AMSI VACATION RESEARCH SCHOLARSHIPS 2020–21

Get a Thirst for Research this Summer

Computational Searches for Combinatorial Designs

Tiana Tsang Ung

Supervised by Dr Thomas Britz

UNSW Sydney

Vacation Research Scholarships are funded jointly by the Department of Education, Skills and Employment and the Australian Mathematical Sciences Institute.

Abstract

In this project, computational tools and techniques were applied to various aspects of combinatorial design theory. This report describes the implementation strategies used in a computer search conducted to find a $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}, 5, 1)$-difference matrix, as well as the method used to construct the end result. A reduction from this original search problem to the clique problem on graphs is outlined. Additionally, a Python program used to re-arrange group divisible designs with block size k into a specific form called Michael's Edge is provided. A brief introduction to assertion-based reasoning is included, outlining how these methods may applied in the verification of this specific program.

Acknowledgements

The success of this project owes a lot equally to Julian Abel, Yudhi Bunjamin, and Diana Combe, for their ongoing assistance with topics explored in this project. Thank you all for your friendly suggestions and advice throughout the summer, and for letting me listen in on your ideas during meetings.

Thanks especially to Thomas Britz, first for suggesting that I take up this opportunity, but also for always being available to provide useful advice and supportive feedback.

Contents

1 Introduction

Design theory is an area of combinatorics concerned with the study of certain mathematical objects called *designs*. A design consists of some arrangement of a finite number of elements, satisfying given conditions which often impose some sort of regularity or symmetry within the structure of these designs.

In design theory we are often concerned with the construction and enumeration of certain types of designs with specific parameters. Other interesting problems involve the study of the structural properties of designs, and the applications of these properties to other fields of science and mathematics. Designs have applications in areas such as coding theory, graph theory, algorithms, and in the design of effective experiments in statistics. More detailed examples, as well as the theory underlying these applications can be found in [5].

Two types of combinatorial designs that are of interest in this report are differences matrices, and group divisible designs.

A difference matrix is a matrix containing entries from a specified group, so that the taking the component-wise differences of any two distinct rows in the matrix produces each element in the group the same number of times. A more precise definition of difference matrices, as well as a method used for their construction is given in Section 2.

$$\begin{array}{ccccccc} 0 & 1 & 4 & 2 & 6 & 3 & 5 \\ 0 & 2 & 1 & 4 & 5 & 6 & 3 \\ 0 & 4 & 2 & 1 & 3 & 5 & 6 \end{array}$$

Figure 1: A difference matrix over $\mathbb{Z}_7$.

In this project, a computer search was conducted to find a difference matrix with specific parameters. Section 3 provides the details of the implementation of this search, including a reduction from the original search problem to a well-studied search problem on graphs.

Difference matrices are mostly used in the construction of other useful designs. Certain difference matrices are equivalent to sets of mutually orthogonal Latin squares, which are of importance in the design of large sampling experiments, as described in [9].

Finding appropriate entries to use in the construction of a large difference matrix with specific parameters can be computationally difficult, even with the method used in this project, due to the number of possibilities involved. So that the search could be conducted in a reasonable amount of time, this search task was split up and conducted in parallel with a cluster of machines. These computation details are also contained in Section 3.

Section 4 describes a program written in Python to assist in the analysis of group divisible designs, which are designs consisting of subsets and partitions of a finite set, structured so that elements within distinct partitions are common to exactly one subset, and otherwise do not interact with each other at all. A clearer definition for this type of design is presented later. Like difference matrices, group divisible designs are useful in the

construction of other types of designs, but also have natural applications to certain real-world situations such as in scheduling problems.

For this project, code was written to re-arrange the elements in a group divisible design so that its associated incidence matrix has a certain ordering within it. This re-arrangement is possible by hand, but automating the process allows for more convenient analysis, and can be incorporated into any future programs written for this purpose. At the end of the section, a short outline of methods which can be used to establish the correctness of computer programs is included.

Overall, this project demonstrates the varying applications of computational tools and related methods, in the exploration of the topics presented in this report.

Statement of Authorship

Thomas Britz was the primary supervisor for the project, and proofread and provided feedback for this report, and provided the graphic in Appendix .

Many of the definitions mentioned in this report have been adapted from other sources. References to the original definitions are provided where necessary.

Julian Abel provided the difference matrix construction method outlined in Section 2, as well as the described base blocks which were used in the actual search.

Everything else is the work of the author, including the writing of the report and the Python code appearing in Appendix B. The C code written for the search task was developed by the author, and the ideas used in the implementation at times originated from discussions with Julian Abel and Yudhi Bunjamin. The implementation also includes calls to functions from the Cliquer C package.

2 Difference Matrix Construction Method

2.1 Search target and definitions for difference matrices

The purpose of the computer search conducted in this project was to search for a specific example of a design called a difference matrix.

