Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering

Scheduling of tournaments with several opponents in one game

Bc. Csaba Filip

Supervisor: Prof. Dr. Ing. Zdeněk Hanzálek
January 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Filip** Jméno: **Csaba** Osobní číslo: **466056**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Rozvrhování turnajů s více soupeři v jedné hře

Název diplomové práce anglicky:

Scheduling of tournaments with several opponents in one game

Pokyny pro vypracování:

1) nastudujte problematiku rozvrhování turnajů s více než 2 soupeři (například karetní hru) a vyberte vhodný podproblém/podproblémy pro spravedlivý rozvrh turnaje
2) specifikujte požadavky a formulujte problém pomocí jazyku optimálního řešiče
3) vyberte vhodnou metodu a navrhněte algoritmy pro řešení tohoto rozvrhovacího problému pro velké instance
4) navrhněte architekturu SW nástroje a implementujte dvě verze algoritmu
5) vyhodnoťte kvalitu řešení a dobu běhu jednotlivých algoritmů na benchmarkových instancích
6) otestujte výsledný SW nástroj včetně jednoduchého uživatelského rozhraní pro zadání vstupních dat a reprezentaci výsledku

Seznam doporučené literatury:

1. Ian Anderson: Combinatorial Designs and Tournaments, Oxford Lecture Series in Mathematics and Its Applications 1998
2. Bernhard Korte, Jens Vygen: Combinatorial Optimization □ Theory and Algorithms
Algorithms and Combinatorics 21, Springer□Verlag, 2018

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Dr. Ing. Zdeněk Hanzálek, katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.02.2021** Termín odevzdání diplomové práce: **04.01.2022**

Platnost zadání diplomové práce: **30.09.2022**

prof. Dr. Ing. Zdeněk Hanzálek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgements

I would like to thank my supervisor prof. Dr. Ing. Zdeněk Hanzálek for his precious advices and his guidance throughout the whole process.

I would also like to thank my parents Anita and Csaba for their support during my studies.

Last I thank my girlfriend Diana for her patience and emotional support during difficult times.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, January 4, 2022

Abstract

The aim of the thesis is scheduling of tournaments with several opponents in a single game. The main focus are tournaments with three player games and whist tournaments. For the scheduling of balanced tournaments, combinatorial designs from the field of combinatorial design theory are used. To schedule more general tournaments with arbitrary number of players and rounds, a constraint programming approach is proposed.

As part of this thesis, a web application that enables the construction of designs and the scheduling of tournaments is designed and implemented.

Keywords: tournament scheduling, block designs, Latin squares, constraint programming, pairwise testing

Supervisor: Prof. Dr. Ing. Zdeněk Hanzálek

Abstrakt

Tato práce se zabývá rozvrhováním turnajů s více soupeři v jedné hře. Hlavním zaměřením jsou turnaje pro tříhráčové hry a pro čtyřhry. K plánování vyvážených turnajů jsou využívány kombinatorické struktury z oblasti kombinatorického designu. K plánování obecnějších turnajů s libovolným počtem hráčů a kol je navržen přístup využívající programování s omezujícími podmínkami.

Součástí práce je i návrh a implementace webové aplikace, která umožňuje konstrukci kombinatorických struktur a plánování turnajů.

Klíčová slova: rozvrhování turnajů, block design, Latinské čtverce, programování s omezujícími podmínkami, pairwise testing

Překlad názvu: Rozvrhování turnajů s více soupeři v jedné hře

Contents

Chapter 1

Introduction

The planning of tournaments is a broad subject. Generally, a tournament consists of several rounds, each round involving multiple games. The parties taking part in a game - be it single players or whole teams - may play at most in one game in each of the rounds. The most common games such as chess or football only involve two parties playing against each others. The construction of tournaments for these games has been studied widely[3, 4, 5] and there are many tools for their scheduling, such as Teamsnap[1] or Tournify[2]. Furthermore, a round in such tournament might be influenced by the results of the previous round and constructed dynamically like in a single-elimination or ladder tournament.

In this thesis, however, we mainly focus on tournaments, where there are more than two opposing parties playing in a single game and the rounds of the tournament are planned statically, in advance. The results of the previous round do not influence the arrangement of games in the next round. An additional requirement we put on these tournaments is that on the balance. Ideally, we want every pair of players to play together in the same amount of games.

Consider the 3 player game Mariasch for example. A completely balanced Mariasch tournament with 9 players is depicted in table 1.1.

Round #	**Game 1**	**Game 2**	**Game 3**
Round 1	1, 2, 3	4, 5, 6	7, 8, 9
Round 2	1, 4, 7	2, 5, 8	3, 6, 9
Round 3	1, 5, 9	2, 6, 7	3, 4, 8
Round 4	1, 6, 8	2, 4, 9	3, 5, 7

Table 1.1: A completely balanced tournament of Mariasch with 9 players.

Each pair of players plays together in exactly one game of the Mariasch tournament. Such balanced tournaments can not be scheduled for an arbitrary number of players. To determine the number of players, for which balanced

[1] Team manager tool including a tournament schedule generator. `https://www.teamsnap.com`.

[2] On-line tournament scheduling tool. `https://www.tournifyapp.com`.

tournament can be scheduled and for the construction itself, we apply the findings from the mathematical field of *combinatorial design theory*. The main focus of the thesis are three player game tournaments and whist (or double tennis) tournaments.

Additionally, we develop an application that can be used to schedule several types of tournaments and to construct combinatorial designs. We also propose a more general method for the scheduling of tournaments with arbitrary number of players and rounds.

1.1 Outline

The following two chapter are introducing the basics of design theory and draw mainly from the books of I. Anderson [1] and C. Lindner [2]. Chapter 2 focuses on block designs and chapter 3 on Latin squares. These structures are crucial to understand the construction methods described in the latter chapters.

Chapter 4 deals with the practical part of this thesis and describes the design and architecture of the developed application.

Chapter 5 focuses on the construction of tournaments for three player games. Chapter 6 then explores the construction of whist tournaments. In chapter 7, a more general approach to the construction of tournaments using constraint programming is proposed.

Chapter 8 showcases, that the combinatorial designs used for the construction of tournaments can also be utilized in other fields - specifically in the field of software testing.

In chapter 9, we describe the testing of the developed application and show several runtime measurements benchmarks of the implemented tournament scheduling methods.

Chapter 2

Design theory: Block designs

When planning a tournament, there are several important parameters that need to be considered. One of the obvious ones is the number of players, that will take part in the whole tournament. Another one is the number of players that can play in a single game of the tournament (whether it is a 2, 3 or k-player game). The number of games in the whole tournament might also be given. It is also desirable for the games in the tournament to be scheduled in a balanced manner. Every pair of players should play in the same number of games.

Block designs from the field of combinatorial design theory provide a sound way to describe balanced tournaments. In this chapter, several block design types are introduced, starting with the simplest and most intensely studied balanced incomplete block designs.

Definition 2.1 (BIBD). *A Balanced Incomplete Block Design or BIBD with parameters v, k, λ is a collection of k-sized subsets (called blocks) of a v-sized set S, such that each pair of elements from S occurs together in exactly λ of the blocks.*

Even though the parameters v, k and λ are sufficient for the definition of BIBD, additional parameters b and r are often used. BIBD(v, k, λ) can then be described as BIBD(v, b, r, k, λ). b denotes the number of blocks in the design and r gives the number of blocks an element can be in. Both b and r can be uniquely determined from the previous parameters:

$$r = \frac{\lambda(v-1)}{k-1} \tag{2.1}$$

$$b = \frac{vr}{k} = \frac{\lambda(v^2 - v)}{k^2 - k} \tag{2.2}$$

Example 2.1. *The following blocks form a BIBD*$(7, 7, 3, 3, 1)$.

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

In the context of tournament planning, the parameters of BIBD(v, b, r, k, λ) can be interpreted in the following way:

- v is the number of players (or teams) in the tournament.
- b is the total number of games played in the tournament.
- r is the number of games each single player takes part in.
- k is the *cardinality* of the game - the number of players that can play in a single game.
- λ denotes the number of games each pair of players has played together.

It is important to note that for the given parameters v, b, r, k, λ, there might exist multiple non-isomorphic (without one-to-one mapping) BIBDs.

