. This is as every point in X is bijectively associated with its stabilizer subgroup, and thus the orbit of a point in X can be associated with a left coset of that stabilizer subgroup.

2.2 Kramer-Mesner matrices

If $G = \text{Aut}(B)$, then G partitions both the points and blocks of a t-design. Thus, the orbits of t-subsets and k-subsets of X form a tactical decomposition of the matrix from Theorem 6. By replacing each submatrix with a specially chosen scalar value, the matrix from Theorem 6 can be made smaller and manageable. The end result is the following:

Theorem 7 (Kramer-Mesner Matrix Equation Theorem)**.** *A t-design exists with automorphism group G if and only if the matrix equation*

$$\mathbf{A}_{t,k}\mathbf{x} = \begin{bmatrix} \lambda \\ \vdots \\ \lambda \end{bmatrix}$$

has a nonnegative integer solution. The matrix $\mathbf{A}_{t,k}$ (or $\mathbf{A}^G_{t,k}$ if there is a need to differentiate between groups) is a $\rho_t \times \rho_k$ matrix where:

- *ρ_t is the number of orbits of t-subsets of X*
- *ρ_k is the number of orbits of k-subsets of X*
- *Each row of $\mathbf{A}_{t,k}$ is associated with a representative of one of the ρ_t t-subset orbits*
- *Each column of $\mathbf{A}_{t,k}$ is associated with a representative of one of the ρ_k k-subset orbits*
- *The (i, j)-entry of $\mathbf{A}_{t,k}$ is the number of sets in the orbit of the k-subset for column j in which the t-subset for row i (or any subset in its orbit) is a subset.*

The matrix from this theorem, first published in [13], is known as the **Kramer-Mesner matrix**, and the equation as a whole is thus the **Kramer-Mesner matrix equation**. Like the equation from Theorem 6, the Kramer-Mesner matrix equation is a constructive method of determining the existence of t-designs; the t-design is reconstructed by taking copies of the orbits under the automorphism group of each of the subsets represented in the solution vector.

There are a few properties of Kramer-Mesner matrices:

Theorem 8. *If $\mathbf{A}_{t,k}$ is a Kramer-Mesner matrix, the following properties hold:*

- *The sum of the entries in any row of $\mathbf{A}_{t,k}$ is $\binom{v}{k}\binom{k}{t}/\binom{v}{t}$.*
- *If $t \leq s \leq k$, then $\mathbf{A}_{t,k} = \binom{k-t}{k-s}^{-1}\mathbf{A}_{t,s}\mathbf{A}_{s,k}$. In particular:*

$$\mathbf{A}_{t,k} = \mathbf{A}_{t,t+1} \ldots \mathbf{A}_{k-1,k}/(k-t)!$$

- *Let L_k be the vector of length ρ_k whose entries are the sizes of the orbits of k-subsets under G. Then $\binom{k}{t}L_k = L_t\mathbf{A}_{t,k}$.*

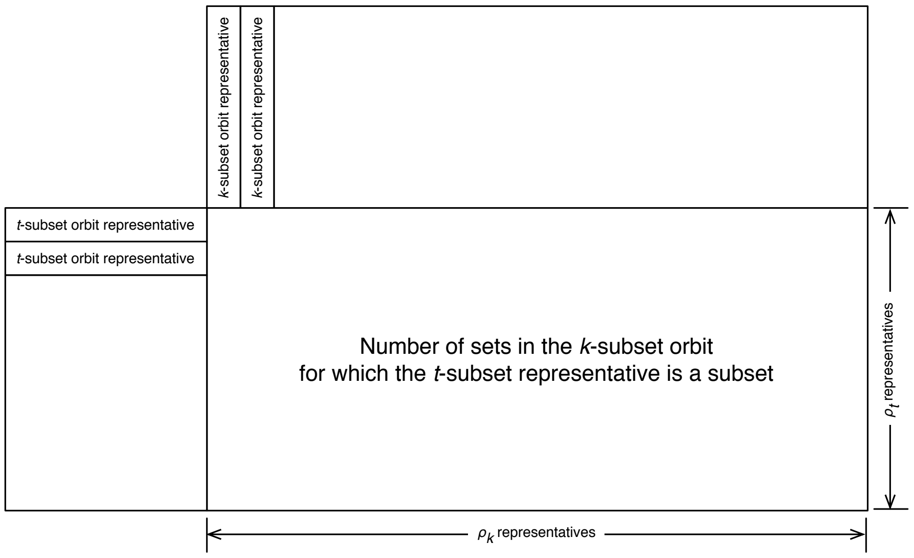

k-subset orbit representative
k-subset orbit representative
t-subset orbit representative
t-subset orbit representative
Number of sets in the *k*-subset orbit
for which the *t*-subset representative is a subset
ρ_t representatives
ρ_k representatives

Figure 2.3: A visualization of a Kramer-Mesner matrix, as presented in Theorem 7.

	$\{1,2,3\}$	$\{1,2,4\}$	$\{1,2,5\}$	$\{1,2,6\}$	$\{1,2,7\}$	$\{1,3,4\}$	$\{1,3,7\}$	$\{1,4,5\}$	$\{2,3,6\}$	$\{2,6,7\}$
$\{1,2\}$	1	1	1	1	1	0	0	0	0	0
$\{1,3\}$	2	0	0	0	0	2	1	0	0	0
$\{1,4\}$	0	1	2	0	0	1	0	1	0	0
$\{1,7\}$	0	0	2	0	2	0	1	0	0	0
$\{2,3\}$	2	0	0	0	0	0	1	0	2	0
$\{2,6\}$	0	0	0	1	2	0	0	0	1	1

Figure 2.4: The Kramer-Mesner matrix $\mathbf{A}_{2,3}$ over $\langle(145)(276),(26)(45)\rangle$, with row and column labels.[15] The value of 1 the top-left entry denotes the fact that $\{1,2\}$ is a subset of one set in the orbit of $\{1,2,3\}$, namely $\{1,2,3\}$ itself. The 2-(7,3,1) design represented by the incidence matrix in Figure 2.1.2 can be constructed from a solution to the corresponding Kramer-Mesner matrix equation.

Today, the Kramer-Mesner matrix equation is the primary method in which new t-designs are discovered. It can be summarized as follows:

Algorithm 1 Generic t-design finding algorithm

```
1: Select a group $G$, and parameters $v$, $t$, and $k$.
2: Construct the Kramer-Mesner matrix.
3: Solve the Kramer-Mesner matrix equation.
4: If a solution exists, reconstruct the $t$-design.
```

2.3 Kramer-Mesner matrix algorithms

The Kramer-Mesner matrix can be computed from the ordinary incidence matrix by simply grouping up and adding all the columns representing k-subsets in the same orbit, and grouping up and keeping only one row for each t-subset in the same orbit. Computing the Kramer-Mesner matrix in this fashion, however, would involve enumerating the t-subsets and k-subsets of X, followed by computing the orbits of each under G, which is horribly impractical.

More practical algorithms for computing the Kramer-Mesner matrix are typically of the following form:

Algorithm 2 Detailed generic t-design finding algorithm

The following computes the Kramer-Mesner matrix $\mathbf{A}_{t,k}$, given the t-design parameters and the automorphism group G.

```
1: for $i$ from 1 to $k$ do
2:     Compute $\rho_i$. This can be done using Burnside's lemma.
3:     Obtain the orbit representatives of $(i-1)$-subsets. The sole orbit representative for
       0-subsets, is, of course, the empty set. All other orbit representatives are computed
       from previous iterations of this loop.
4:     Populate the set of $i$-representative candidates by taking each $(i-1)$-representative
       and adding in each member of $X$ not present therein.
5:     Prune the set of candidates until $\rho_i$ candidates remain, each of which representing a
       different orbit.
6: end for
7: Compute the Kramer-Mesner matrix given the $t$-candidates and $k$-candidates.
8: Solve the Kramer-Mesner matrix equation.
9: If a solution exists, reconstruct the $t$-design.
```

The loop at the beginning of this algorithm reduces the number of candidates being considered to a value that is much more manageable compared to obtaining the row and column orbit representatives from a list of $\binom{v}{t}$ and $\binom{v}{k}$ candidates, respectively. However, note also that the method of pruning the number of candidates down to the required ρ_i candidates is not explicitly mentioned.

It is also to be noted that there is the additional consideration of choosing a set that is representative of its orbit. This would imply the use of some form of ordering relation on subsets of the same size. Lexicographical ordering is a natural subset ordering that is often used for this purpose, and will be used for the remainder of the discussion on the algorithm.

2.3.1 Candidate pruning

If $\rho_k = 1$, then every candidate belongs to the same orbit, and we can arbitrarily pick one candidate to be representative of all k-subsets. Pruning the candidate list down to the required number ρ_k is considerably more difficult if $\rho_k > 1$, though.

The task of reducing the collection of candidates to the collection of orbit representatives, at the lowest level, boils down to this decision problem:

Definition 13 (Set Image Problem)**.** Given a group G acting on a set of points X, and two subsets A and B of X, the **set image problem** (alternately named the **orbit discrimination problem**) is a decision problem that accepts its input if and only if there exists an element $g \in G$ such that $g(A) = B$.

A naive way of solving the set image problem is simply by iterating through the elements of G explicitly and seeing if the image of A under the current group element is in fact B. This leads to the following pruning algorithm:

Algorithm 3 Explicit pruning algorithm

This algorithm reduces the set of k-subset representative candidates down to the ρ_k orbit representatives of k-subsets.

```
1: repeat
2:   for all candidates $S$ do
3:     for all $g \in G$ do
4:       Compute the image $g(S)$ of $S$ under $g$.
5:       Compare $S$ to $g(S)$ using the ordering relation. If $g(S)$ compares favourably,
         remove $S$ from the set of candidates, and, if necessary, insert $g(S)$ to the set of
         candidates.
6:       Continue the iteration with $S$ as the set that was compared more favourably.
7:     end for
8:   end for
9: until the set of candidates is of size $\rho_k$
```

Of course, if the ordering relationship is stronger (as is the case for lexicographical ordering), the candidate set can itself be ordered, which reduces the pruning algorithm to that of removing the least element from the candidate set and eliminating candidate sets in the same orbit from further consideration, and repeating this ρ_k times. Even then, this is an extremely expensive operation, especially as k or $|G|$ get larger.

Fortunately, there are many approaches to reduce the time it takes to compute orbit representatives. In the end, however, it is well known that the set image problem is at least as hard as the graph isomorphism problem, and thus we should not be expecting polynomial runtimes with any specific approach.

Three specific approaches are explored here: the first involves using functions that do not depend on the specific element of G that is currently being iterated upon, the second involves iterating through a smaller subset of G that serves to represent the whole of G, and a third exploits the relationship between the set image problem and a related problem.