The following definition is adapted from [3]. A *difference matrix* with parameters k and λ over a group G is a matrix $D := (d_{ij})$, with k rows and $\lambda|G|$ columns containing entries from G such that for rows $i \neq j$, the multi-set of differences

$$\{d_{i\ell}(d_{j\ell})^{-1} : \ell = 1, \ldots, \lambda|G|\}$$

contains each element of G exactly λ times. The notation (G, k, λ)-difference matrix (DM) may be used to denote such a matrix.

This project was focused on the construction of a $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}, 5, 1)$-DM. Difference matrices with $\lambda = 1$ are useful in the construction of other designs, including the construction of difference matrices with greater values of λ. Here the group $G := \mathbb{Z}_4 \oplus \mathbb{Z}_{32}$ is the set of pairs with first and second component from $\mathbb{Z}_4$ and $\mathbb{Z}_{32}$ respectively. Later we write $\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}$ for the subgroup of G containing the pairs from G with a second component which is divisible by 8. Usual addition is the group operation, as described in Subsection 3.1.

Results exist for a difference matrix with the same parameters over the group $\mathbb{Z}_4 \oplus \mathbb{Z}_4$, given in [[**?**]]. Since $\mathbb{Z}_4 \oplus \mathbb{Z}_4$ is isomorphic (structurally equivalent) to $\mathbb{Z}_4 \oplus 8\mathbb{Z}_4$ which is contained in our desired group, to complete the construction we just need to find the portion of the matrix which gives the remaining differences. Once we have this incomplete matrix, we can produce the complete difference matrix by first re-labelling the entries of the existing difference matrix over $\mathbb{Z}_4 \oplus \mathbb{Z}_4$, and then appending this to the search result.

To describe this part of a difference matrix, we use the following definition from [8].

Let H be a subgroup of G. An *incomplete difference matrix* with parameters k and λ over G is a matrix $A := (a_{ij})$, with k rows and $\lambda|G\backslash H|$ columns containing entires from G such that for rows $i \neq j$, the multi-set of differences

$$\{d_{i\ell}(d_{j\ell})^{-1} : \ell = 1, \ldots, \lambda|G\backslash H|\}$$

contains each element of $G\backslash H$ exactly λ times. The notation (G, H, k, λ)-incomplete difference matrix (IDM) may be used to denote such a matrix.

With the construction method described below, the required search target becomes an $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}, \mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}, 5, 1)$-IDM.

2.2 Construction of incomplete difference matrix from base blocks

The search begins with 28 base blocks (a block can be thought of as an ordered set), each containing four elements from $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$. These base blocks are selected to begin with some, but not all, of the necessary properties for the construction of the desired IDM. The search program described in Section 3 modifies the 28 base blocks to find new blocks which satisfy all the properties required to complete the construction.
Write the i-th block as (a_i, b_i, c_i, d_i) and let $e := (0, 0)$ (the identity element within $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$). We want the following properties to be satisfied:

i. $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32})\backslash(\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}) = \bigcup_i (a_i - d_i, d_i - a_i, b_i - c_i, c_i - b_i)$,

ii. $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32})\backslash(\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}) = \bigcup_i (a_i - b_i, b_i - a_i, c_i - d_i, d_i - c_i)$,

iii. $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32})\backslash(\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}) = \bigcup_i (a_i - c_i, c_i - a_i, b_i - d_i, d_i - b_i)$, and

iv. $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32})\backslash(\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}) = \bigcup_i (a_i, b_i, c_i, d_i)$.

If we have this, then the matrix

$$B_i := \begin{pmatrix} a_i & b_i & c_i & d_i \\ b_i & a_i & d_i & c_i \\ c_i & d_i & a_i & b_i \\ d_i & c_i & b_i & a_i \\ e & e & e & e \end{pmatrix}$$

is structured so that for all values of i, if we append the above matrices together, the produced matrix

$$B := (B_1 \,|\, B_2 \,|\, \ldots \,|\, B_{28}),$$

is the desired $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}, \mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}, 5, 1)$-incomplete difference matrix. The 28 initial base blocks are selected to satisfy properties i-iii, but not yet property iv above.

3 Difference Matrix Search Implementation

To construct the required IDM from the base blocks described in Subsection 2.2, what remains is to modify the base blocks in a way which preserves the differences between elements within a block, so that the union of all resulting blocks (when viewed as sets) forms exactly the set of elements in $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}) \backslash (\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32})$. Once this is done, the produced blocks also satisfy the final property iv required for the construction.

We achieve this by conducting a large computational search. This section outlines the implementation details and strategies used in this search.

The search code for this task was written using the C programming language. Some data for the search was pre-computed using programs written with Python. Computations were later run on the Katana high performance computing cluster at UNSW Sydney.