2.1 Necessary conditions for the existence of BIBD

The choice of the parameters of the BIBD can not be completely arbitrary. Obviously, all of the parameters have to be positive integers - including the parameters b and r in equations 2.1, 2.2. The other necessary conditions can be summarized with the following inequalities.

$$k < v \tag{2.3}$$

$$r > \lambda \tag{2.4}$$

$$b \geq v \tag{2.5}$$

Sufficient conditions for the existence of general BIBD are not known so far. Fortunately, for most of the subtypes of BIBD we use in this work, the sufficient conditions are known.

2.2 Resolvability

A general BIBD structure is not that useful when it comes to tournament scheduling. There is no guarantee that the games can be grouped into rounds of the tournament. As in example 2.1, all of the blocks have at least one element in common so that no games of the design can be played in parallel. The property ensuring that the blocks of the design can be split into parallel classes of the same sizes is called **resolvability**.

Definition 2.2 (Resolvability). *A BIBD is resolvable, if its blocks can be arranged into r classes so that the blocks of each class are disjoint and the union of blocks in a class contains each element exactly once. These classes are called parallel or resolution classes.*

Clearly, each parallel class should contain $\frac{b}{r} = \frac{v}{k}$ blocks.

Example 2.2. *The blocks and parallel classes of a resolvabe BIBD(9, 3, 1) are depicted in the table bellow:*

Parallel class	Blocks
Parallel class π_1	{1, 2, 3}, {4, 5, 6}, {7, 8, 9}
Parallel class π_2	{1, 4, 7}, {2, 5, 8}, {3, 6, 9}
Parallel class π_3	{1, 5, 9}, {2, 6, 7}, {3, 4, 8}
Parallel class π_4	{1, 6, 8}, {2, 4, 9}, {3, 5, 7}

Parallel classes are usually denoted with the symbol π. The resolvable BIBD in example 2.2 is actually equivalent to the Mariasch tournament schedule depicted in table 1.1. The parallel classes can clearly be interpreted as rounds of a tournament.

2.3 BIBD subtypes

2.3.1 Steiner triple systems

Definition 2.3. *A Steiner triple system of order v or STS(v) is a BIBD$(v, k = 3, \lambda = 1)$.*

STS is just a subtype of BIBD, in which the block sizes are 3. Unlike the general BIBDs, the sufficient conditions for the existence os STSs are known.

Theorem 2.1. *STS of order v exists if and only if $v = 6n + 1$ or $v = 6n + 3$, $n \in \mathbb{N}$.*

Regarding the tournaments, a much more important design is the Kirkman triple system.

2.3.2 Kirkman triple systems

Definition 2.4. *A Kirkman triple system of order v or KTS(v) is a STS(v) with the additional property of being resolvable.*

The resolvable BIBD in the example 2.2 was in fact a KTS(9).
A Kirkman Triple System was first proposed by Thomas Kirkman in disguise as a problem in 1850 in the recreational mathematics magazine *The Lady's and Gentleman's Diary*. The problem states:

"Fifteen young ladies in a school walk out three abreast for seven days in succession: it is required to arrange them daily so that no two shall walk twice abreast."

The solution of the problem is clearly a KTS(15). Later, the problem became known as the *Kirkman's schoolgirl problem* and is often referenced in literature and scientific papers concerning design theory.

It took more than 100 years (Chaudhuri & Wilson [6]) to discover general construction methods for KTSs and prove the following sufficient conditions for their existence.

Theorem 2.2. *A KTS(v) exists if and only if $v = 6n + 3$, $n \in \mathbf{N}$.*

	Row 1	Row 2	Row 3	Row 4	Row 5
Day 1	{1, 2, 3}	{4, 8, 12}	{5, 10, 14}	{6, 11, 13}	{7, 9, 15}
Day 2	{1, 4, 5 }	{2, 8, 10 }	{3, 13, 15}	{6, 9, 14}	{ 7, 11, 12}
Day 3	{1, 6, 7}	{2, 9, 11}	{3, 12, 14}	{4, 10, 15}	{5, 8, 13}
Day 4	{1, 8, 9}	{2, 12, 15}	{3, 5, 6}	{4, 11, 14}	{7, 10, 13}
Day 5	{1, 10, 11}	{2, 13, 14}	{3, 4, 7}	{5, 9, 12}	{6, 8, 15}
Day 6	{1, 12, 13}	{2, 4, 6}	{3, 9, 10}	{5, 11, 15}	{7, 8, 14}
Day 7	{1, 14, 15}	{2, 5, 7}	{3, 8, 11}	{4, 9, 13}	{6, 10, 12}

Table 2.1: One possible solution to the Kirkman's schoolgirl problem. The days can be interpreted as the parallel classes.

KTSs are especially useful when it comes to three player balanced tournaments. The method for their construction is described later in chapter 5.

2.4 Other Block Design types

As was mentioned in the beginning, Balanced incomplete block design is the simplest block design type. Other block designs, that will be discussed, are *Pairwise balanced designs* and *Group divisible designs*. Fortunately, most of the concepts introduced for BIBD can be (at least partially) applied to the other block designs as well.

2.4.1 Pairwise balanced design

In BIBD, the size of all the blocks has to be the same and it is given by the parameter k. In a Pairwise balanced design, however, the blocks might have different sizes given by a whole set K.

Definition 2.5. *A Pairwise balanced design PBD*(v, K, λ) *is a collection of blocks on a* v*-sized set* S*, such that each pair of elements from* S *occurs together in exactly* λ *of the blocks and the size of each block is in the set* K*.*

Example 2.3. *Blocks* $\{1,2,4\}$, $\{1,6,7\}$, $\{1,3,8\}$, $\{2,5,6\}$, $\{2,7,8\}$, $\{3,5,7\}$, $\{3,4,6\}$, $\{4,5,8\}$ *of size* 3 *and blocks* $\{1,5\}$, $\{2,3\}$, $\{4,7\}$, $\{6,8\}$ *of size* 2 *form a PBD*$(8, \{2,3\}, 1)$.

One method for the construction of PBD is to take a BIBD(v, k, λ) and remove a single element. This way a PBD$(v-1, \{k-1, k\}, \lambda)$ is obtained. In example 2.3 a BIBD$(9, 3, 1)$ was used for the construction of the PBD.

BIBD is actually a sub type of PBD, where $K = \{k\}$.

2.4.2 Group divisible design

A design somewhat similar to PBD and the most complicated design discussed here is the *Group divisible design*.

Definition 2.6. *A Group divisible design or GDD of order v consists of a v-sized set S of elements, a collection of subsets G called* **groups** *and a collection of blocks B such that*

- *the groups form a partition of S;*
- *each pair of elements from the same group do not occur together in any of the blocks;*
- *each pair of elements from different groups occur together in exactly one of the blocks.*

All the designs mentioned before are usually defined in a very similar way across the literature but there is quite a divergence when it comes to GDDs. In [1] for example, only uniform sized blocks and groups are considered. Furthermore, parameters λ_1 and λ_2 are used to describe the number of occurrences of pairs from the same group as opposed to the number of occurrences of pairs from different groups. Here, we only consider $\lambda_1 = 0$ and $\lambda_2 = 1$. In [2], however, GDDs are defined with both variable sized blocks and groups.

The definition used here is the most convenient for the scope of this thesis. We mostly work with uniform sized block and groups and denote such designs as $\mathrm{GDD}(v, g, k)$ where g is the size of groups and k is the size of blocks. In some of the construction methods, however, we require variable sized groups and blocks, so the definition was chosen to include these cases.

2.5 BIBD construction methods

There are several BIBD construction methods. Most of them, however, use already existing BIBDs to construct specific new designs.

Example 2.4. *Take the 7 blocks of the BIBD$(7, 3, 1)$ in example 2.1 twice to obtain a BIBD$(7, 3, 2)$.*

A convenient general construction method would take the parameters of the BIBD as the input and it would output the desired design. Unfortunately, as the necessary conditions for the existence of BIBDs are not known, there are no efficient general construction methods. The task of the BIBD construction, however, can be formulated as an *integer quadratic program.*

2.5.1 Quadratic programming approach

Consider a binary incidence matrix $X = (x_{i,j})$ of dimensions $v \times b$ representing the given BIBD. $x_{i,j} = 1$ only if element i is part of the block j. A valid BIBD can then be described with the following equations:

$$\sum_{j=1}^{b} x_{i,j} = r, \qquad i = 1, \ldots, v \tag{2.6}$$

$$\sum_{i=1}^{v} x_{i,j} = k, \qquad j = 1, \dots, b \tag{2.7}$$

$$\sum_{j=1}^{b} x_{i,j} x_{i',j} = \lambda, \qquad i, i' = 1, \dots, v, \quad i < i' \tag{2.8}$$

Equation 2.6 ensures that all of the elements are present in exactly r of the blocks. Equation 2.7 guarantees that all of the block are of the same size k. And the last equation 2.8 ensures, that all the elements occur together in exactly λ of the blocks.