2.3.2 The table method

The naive way of solving the set image problem is to evaluate a boolean function on three inputs: a group element $g \in G$ and the two input sets; if the function rejects its input for every $g \in G$, then the two input sets must be in different orbits. Iteration of the elements in a group is a fairly expensive operation especially as $|G|$ gets large, and the central idea of the table method is to consider functions that do not make use of g in determining acceptance or rejection; this would render iterating through G unnecessary.

Formally, if a function f takes in an element of G and a k-subset of X as input and returns a value from a set Y, then this function is **G-invariant** if $f(g, x)$ for a fixed x returns the same value for all $g \in G$.

It is thus apparent that for any G-invariant function over k-subsets, the size of the codomain is at most the size of the quotient of k-subsets under G, i.e. ρ_k. If in fact the codomain of such a function is of size ρ_k, then the set image problem is reduced to evaluating this function finding whether or not their outputs are the same. Such functions are known as **discriminator**s.

Clearly, G-invariant functions and discriminators exist: if $\rho_k = 1$, then a constant function is trivially the discriminator over k-subsets. Several families of nontrivial G-invariant functions also exist, and will be discussed later. Furthermore, it is to be noted that the cartesian product of G-invariant functions is itself a G-invariant function, and its codomain is at least as big as the codomain of its largest component. Thus, given enough G-invariant functions, it is possible to create a discriminator for k-subsets. This gives us the following algorithm:

Algorithm 4 Table method for pruning candidates

This algorithm finds the ρ_k orbit representatives from a complete list of candidates.

```
1: Start with a table with no rows. Associate one column of the table to each orbit
   representative candidate.
2: repeat
3:    Construct a new $G$-invariant function, $f$, and add a row to the table.
4:    Evaluate $f$ on every candidate, and store its result in its corresponding table cell.
5: until the number of distinct columns in the table is $\rho_k$
6: Partition the orbit representatives based on their associated column values, and take
   one candidate from each grouping in the partition.
```

This algorithm remains open-ended with regards to how each G-invariant function is to be added. Ideally, these functions should be simple to construct and cheap to evaluate, while having large codomains. To this end, Magliveras and Leavitt, in [21] identify three nontrivial G-invariant functions suitable for use in this algorithm.

Anchor Sets

An **anchor set** is a large subset of X that can be used to create a G-invariant function, evaluated as follows.

Algorithm 5 Anchor set evaluation

This algorithm evaluates the G-invariant function created from an anchor set on a k-subset B of X.

```
1: Precompute the orbit of the anchor set under $G$.
2: Initialize a running tally of integers.
3: for all set $S$ in the orbit do
4:     Compute $|S \cap B|$ and add it to the running tally.
5: end for
6: return the running tally
```

Note that anchor sets may be any size; however, the reliability (codomain size) of the function is maximized when the anchor set is half the size of X and the size of B is small relative to the size of the anchor set. Note also that anchor sets may also be reused for different sizes of input.

Taxonomy 1

This method relies on a specified element of G. If $g \in G$, then g can be expressed as a disjoint union of cycles. If all of the points in one of these cycles were to be treated as a set, then g would fix the set. Furthermore, if each of these cycles was treated as a set, then the resulting collection of sets would partition X. If G acts on a partition of X by acting on each of the sets in the partition separately, then we have a G-invariant function that is evaluated as follows:

Algorithm 6 Taxonomy 1 evaluation

This algorithm evaluates the G-invarjant function created from a group element g via the Taxonomy 1 method on a k-subset B of X.

```
1: Precompute the orbit of the partition created by $g$.
2: Initialize a running tally of running tallies.
3: for all collection of sets $S$ in the orbit do
4:     Initialize a running tally of integers.
5:     for all sets $T$ in $S$ do
6:         Compute $|T \cap B|$ and add it to the running tally of integers.
7:     end for
8:     Add the running tally of integers to the running tally of running tallies.
9: end for
10: return the running tally of tallies
```

The G-invariant function created from the Taxonomy 1 method is, in many ways, similar to that of the one created from an anchor set; it can be viewed from another standpoint as evaluating a large number of small anchor sets. Like anchor sets, this may also be reused for different sizes of input, though unlike them, the reliability of this method is subject to both the selection of g as well as the structure of G itself. In particular, this method is entirely useless if G is cyclic or if the chosen element g is the identity element; in both of these cases a constant function is returned.

Taxonomy 2

This method relies on the existence of a discriminator of size $t < k$, and is evaluated as follows:

Algorithm 7 Taxonomy 2 evaluation

This algorithm evaluates the G-invariant function created from a discriminator of t-subsets ($t < k$) via the Taxonomy 2 method on a k-subset B of X.

```
1: Initialize a running tally of the values returned by the discriminator.
2: for all $t$-subsets $S$ of $B$ do
3:     Evaluate the discriminator on $S$, and add its result to the tally.
4: end for
5: return the running tally
```

The G-invariant function created from this method is, unlike the others, less reusable for different sizes of input due to the presence of the discriminator. It is said that the reliability of functions created from this method is consistently high, though the performance of this function is highly dependent on that of the discriminator.

2.3.3 Pruning via minimum orbit representatives

Lexicographical ordering, like other total linear orderings, allows us to speak of the notion of a minimum among a finite collection. Specifically, among different k-subsets in the same orbit, we can now speak of a set that is minimum among them to represent the orbit as a whole. There are various ways of finding the minimum orbit representatives for k-subsets. Efficient methods to find them, however, depend on the way permutation groups are stored in memory. Permutation groups are often stored via a construction known as a **base and strong generating subset**, or **BSGS** for short.

Definition 14 (Base of a Group)**.** A sequence of distinct points $(a_1, \ldots, a_n)$ is known as a **base** for a group G if the only element in G to fix the sequence pointwise is the identity element; that is, if $g \in G$ and $g(a_1) = a_1, \ldots, g(a_n) = a_n$, then g is the identity element.

It is to be noted that a permutation $g \in G$ can be thought of as a base for G if it is treated as the sequence $(g(1), g(2), \ldots, g(v))$.

Given the base $B = (a_1, \ldots, a_n)$, let G_i be the pointwise stabilizer $G_{(a_1,\ldots,a_i)}$. Then, it can be shown that G_i is a subgroup of G_j for all $i > j$. The sequence of G_i is known as the **stabilizer chain** of G for the base B, and B is said to be **nonredundant** if all of these subgroup inclusions are proper.

Definition 15 (Strong Generating Set)**.** A **strong generating set** relative to a base B of a group G is a set of group elements S such that the group generated by $S \cap G_i$ is G_i itself for each of the G_i in the stabilizer chain.

The stabilizer chain for a base also leads to the following: suppose H is a subgroup of G. Then, we have it that the group elements of G are partitioned by the right cosets of H. A **right transversal** of G modulo H is a set which contains one representative of each of the sets in the partition. (A similar construction with left cosets produces the left transversals of G modulo H.) Because each G_i is a subgroup of G_{i-1}, we can define the right transversal sets U_{i-1} of G_{i-1} modulo G_i, and claim that every group element can be uniquely decomposed into the product of elements $u_n u_{n-1} \ldots u_1$, where $u_i \in U_i$.

Bases, even nonredundant bases, are not unique for a given group: for example, if G is cyclic, then any single point (and thus any sequence of distinct points) would be a base. This in turn implies the ability to reconstruct the same group using a different base. Efficient algorithms exist for changing the base and its associated data structures.

The judicious use of base changing allows us to efficiently find the minimum orbit representative of a k-subset given any other k-subset. A simple algorithm that does this using backtracking is provided by Kreher and Stinson in [16], while another algorithm involving breadth-first search is given by Linton[20].

Backtracking algorithm

A main premise of the backtracking algorithm from Kreher and Stinson in [16] is that each k-subset is also associated with a k-tuple consisting of the elements of the subset presented in increasing point order. Thus, the minimum orbit representative for a k-subset K is a set associated with the sequence $(s_0, \ldots, s_{k-1})$ satisfying three properties:

- $s_0 < s_1 < \ldots < s_{k-1}$

- There exists $h \in G$ such that $h(K) = \{s_0, \ldots, s_{k-1}\}$
- If there is another set $\{t_0, \ldots, t_{k-1}\}$ satisfying the first two properties, then $\{s_0, \ldots, s_{k-1}\}$ is smaller in lexicographical order: that is, $s_0 \leq t_0$, $s_1 \leq t_1$, $\ldots$, and $s_{k-1} \leq t_{k-1}$.

The backtracking algorithm's subroutine attempts to determine the elements in the sequence s_i one element at a time. In determining element s_j, the algorithm first assumes the existence of $h \in G$ which in turn determine the elements s_i for $i < j$. Then, the algorithm finds an element $g \in G$ fixing s_i for $i < j$ that minimizes $g(x)$ for the elements $x \in h(K)$ not in $\{s_0, \ldots, s_{j-1}\}$. The use of base changing allows us to efficiently find g, as we only need to search through the elements of one transversal set rather than the whole group.

The process of doing so is as follows:

Algorithm 8 Backtracking algorithm for minimum orbit representative

This algorithm computes the minimum orbit representative for a given k-subset of X.

```
1: Let $\mathcal{C}_0$ be the input set.
2: Let $S_i$ be a sequence, presently initialized to the sequence of elements in $\mathcal{C}_0$ in point
   order.
3: Perform the subroutine, passing in the parameter $j = 0$.
4: return  the set of points in the sequence $S_i$
```

Algorithm 9 Subroutine for backtracking algorithm

This subroutine is a recursive subroutine, used to find the minimum orbit representative for a given k-subset of X. It requires the sequence S from the main algorithm, the set of globals $\mathcal{C}_i$, and takes in an additional parameter j.

```
1: Let $m$ be a point, presently uninitialized.
2: for all $x \in \mathcal{C}_j$ do
3:    Let $r$ be the first index before $j$ for which $s_r = S_r$; if no such index exists, set $r$ to $j$.
4:    If $r < j$ and $s_r \geq S_r$, return.
5:    Change the base of $G$ to the sequence $(s_0, \ldots, s_{j-1}, x)$.
6:    Let $U_j$ be the right transversal set. (This will be the one corresponding to the base point $x$.)
7:    Let $g$ be an element of $U_j$ such that $g(x)$ is the smallest point.
8:    if $m$ is uninitialized or $g(x) \leq m$ then
9:       Set $m$ to $g(x)$, and set $s_j$ to $m$.
10:      if $k = j + 1$ then
11:         Let $p$ be the first index after $r$ for which $s_p \neq S_p$; if no such index exists, skip the next step.
12:         If $s_p < S_p$ then replace the sequence $S_i$ with the sequence $s_i$.
13:      else
14:         Set $\mathcal{C}_{j+1}$ to the set $g(\mathcal{C}_j) - \{m\}$.
15:         Perform this subroutine, passing in the parameter $j = j + 1$.
16:      end if
17:   end if
18: end for
```

Linton's algorithm

Linton's algorithm consists of a series of k iterations, in which iteration t computes M_t, the lexicographically smallest t-subset among the orbits under G of every t-subset of the input set. The actual t-subsets that map to the minimum are computed through this process and kept for use by the next iteration in determining the lexicographically smallest $(t + 1)$-subset.