3.1 Description of the search problem

The initial base blocks may be modified using two different operations. First, a constant element from the group $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$ may be added to each element in the base block. Within $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$, arithmetic occurs modulo 4 for the first component, and modulo 32 for the second component. For a base block $B := (\,(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\,)$, if we add the pair (a, b) the resulting block is given by

$$(\,(a + x_i \pmod 4,\, b + y_i \pmod{32}) : i = 1, \ldots, 4\,).$$

The other allowed operation involves multiplying each component of each pair within a block by -1. Using the same notation as above, this produces the block

$$(\,(-a \pmod 4,\, -b \pmod{32}) : i = 1, \ldots, 4\,).$$

Notice that applying either of these operations does not change the differences between any two elements in the base block. With the given base blocks, we begin with 112 elements in total. This is exactly the number of elements contained in $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}) \backslash (\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32})$. So in the described implementation, once we apply these operations to find a mutually disjoint collection of resulting blocks containing only elements from $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}) \backslash (\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32})$, we know that taking the union of the elements in these blocks must give us this entire set. Since the above operations preserve differences between elements in the same block, we would then have all properties required for the difference matrix construction.

Before implementing the search, it is worthwhile to consider whether any aspects of the search problem can be simplified to reduce the amount of computation required.

3.2 Pre-computation

Pre-computation involves calculating any repeatedly used values in advance, and storing them in a suitable data structure so that they may be accessed when needed instead of re-computed every time. This often results in a speed-up of the program, with the trade-off of requiring more memory. With pre-computation, results of calculations which would otherwise take multiple operations to compute may be obtained in a single memory-access operation, once the data structure containing these required results has been established.

Each subsequent stage of pre-computation reduces the significance of any previously used pre-computation steps. Here every stage is mentioned to outline the program's development process.

A naive approach to the computer search would be to trial every possible combination of possible operations across all the base blocks. Refinement of this search begins with the pre-computation of any values which may be needed while trialing these operations on each base block.

We note that the addition of two elements from $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$ involves taking the sum of each component from the first element with the corresponding component from the second element, modulo 4 and 32 for the first and second component respectively. Implementing this addition in a computer program involves multiple arithmetic operations and the access of multiple data values, since we must read in both components of each pair involved in the addition.

Rather than manipulating these pairs in the implementation, each pair (a, b) from $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$ is uniquely represented using the integer given by $32a + b$. The disadvantage of this is that it slightly complicates operations between two elements where modulo arithmetic is involved. Generally in the program, conversion from the single integer to the ordered pair representation occurs prior to applying any operations, and then any results are converted back into their integer representations for storage, again increasing the number of computational steps required. To avoid conversion and the multiple arithmetic steps here, an addition table containing the sum of any two elements from $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$ is constructed in advance. This table represents each sum with a single integer, and the integer representation is also used instead of the pair representation to index the entries of this table. With this, we only need one memory access operation to read each element, and obtaining the result of the addition requires just one step.

This table is symmetric across the main diagonal (upper-left corner to bottom-right corner) since addition in $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$ is commutative. When constructing similar operation tables for larger groups, or on machines with tighter space constraints, the entire table does not need to be pre-computed provided that the corresponding group with the desired operation is Abelian. Space limitations were not a concern in this project, so the complete table was constructed for simplicity.

It would also be possible to extend this addition table, or generate a separate table to include the results of subtracting a given element from another (to account for the case where we first multiply the base block by -1). Alternatively, a separate table could be constructed for this purpose. This was not implemented in the search conducted for this project, as further pre-computation would make any potential improvements from this step negligible.

After the pre-computation of this addition table, a next step would be to filter the list of potential combinations of operations for each base block. For the construction of the incomplete difference matrix, none of the resulting blocks we obtain may contain any element from $\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}$. Before considering the interactions between different combinations of operations over all of the blocks, the set of possible operations applicable to each individual base block may be filtered to exclude possibilities which would produce any of these elements. This step significantly reduces size of the search space in the end implementation.

For each base block, we first try to add every possible constant from $\mathbb{Z}_4 \oplus \mathbb{Z}_{32}$, keeping a record of the constants that do not produce elements from $\mathbb{Z}_4 \oplus 8\mathbb{Z}_{32}$. This process is also repeated for the block obtained by multiplying the components of the elements within this initial block by -1. From this we two lists of potential constants for each base block, accounting for all valid combinations of the two allowed operations for each base block. Once this filtering has been done in advance, the main body of the search program never has to check and discard any produced blocks for these prohibited elements.

Extending this further, each possible constant in the two lists for each base block can instead be replaced with the resulting block it would produce. All that remains at this point is to select one resulting block for each base block, so that none of the chosen blocks have an element in common.

3.3 Reduction to the clique problem on graphs

In languages with well-developed implementations for sets, overlapping elements between blocks can be tested for in constant time after they have been converted into set data structures. Since no such implementation exists in the standard C library, a relation table indicating whether two blocks do not share any elements is constructed instead. In the implementation a four-dimensional, fixed-size array is used, but abstractly this structure can be viewed as a square matrix M with as many rows and columns as the total number of possible resulting blocks. Then the entry M_{ij} is a zero if the i-th and j-th blocks originate from the same blocks, or if the two blocks share an element. Otherwise, the value of M_{ij} is set to one.