These conditions can be easily translated into a quadratic program, which can then be solved by any quadratic solver. If there is no BIBD for the instance given by input parameters, the solver should label the instances as infeasible. Otherwise, it should return a valid BIBD with the given parameters in some finite time. Unfortunately, even with relatively small instances and with a state-of-the-art solver, the calculation takes a long time.

The quadratic programming BIBD construction is implemented in the `QuadraticProgrammingBIBDConstruction` class in the application accompanying this thesis.

2.5.2 Incremental block building approach

A more efficient alternative to the quadratic programming approach was proposed by Yokoya and Yamada in [7]. This approach combines simple backtracking with linear programming.

Instead of computing the whole incidence matrix X as in 2.5.1, we try to obtain its rows incrementally, one by one.

$$X_{k+1} = \begin{bmatrix} X_k \\ \mathbf{x} \end{bmatrix} \tag{2.9}$$

This way, because we only need to determine a single binary row vector $\mathbf{x}$, the problem can be written as linear binary program instead of the quadratic one.

$$\text{Maximize} \quad \sum_{j=1}^{b} x_j + \sum_{j=1}^{b} (\sum_{i=1}^{k} \bar{x}_{ij}) x_j \tag{2.10}$$

$$\text{subject to} \quad \sum_{j=1}^{b} x_j \leq r, \tag{2.11}$$

$$x_j \leq k - \sum_{i=1}^{k} \bar{x}_{ij}, \qquad j = 1, \dots, b, \tag{2.12}$$

$$\sum_{j=1}^{b} \bar{x}_{ij} x_j \leq \lambda, \qquad i = 1, \dots, k, \tag{2.13}$$

$$x_j \in \{0, 1\}, \qquad j = 1, \dots, b. \tag{2.14}$$

It can be shown, that if the optimal value of the objective function 2.10 is less than $r + k\lambda$, no BIBD exists as an extension of X_k. In that case, we need to backtrack to the previous X_{k-1} problem, find a different optimal solution and try again - now with a different X_{k-1} input matrix.

The first two rows of X_2 can always be initialized as

$$X_2 = \begin{array}{c} \overbrace{\begin{matrix} 1 & 1 & \cdots & 1 & 1 & \cdots & 1 \end{matrix}}^{r} \begin{matrix} 0 & \cdots & 0 & 0 & \cdots & 0 \end{matrix} \\ \underbrace{\begin{matrix} 1 & 1 & \cdots & 1 \end{matrix}}_{\lambda} \begin{matrix} 0 & \cdots & 0 \end{matrix} \underbrace{\begin{matrix} 1 & \cdots & 1 \end{matrix}}_{r-\lambda} \begin{matrix} 0 & \cdots & 0 \end{matrix} \end{array} \tag{2.15}$$

This incremental approach to the construction of BIBDs is implemented in the `IncrementalBIBDConstruction` class. As described in the original paper, this method can be further augmented with *tabu search*.

Chapter 3

Design theory: Latin squares

So far, only several types of block designs have been discussed. Another closely related structure in the field of combinatorial design are Latin squares. Their relation to tournament scheduling is not immediately obvious. Latin squares are, however, quite useful on their own and are required for the construction of several of the block design types which are in turn used for the scheduling of tournaments.

Definition 3.1. *A Latin square of n symbols is an $n \times n$ matrix, such that each row and column of the matrix contains each of the n symbols exactly once. The number n is called the order of the Latin square.*

Example 3.1. *A Latin square of order* 5.

$$\begin{matrix} A & B & C & D & E \\ B & C & D & E & A \\ C & D & E & A & B \\ D & E & A & B & C \\ E & A & B & C & D \end{matrix}$$

A Latin square of any order $n \geq 2$ can be constructed. Furthermore, it can be shown that any $r \times n$, $r < n$ *Latin rectangle* (an incomplete Latin square with some missing rows) can be expanded into a Latin square.

3.1 Bipartite Tournaments

Latin squares can be utilized to construct bipartite tournaments. In a bipartite tournament, there are two teams of the same size, say n. The goal is to construct the tournament in such a way, that every player from the first team will play exactly once against every player from the second team. The tournament should have n rounds, all of the rounds consisting of n games.

It turns out, that the construction of bipartite tournament only requires a single Latin square of order n. The players from the first team will be fixed to the same game in each of the rounds and players from the second team will be distributed according to the Latin square. The scheduling of bipartite tournaments is implemented in the `BipartiteTournamentPlanner` class.

Example 3.2. *The bipartite tournament with teams* $T_a = \{a_1, a_2, a_3, a_4\}$, $T_b = \{b_1, b_2, b_3, b_4\}$ *and the Latin square of order* 4 *used to construct the tournament.*

Round 1:	a_1 v b_1	a_2 v b_2	a_3 v b_3	a_4 v b_4
Round 2:	a_1 v b_2	a_2 v b_1	a_3 v b_4	a_4 v b_3
Round 3:	a_1 v b_3	a_2 v b_4	a_3 v b_1	a_4 v b_2
Round 4:	a_1 v b_4	a_2 v b_3	a_3 v b_2	a_4 v b_1

Table 3.1: The constructed bipartite tournament.

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

3.2 Mutually orthogonal Latin squares

The concept of Latin square orthogonality was studied in detail by Leonhard Euler but it dates back even further. A nice way to introduce this concept is with Jacques Ozanam's 1725 puzzle [8]. The goal is to arrange the 16 face cards of a standard deck into a 4 × 4 grid, such that each row and each column will contain all four face values and all four suits.

Example 3.3. *One possible solution of the playing card puzzle.*

A♠	*K*♥	*Q*♦	*J*♣
Q♣	*J*♦	*A*♥	*K*♠
J♥	*Q*♠	*K*♣	*A*♦
K♦	*A*♣	*J*♠	*Q*♥

The grid of face values can be viewed as the first Latin square and the grid of the suits can be viewed as the second Latin square. The fact, that the superimposed grid contains 16 unique pairs - the 16 face cards of the deck - is ensured via the mutual orthogonality of the two squares.

The formal definition is as follows:

Definition 3.2. *Let* L_1 *and* L_2 *be Latin squares of the same order. We say that* L_1 *and* L_2 *are* mutually orthogonal *Latin squares (MOLS) if, when superimposed, each of the possible ordered pair occurs exactly once.*
Latin squares $L_1, \ldots, L_n$ *are mutually orthogonal if they are orthogonal in pairs.*

Example 3.4. *Three MOLS of order 4.*

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array} \qquad \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \end{array} \qquad \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \end{array}$$

Another famous problem, similar to the playing cards puzzle and studied by Euler, was the **36-officer problem**. In it, the goal was to arrange 36 officers of 6 different ranks drawn from 6 different regiments (no two officers have the same rank and regiment) in a square so that in each row and column, there are 6 officers of different ranks and different regiments. In other words, the goal was to find two mutually orthogonal Latin squares of order 6.

Euler was not able to solve this problem, but he conjectured that there are no two MOLS of order 6. Furthermore, he conjectured that there are no MOLS of order $n = 4k + 2$. The non-existence of MOLS of order 6 was confirmed in 1901 through a proof by exhaustion [9]. However, in 1959 Bose and Shrikhande were able to construct MOLS of order 22, thereby disproving Euler's conjecture [10]. Furthermore, in their follow-up work, they have shown that at least 2 MOLS exist for any order $n \geq 10$ [11].

Theorem 3.1. *A pair of MOLS exists for all orders n except for 2 and 6.*

The fact that there are no MOLS of order 2 can be proven easily by exhaustion.

3.3 Latin squares and quasi-groups

Definition 3.3 (Quasi-group). *A (combinatorial) quasi-group $(Q, \circ)$ is a set Q equipped with a binary multiplication operation*

$$Q \times Q \to Q;\ (x, y) \to x \circ y$$

denoted by $\circ$, such that any two of x, y, z in the equation $x \circ y = z$ determine the third uniquely [12].