Algorithm 10 Linton's algorithm

This algorithm computes the minimum orbit representative for a given k-subset of X.

```
1: Let $\mathcal{C}_t$ be a list of candidate records for $t$-subsets. A **candidate record** is a pair
   $(C, \Delta)$ where $C$ is a $t$-subset of the input set $B$, and $\Delta$ is the image of $B$ under a group
   element $g \in G$ such that $g(C) = M_t$.
2: From the above, let $\mathcal{C}_0$ be a singleton list containing the record $(\emptyset, B)$ and $M_0 = \emptyset$.
3: Let $G_i$ be the $i$th group in the stabilizer chain of $G$.
4: for $i$ from 1 to $k$ do
5:    Let $m$ be a point, presently uninitialized.
6:    Let $\mathcal{P}$ be a list of candidate records.
7:    for all candidate records $(C, \Delta)$ in $\mathcal{C}_{i-1}$ do
8:       Let $m_{(C,\Delta)}$ be the smallest point in the orbit of any point in $\Delta - C$ under $G_{i-1}$
9:       if $m$ is uninitialized or $m_{(C,\Delta)} < m$ then
10:         Set $m$ to be $m_{(C,\Delta)}$.
11:         Set $\mathcal{P}$ to the singleton list containing the record $(C, \Delta)$.
12:      else if $m = m_{(C,\Delta)}$ then
13:         Add the record $(C, \Delta)$ to $\mathcal{P}$.
14:      end if
15:   end for
16:   Set $M_i$ to be the set $M_{i-1} \cup \{m\}$.
17:   If $i = k$ then return $M_k$.
18:   Change the base of $G_{i-1}$ so that $m$ is the first point in the base. (After this step, the
      base of $G$ itself will have $M_i$ ordered in ascending point order as the initial portion
      of its base.)
19:   Let $U_i$ be the right transversal set. (This will be the one corresponding to the new
      base point $m$.)
20:   for all candidate records $(C, \Delta)$ in $\mathcal{P}$ do
21:      for all points $x$ in $\Delta$ do
22:         If there exists a permutation $u \in U_i$ mapping $m$ to $x$, add the record $(C \cup
            \{m\}, u^{-1}(\Delta))$ to $\mathcal{C}_i$.
23:      end for
24:   end for
25: end for
```

Candidate pruning algorithms

Given an algorithm for finding the minimum orbit representative of a given k-subset, then, one can make a simple candidate pruning algorithm by seeing whether the minimum orbit representative of a specific k-subset is itself, and discarding those that are not. Kreher and Stinson in [16] also propose this algorithm for finding the orbit representatives as they are generated:

Algorithm 11 Backtracking method for pruning candidates

This algorithm computes the ρ_k orbit representatives of k-subsets from the set of orbit representative candidates. The list of candidates must have been formed by taking a $(k-1)$-subset representative and adding a point larger than all points therein.

```
1: Start with an empty set of representatives
2: for all candidates $S$ do
3:     Take the set $A$ to be $S$, less its largest element. (i.e. the $(k-1)$-representative
       generating $S$)
4:     Find the minimum orbit representative $R$ of $S$.
5:     Add $R$ to the representative set, if not already present.
6:     Let $S'$ be a sequence of points, ordered as follows: first, include the elements of $S$
       in numerical order. Then, include the remaining elements of $X$, again in numerical
       order. Note that the point $x$ added to $A$ to form $S$ is in position $k$ in this sequence.
7:     Change the base of $G$ to $S'$.
8:     for all $g \in U_k$ do
9:         Remove $A \cup \{g(x)\}$ from the set of candidates.
10:    end for
11: end for
```

2.3.4 Pruning via double cosets

Another method of generating orbit representative candidates and pruning them comes from Schmalz in [26], and his method is generally the method that is used in practice when constructing Kramer-Mesner matrices.

Recall that the right cosets of G under a subgroup H partition G; the partition is often denoted $H\backslash G$. Similarly, the left cosets of G under a subgroup K partition G; the partition is denoted G/K. The two concepts can be combined together to partition G to **double coset**s under H and K, denoted $H\backslash G/K$.

Definition 16 (Young Subgroup)**.** Suppose $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is a partition of X, and G_i be the permutations of S_X, the symmetric group over X, fixing every element of X not in α_i. The **Young subgroup** corresponding to the partition is the group $S_{(\alpha_1,\alpha_2,\ldots,\alpha_n)} = G_1 \times G_2 \times \ldots \times G_n$.

Essentially, if g is a permutation in a Young subgroup, then g maps the elements of α_i only to elements of α_i for each i. For the purposes of brevity, we denote the group $S_{(\{1,\ldots,a_1\},\{a_1+1,\ldots,a_1+a_2\},\ldots,\{v-a_n+1,\ldots,v\})}$ as $S_{[a_1,a_2,\ldots,a_n]}$. We also note that the first component group in $S_{(A,B)}$ is the stabilizer of the set B in S_X.

It was noted in [26] that the computation of orbit representatives is equivalent to finding a transversal of a double coset partition (i.e. selecting one permutation from each set in the partition). Specifically, each right coset $S_{[v-k,k]}\pi$ of $S_{[v-k,k]}$ in S_X corresponds bijectively to the set of points that π maps to the set $\{v-k+1,\ldots,v\}$. If G acts on the former from the right and the latter on the left, then each double coset $S_{[v-k,k]}\pi G$ corresponds to the orbit under G of the points that π maps to the set $\{v-k+1,\ldots,v\}$.

To find the orbit representatives of t-subsets and k-subsets, then, a transversal of $S_{[v-t,t]}\backslash S_X/G$ and $S_{[v-k,k]}\backslash S_X/G$, must be computed. This, in turn, is done through a version of the generic t-design algorithm that Schmalz, in [26], calls "Leiterspiel".

Definition 17 (Ladder of Groups)**.** A sequence of groups $(G_0, G_1, \ldots, G_n)$ is known as a **ladder** of groups if it is the case that G_i is a subgroup of either G_{i-1} or G_{i+1} for each group in the sequence. The transition between G_{i-1} to G_i is known as a **step-down** if G_i is a subgroup of G_{i-1}, and is known as a **step-up** if G_{i-1} is a subgroup of G_i.

The Leiterspiel algorithm traverses a ladder from S_X to $S_{[v-k,k]}$, and computes the double coset representatives of S_X under the group in the ladder and the input group G. Because of the structure of a Young subgroup, a ladder will always exist between S_X and any Young subgroup therein.

Each iteration of the generic t-design finding algorithm except the first consists of a step-down and a step-up; in the first iteration no step-up is performed. In each step-down, the group $S_{[v-(k-1),k-1]}$ is transitioned to $S_{[v-k,1,k-1]}$, and each double coset of $S_{[v-(k-1),k-1]}$ and G is partitioned in such a way as to get the double cosets of $S_{[v-k,1,k-1]}$ and G. Conversely, in each step-up, the group $S_{[v-k,1,k-1]}$ transitions to $S_{[v-k,k]}$, and the double cosets of $S_{[v-k,1,k-1]}$ and G are combined in a disjoint union to get the double cosets of $S_{[v-k,k]}$.

In each step-down, each double coset $S_{[v-k+1,k-1]}\pi G$ is partitioned into a number of double cosets $S_{[v-k,1,k-1]}t\pi G$, where t is an element from the transversal of $S_{[v-k,1,k-1]}\backslash S_{[v-k+1,k-1]}$. Computing a transversal for this latter partition is simple, as each coset in the partition consists of permutations that maps the points in $\{v-k+2,\ldots,v\}$ amongst themselves, maps a specific point in $\{1,\ldots,v-k+1\}$ to $v-k+1$, and maps the remaining points amongst themselves. However, it is very likely that some of these $v-k+1$ double cosets $S_{[v-k,1,k-1]}t\pi G$ generated from $S_{[v-k+1,k-1]}\pi G$ are in fact the same.

In each step-up, each double coset $S_{[v-k,k]}\pi G$ is formed from a number of double cosets $S_{[v-k,1,k-1]}t\pi G$, where t is a permutation in a transversal of $S_{[v-k,1,k-1]}\backslash S_{[v-k,k]}$. Again, such a transversal is easy to compute, as each coset in the partition consists of permutations that maps the points in $\{1,\ldots,v-k\}$ amongst themselves, maps a specific point in $\{v-k+1,\ldots,v\}$ to $v-k+1$, and maps the remaining points amongst themselves.

A detailed process of performing the step-downs and step-ups is provided in [18].

2.3.5 Computing the Kramer-Mesner matrix

Given $\mathcal{T}$ and $\mathcal{K}$, the orbit representatives for both t-subsets and k-subsets, respectively, a simple algorithm can be used to compute the Kramer-Mesner matrix:

Algorithm 12 Kramer-Mesner matrix computation algorithm

This algorithm computes the Kramer-Mesner matrix $\mathbf{A}_{t,k}$ over the automorphism group G, given $\mathcal{T}$, the set of orbit representatives for t-subsets, and $\mathcal{K}$, the set of orbit representatives for k-subsets.

```
1: Initialize $\mathbf{A}_{t,k}$ as the $\rho_t \times \rho_k$ zero matrix.
2: Initialize stab, a $\rho_k$ column vector, as the zero vector.
3: for all $g \in G$ do
4:    for all $K \in \mathcal{K}$ do
5:       Compute the image $g(K)$ of $K$ under $g$.
6:       if $K = g(K)$ then
7:          Increment the corresponding entry in stab.
8:       end if
9:       for all $T \in \mathcal{T}$ do
10:         if $T \subseteq g(K)$ then
11:            Increment the corresponding entry in $\mathbf{A}_{t,k}$.
12:         end if
13:      end for
14:   end for
15: end for
16: Divide each row in $\mathbf{A}_{t,k}$ by its corresponding entry in stab.
```

It's to be noted above that in addition to explicit iteration through G, the size of the setwise stabilizer G_K of K for each k-representative is also explicitly computed.