Instead of trialing each possible combination of operations over all base blocks, we first perform the described pre-computation and construct the required look-up tables. Then after indexing all possible result blocks, to trial a potential combination of operations we select a subset of indices and check the compatibility of result blocks which they correspond to. If none of the selected blocks do not overlap, then for any two indices i and j from this set, the relation table entry M_{ij} must equal 1. This gives a natural reduction to a well-studied problem on graphs.

We may interpret this relation table directly as an incidence matrix for a graph. The resulting graph is simple and undirected, and the vertices within it can also be partitioned into 28 partite sets (one set for each base block) such that any two vertices within the same set do not share an edge.

Now we must select a subset of vertices, one from each partite set, such that any two vertices within the subset are connected by an edge. No such subset containing more than 28 vertices could exist, since we are considering a 28-partite graph. Therefore, the subset of vertices that we are looking for has maximal size. This problem is

exactly the maximal clique problem.

Clique problems generally tend to be computationally difficult, but are problems of interest in many aspects of computer science due to both their complexity and applications. Due to this, well-developed implementations of clique-finding algorithms already exist. Here search algorithms from the Cliquer C package by Niskanen and Østergård [7] are used in the search to simplify the implementation. Using known algorithms also decreases the likelihood of potential bugs in the program. The C code produced for this project implements the required pre-computation steps described above, and then constructs the relevant graph before calling functions from Cliquer. Additional effort is required to output the graph into DIMACS graph file format, which is the input graph format Cliquer uses.

3.4 Dividing up the search

If we do not split up the search, the constructed input graph ends up with 3584 vertices which is too large for feasible computations. Before beginning the search, this graph is divided into smaller subgraphs, and then each subgraph is passed as input to one running instance of the program. This ensures that each instance of the program may be run in a feasible amount of time, and also allows the sub-searches to occur in parallel, increasing the amount of the search space which may be explored in real-time.

To split up the search, we pre-select three vertices to always include in the clique for the current sub-search. If these three vertices do not form a clique themselves (here just a cycle of size 3), then they could not be contained within a larger clique. To account for this, Python code was written to enumerate the possible vertex sets which form 3-cycles, outputting these sets as sequences of numbers to a text file. Before starting multiple instances of the search program, a batch of individual sub-graph files is first generated. When generating each of these graph files, the C program reads in one set of vertices from this text file, and then takes the sub-graph containing the three vertices themselves, along with all vertices that share an edge with all vertices in this set. Selecting three vertices gives 1294510 possible subgraphs, each with just under 2000 vertices.

For the computation itself, each sub-search program is run on a machine contained within the Katana computing cluster. Initially, a few hundred subgraphs were generated and searched simultaneously. These searches were enough to find a required solution.

3.5 Limitations and possible improvements

The ordering of the vertices in the clique search has a considerable impact on the speed of the algorithm. Through experimentation, it was determined that the vertices should be listed so that the three pre-selected vertices appear first, and all other vertices are grouped based on which partite set they belong to. This is already the order in which the C code constructs each subgraph. The Cliquer package contains its own ordering functions, for example, functions for a random ordering or ordering based on a greedily selected colouring. These options were not used in the project as they did not improve the speed of the search when compared to the original ordering.

A potential improvement would be to maintain the current partite set grouping within the current ordering, but then to apply Cliquer's ordering functions to the list of vertices in each individual partite set. Any other improvements would have to take advantage of the structure and properties specific to the given base blocks and the constructed graph in this problem. As noted above, the clique search in this project occurs in a 28-partite graph. The Cliquer search functions use a generic algorithm which make no assumptions about the structure of the given input. Possibly, the use of a more specialised algorithm, such as a k-clique algorithm for k-partite graphs described in [4] could improve the speed of the search, however, using this would require the implementation details of the clique-search to be handled ourselves. It was decided that for the purpose of this project, the resources required for this would not be worth the potential performance increase, but this may be considered in any future searches.

4 A Program for the Analysis of Group Divisible Designs

4.1 Required definitions for group divisible designs

The definition below has been adapted from [2].

Let X be a finite, non-empty set X. Consider a partition $\mathcal{G}$ of the elements of X and a collection of k-subsets $\mathcal{B}$ of X. In a *group divisible design*, we call the elements in X *points*, the parts within $\mathcal{G}$ *groups*, and the subsets in $\mathcal{B}$ *blocks*. The triple $(X, \mathcal{G}, \mathcal{B})$ is a *group divisible design* with block size k (or a k-GDD) if, any two distinct points from different groups appear together in exactly one block, and each block does not contain more than one point from each group.

The following definitions are given in [6].