Definition 3.4 (Orthogonal quasi-groups). *Two quasi-groups $(Q, \circ_1)$ and $(Q, \circ_2)$ on the same set Q are orthogonal, if the system of equations*

$$x \circ_1 y = a \tag{3.1}$$

$$x \circ_1 y = b \tag{3.2}$$

has a unique solution for any $a, b \in Q$.

Considering the properties of Latin squares, it should be clear that they yield the multiplication tables of finite quasi-groups, as in table 3.2. Furthermore, mutually orthogonal Latin squares yield the multiplication tables of orthogonal quasi-group.

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

Table 3.2: Multiplication table of a quasi-group yielded by Latin square of order 7.

Quasi-groups and orthogonal quasi-groups are useful in many mathematical fields. In design theory, they can be used to construct Steiner triple systems or several types of PBDs (see 5.2.1).

3.4 Orthogonal arrays

The definitions of orthogonal arrays differ across literature, but the main concept is the same. Here, we use a simplified definition which is sufficient for the scope of this thesis.

Definition 3.5. *An orthogonal array or OA(k, m) on m elements is a $k \times m^2$ matrix, such that any two rows yield each of the possible ordered pairs of elements in their pairs of columns.*

An OA(k, m) is equivalent to a set of $k - 2$ MOLS of order m. This fact is illustrated in the following example.

Example 3.5. *Consider the following 2 MOLS of order* 4

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array} \qquad \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \end{array}$$

The first two rows of the equivalent orthogonal array are the row and column coordinates of the elements in the MOLS. The third row lists the elements of the first Latin square according to the coordinates and the fourth row lists the elements of the second Latin square.

$$\begin{array}{cccccccccccccccc} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 2 & 1 & 4 & 3 & 3 & 4 & 1 & 2 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 3 & 4 & 1 & 2 & 4 & 3 & 2 & 1 & 2 & 1 & 4 & 3 \end{array}$$

The number k of the orthogonal array is called its *factor* and m is called its *level.* In some literature, the definition of an orthogonal array is actually the transpose of the orthogonal array defined here [25].

3.5 Transversal designs

Definition 3.6. *A transversal design or TD(k, m) on km elements consists of k-sized blocks and a collection m-sized groups such that the k groups partition the set of elements, each block contains exactly one element from each group and any pair of elements from different groups occurs together in exactly one block.*

On the first glance, transversal designs might seem more similar to block design as a specific GDD. They are, however, very closely related to MOLS and orthogonal arrays.

Example 3.6 (TD$(4, 4)$)**.** *Consider the OA$(k = 4, m = 4)$ form example 3.5. Modify its rows by adding $(i - 1)m$ to each of the elements in the ith row.*

$$\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\
5 & 6 & 7 & 8 & 5 & 6 & 7 & 8 & 5 & 6 & 7 & 8 & 5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 & 10 & 9 & 12 & 11 & 11 & 12 & 9 & 10 & 12 & 11 & 10 & 9 \\
13 & 14 & 15 & 16 & 15 & 16 & 13 & 14 & 16 & 15 & 14 & 13 & 14 & 13 & 16 & 15
\end{array}$$

Consider the columns of the modified array as blocks. These blocks together with groups $G_1 = \{1, 2, 3, 4\}$, . . . , $G_4 = \{13, 14, 15, 16\}$ form a TD$(4, 4)$.

TD(k, m) is clearly equivalent to OA(k, m) and to $k - 2$ MOLS of order m. The construction of transversal designs is implemented in the `TDConstruction` class.

Theorem 3.2. *A resolvable TD(k, m) exists only if a TD$(k + 1, m)$ exists.*

Having constructed a TD$(k + 1, m)$, it is easy to obtain a resolvable TD(k, m). Take the $(k+1)$th groups of TD$(k+1, m)$ and remove its elements from the blocks. This leaves blocks of size k and k groups of size m. For each removed element of the removed group, take the blocks that contained said element. These blocks form a parallel class. This construction is implemented in the `RTDConstruction`.

3.6 Construction of MOLS

The construction of complete sets of MOLS of any order is still an open problem [17]. We show and implement the construction of pairs of MOLS of odd orders and of orders that are multiples of 4 according to Guichard[18]. Unfortunately, some of the designs required for the scheduling of tournaments can not be constructed without larger sets of MOLS. These larger sets were therefore obtained using Sage Math[1] and stored in the data store of the application accompanying this thesis. It should be noted however, that the larger sets of MOLS and MOLS of order $4k + 2$ are the only designs, that are not constructed directly by our application.

[1] Sage Math is a mathematical package for Python. It covers many fields of mathematics including statistics, calculus and combinatorics.

3.6.1 Pairs of MOLS of odd orders

Construction of pairs of MOLS of odd order n is very simple. Define the matrices A and B as follows:

$$a_{i,j} = i + j \pmod{n}, \tag{3.3}$$

$$b_{i,j} = 2i + j \pmod{n}. \tag{3.4}$$

Both A and B form Latin squares. Matrix B contains the same rows as A, only in different order. When superimposed, they yield all the elements of $\{0, \dots, n-1\} \times \{0, \dots, n-1\}$.

Example 3.7. *Two MOLS of order 3 constructed using this method.*

$$\begin{array}{ccc} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{array} \qquad \begin{array}{ccc} 0 & 1 & 2 \\ 2 & 0 & 1 \\ 1 & 2 & 0 \end{array}$$

This construction method is implemented in `OddMOLSPairConstruction`.

3.6.2 Pair of MOLS of order $4 \times k$

Construction of pairs of MOLS of order $4k$ is a bit more involved. It uses two pairs of MOLS of smaller orders m, n and a special matrix product operation to construct a larger pair of MOLS of order mn.

Let A be a Latin square of order m and B of order n. Let $c_{i,j}$, $1 \leq i \leq m$, $1 \leq j \leq n$ be new mn symbols. Form a $mn \times mn$ grid by replacing each element of B with a copy of A. Then replace each element i in this copy of A with c_i, j where j is the element of B that has been replaced. We denote this new matrix as $A \circ B$.

Example 3.8. *The special product operation on Latin square A of order 4 and B of order 3.*

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{array} \circ \begin{array}{ccc} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{array} = \begin{array}{cccc|cccc|cccc}
c_{1,1} & c_{2,1} & c_{3,1} & c_{4,1} & c_{1,2} & c_{2,2} & c_{3,2} & c_{4,2} & c_{1,3} & c_{2,3} & c_{3,3} & c_{4,3} \\
c_{2,1} & c_{3,1} & c_{4,1} & c_{1,1} & c_{2,2} & c_{3,2} & c_{4,2} & c_{1,2} & c_{2,3} & c_{3,3} & c_{4,3} & c_{1,3} \\
c_{3,1} & c_{4,1} & c_{1,1} & c_{2,1} & c_{3,2} & c_{4,2} & c_{1,2} & c_{2,2} & c_{3,3} & c_{4,3} & c_{1,3} & c_{2,3} \\
c_{4,1} & c_{1,1} & c_{2,1} & c_{3,1} & c_{4,2} & c_{1,2} & c_{2,2} & c_{3,2} & c_{4,3} & c_{1,3} & c_{2,3} & c_{3,3} \\ \hline
c_{1,2} & c_{2,2} & c_{3,2} & c_{4,2} & c_{1,3} & c_{2,3} & c_{3,3} & c_{4,3} & c_{1,1} & c_{2,1} & c_{3,1} & c_{4,1} \\
c_{2,2} & c_{3,2} & c_{4,2} & c_{1,2} & c_{2,3} & c_{3,3} & c_{4,3} & c_{1,3} & c_{2,1} & c_{3,1} & c_{4,1} & c_{1,1} \\
c_{3,2} & c_{4,2} & c_{1,2} & c_{2,2} & c_{3,3} & c_{4,3} & c_{1,3} & c_{2,3} & c_{3,1} & c_{4,1} & c_{1,1} & c_{2,1} \\
c_{4,2} & c_{1,2} & c_{2,2} & c_{3,2} & c_{4,3} & c_{1,3} & c_{2,3} & c_{3,3} & c_{4,1} & c_{1,1} & c_{2,1} & c_{3,1} \\ \hline
c_{1,3} & c_{2,3} & c_{3,3} & c_{4,3} & c_{1,1} & c_{2,1} & c_{3,1} & c_{4,1} & c_{1,2} & c_{2,2} & c_{3,2} & c_{4,2} \\
c_{2,3} & c_{3,3} & c_{4,3} & c_{1,3} & c_{2,1} & c_{3,1} & c_{4,1} & c_{1,1} & c_{2,2} & c_{3,2} & c_{4,2} & c_{1,2} \\
c_{3,3} & c_{4,3} & c_{1,3} & c_{2,3} & c_{3,1} & c_{4,1} & c_{1,1} & c_{2,1} & c_{3,2} & c_{4,2} & c_{1,2} & c_{2,2} \\
c_{4,3} & c_{1,3} & c_{2,3} & c_{3,3} & c_{4,1} & c_{1,1} & c_{2,1} & c_{3,1} & c_{4,2} & c_{1,2} & c_{2,2} & c_{3,2}
\end{array}$$

This $\circ$ operation preserves orthogonality. So if we performed this operation on the orthogonal pair of A and B, we would obtain a Latin square orthogonal to $A \circ B$.