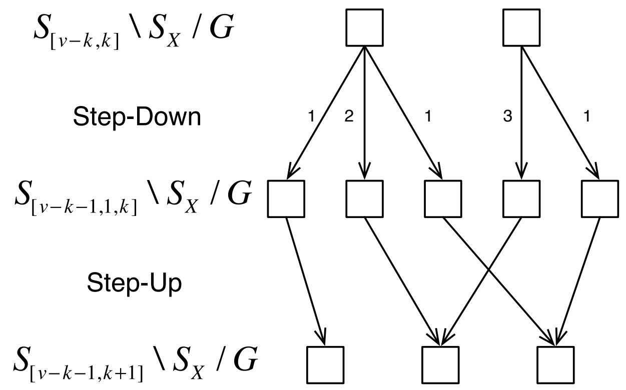
However, given the properties of Kramer-Mesner matrices, we can also compute the Kramer-Mesner matrix piecemeal after the candidate pruning process but before moving to the next iteration for subset sizes between t and k. This process creates the sequence Kramer-Mesner matrices $\mathbf{A}_{t,t+1}, \mathbf{A}_{t+1,t+2}, \ldots, \mathbf{A}_{k-2,k-1}, \mathbf{A}_{k-1,k}$, which can then be combined via Theorem 8 to produce $\mathbf{A}_{t,k}$.

Algorithm 13 Incremental Kramer-Mesner matrix computation algorithm

This algorithm computes the Kramer-Mesner matrix $\mathbf{A}_{k,k+1}$ given $\mathcal{T}$ the list of orbit representatives for k-subsets, and $\mathcal{K}$, the list of representatives for $(k+1)$-subsets.

```
1: Initialize $\mathbf{A}_{k,k+1}$ as the $\rho_t \times \rho_k$ zero matrix.
2: for all $T \in \mathcal{T}$ do
3:     for all $x \in X - T$ do
4:         for all $k$-subsets $T'$ of $T \cup \{x\}$ do
5:             if $T' \in \mathcal{T}$ then
6:                 Let $K'$ be the $(k+1)$-subset representative in $\mathcal{K}$ for $T \cup \{x\}$.
7:                 Increment the entry in $\mathbf{A}_{k,k+1}$ for the row representing $T'$ and column representing $K'$.
8:             end if
9:         end for
10:    end for
11: end for
```

It is also to be noted that the entries of $\mathbf{A}_{k,k+1}$ can also be computed without trying to find the orbit representative corresponding to a $(k+1)$-subset. In Schmalz's description of the Leiterspiel algorithm, he describes the construction of a layered graph called the **double coset graph** for which each layer corresponds to a particular group H in the algorithm's subgroup ladder. An example of such a graph is outlined in Figure 2.5. Vertices in each layer corresponds to the double coset representatives $H \backslash S_X / G$ (and thus their associated sets and tuples), while edges exist between layers only where the double coset representatives are split and merged to form the double coset representatives of the layer below. The edges in step-downs are further labeled with the value $|G_1|/|G_2|$, where $|G_1|$ and $|G_2|$ are the setwise and tuplewise (respectively) stabilizers under G of the objects associated with the tail and head of the edge, respectively. In constructing $\mathbf{A}_{k,k+1}$, only three layers (two in the first iteration, as there is no step-up) are constructed: the top layer representing k-subsets, the bottom layer representing $(k+1)$-subsets, and the middle layer representing singleton subset and k-subset pairs. Only edges between the top layer and the middle layer are weighted, as the transition between the two layers is a step-down. To



$S_{[v-k,k]} \backslash S_X / G$
Step-Down
1
2
1
3
1
$S_{[v-k-1,1,k]} \backslash S_X / G$
Step-Up
$S_{[v-k-1,k+1]} \backslash S_X / G$

Figure 2.5: Three layers of a double coset graph. An entry of $\mathbf{A}_{k,k+1}$ is computed from adding up the edge weights in every labelled edge on every path between the k-subset and the $(k+1)$-subset associated with the entry.

compute the value of an entry in $\mathbf{A}_{k,k+1}$, take all of the paths from the vertex representing the k-subset representing the entry's row and the $(k+1)$-subset representing the entry's column and add up all of the edge weights therein.

This process can be generalized to compute the final Kramer-Mesner matrix $\mathbf{A}_{t,k}$ without computing any of the intermediate matrices by constructing a graph with additional layers; however, the edge weights of all of the edges in a path from a t-subset vertex to a k-subset vertex must first be multiplied together to form a path weight before all the path weights are added together, after which a factor of $(k-t)!$ is divided to represent the fact that the order in which additional points are added to create the k-representative from the t-representative does not matter. This generalized process is thus effectively identical to the incremental construction of Kramer-Mesner matrices via matrix multiplication, as stated in Theorem 8.

2.4 Solving the Kramer-Mesner matrix equation

As the initial theorem implies, the question of the existence of t-designs is reduced to an application of integer programming. In particular, the Kramer-Mesner matrix equation has made the problem more manageable by reducing the size of the matrix to be solved. Finding {0,1}-solutions (for simple t-designs) or nonnegative integer solutions (for general t-designs) to integer matrix equations is known to be NP-complete: if $\mathbf{A}_{t,k}$ has only one row, then finding a {0,1}-solution to the Kramer-Mesner matrix equation is equivalent to the subset-sum problem.

However, even a Kramer-Mesner matrix may still be too large and unwieldy; this is apparent if:

- there is a large difference between t and k
- there is a large difference between v and k
- k is large
- G has small order

All of these factors would cause a large difference between the number of rows and the number of columns. Thus, it is advisable that the Kramer-Mesner matrix be a starting point towards finding an optimized matrix that will still produce the same t-designs; such a matrix will hopefully have fewer columns even if it means more rows will have to be added.

In practice, the use of the generic algorithm typically revolves around using either large input groups or small values of t and k to keep the size of the matrix manageable. Otherwise, additional input used in post-processing a Kramer-Mesner matrix to reduce it to a more manageable size is needed. Methods such as those in [14] have been used with some success to find 2-designs for large v and small k, and can be easily adapted to t-designs in general.

2.4.1 Solution-finding Algorithms

Whether or not post-processing (or "pre-solving") occurs, there are ultimately three approaches to solving the resulting integer matrix equation: backtracking, integer programming techniques, and lattice basis reduction. These methods can be used to find individual t-designs (and t-designs that can be constructed from them), enumerate them, or even partitioning all k-subsets of X into a large set of t-designs.

Integer programming

Integer programming is the process of finding an optimal solution to a matrix equation given certain constraints. Specifically, given a matrix $\mathbf{A}$ and vectors $\mathbf{b}$ and $\mathbf{c}$, we must find a solution $\mathbf{x}$ satisfying $\mathbf{Ax} = \mathbf{b}$ that minimizes (or maximizes) the scalar value $\mathbf{c}^T\mathbf{x}$. Furthermore, the values in $\mathbf{x}$ must all be nonnegative integers. A vector $\mathbf{x}$ satisfying only the matrix equation is said to be a **feasible** solution, and a feasible solution that also satisfies the second condition (sometimes referred to as the **objective function**) is said to be **optimal**.

Integer programming is thus related to linear programming (in the latter problem the values in $\mathbf{x}$ need not be integers and that each entry in $\mathbf{Ax}$ need only to be less than or equal to the corresponding entry in $\mathbf{b}$), but whereas linear programming can be solved efficiently, it is known that integer programming is NP-hard in general, and **binary integer programming**, the special case of integer programming where the values in $\mathbf{x}$ must be 0 or 1, is listed among the list of Karp's 21 NP-complete problems. [11]

Clearly, finding a t-design using the Kramer-Mesner matrix equation can be expressed as an integer programming problem, and in many cases, altering the linear objective function $\mathbf{c}^T\mathbf{x}$ can give us t-designs with particular characteristics. For example, the t-design with the fewest number of blocks, if it exists, can be found if we let $\mathbf{c} = L_k$. It must also be noted that if $\mathbf{c} = 0$, then we are claiming that any solution to the Kramer-Mesner matrix equation is considered to be optimal, allowing us to solve the original problem of the general existence of t-designs.

A common heuristic for solving integer programming problem is the **cutting plane method**: it first **relax**es the problem by ignoring the integrality constraint and solving the related linear program. If a solution exists and the solution obtained from that is, in fact, an integer vector, then we have a solution to the original matrix equation. However, if it is not, a second heuristic (known as a **cut**) will attempt to use the non-integer vector to find an integer solution to the matrix equation that is near it, or at the very least find a related problem which will assist in doing the same.

One example of this heuristic is the **feasibility pump**, described in detail in [9]. In this heuristic, the linear programming relaxation is solved as usual, and if a solution vector exists but is not an integer vector, then the closest integer vector (as defined by rounding each entry in the vector to the nearest integer) is computed. This, of course, is not a solution to the matrix equation (original or relaxed). However, from it, we can find the solution (again, to the linear programming relaxation) that is closest to this new vector. It is the hope that the feasibility pump would produce results that would converge to their integer roundings, and thus arrive at an integer solution.

Note that if optimality is a concern, then if a solution is found, then it may not be an optimal solution, and other methods will be needed to enhance the optimality of the obtained solution. The **branch and bound** technique is a method that is used to find an optimal solution. When both the cutting plane method and the branch and bound method is used, the resulting algorithm is often called a **branch and cut**. As the name implies, the branch and bound method is one that splits the problem into a number of subproblems (the "branching") whose constraints together cover the main problem. Heuristics can then determine upper and lower bounds (the "bounding") for the objective function's value for the optimal solution for each subproblem. If the lower bound of one subproblem is larger than the upper bound of another, then the first subproblem cannot contain the optimal solution, and thus it can be excluded from further consideration.

A more detailed treatise on integer programming is available in [29].

Lattice basis reduction

It was observed by Kreher and Radziszowski in [15] that if $\mathbf{x}$ was a (0,1)-solution to the Kramer-Mesner matrix equation, then the vector $[\mathbf{x}^T, 0, \ldots, 0]^T$ would be an integer linear combination of the columns of the matrix

$$B = \left[\begin{array}{c|c} I & 0 \\ \hline & -\lambda \\ \mathbf{A}_{t,k} & \vdots \\ & -\lambda \end{array}\right]$$

In other words, the vector $[\mathbf{x}^T, 0, \ldots, 0]^T$ is a vector in the **lattice** spanned by the columns of B. Since $\mathbf{x}$ is a (0,1)-solution to the Kramer-Mesner matrix equation, $[\mathbf{x}^T, 0, \ldots, 0]^T$ will tend to be a short vector in the lattice. Based on this, Kreher and Radziszowski then proceed to adapt the Lenstra-Lenstra-Lovasz (LLL) algorithm, first introduced in [19], for lattice basis reduction to solve the Kramer-Mesner matrix equation.

According to Kreher and Radziszowski, the LLL algorithm alone often cannot find simple t-designs in practice, but if augmented by two supplementary subroutines, called size reduction and weight reduction, then the resulting algorithm can be made more effective in finding simple t-designs.