The *incidence matrix* of a k-GDD is a matrix with $|X|$ rows and $|B|$ columns. The entry in row i and column j of this matrix is equal to 1 if point i is contained in block j, and 0 otherwise.

Two k-GDDs are *isomorphic* or *equivalent* if their corresponding incidence matrices may be obtained from one another by a series of row and column swaps. If M_1, M_2 are the incidence matrices of the two k-GDDs, then another way of writing this is that there exist two permutation matrices A and B, such that

$$M_1 = AM_2B.$$

4.2 Description of Michael's Edge form

Another part of this project involved writing a program using Python, which could be used to assist in analysing the structure of k-GDDs. This program takes the incidence matrix of a k-GDD as input, and outputs the incidence matrix of an equivalent k-GDD arranged in a particular form called Michael's Edge form, as described below and in [1]. An example of an incidence matrix in this form is given in Appendix A.

Consider a k-GDD with m groups, and the group sizes $g_1, \ldots, g_m$. Then for a point in a group of size g_i, write r_i for the number of blocks which contain this particular point.

Suppose that we are given the two groups $G_1 := \{a_1, \ldots, a_{g_1}\}$ and $G_2 := \{b_1, \ldots, a_{b_2}\}$. In *Michael's Edge* form, the incidence matrix of a k-GDD is structured as follows:

- For each point $a_i \in G_1$, the row corresponding to a_i contains a one in the columns $r_1(i-1)+1, r_1(i-1)+2, \ldots, r_1(i-1)+r_1 = r_1 i$.
- For each point $b_j \in G_2$, the row corresponding to b_i contains a one in the columns $j, r_1 + j, 2r_1 + j, \ldots, (g_1 - 1)r_1 + j$. If $r_2 > g_1$, a one also occurs columns $g_1 r_1 + (r_2 - g_1)(j-1) + 1, g_1 r_1 + (r_2 - g_1)(j-1) + 2, \ldots, g_1 r_1 + (r_2 - g_1)j + 1$.

The Python program is written so that additionally, points which don't belong to the two selected groups for the Michael's Edge form are also ordered in a natural way. Once an equivalent GDD in Michael's Edge form is obtained, further permuting of the resulting incidence matrix is done such that:

- points within each group appear together (their corresponding rows in the incidence matrix are adjacent),
- groups are listed in decreasing size order, and if two groups are the same size, the one containing the point with the earliest non-zero entry appears first,
- points within each group are ordered based on increasing block number (their corresponding rows are ordered by their earliest non-zero entry).

The Python code which arranges a given k-GDD into this form can be found in Appendix B.
Re-ordering designs into a unique form such as the Michael's Edge form is useful in the analysis and comparison of designs. Doing so can provide a clearer view of the structure of a particular design, or can be used to distinguish two potentially different designs of the same size. The ability to tell distinct designs apart is especially useful in problems about enumeration, where the aim is to determine how many structurally unique designs of a certain kind exist for given parameters.

4.3 Checking the correctness of computer programs

Parts of the Python code written in this project may be re-used in larger programs in the future. Once the code has been incorporated into a larger program, identifying any previously uncaught errors or missed assumptions require far more effort. To prevent this, additional efforts were made to ensure that the functions within the code behaved as expected in isolation. Apart from routine testing, more structured, assertion-based verification methods were used to give a greater guarantee of the program's correctness, and to identify any key assumptions. The usefulness of these methods are discussed below.
Verification of computer programs using assertions involves devising a set of desirable properties about the state of the program which imply its correctness when execution ends. In other words, it involves coming up with properties about the data structures and variables used in the code which have to hold at certain parts of the program for the resulting output to be correct. Since we have to check for these properties, they need to be expressed in a way which doesn't allow for any ambiguity. Once we have these properties, all that remains is to justify that they hold when the program ends. Then we would know that the output result is correct, given that the selected properties are an accurate description of the result we wanted the program to compute, and that any assumptions made along the way are met.
Due to the precise nature of the specification of the program (we have a clear definition of what it means for a k-GDD to be in Michael's Edge form), and the simplicity of the implementation, the correctness of this code could be justified by inspection. Still, the required properties for its correctness are identified below. Here the act of analysing the code to come up with the necessary properties may have been more useful than the justification itself, since doing so allows for the identification of the important assumptions made about the program's usage which may have not been considered previously.
Similar methods can be applied to more complex programs. In these cases, it may be necessary to reason more formally about the programs, especially if the required properties are less obvious or harder to state.

4.4 Necessary properties for the written program

Suppose that we wish to fix two groups of size g_1 and g_2 when re-arranging the given GDD into Michael's Edge form.

In the implemented program, we first generate a matrix with $g_1 + g_2$ rows which has the structure of the incidence matrix only containing the points from these fixed groups after the GDD has been placed in Michael's Edge form. If this matrix is constructed directly from the definition of Michael's Edge above, then it is easy to check that it corresponds to part of the correct output. So one simple way of stating the property that the two fixed groups have been arranged in the correct form is to say that the sub-matrix containing rows $1 - (g_1 + g_2)$ from the output M is equal to this generated matrix.