Now, the order $4k$ can be written as $4k = n \cdot 2^m$ where n is odd and $m \geq 2$. We can construct a pair of MOLS of order n using the method from the previous subsection. The only thing left is to construct a pair of MOLS of order 2^m.

Suppose that we are able to construct pairs of MOLS of orders 4 and 8. Using these orders and the (repeated) application of special product operation $\circ$, we are able to construct any pair of MOLS of order 2^m. So we only require MOLS of order 4 and 8. MOLS of order 4 are listed in example 3.4 and MOLS of order 8 are shown bellow.

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

Figure 3.1: A pair of MOLS of order 8.

This construction is implemented in the `FourKMOLSPairConstruction` class.

Chapter 4

Architecture and design of the application

The software part of this work is a web application named `TournamentPlanner`. In this chapter, we describe the software architecture including the modelling of designs, tournaments and their construction methods. But first, we summarize the basic requirements we put on the system.

4.1 General requirements

The process of requirement gathering is one of the most important steps in software engineering. Normally, it involves consultations with the client and the stakeholders. With this project however, there is no client and the only stakeholder is me as the sole designer and developer. It would be convenient for me to retrofit the requirement specifications to fit the resulting system perfectly. To avoid this and to have more freedom during the development, I kept the initial requirements fairly general and broad. They were the following:

- implement the construction of several types of design structures;
- implement the planning of tournaments using the constructed designs;
- propose and implement a more general method for tournament planning using constraint programming;
- provide a simple GUI that allows the construction of designs and tournaments according to the input parameters; display the constructed designs and tournaments.

4.2 Architecture of the TournamentPlanner

`TournamentPlanner` is a monolithic web application developed using the `ASP.NET Core`[1] `net5.0` framework. Web application was chosen over desktop application because it does not require any additional software installations

[1] Free and open-source web framework developed by Microsoft. `https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0`

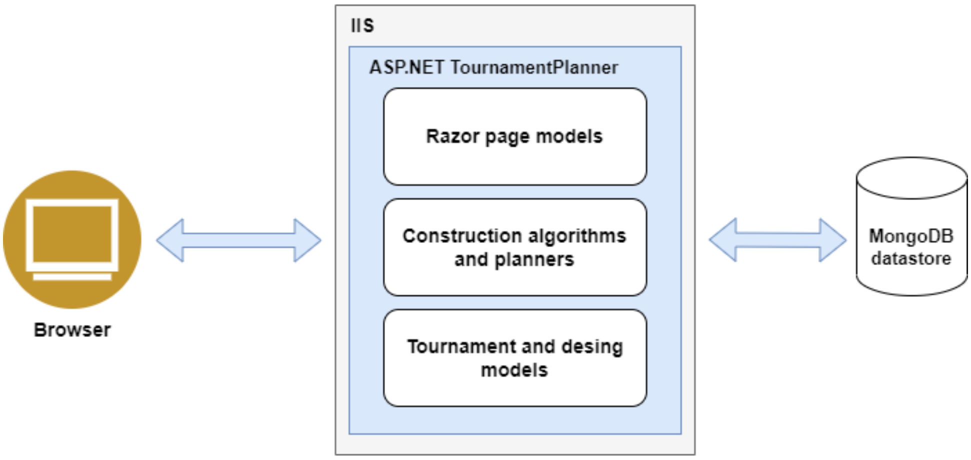

IIS
ASP.NET TournamentPlanner
Razor page models
Construction algorithms and planners
Tournament and desing models
Browser
MongoDB datastore

Figure 4.1: Diagram illustrating the architecture of the `TournamentPlanner` and showcasing its main components.

except a web browser on the client side and because I was already familiar with the `ASP.NET` framework.

The server side of the application includes the construction of the designs and tournaments. The only programming language used here is `C#`. `MongoDB`[2] is used as the data store for larger sets of MOLS. As has been stated before, these are the only designs that are not constructed directly by the application. The modelling of the designs, tournaments and their construction methods is addresses in the following sections.

The user interface of the application is developed using `Razor pages`[3] which allow for combination of `HTML`, `CSS` and `JavaScript` with pieces of `C#` code. There are several forms to specify the parameters of the design or tournament. When submitted, an according design or tournament is constructed on the server side and then visualized on the client side. With the combination of `ASP.NET` and `Razor`, there is no need to explicitly specify the Web API through which the client and the server communicate.

4.3 Modelling the designs and tournaments

There are 2 main groups of models in the `TournamentPlanner` - one for the combinatorial designs and the other for the tournaments. They both implement their separate interfaces and are kept in separate namespaces `TournamentPlanner.Model.Designs` and `TournamentPlanner.Model.Tournaments`.

4.3.1 Design structures

When modelling the designs introduced in the previous 2 chapters, the goal was to keep them as close as possible to their original definitions. All the

[2] `MongoDB` is a document based data store which is fairly convenient for storing large arrays.

[3] `Razor` is an `ASP.NET` programming syntax used to create dynamic web pages.

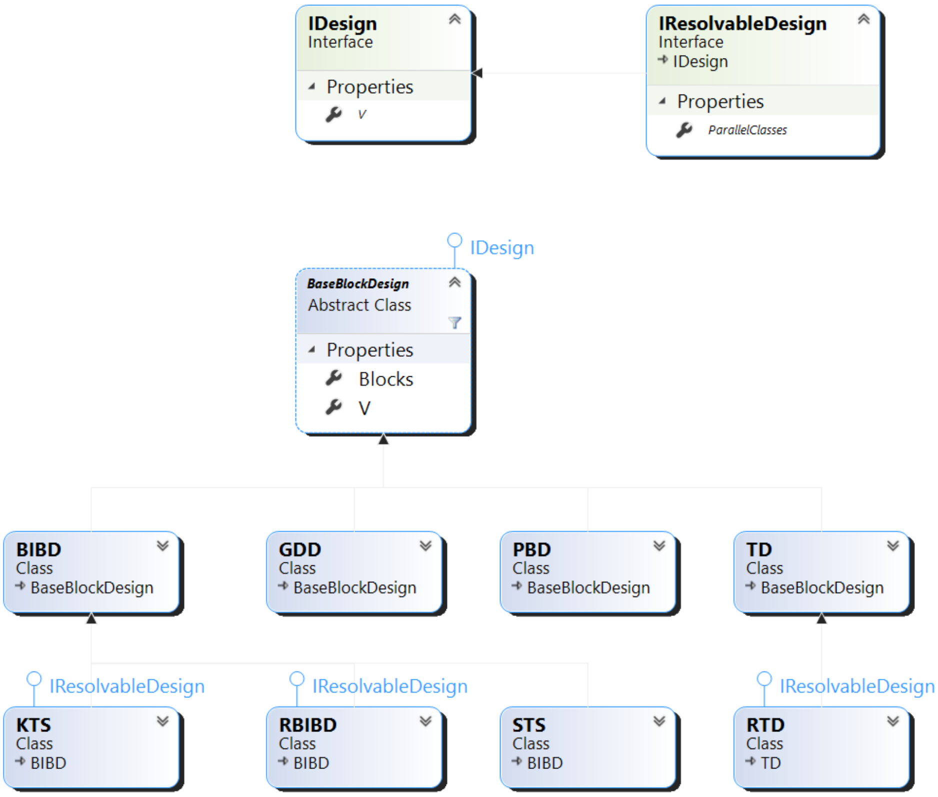


Figure 4.2: Class diagram of the block designs. Interfaces are shown using the lollipop notation.

classes representing a design implement the `IDesign` interface.

```
namespace TournamentPlanner.Model.Designs
{
    public interface IDesign
    {
        public int V { get; }
    }
}
```

Listing 4.1: Code of the `IDesign` interface without documentation.