Definition 18 (Weight of a Lattice Basis)**.** Let $B = (b_1, \ldots, b_n)$ be a lattice basis. The **weight** of the basis is defined as $\mathrm{w}(B) = \sum_{i=1}^{n} ||b_i||^2$.

The first of these two subroutines is called **weight reduction**. Weight reduction is based on the premise that, in practice, given the output of LLL, an ordered basis of a lattice closely approximating an orthogonal basis, that it is often the case that there are two distinct basis vectors b_i and b_j such that either $||b_i + b_j||$ or $||b_i - b_j||$ is smaller than the larger of $||b_i||$ and $||b_j||$. Replacing the latter vector (b_i or b_j, whichever is longer) with the former ($b_i + b_j$ or $b_i - b_j$, whichever is shorter) will result in a basis with smaller weight. Since this basis is "less orthogonal", the LLL algorithm can then be reapplied to get a "more orthogonal" lattice basis. The weight reduction subroutine performs this process repeatedly until the weight of the basis cannot be reduced further.

For **size reduction**, recall that we must find an integer linear combination of the columns of B that sums to $[\mathbf{x}^T, 0, \ldots, 0]^T$. If there is a row in the lower portion of B with exactly one nonzero entry, then the column corresponding to the nonzero entry does not contribute to the linear combination. This, then, allows us to remove the row and column containing the nonzero entry from B (and the corresponding entry from $[\mathbf{x}^T, 0, \ldots, 0]^T$), reducing our original problem to that of a smaller size. Of course, this may rarely occur, but it was observed that if a row in the lower portion of B was multiplied by $||b_i||^2$, where b_i is the longest lattice basis vector, and then LLL was subsequently applied, the affected row would almost always end up being of this form. The size reduction subroutine thus performs this for every row in the lower portion of B, which would ideally reduce an $(n+m)\times(n+1)$ matrix (given that $\mathbf{A}_{t,k}$ is an $m \times n$ matrix) to an $n \times (n-m+1)$ matrix.

The Kreher-Radziszowski algorithm consists of using LLL, size reduction and weight reduction, in this order, on B, to find t-designs. Since then, there have been various improvements to this algorithm or to LLL itself that have made this algorithm even more efficient. More details on improvements to LLL can be found in [7], and details on other lattices based on Kramer-Mesner matrices which also lead to t-designs may be found in [28].

Chapter 3

Implementation

This chapter will describe our implementation of a flexible generic program used to find t-designs based on computing the Kramer-Mesner matrix and solving the Kramer-Mesner matrix equation, with a concrete implementation of the table method. This program is written in C++, with the C++03 standard in mind, but can be easily fitted for C++11 through a small number of edits.

The source code is available in a git repository at `https://github.com/kelvSYC/MatrixGenerator`.

3.1 permlib and the `Group` class

The functionality needed for permutation group manipulation is done through the use of the permlib [24] library, a C++ language library for computational group theory. This library also makes use of the Boost C++ Libraries [8] for its implementation, and as such the rest of the program also makes use of Boost. We have chosen permlib for this program because of its its relatively simple API and the lack of dependencies on computer algebra systems such as GAP [10] and Magma [6], as well as the fact that it had shown to be faster than both in various computational group operations.

The `Group` class is the main interface to permlib in our program, and contains other convenience methods and supporting classes that facilitate iteration through the elements of the group in `GroupElementIterator` and the calculation of ρ_k for different values of k through `GroupBurnsideCache`. `Group` is also accompanied by supporting operations such as equality operators, hashing functions (suitable for use with Boost's unordered associative containers), and a weak ordering (suitable for use with standard associative containers).

3.2 The `Pruner` hierarchy

In order to generate the orbit representative candidates for k-subsets and then pruning them down to the ρ_k representatives that are needed, we have commissioned the `Pruner` class hierarchy. The `Pruner` class itself is an abstract class that provides basic functionality for generating the candidates, retrieving the list of representatives after the candidates have been pruned, and determining the representative for a given subset, for the purposes of populating the entries of the Kramer-Mesner matrix. `Pruner`s may also rely on additional data that they may need from one iteration to the next, which they may create and pass on. The generic algorithm does not prescribe whether or not the same `Pruner` is used throughout the algorithm, nor does it describe whether or not `Pruner`s must interpret the auxiliary data generated by other `Pruner`s; in this implementation, we have chosen to create a new `Pruner` in each iteration, with all `Pruner`s created being of the same type.

Four pruners are provided with this implementation:

- `ExplicitPruner` explicitly computes the orbits of representatives to determine whether or not a candidate is in the orbit of a known representative.
- `SetImagePruner` uses backtracking search to determine the existence of a permutation in G mapping a known representative to a candidate.
- `TablePruner` uses the table method by Magliveras and Leavitt to find representatives from candidates.
- `MinRepPruner` uses Linton's algorithm, as implemented in permlib, to find the minimum representative of a candidate.

The table method, through `TablePruner`, is very complex, and is implemented via a large number of classes.

3.2.1 The `CandidateGenerator` hierarchy

To generate candidates, `Pruner`s rely on the `CandidateGenerator` class hierarchy. `Pruner`s are free to use whatever `CandidateGenerator`s they desire, and two general-purpose `CandidateGenerator`s are provided with this implementation. The `FullCandidateGenerator` generates the full complement of candidates, while the `DefaultCandidateGenerator` only generates k-candidates for which the newly-inserted point is larger than the existing points of the $(k-1)$-representative. All of the existing `Pruner`s use the latter `CandidateGenerator` by default.

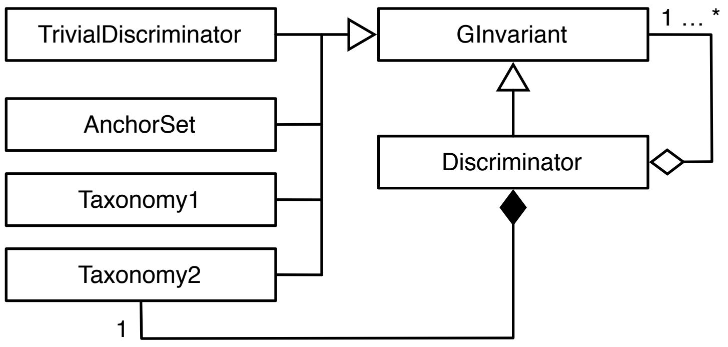


Figure 3.1: The `GInvariant` class hierarchy.

3.3 `TablePruner` and supporting classes

The table method lends itself well to an object-oriented implementation. Besides `TablePruner` itself, which manages the actual evaluation table, there are two main groups of classes that assist in the implementation of the table method: the `GInvariant` class hierarchy encompasses the G-invariant functions that are needed for the table method, and the `KMStrategy` class hierarchy governs the selection of `GInvariant`s for use in the building of the table.

3.3.1 The `GInvariant` class hierarchy

As the name implies, `GInvariant` is the base class for all G-invariant functions. There are five concrete subclasses to `GInvariant`, as summarized in Table 3.1: one subclass for each of the three nontrivial G-invariant functions, one for discriminators that are built from these, and the fifth for trivial discriminators, built when $\rho_k = 1$ for a specific k. `GInvariant` itself does not contain any behaviour, though it does provide storage for the `Group` the `GInvariant` is invariant over, as well as some partially-defined functions such as those needed for equality comparison and hashing. Above all else, however, is the `evaluate()` method, which acts as the common interface for the evaluation of the G-invariant function represented by the `GInvariant`.

Along with the classes representing the G-invariant functions themselves, there are a number of supporting classes that assist in imparting metadata about G-invariant functions into type traits used in template metaprogramming. For example, the `isDiscriminator` class is a template metafunction that separates the two discriminator subclasses from the other three.

Each of the G-invariant functions have been implemented factoring in various practical concerns. For example, the anchor set used in `AnchorSet` is always randomly generated, and is always half the size of X. Similarly, `Taxonomy1` is backed by a randomly-chosen element of the group, and a `Taxonomy2` represents a G-invariant function over $(k+1)$-subsets given a k-discriminator.

Most subclasses also have a collection of supporting classes that assist in implementing the behaviour of the G-invariant functions that they represent. Key among them are evaluation caches that help avoid the overhead of re-evaluating the G-invariant functions repeatedly, as evaluations may be expensive to perform.

Evaluators

For each of the four nontrivial subclasses of `GInvariant`, the actual functionality of evaluating their corresponding G-invariant functions is delegated to an evaluator class. These evaluators are similar in interface, and follow a convention that, in C++ parlance, is known as a **concept**.

Evaluators are C++ function objects that define a type called `FrequencyVector`, named as such as the return types are often vectors returned from running tallies. The function application operator takes in only one argument, the input set to the G-invariant function, and return a `FrequencyVector`. The input set is assumed to be compatible with the G-invariant function being modelled by the evaluator class. Evaluators must also be copy-constructible and copy-assignable, and `FrequencyVector` must be of a type that has a strict weak ordering. The reason for this requirement is that the `evaluate()` method of `GInvariant` specifically demands that an integral value is returned, and thus after the evaluator class returns a result, it must then be translated into an integral value via a lookup table.

There is no specific requirement on how evaluator classes are created by their corresponding `GInvariant` subclass, though in this implementation this is done through a call to the `createEvaluator()` call in all `GInvariant` subclasses that use evaluator classes.

Result caches

Result caches are used to both translate the returned `FrequencyVector`s from evaluator classes to the integral value required by `GInvariant`'s `evaluate()` method and to store the results of previous evaluations. This framework also separates out the results of G-invariant functions which can be reused for different input sizes, such as `AnchorSet`, by input size. They are effectively a thin template-based layer that separates out the needs of the cache's key and value types from the associative container type that backs them. The overall structure of the result caches used in our program is summarized in Figure 3.2.

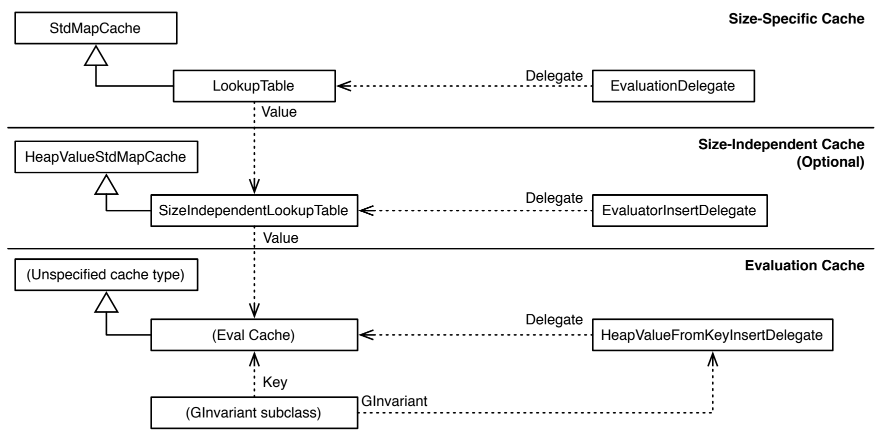


Figure 3.2: Overview of the result cache structures used in the implementation of the table method. The type of the evaluation cache varies based on the nature of the `GInvariant` being modelled; hence, this diagram may have minor variations for specific `GInvariant` subclasses.