An obvious property is that the output GDD must be isomorphic to the GDD to the one provided as input. By definition, we can restate this as the property that the output matrix M can be obtained from the input matrix through a series of row and/or column swaps.

The three additional ordering requirements are less simple to state as well-defined properties that can also be easily checked for. One way would be to define a function which takes in a list and identifies it's first non-zero entry, and then check that the value of this function strictly increases when applied to the points in each group, but this is not much easier to reason about. Occasionally in these cases, we have to settle for intuitive but less rigorous arguments, or accept that we must make certain assumptions about the program and its inputs. For example, the correctness of this implementation heavily relies on the correctness of the built-in sorting functions within Python. If we can assume that these functions work as expected, the correctness of our own code often follows immediately.

Although these properties may not always be useful for complete verification of programs without spending a large amount of resources to formally prove that they hold, designing the code with these properties in mind allows for more structured debugging, and produces more elegant code in general.

5 Conclusion

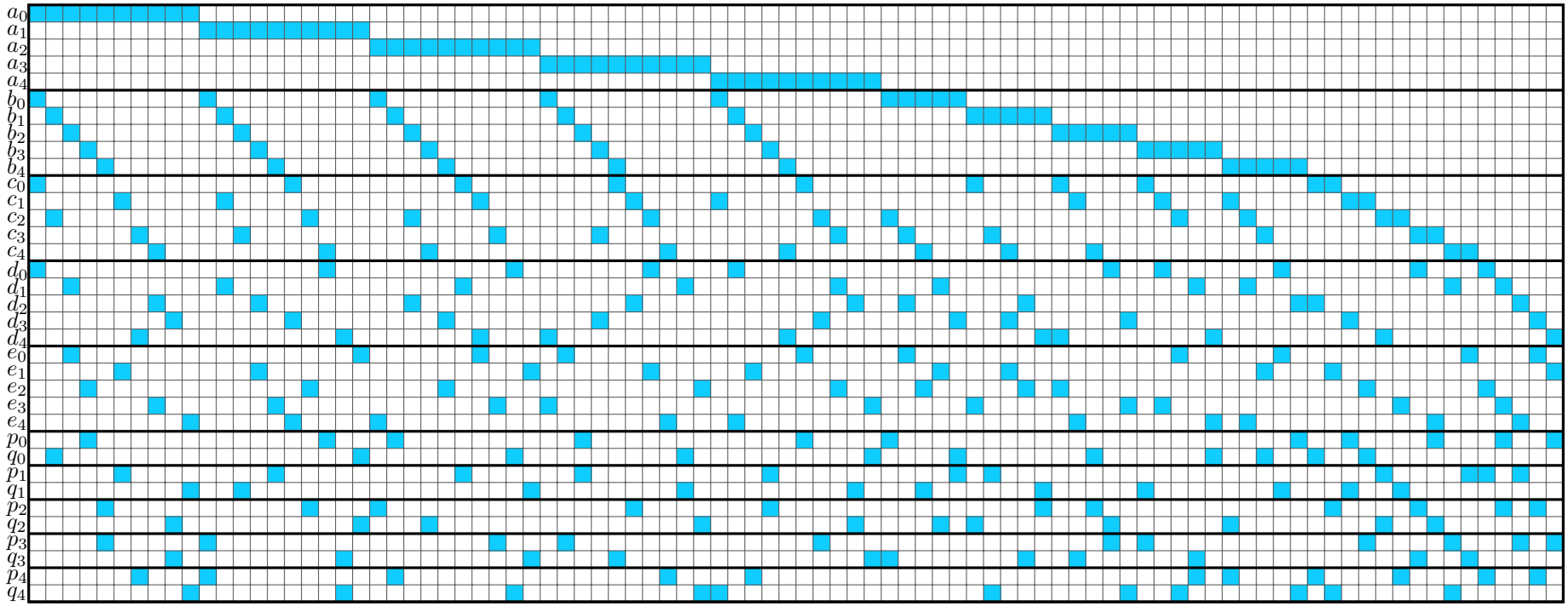
This projected involved the exploration of multiple different aspects of design theory, through the use of mostly computational methods. Combinatorial design theory is the study of deliberately structured objects called designs. A design can be viewed as a finite set of elements, arranged in a way that satisfies certain conditions. The particular conditions depend on the type of design, but generally provide some kind of symmetry, balance, or regularity to the design's structure.

A difference matrix is a type of design which consists of elements from a group which have been arranged into a matrix, such that the differences obtained by subtracting corresponding entries in two distinct rows produces each group element the same number of times.