Its only property - the integer `V` - denotes the size of the set the design is constructed on. For simplicity, we always consider the set $\{0, 1 \ldots, \texttt{V - 1}\}$ as the elements of the design structure and do not store this set explicitly.

Designs are further divided into Latin square like structures (Latin squares, MOLS and orthogonal arrays) in the `LatinSquares` namespace and block designs in the `BlockDesigns` namespace.

A single block of a block design is represented as a list of integers `List<int>`, and all the blocks of a design as `List<List<int>>`. The block designs derive from the abstract `BaseBlockDesign` class. The resolvable blocks designs also implement the `IResolvableDesign` interface with the `ParallelClasses` property (`List<List<List<int>>>`). All the class names follow the same abbreviations that are used in this text.

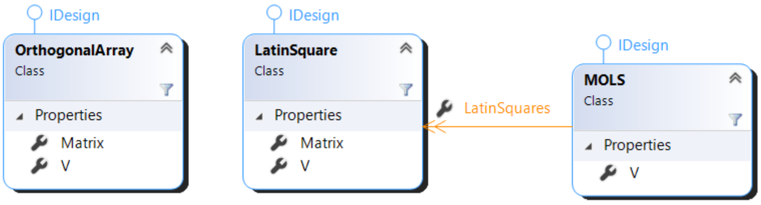


Figure 4.3: Class diagram of `OrthogonalArray`, `LatinSquare` and `MOLS`.

The Latin square like structures are listed in figure 4.3. The matrices in `LatinSquare` and `OrthogonalArray` are represented as two-dimensional arrays `int[,]`. `MOLS` has a list with 2 or more `LatinSquare` instances that are mutually orthogonal.

4.3.2 Tournaments

Instead of implementing an interface, all the tournaments in the `TournamentPlanner` inherit from the abstract `BaseTournamentClass`.

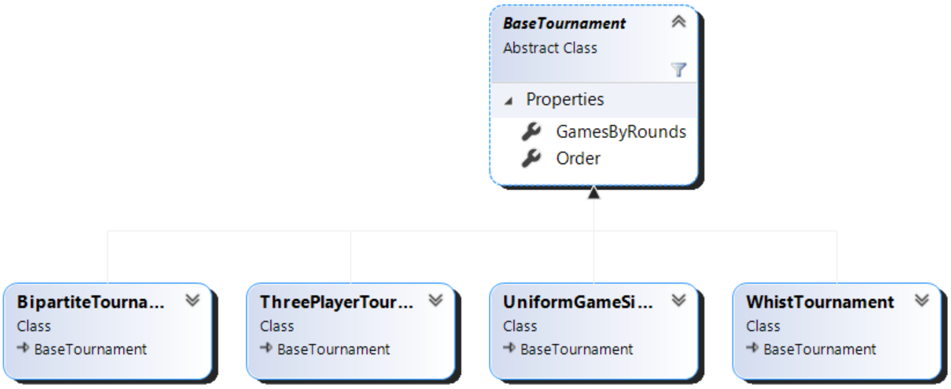


Figure 4.4: Class diagram of the tournaments.

Players in the tournaments are also represented as integers and the rounds of the tournament are represented in the same way as the parallel classes of resolvable block designs. The `int Order` property of the `BaseTournament` denotes either directly the number of players, or the number of players in a single team when it comes to the `BipartiteTournament`.

4.4 Construction algorithms

It has already been established that the designs and the tournaments form 2 separated groups of classes in the `TournamentPlanner`. The same applies to the classes facilitating the construction of these objects. We call the classes dealing with the construction of design structures *Construction algorithms*. They are placed in the `TournamentPlanner.Algorithms.ConstructionAlgorithms` namespace and their names always end with the word `Construction` (`RecursiveKTSConstruction` for example).

```
namespace TournamentPlanner.Algorithms.ConstructionAlgorithms
{
    public interface IDesignConstruction<IDesign>
    {
        public IDesign ConstructDesign();
    }
}
```

Listing 4.2: Code of the `IDesignConstruction` interface without documentation.

All the construction algorithms implement the generic `IDesignConstruction` interface. Its single method `public IDesign ConstructDesign()` returns a design structure implementing the `IDesign` interface. Then, a concrete construction algorithm can look as the code in listing 4.3.

```
public class TDConstruction : IDesignConstruction<TD>
{
  // constructor and variables
  // ...

  public TD ConstructDesign()
  {
    // construction of the groups and blocks
    // ...
    return new TD(k, m, groups, blocks);
  }
}
```

Listing 4.3: Snippet of the transversal design construction class.

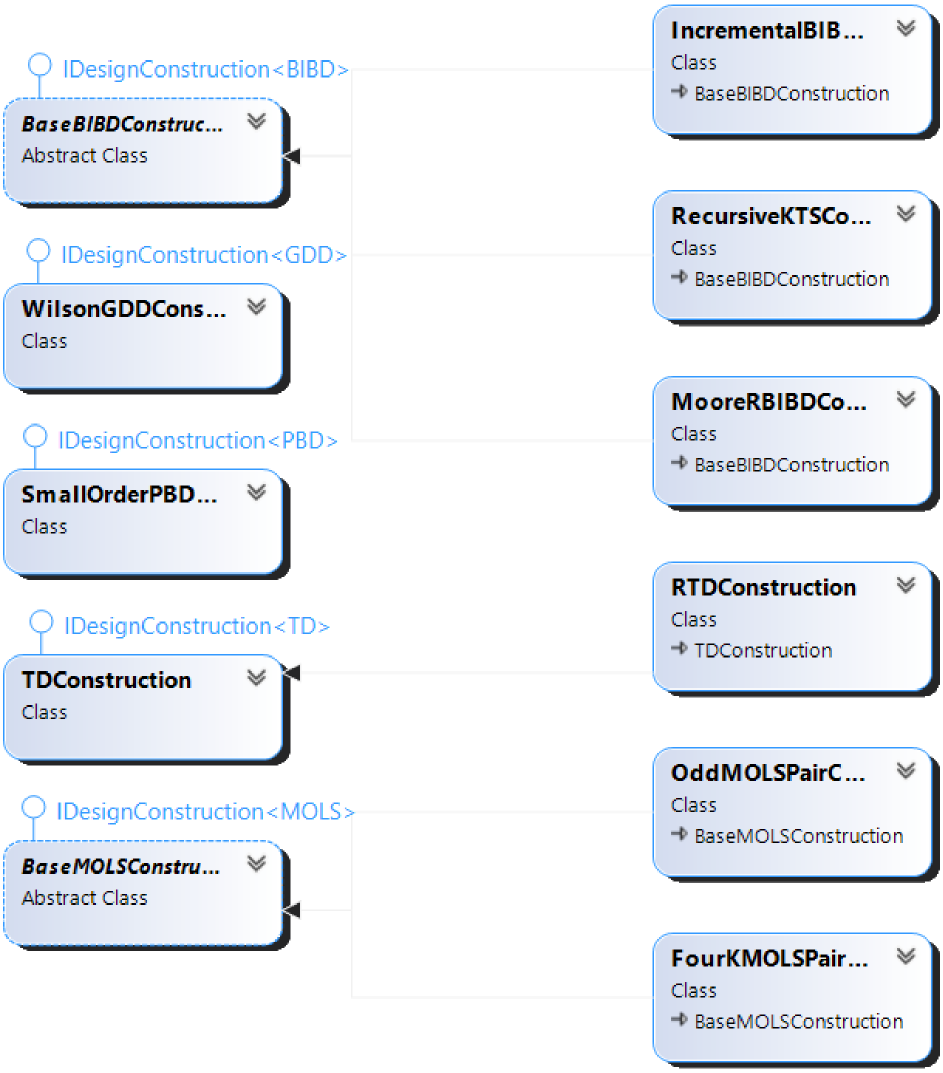


Figure 4.5: Several of the construction algorithm classes.

Some of the construction algorithm classes are displayed in figure 4.5. Their hierarchy copies the hierarchy of the design structures they are supposed to construct to some extent. All the BIBD subtype construction algorithms inherit from the abstract `BaseBIBDConstruction` class (similarly as `KTS` and `STS` inherit from BIBD). Analogically, `RTDConstruction` inherits from `TDConstruction` as `RTD` inherits from `TD`.