One important consideration in the implementation of result caches is that the contents of each cell in the table created from the table method is independent of the contents of any other cell in the table. Because of this, each cell in a row may be concurrently evaluated. The conversion of the `FrequencyVector` to an integral value as well as the caching of this value thus creates a synchronization point between the threads that are evaluating the G-invariant function on different inputs. Because of this, caches also act as a thread-safe wrapper for their backing containers. Thread safety is provided through the use of Boost's thread library, and read-write locks are used extensively.

The root interface for result caches is the `Cache` class, which has two templated subclasses: `MapCache` for caches backed by an associative container, and `CacheAdapter` for caches that are backed by other caches (used in cases such as transforming the value type of the backing cache to a different type). A key template parameter is the delegate class, which is used to insert a value into a cache when the key is not found. In the case above, the translation from `FrequencyVector` to integral value is performed using an `EvaluatorDelegate`.

3.3.2 The `KMStrategy` hierarchy

A key step in the table method is to construct a new G-invariant function and add a new row to the table. Since there are many types of G-invariant functions, an overall strategy to govern what to add is needed. `KMStrategy` is a class that represents this. Specifically, a `KMStrategy` answers two specific questions:

- Given the results of the previous iteration, which G-invariant functions over k-subsets should we use to attempt to find a k-discriminator?
- What should be done if the G-invariant functions above fail to produce a k-discriminator?

Two sample `KMStrategy` subclasses are provided as concrete examples. The first is the `Taxonomy2Strategy`, which, provided the discriminator from the previous iteration is non-trivial, uses the `Taxonomy2` backed by the $(k-1)$-discriminator as its sole initial function; if the previous iteration is trivial, no initial function is provided. The second strategy, `RecyclerStrategy`, leverages the fact that anchor sets are G-invariants over subsets of any size. Because of this, the anchor sets that are used by the $(k-1)$-discriminator can be reused as G-invariant functions over k-subsets. In both strategies, should the initial functions fail to produce a k-discriminator, additional anchor sets are created.

Concurrent cell evaluation

Recall that the result caches are thread-safe so as to facilitate the concurrent evaluation of all of the cells in the table. Two compile-time macros are provided as an option for determining the granularity of concurrent evaluations. Concurrent evaluations are provided through the use of Boost's Thread and Asio libraries, the latter of which being used to implement a worker thread pool.

The evaluation of G-invariant functions may depend on the results of evaluating other G-invariant functions, as is the case when discriminators are evaluated. Thus, evaluating a single table cell will result in a tree of dependent evaluations, and evaluating an entire table row will result in a graph. (We can further assume that this graph is acyclic, as the two types of G-invariant functions that have dependencies only depend on evaluations on inputs of the same or smaller size.) A compile-time constant allows us to control the granularity of the concurrency desired: that is, whether or not this evaluation graph needs to be computed. In the event that this is done, the graph is built, nodes representing cached evaluations and their dependencies are eliminated, and the remaining nodes are then topologically sorted before each evaluation is performed; the net result is that each G-invariant function evaluation depend only on previously cached values. The graph construction is implemented through the use of the Boost Graph Library.

Note that the graph construction is only relevant if at some point, the prevalent `KMStrategy` dictates that a row for a G-invariant function with dependencies is to be added to the table. In certain situations, the penalty of having to construct the dependency graph may be too great to overcome, which in turn influences the `KMStrategy` design.

3.4 The `KMBuilder` class

The `KMBuilder` class is a class that produces `KMBuilderOutputs`, which in turn encapsulates the intermediate outputs produced by each iteration of the generic t-design finding algorithm: the set of orbit representatives for k-subsets, the Kramer-Mesner matrix $\mathbf{A}_{k-1,k}$, and whatever auxiliary data is produced by the `Pruner` class. In the generic t-design finding algorithm, each iteration begins by creating a `KMBuilder`, which in turn creates the appropriate `Pruner` with the data from the previous iteration's `KMBuilderOutput`. The `Pruner` is then responsible for generating the orbit representatives for this iteration, before the new Kramer-Mesner matrix for this iteration is constructed by `KMBuilder` itself. At this point, the new `KMBuilderOutput` is created.

Though the generic algorithm starts from finding the orbit representatives for singleton sets from the single orbit representative for the empty set, this implementation unrolls the first iteration, and thus begins with the orbit representatives for singleton sets, which in turn are explicitly computed.

Trivial iteration detection

If $\rho_k = 1$ for a particular iteration k, then the use of a `Pruner` is unnecessary. As the sequence of ρ_k is nondecreasing, we can conclude that the Kramer-Mesner matrix is a 1×1 matrix whose entry is $v - k$. The use of lexicographical ordering throughout the generic t-design finding algorithm also allows us to use the set $\{1, \ldots, k\}$ as the orbit representative for k-subsets. Thus, a `KMBuilderOutput` can be built right away.

Currently, ρ_k is computed by `KMBuilder` itself, but it may be the case that a future version of the algorithm will delegate this to the `Pruner` classes, as not all `Pruner`s would need ρ_k to be computed. To this end, all existing `Pruner`s also detect the case where $\rho_k = 1$, even though this is strictly unnecessary.

3.5 The `Solver` class hierarchy

After a Kramer-Mesner matrix is created, we employ the `Solver` class hierarchy is used to find solutions to the Kramer-Mesner matrix equation. Our program directly passes in the Kramer-Mesner matrix to the applicable solver; no attempts have been made to presolve or otherwise further reduce the size of the matrix before it is processed by the solver. Rather than providing a concrete implementation of any specific algorithm for finding nonnegative integer solutions to the Kramer-Mesner matrix equation, a solver that leverages CPLEX[1] and its integer programming tools is provided through the `CPlexSolver` class.

CPLEX[1] is a commercially available solver for linear programming and integer programming problems. For integer programming, it relies on a branch and cut technique to obtain solutions, and it has the ability to analyze the input and use the appropriate algorithms to find an optimal solution. Like permlib, it has a C++ API so that the transition from the `KramerMesnerMatrix` class to t-design solution vectors can be made as seamless as possible.

The `CPlexSolver` class will attempt to find any 0-1 solution, treating all solutions as optimal, by using a null (constant) objective function.

3.6 Creating custom `Solvers`

The program can be modified to make use of other solvers that may be more efficient for specific types of integer matrices, or for when different optimization conditions are needed. A custom `Solver` must implement the `solve()` method, which must return a boolean value to indicate whether a solution is found. If so, the `getSolutionVectors()` method can be used to retrieve any solutions that have been found. `Solver` does not prescribe how the information from the Kramer-Mesner matrix (specifically, the `KramerMesnerMatrix` class) and the input parameter λ, otherwise unused by the algorithm up to this point, is to be input.

Chapter 4

Results

Our implementation of the generic t-design finding algorithm leaves room for the use of many different candidate generators, pruners, and, for the table method, selection strategies. The different selections each affect the runtime of the algorithm, and the following are some empirical conclusions drawn from their use in practice.

4.1 Selection strategy with `TablePruner`

Among one of the key factors in the table method is the overall `KMStrategy` used to insert G-invariant functions to the table. The paper [21] introducing the table method had used it to construct a 6-(33,8,36) designs over $\mathrm{P\Gamma L}_2(32)$, the projective semilinear group on the two-dimensional vector space over field of 32 elements, which serves as the basis of strategy comparison in this section.

4.1.1 The `Taxonomy2` Strategy

As `Taxonomy2` is the easiest function to construct given a non-trivial `Discriminator` and `AnchorSets` being the cheapest `GInvariant` to create in general, this forms the basis of our first strategy, the `Taxonomy2Strategy`. According to [21], `Taxonomy2` tends to be one of the more reliable `GInvariants` in terms of getting large codomain sizes, and this claim is indeed justified: we were able to show with our program that the $k = 7$ and $k = 8$ iterations fully discriminated the 32 and 97 orbit representatives respectively using only the `Taxonomy2` alone, and the one iteration where this was not the case ($k = 6$) only one additional `AnchorSet` is needed. The results of this are summarized in Table 4.1.

However, for all its purported efficiency, there is a massive downside to using a `Taxonomy2`:

Table 4.1: The `Taxonomy2Strategy` in finding a 6-(33,8,36) design over $P\Gamma L_2(32)$

k	ρ_k	Columns in Table	`GInvariant` class	Discrimination	Uncached evaluations
5	3	29	`AnchorSet`	3/3	29
6	13	81	`Taxonomy2`	8/13	729
			`AnchorSet`	13/13	81
7	32	308	`Taxonomy2`	32/32	9779
8	97	684	`Taxonomy2`	97/97	44004

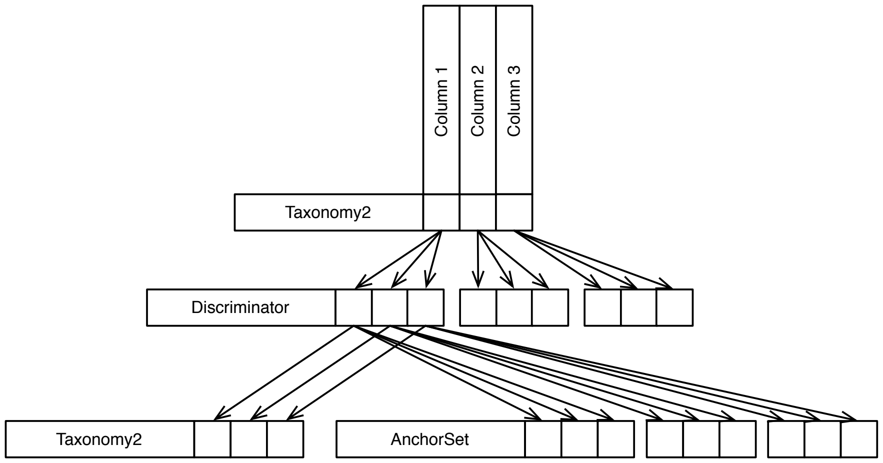


Figure 4.1: The major issue with `Taxonomy2Strategy` is that each successive `Taxonomy2` is more expensive than the last to evaluate.

the exponential growth in table columns with each iteration is only exacerbated by the number of uncached evaluations that must be performed in order to fully populate the row, as illustrated in Figure 4.1. Each evaluation of a `Taxonomy2` spawns $k-1$ `Discriminator` evaluations (of which at least one is cached), which in turn minimally spawns a `Taxonomy2` evaluation and (due to the `Discriminator`-finding process being probabilistic) a potentially unbounded number of `AnchorSet` evaluations. These `Taxonomy2` evaluations on $(k-1)$-sets can then be further unrolled into $(k-2)$-`Discriminator` calls, and so on down to the first value of k for which the discriminator is nontrivial. In practice, each value of k on our input increases the number of uncached evaluations for the row by an order of magnitude. On a quad-core laptop, compiled with clang 3.0 with full optimizations, creating the Kramer-Mesner matrix using this strategy took eight hours, of which the majority was dedicated to the last iteration alone.