In this project, a successful search was conducted using the Katana computing cluster at UNSW Sydney. The resulting $(\mathbb{Z}_4 \oplus \mathbb{Z}_{32}, 5, 1)$-difference matrix is to appear in a co-authored paper in progress [8]. Implementation of this search involved first reducing the search for the necessary values for the DM construction to the well-studied clique problem on graphs. Construction and more implementation details are outlined in Sections 2 and 3 of this report.

Group divisible designs consist of a finite set of elements which have been arranged into parts and subsets so that certain conditions are met. We write k-GDD for an arrangement where all each of the subsets contains exactly k elements. In Section 4, a form called Michael's Edge for these designs is described, and the relevant Python code used to automatically re-arrange these k-GDDs designs into Michael's edge form for clearer structural analysis is provided. The report concludes with a brief overview of how verification methods can be used to increase our confidence in computer programs we write, and to identify potential assumptions necessary when code is to be re-used.

A Appendix: Example Incidence Matrix in Michael's Edge form



B Appendix: Python Program for Designs in Michael's Edge form

```
# Functions re-order a k-GDD incidence matrix into Michael's Edge form
from numpy import array, zeros, array_equal, copy, asarray

def genPartM(g1, g2, k, b, v):
    # Generates sub-incidence matrix corresponding to points in G1 \cup G2
    # when placed in Michael's Edge form

    # Given parameters:
    # g1 - size of first fixed group in edge
    # g2 - size of seond fixed group in edge
    # k  - size of blocks
    # b  - number of blocks
    # v  - number of points

    # g1, g2 renamed to ensure g1 >= g2
    gs= sorted([g1,g2], reverse=True)
    g1= gs[0]
    g2= gs[1]

    r1 = (v-g1)//(k-1)
    r2 = (v-g2)//(k-1)

    # start with an empty (g1+g2) by b matrix of zeroes
    P= zeros([g1+g2,b], dtype=int)

    for i in range(1,g1+1):
        xs= [ (r1*(i-1))+a for a in range(1,r1+1) ]
        for j in xs:
            P[i-1][j-1] = 1

    for i in range(1,g2+1):
        xs= [ (r1*a)+i for a in range(g1) ]

        if r2>g1:
           xs+= [(g1*r1)+((r2-g1)*(i-1))+a for a in range(1,(r2-g1)+1)]

        for j in xs:
            P[g1+i-1][j-1] = 1

    return P