4.5 Tournament planners

Tournament planners are conceptually similar to the *construction algorithms*, but they facilitate the scheduling of tournaments. They are placed in the `TournamentPlanner.Algorithms.TournamentPlanners` namespace. All the tournament planners implement the `ITournamentPlanner` interface which is akin to the IDesignConstruction.

```
namespace TournamentPlanner.Algorithms.TournamentPlanners
{
    public interface ITournamentPlanner<BaseTournament>
    {
        public BaseTournament PlanTournament();
    }
}
```

Listing 4.4: Code of the ITournamentPlanner interface without documentation.

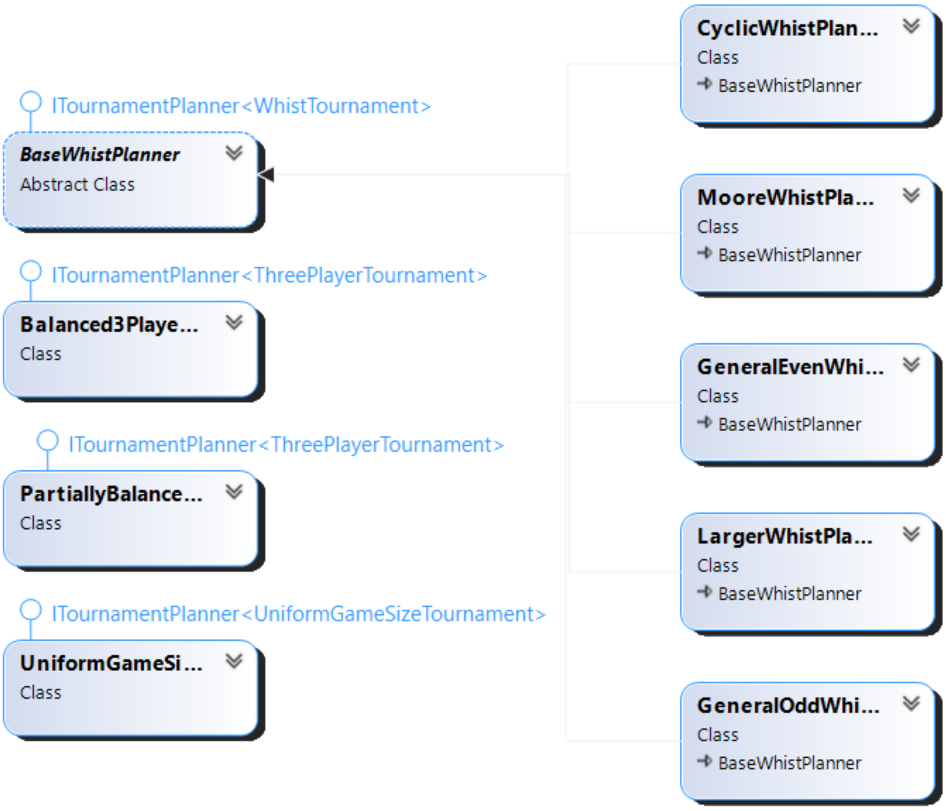
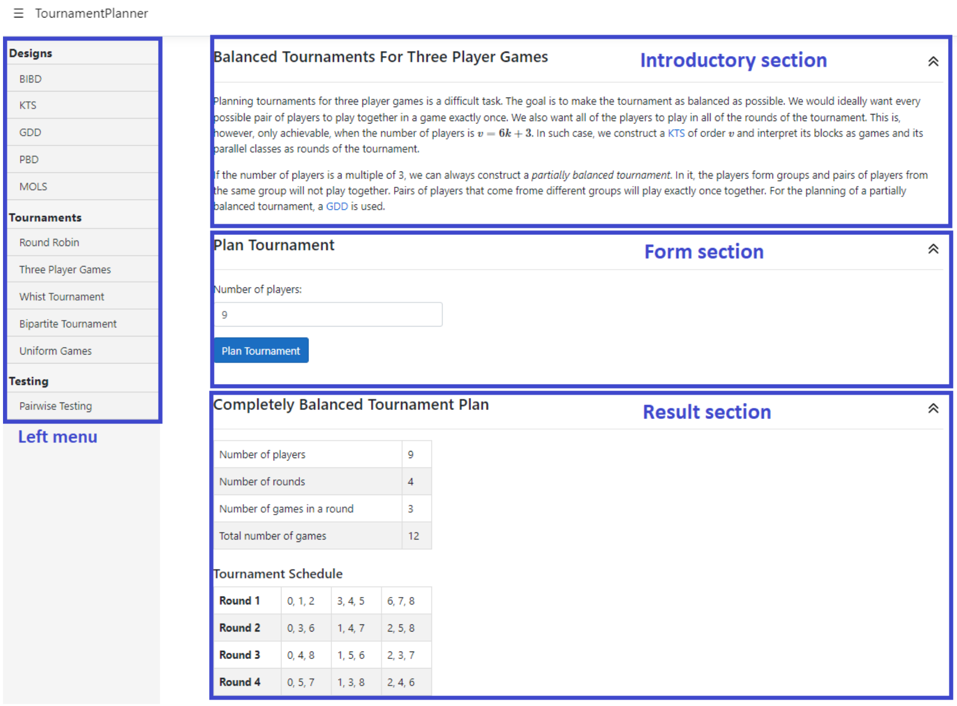


Figure 4.6: Class diagram of several tournament planners. Whist tournaments are difficult to plan and require several planners.

A concrete tournament planner then specifies the type of the BaseTournament and returns it on calling the PlanTournament() method. Several of the tournament planners are displayed in figure 4.6.

Both the construction algorithms and the tournament planners are using the *command pattern*. They encapsulate all the data required for the construction of the given object and the construction itself is performed on the call of the corresponding method.



Balanced Tournaments For Three Player Games

Planning tournaments for three player games is a difficult task. The goal is to make the tournament as balanced as possible. We would ideally want every possible pair of players to play together in a game exactly once. We also want all of the players to play in all of the rounds of the tournament. This is, however, only achievable, when the number of players is $v = 6k + 3$. In such case, we construct a KTS of order v and interpret its blocks as games and its parallel classes as rounds of the tournament.

If the number of players is a multiple of 3, we can always construct a *partially balanced tournament*. In it, the players form groups and pairs of players from the same group will not play together. Pairs of players that come frome different groups will play exactly once together. For the planning of a partially balanced tournament, a GDD is used.

Plan Tournament

Number of players: 9

Completely Balanced Tournament Plan

Number of players	9
Number of rounds	4
Number of games in a round	3
Total number of games	12

Tournament Schedule

Round 1	0, 1, 2	3, 4, 5	6, 7, 8
Round 2	0, 3, 6	1, 4, 7	2, 5, 8
Round 3	0, 4, 8	1, 5, 6	2, 3, 7
Round 4	0, 5, 7	1, 3, 8	2, 4, 6

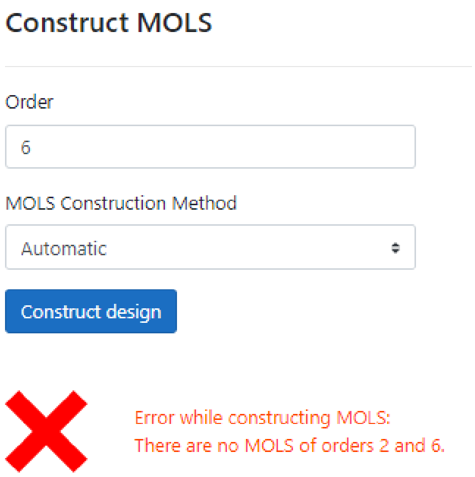
Figure 4.7: Page for the construction of balanced tournaments for three player games. Individual sections are highlighted with rectangles.

4.6 User interface design

The design of the rendered page, which is displayed to the user of the `TournamentPlanner` application, can be seen in figure 4.7 and is fairly simple. The user can select the type of the design or tournament in the left menu and the content of the page changes accordingly.

The content of the page is divided into several sections, but only 2 sections are displayed initially. The first introductory section introduces the selected tournament or design and suggests the input parameters for which the tournament or design can be constructed.