Table 4.2: The RecyclerStrategy in finding a 6-(33,8,36) design over $P\Gamma L_2(32)$

k	ρ_k	Columns in Table	GInvariant class	Discrimination	Uncached evaluations
5	3	29	AnchorSet	3/3	29
6	13	81	AnchorSet	13/13	81
7	32	308	AnchorSet	32/32	308
8	97	684	AnchorSet	97/97	684

4.1.2 The Discriminator-free Strategy

It becomes apparent that the Taxonomy2Strategy becomes untenable if k is large but the first nontrivial iteration occurs early, assuming that each uncached evaluation takes roughly the same amount of time to evaluate (given that a function's dependent evaluations, if any, are computed before the function itself is evaluated). It's also to be noted that it is mentioned in [21] that a Taxonomy1 alone could almost fully discriminate the $k = 8$ iteration. Though that claim is not fully tested, the difference between having only 1368 uncached evaluations from a Taxonomy1 and an AnchorSet compared to 44004 uncached evaluations from a Taxonomy2 is too great to ignore.

Thus, we have created a second strategy where the aim is to avoid evaluating (and thus computing) the Discriminator or any other GInvariant which rely on other GInvariant evaluations, which removes Taxonomy2 from consideration. This strategy takes advantage of the fact that both AnchorSet and Taxonomy1 work for different input sizes, and involves exclusively using either of these two GInvariants to construct a Discriminator. Once one is found, the same GInvariants are used as the initial collection in the search for the next Discriminator.

The RecyclerStrategy proved to be far more effective on the same input compared to the Taxonomy2Strategy, as summarized in Table 4.2: it was capable of full discrimination in all iterations using a single AnchorSet in roughly one hour on an quad-core laptop compiled with clang 3.0 with full optimizations. The strategy also appears to refute the claim in [21] that AnchorSet is unreliable in discriminating orbits, at least with the same group and parameters that they had used.

Both tables were compiled using the FullCandidateGenerator rather than the DefaultCandidateGenerator used by default in the TablePruner.

Table 4.3: Run times for a 6-(33,8,36) design over $\mathrm{P\Gamma L}_2(32)$ using different `Pruner`s

k	ρ_k	Number of candidates	`ExplicitPruner`	`TablePruner` (`RecyclerStrategy`)	`SetImagePruner`	`MinRepPruner`
5	3	29	6:03.81	1:36.07	0:00.43	0:00.08
6	13	78	1:03:47.50	3:46.68	0:07.12	0:00.37
7	32	231	13:02:58.53	14:40.12	1:54.75	0:01.93
8	97	589	113:33:08.57	33:21.55	20:13.39	0:06.64

4.2 Pruner comparison

One apparent weakness to `TablePruner` and the table method is that none of the G-invariant functions proposed by Magliveras and Leavitt in [21], in fact, remove the requirement of iterating through the group: both anchor sets and the Taxonomy 1 method rely on precomputation of the images of items under the group, and these images may be the size of the group itself. Thus, the evaluations of these G-invariant functions and, to a lesser extent, G-invariants which use them as a dependent, also effectively iterates through the group. Furthermore, the table-based method does not provide any guarantees as to the number of functions that are needed to ensure full discrimination, meaning that a poor choice of `KMStrategy` can result in unbounded runtime (even worse than explicit iteration through the elements of G).

Neither the traditional backtracking algorithm (as implemented by `SetImagePruner`) or Linton's algorithm have these disadvantages. Neither algorithm iterates through the elements of G, instead iterating through the transversal sets for G (or a subgroup therein), which may be efficiently computed on an as-needed basis. It is also to be noted that, like the G-invariant functions used in the table method, the minimum orbit representative function is itself G-invariant (it is, by definition, a discriminator). Thus, we can claim that any `KMStrategy` involving `GInvariant`s which do not guarantee full discrimination in only one row will perform poorly compared to `MinRepPruner`. In practice, we will expect that `MinRepPruner` will thus greatly outperform `TablePruner` with `RecyclerStrategy`.

In Table 4.3 above, the various `Pruner`s are compared to each other by comparing the lifetime of each `KMBuilder` object. All of the tests were run on a quad-core laptop, with the program compiled with clang 3.0 with no optimizations. The times in the table were measured using the Boost Timer library, and does not include the time needed to compute ρ_k, which is performed in `KMBuilder`'s constructor.

Table 4.4: `MinRepPruner` timings for a 6-(33,8,36) design over PΓL$_2$(32)

k	ρ_k	Candidates	Computation Time
5	3	29	0:00:00.02
6	12	78	0:00:00.14
7	32	231	0:00:00.85
8	97	589	0:00:03.25

Table 4.5: CPLEX timings for a 6-(33,8,36) design over PΓL$_2$(32)

Generators for G	(1 2 4 8 16)(3 6 12 24 17)(5 10 20 9 18)(7 14 28 25 19)(11 22 13 26 21)(15 30 29 27 23) (1 18 30)(2 21 12)(3 10 28)(4 31 32)(5 24 14)(6 7 17)(8 25 27)(9 19 20)(11 15 13)(16 23 29)(22 33 26)
Kramer-Mesner matrix size	13×97
Design Found by CPLEX	Orbit of $\{1,2,3,4,5,6,7,8\}$ Orbit of $\{1,2,3,4,5,6,8,13\}$ Orbit of $\{1,2,3,4,5,6,8,26\}$ Orbit of $\{1,2,3,4,5,6,9,22\}$ Orbit of $\{1,2,3,4,5,6,10,15\}$ Orbit of $\{1,2,3,4,5,6,11,12\}$ Orbit of $\{1,2,3,4,5,6,12,26\}$ Orbit of $\{1,2,3,4,5,6,14,24\}$ Orbit of $\{1,2,3,4,5,6,17,19\}$ Orbit of $\{1,2,3,4,5,6,17,33\}$ Orbit of $\{1,2,3,4,5,7,10,20\}$ Orbit of $\{1,2,3,4,5,7,10,32\}$ Orbit of $\{1,2,3,4,5,9,12,24\}$
Total Computation Time	0:01:05.97

4.3 Effectiveness of `CPlexSolver`

Regardless of the choice of `Pruner`, once the Kramer-Mesner matrix is generated, our program use CPLEX to solve it. In this section, all of the results were obtained from running the program on a 64-core server with gcc 4.1.

CPLEX proved to be very capable in finding a simple 6-(33,8,36) design over PΓL$_2$(32), as shown in Table 4.4 and 4.5. Other well-known designs over PΓL$_2$(32) were also found quickly, as summarized in 4.6.

As summaries in Tables 4.7 and 4.8, a simple 5-(36,6,1) design over PGL$_2$(17)$\times C_2$, first discovered in [5], shows that certain Kramer-Mesner matrix equations are more amenable to CPLEX, as its presolve routines may eliminate a large number of rows and columns (compared to the Kramer-Mesner matrix equations over PΓL$_2$(32), for which the CPLEX presolver were unable to do so). This may be attributed to the small value of λ.

However, it appears that integer programming has its limitations. Though finding the Kramer-Mesner matrix was a non-issue (as shown in Table 4.9), in attempting to find an 8-(31,10,93) design over PSL$_3$(5), CPLEX was unable to find a $\{0,1\}$-solution after allowing

Table 4.6: CPLEX timings for a 7-(33,8,10) design over PΓL$_2$(32)

Generators for G	(1 2 4 8 16)(3 6 12 24 17)(5 10 20 9 18)(7 14 28 25 19)(11 22 13 26 21)(15 30 29 27 23)
	(1 18 30)(2 21 12)(3 10 28)(4 31 32)(5 24 14)(6 7 17)(8 25 27)(9 19 20)(11 15 13)(16 23 29)(22 33 26)
Kramer-Mesner matrix size	32×97
Design Found by CPLEX	Orbit of $\{1, 2, 3, 4, 5, 6, 7, 11\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 7, 32\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 8, 15\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 8, 20\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 8, 21\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 8, 23\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 8, 24\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 8, 27\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 9, 11\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 9, 13\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 9, 15\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 9, 17\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 9, 24\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 9, 26\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 10, 11\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 10, 13\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 10, 18\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 10, 19\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 10, 25\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 11, 16\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 11, 26\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 11, 27\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 11, 33\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 12, 20\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 12, 24\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 12, 26\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 12, 32\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 16, 17\}$
	Orbit of $\{1, 2, 3, 4, 5, 6, 17, 33\}$
	Orbit of $\{1, 2, 3, 4, 5, 7, 9, 12\}$
	Orbit of $\{1, 2, 3, 4, 5, 7, 9, 32\}$
Total Computation Time	0:01:03.91

the program to run for over a month, even with the most aggressive optimizations geared towards finding a feasible solution, despite the fact that $\mathbf{A}_{8,10}^{\mathrm{PSL}_3(5)}$ has roughly the same number of rows and fewer columns than $\mathbf{A}_{5,6}^{\mathrm{PGL}_2(17)\times C_2}$. In contrast, Betten et al., in [3], found and enumerated all 138 such designs in an hour using the DISCRETA [4] software suite, which uses lattice basis reduction to solve the Kramer-Mesner matrix equation. It may be the case that the linear relaxation of the Kramer-Mesner matrix equation has found a local minimum, for which there are no integer solutions nearby.