```

```
43 def reorderGDD(incidenceM, g1, g2, k, b, v, G, typeL):
44     # Arranges the incidence matrix into Michael's Edge form
45 
46     # Given parameters:
47     # incidenceM - input incidence matrix as 2d list
48     # g1 - size of first fixed group in edge
49     # g2 - size of seond fixed group in edge
50     # k  - size of blocks
51     # b  - number of blocks
52     # v  - number of points
53     # G  - collection of groups as dictionary
54     # typeL - list of group sizes
55 
56     # Construct a list of group sizes corresponding to groups
57     # that still need to be added to temporary matrix
58     fixedGrps = sorted([g1,g2], reverse=True)
59     grpSizesLeft = typeL[:]
60     grpSizesLeft.remove(g1)
61     grpSizesLeft.remove(g2)
62     grpSizesLeft = sorted(set(grpSizesLeft), reverse=True)
63 
64     tempM = []
65     # Copy rows into temporary matrix
66     if g1 == g2:
67         # First place the two fixed groups at the top of the matrix
68         for l in G[g1][:2]:
69             for p in l:
70                 tempM.append(incidenceM[p])
71 
72         # Then fill in remaining groups in decreasing size order
73         for t in grpSizesLeft:
74             if t == g1:
75                 for l in G[g1][2:]:
76                     for p in l:
77                         tempM.append(incidenceM[p])
78             else:
79                 for l in G[t]:
80                     for p in l:
81                         tempM.append(incidenceM[p])
82     else: # If g1, g2 distinct
83         for t in fixedGrps:
84             for p in G[t][0]:
85                 tempM.append(incidenceM[p])
86 
87         for t in grpSizesLeft:
88             if t in fixedGrps:
```

```
89                  for l in G[t][1:]:
90                       for p in l:
91                          tempM.append(incidenceM[p])
92              else:
93                  for l in G[t]:
94                   for p in l:
95                          tempM.append(incidenceM[p])
96
97      assert len(incidenceM) == len(tempM)
98      incidenceM = copy(tempM)
99
100     # Reset temporary matrix
101     tempM = zeros([v,b], dtype=int)
102
103     # List of columns which are yet to be copied into temporary matrix
104     remainingCols = list(range(b))
105
106     # Generate model sub-matrix
107     A = genPartM(g1, g2, k, b, v)
108
109     # Copy
110     for j1 in range(b):
111         for j2 in remainingCols:
112             if array_equal(A[0:g1+g2, j1:j1+1], incidenceM[0:g1+g2,j2:j2+1]):
113                 tempM[:, j1] = incidenceM[:, j2]
114                 remainingCols.remove(j2)
115                 break
116
117     incidenceM = copy(tempM)
118
119     # Check that all columns have been copied into temporary matrix
120     assert not remainingCols
121
122     # List of remaining groups to be ordered
123     # Used to keep track of which rows correspond to remaining groups
124     grpsNotIncl = typeL[:]
125     grpsNotIncl.remove(g1)
126     grpsNotIncl.remove(g2)
127     grpsNotIncl.sort(reverse=True)
128
129     # Reorder rows in remaining groups
130     reStart = g1+g2
131     for s in grpsNotIncl:
132         # Save the position of the first non-zero element in each row
133         # By constructing a list containing the pairs of point labels, with
134         # the index of the first non-zero element in their corresponding rows
```

```
135         fstNonzero = []
136         for i in range(s):
137             for j in range(b):
138                 if tempM[reStart+i][j] == 1:
139                     fstNonzero.append((reStart+i,j))
140                     break
141 
142         # Order pairs by second component
143         # Points within groups sorted by increasing index of first non-zero component
144         fstNonzero = [ t[0] for t in sorted(fstNonzero, key=lambda x: x[1]) ]
145         for i, row in enumerate(fstNonzero):
146             incidenceM[reStart+i, :] = tempM[row, :]
147 
148         reStart += s
149 
150     return incidenceM
151 
152 
153 def inferGDDParams(incidenceM):
154     # Function which can infer the parameters of a k-GDD given the incidence matrix
155     # Assumes that given incidence matrix corresponds to a valid k-GDD,
156     # and if any groups are the same size, points from the same group appear together.
157 
158     typeL = []
159 
160     # Dictionary tracks current groups
161     G = {}
162 
163     # Some sanity checks on provided incidence matrix
164     v = len(incidenceM)
165     assert v > 0
166     b = len(incidenceM[0])
167     assert b > 0
168 
169     k = 0
170     for i in range(v):
171         k += incidenceM[i][0]
172 
173     for i in range(v):
174 
175         # Value of r_i
176         rCurr = 0
177         for j in range(b):
178             assert incidenceM[i][j] in [0,1]
179             rCurr += incidenceM[i][j]
180 
```

```
        # Size of group containing current point i
        gCurr = v - rCurr*(k-1)

        if not gCurr in G:   # Create a dictionary entry for groups of size gCurr if needed
            G[gCurr] = [[i]]
        else:                # Find the first suitable group with room and add new point
            added = 0
            for grpL in G[gCurr]:
                if len(grpL) < gCurr:
                    grpL.append(i)
                    added = 1
                    break

            # Need to construct a new list if all existing groups contain enough points
            if not added:
                G[gCurr].append([i])

    # G is a python dictionary containing all the groups
    # Each dictionary key is an integer used to access the list of groups of that size
    for key in G:
        for i in range(len(G[key])):
            typeL.append(key)

    return k, b, v, typeL, G
```

References

[1] R.J.R. Abel, Y.A. Bunjamin, and D. Combe, Pairwise non-isomorphic 4-GDDs of type $3^5 6^2$, in progress.

[2] R.J.R. Abel, Y.A. Bunjamin, and D. Combe, Some new group divisible designs with block size 4 and two or three group sizes, *J. Comb. Des.* **28** (2020), 614–628. https://doi.org/10.1002/jcd.21719

[3] *Handbook of Combinatorial Designs*, Edited by C.J. Colbourn and J.H. Dinitz, 2nd ed., Chapman & Hall/CRC, Boca Raton, FL, 2007.

[4] T. Grünert, S. Irnich, H.-J. Zimmermann, M. Schneider, and B. Wulfhorst, Finding all k-cliques in k-partite graphs, an application in textile engineering, *Comput. Oper. Res.* **29** (2002), 13–31.
www.sciencedirect.com/science/article/abs/pii/S0305054800000538

[5] T. Beth, D. Jungnickel and H. Lenz, *Design Theory. Vol. I.*, 2nd ed., Encyclopedia of Mathematics and its Applications 78, Cambridge University Press, Cambridge, 1999.

[6] J.H. van Lint and R.M. Wilson, *A Course in Combinatorics*, Cambridge University Press, 2001.

[7] S. Niskanen and P.R.J. Östergård, *Cliquer User's Guide, Version 1.0*, Communications Laboratory, Helsinki University of Technology, Espoo, Finland, Tech. Rep. T48, 2003.
https://users.aalto.fi/~pat/cliquer/cliquer.pdf

[8] R. Pan, R.J.R. Abel, Y.A. Bunjamin, T. Feng, T.J. Tsang-Ung, and X. Wang, Difference matrices with five rows over finite abelian groups, in progress.

[9] Maximal Sets of Mutually Orthogonal Latin Squares. I, A.B. Evans,*European Journal of Combinatorics* **12** (1991), 477–482.
https://www.sciencedirect.com/science/article/pii/S0195669813800984