Table 4.7: `MinRepPruner` timings for a 5-(36,6,1) design over $\mathrm{PGL}_2(17) \times C_2$

k	ρ_k	Candidates	Computation Time
2	3	35	0:00:00.00
3	3	67	0:00:00.01
4	17	65	0:00:00.02
5	48	249	0:00:00.16
6	259	545	0:00:00.73

Table 4.8: CPLEX timings for a 5-(36,6,1) design over $\mathrm{PGL}_2(17) \times C_2$

Generators for G	(1 19)(2 20)(3 21)(4 22)(5 23)(6 24)(7 25)(8 26)(9 27)(10 28)(11 29)(12 30)(13 31)(14 32)(15 33)(16 34)(17 35)(18 36) (3 5 11 12 15 7 17 13 18 16 10 9 6 14 4 8)(21 23 29 30 33 25 35 31 36 34 28 27 24 32 22 26) (3 8 4 14 6 9 10 16 18 13 17 7 15 12 11 5)(21 26 22 32 24 27 28 34 36 31 35 25 33 30 29 23) (2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18)(20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36) (1 3 11 8 15 9 5 7 17 4 14 16 12 6 13 10 18)(19 21 29 26 33 27 23 25 35 22 32 34 30 24 31 28 36)
Kramer-Mesner matrix size	48×259
Design Found by CPLEX	Orbit of $\{1,2,3,4,5,30\}$ Orbit of $\{1,2,3,4,7,34\}$ Orbit of $\{1,2,3,4,8,26\}$ Orbit of $\{1,2,3,4,19,21\}$ Orbit of $\{1,2,3,5,9,26\}$ Orbit of $\{1,2,3,5,27,32\}$ Orbit of $\{1,2,3,6,26,30\}$ Orbit of $\{1,2,3,19,22,36\}$ Orbit of $\{1,2,3,19,23,35\}$ Orbit of $\{1,2,3,19,25,33\}$ Orbit of $\{1,2,3,19,27,31\}$ Orbit of $\{1,2,3,22,23,24\}$ Orbit of $\{1,2,3,22,25,28\}$ Orbit of $\{1,2,3,22,31,35\}$ Orbit of $\{1,2,3,24,25,31\}$
Total Computation Time	0:00:03.41

Table 4.9: `MinRepPruner` timings for an 8-(31,10,93) design over $\mathrm{PSL}_3(5)$

k	ρ_k	Candidates	Computation Time
3	2	29	0:00:00.00
4	3	52	0:00:00.02
5	5	69	0:00:00.06
6	12	101	0:00:00.16
7	22	193	0:00:00.48
8	42	321	0:00:01.11
9	92	440	0:00:02.24
10	174	869	0:00:06.14

Chapter 5

Future Work

The existence and construction of t-designs can be summed up using the generic t-design construction algorithm for solving the Kramer-Mesner matrix equation. In this thesis, we have summarized various approaches to generating the Kramer-Mesner matrix as well as solving the matrix equation, and created a program that does just that. A result from the use of our program showed that Linton's algorithm was sufficiently fast in generating the Kramer-Mesner matrix, and integer programming techniques were able to reproduce many previously-discovered t-designs in reasonable time.

This generic t-design construction program can certainly be refined, be it though new orbit representative construction methods, better candidate pruning methods, or better solvers. Our program has made some accommodations for some of these potential refinements. In particular, we have not used the full functionality of permlib, whose advanced algorithms for various types of searches can be leveraged for greater refinements and new approaches to candidate generation of pruning. Techniques used in the Leiterspiel algorithm's step-downs can be used to create a better `CandidateGenerator` resulting in fewer candidates to prune via Linton's algorithm or traditional backtracking search. Alternatively, it is possible to implement Leiterspiel's step-ups via a `Pruner`. It is believed that permlib already has the tools needed to do all of this, though it may, in the end, perform worse than `MinRepPruner` in practice.

Though we have shown that the table method is largely impractical with the demonstrated `GInvariant` functions due to the need to iterate through the whole group, there may yet exist an easily constructible and cheap to evaluate `GInvariant` function that may allow `TablePruner` to be used as a base for a probabilistic approach to finding Kramer-Mesner matrices. However, such a function would have to compete against Linton's algorithm, which imposes steep performance requirements, in addition to having to guarantee full discrimination with high probability.

Given such functions, a useful addition to the table method is a way to determine a priori that a nontrivial `GInvariant` will evaluate to a constant value for all candidates, making it a useless and unnecessary addition. Currently, this function would still be added rather than discarded. Another useful addition to the table method would be a `KMStrategy` that makes use of state, which may help in improving the table method with regards to `GInvariant` selection in order to make the table method more competitive to Linton's algorithm.

Similarly, we have not fully utilized the advanced functionality of CPLEX, and fine-tuning the use of CPLEX may offer better performance.

Relating to this is the development of better `Solver`s that do not directly use the Kramer-Mesner matrix equation. One such `Solver` which may prove to obtain results in a more timely manner comes from Moura [22], which states that a $\{0,1\}$-solution to the matrix equation of Theorem 6 is also the optimal solution to both the **set-packing model** and the **set-covering model**, both of which are binary integer programming problems. The former maximizes $[1,\ldots,1]^T\mathbf{x}$ subject to $\mathbf{A}_{t,k}\mathbf{x} \leq [\lambda,\ldots,\lambda]^T$, and the latter minimizes the same objective function subject to $\mathbf{A}_{t,k}\mathbf{x} \geq [\lambda,\ldots,\lambda]^T$. It is believed that this can be adapted to solving Kramer-Mesner matrix equations in a straightforward manner by substituting in L_k in the objective function and replacing the regular incidence matrix with the Kramer-Mesner matrix.

As an alternative to CPLEX, a hybrid approach incorporating techniques from both lattice basis reduction and integer programming may prove to solve the Kramer-Mesner matrix equation faster than either technique alone. The `Solver` hierarchy does not prescribe in any way how the Kramer-Mesner matrix equation is to be solved, or whether or not the Kramer-Mesner matrix equation is used as-is; with some sets of inputs, the matrix may be too large or otherwise too difficult to solve without additional processing. Some progress, such as those in [14], have been made to this effect, and the incorporation of these efforts into a better `Solver` may prove to be useful.

Another potential addition to our program comes from work that has, up until this point, been specialized into finding 2-designs but have yet to be generalized for larger t. An example of this is in [23], which proposes that a t-design exists if and only if a specific system of equations of variables with degree t has a solution. Whether this can be combined with the Kramer-Mesner matrix equation approach in any meaningful way is still unknown.

Chapter 6

References

[1] IBM ILOG CPLEX optimization studio. `http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/`.

[2] W. O. Alltop. Extending t-designs. *J. Combinatorial Theory Ser. A*, 18:177–186, 1975.

[3] Anton Betten, Adalbert Kerber, Reinhard Laue, and Alfred Wassermann. Simple 8-designs with small parameters. *Des. Codes Cryptogr.*, 15(1):5–27, 1998.

[4] Anton Betten, Reinhard Laue, and Alfred Wassermann. *DISCRETA – a tool for constructing t-designs*. Lehrstuhl II für Mathematik, Universität Bayreuth, 1998. `http://www.mathe2.uni-bayreuth.de/betten/DISCRETA/Index.html`.

[5] Anton Betten, Reinhard Laue, and Alfred Wassermann. A Steiner 5-design on 36 points. *Des. Codes Cryptogr.*, 17(1-3):181–186, 1999.

[6] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[7] Murray R. Bremner. *Lattice basis reduction*, volume 300 of *Pure and Applied Mathematics (Boca Raton)*. CRC Press, Boca Raton, FL, 2012. An introduction to the LLL algorithm and its applications.

[8] Beman Dawes, David Abrahams, and Rene Rivera. Boost C++ Libraries. `http://www.boost.org/`.

[9] Matteo Fischetti, Fred Glover, and Andrea Lodi. The feasibility pump. *Math. Program.*, 104(1):91–104, September 2005.

[10] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008.

[11] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, pages 85–103. Plenum, New York, 1972.

[12] Earl S. Kramer, David W. Leavitt, and Spyros S. Magliveras. Construction procedures for t-designs and the existence of new simple 6designs. In C.J. Colbourn and M.J. Colbourn, editors, *Annals of Discrete Mathematics 26 Algorithms in Combinatorial Design Theory*, volume 114 of *North-Holland Mathematics Studies*, pages 247 – 273. North-Holland, 1985.

[13] Earl S. Kramer and Dale M. Mesner. t-designs on hypergraphs. *Discrete Mathematics*, 15(3):263 – 296, 1976.

[14] Vedran Krčadinac, Anamari Nakić, and Mario Osvin Pavčević. The Kramer-Mesner method with tactical decompositions: some new unitals on 65 points. *J. Combin. Des.*, 19(4):290–303, 2011.

[15] Donald L. Kreher and Stanisław P. Radziszowski. Constructing 6-$(14, 7, 4)$ designs. In *Finite geometries and combinatorial designs (Lincoln, NE, 1987)*, volume 111 of *Contemp. Math.*, pages 137–151. Amer. Math. Soc., Providence, RI, 1990.

[16] Donald L. Kreher and D.R. Stinson. *Combinatorial algorithms: generation, enumeration, and search.* CRC Press series on discrete mathematics and its applications. CRC Press, 1999.

[17] Hans Kurzweil and Bernd Stellmacher. *The theory of finite groups.* Universitext. Springer-Verlag, New York, 2004. An introduction, Translated from the 1998 German original.

[18] Reinhard Laue. Construction of combinatorial objects—a tutorial. *Bayreuth. Math. Schr.*, (43):53–96, 1993. Konstruktive Anwendungen von Algebra und Kombinatorik (Bayreuth, 1991).

[19] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4):515–534, 1982.

[20] Steve Linton. Finding the smallest image of a set. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, ISSAC '04, pages 229–234, New York, NY, USA, 2004. ACM.

[21] Spyros S. Magliveras and David W. Leavitt. Simple 6-(33, 8, 36) designs from $\mathrm{P\Gamma L}_2(32)$. In *Computational group theory (Durham, 1982)*, pages 337–352. Academic Press, London, 1984.

[22] Lucia Moura. Polyhedral methods in design theory. In *Computational and constructive design theory*, volume 368 of *Math. Appl.*, pages 227–254. Kluwer Acad. Publ., Dordrecht, 1996.

[23] Lucia Moura. *Polyhedral aspects of combinatorial designs.* PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 1999. AAINQ41034.

[24] Thomas Rehn. Funadmental permutation group algorithms for symmetry computation. diploma thesis, Otto von Guericke University Magdeburg, February 2010.

[25] John S. Rose. *A course on group theory.* Dover Publications Inc., New York, 1994. Reprint of the 1978 original [Dover, New York; MR0498810 (58 #16847)].

[26] Bernd Schmalz. The t-designs with prescribed automorphism group, new simple 6-designs. *J. Combin. Des.*, 1(2):125–170, 1993.

[27] Luc Teirlinck. Non-trivial t-designs without repeated blocks exist for all t. *Discrete Math.*, 65(3):301–311, July 1987.

[28] Alfred Wassermann. Finding simple t-designs with enumeration techniques. *J. Combin. Des.*, 6(2):79–90, 1998.

[29] Laurence A. Wolsey. *Integer programming.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons Inc., New York, 1998. A Wiley-Interscience Publication.