The second form section contains one or more input fields to specify the input parameters and a submit button. On submitting the form, the `TournamentPlanner` attempts to construct the selected structure for the specified input parameters. If the construction fails an error message is displayed as in figure 4.8. Otherwise, one or more result sections appear visualizing the constructed structures. The rounds of tournaments or the blocks of the designs are displayed in tables.



Construct MOLS

Order

6

MOLS Construction Method

Automatic

Error while constructing MOLS:
There are no MOLS of orders 2 and 6.

Figure 4.8: Error message message when constructing MOLS of order 6 which do not exist.

Chapter 5

Balanced three-player game tournaments

In this chapter, balanced and partially balanced three-player tournaments will be discussed and methods for constructing them will be shown. We will only consider situations, in which the number of players is a multiple of 3 - so there are no left over players in any of the rounds of the tournament. Ideally, in a balanced tournament, each possible pair of player will play a game together exactly once. This is however only achievable for certain numbers of players.

Fortunately, for any number of players (that is divisible by 3), it is possible to construct a partially balanced tournament. In such a tournament, every player will take part in the same number of games and will play a game in each of the rounds of the tournament. Contrary to the completely balanced tournament, there will be groups of players, that are not going to play a game together during the tournament.

An attentive reader could have already noticed that there is a close connection between a KTS and completely balanced three-player game tournaments. To reiterate, a KTS(v) is a BIBD($v, \lambda = 3, k = 3$) with the additional property of being resolvable. Indeed, the blocks of the KTS can be interpreted as the games in the tournament and the parallel classes as the individual rounds of the tournament. As has been stated in theorem 2.2, KTS only exist for orders v, such that $v = 6n + 3$, $n \in \mathbb{N}$. So to construct a completely balanced three-player games tournament with $v = 6n + 3$ players, we only need to construct a KTS(v). Unfortunately, the construction of KTSs is a difficult task and so a large part of this chapter is dedicated to describing this construction.

5.1 Construction of Kirkman triple systems

There are several KTS construction methods, most of them recursive in nature [13][14]. The method used here uses smaller KTSs to construct larger ones as in Lindner and Rodger [2]. The main theorem, this construction method is based on goes as follows:

Theorem 5.1. *If there exists a PBD$(3n+1)$ with block sizes $k_1, \dots, k_x$ and if there exists a KTS$(2k_i+1)$ for all $1 \leq i \leq x$, then there exists a KTS$(6n+3)$.*

Before we describe the steps of the construction, we need to clarify a few things. The PBDs (or pairwise balanced designs) mentioned in the theorem were introduced in section 2.4.1. For the KTS construction, we will only consider PBDs with block sizes $k \in \{4, 7, 10, 19\}$. Later in this chapter, it will be shown how to construct such PBD$(3n+1)$ for any $n \geq 1$. For now, just suppose we are able to construct them.

For all of the listed block sizes $k \in \{4, 7, 10, 19\}$ of a PBD, we are able to construct small KTSs of orders $2k + 1$: $9, 15, 21, 39$. These smaller auxiliary KTSs are constructed directly (in the code, their blocks are defined statically) and will be used to construct the blocks of the larger KTS.

In the construction, instead of denoting the elements of the design as single integers as we are used to, we use pairs of integers and a special element denoted as ∞. Each unique pair represents a unique element of the design. After the design is constructed, an arbitrary bijection can be used to map the pairs back to single integers.

Now we can finally show, how to construct the blocks of the new, larger KTS. Suppose we have already constructed a PBD$(3n+1)$ with elements $\{1, 2, \dots, 3n+1\}$ and with block sizes in $\{4, 7, 10, 19\}$. We define the elements of the new KTS$(6n+3)$ as $S = \{\infty\} \cup \{\{1, 2, \dots, 3n+1\} \times \{1, 2\}\}$ with blocks B. We obtain the blocks of the new KTS using the following 2 approaches:

1. For $1 \leq i \leq 3n+1$, construct blocks $\{\infty, (i, 1), (i, 2)\}$ and

2. for each block b of the PBD$(3n+1)$, use the smaller auxiliary KTS$(2|b|+1)$ with elements $S(b) = \{\infty\} \cup \{b \times \{1, 2\}\}$ renamed in such a way, that blocks $\{\infty, (i, 1), (i, 2)\}$ for all $i \in b$ will be amongst the blocks of the auxiliary KTS. We will call this renamed auxiliary KTS as b *induced KTS*. All the blocks of the b induced KTS except $\{\{\infty, (i, 1), (i, 2)\} \mid i \in b\}$ are also blocks of the new KTS.

It might be hard to see, that the design constructed in a such a way is really a KTS$(6n+3)$. We have constructed all the blocks and later we will show, how to obtain the parallel classes of the design. First, however, we will try to demonstrate, that the constructed blocks truly form a STS (i.e. KTS without resolvability).

Clearly, all the pairs of elements in block b of PBD are unique to block b and do not occur together in any other block of PBD. Then, in the b induced KTS, all the blocks except $E = \{\{\infty, (i, 1), (i, 2)\} \mid i \in b\}$ form a set $\{\{(i, *), (j, *), (k, *)\} \mid i, j, k \in b,\ * \in \{1, 2\}, i \neq j \neq k\}$. All the elements $(i, *), (j, *), (k, *)$ occur together exactly once (property of KTS) and as the pairwise occurrences of i, j, k are unique to block b, none of these elements occur together in any other block induced KTS.

Now, considering all the block induced KTSs for all the blocks of the PBD, all pairs of element in S should occur together exactly once. So the constructed design should be a STS.

n	new KTS	PBD	block sizes	aux. KTSs
1	9	4	4	-
2	15	7	7	-
3	21	10	10	-
4	27	13	4	9
5	33	16	4	9
6	39	19	19	-
7	45	22	4, 7	9, 15

Table 5.1: Orders of PBDs, their block sizes and the orders of auxiliary KTSs required for the construction of new KTSs. Several values of n.

Example 5.1 (Construction of KTS(27))**.** *The smallest order of KTS for which the recursive construction can be applied is* 27 *(see 5.1). The blocks of the required PBD*(13) *are of size* 4 *and can be constructed in a cyclic manner (see 5.2.1). For the auxiliary KTS*(9)*, the design in example 2.2 can be used. The* 27 *elements of the resulting KTS are* $S = \{\infty\} \cup \{\{1, 2, \dots, 13\} \times \{1, 2\}\}$.

Now, let us describe how to find the parallel classes of the newly constructed KTS. Every KTS$(6n + 3)$ is going to have $3n + 1$ parallel classes - that is of course the number of elements of the PBD used during the construction.

For each element i of the PBD, we will obtain a parallel class π_i. We continue with example 5.1 and try to find π_3. First, we need to find all the blocks of PBD(13) containing the element 3.

$$B_3 = \{\{1, 2, 3, 4\}, \{5, 10, 3, 8\}, \{9, 6, 3, 12\}, \{3, 7, 11, 13\}\}$$

For each block $b \in B_3$, we find the parallel class of b induced KTS containing block $\{\infty, (3, 1), (3, 2)\}$ (generally for i, find the parallel classes containing $\{\infty, (i, 1), (i, 2)\}$).

Block b	π of b induced KTS containing $\{\infty, (3, 1), (3, 2)\}$
$\{1, 2, 3, 4\}$	$\{\infty, (3, 1), (3, 2)\}$, $\{(1, 1), (4, 1), (2, 2\}$, $\{(1, 2), (2, 1), (4, 2)\}$
$\{5, 10, 3, 8\}$	$\{\infty, (3, 1), (3, 2)\}$, $\{(5, 1), (8, 1), (10, 2)\}$, $\{(5, 2), (10, 1), (8, 2)\}$
$\{9, 6, 3, 12\}$	$\{\infty, (3, 1), (3, 2)\}$, $\{(9, 1), (12, 1), (6, 2)\}$, $\{(9, 2), (6, 1), (12, 2)\}$
$\{3, 7, 11, 13\}$	$\{\infty, (3, 1), (3, 2)\}$, $\{(7, 1), (11, 1), (13, 1)\}$, $\{(7, 2), (13, 2), (11, 2)\}$

The union of these parallel classes forms the parallel class π_3 of the newly constructed KTS. The block $\{\infty, (3, 1), (3, 2)\}$ is naturally only listed once.

The construction described in this section is implemented in the `RecursiveKTSConstruction` class.

5.2 Construction of pairwise balanced designs

In the previous section, we claimed that the PBD$(3n + 1)$ with block sizes in $\{4, 7, 10, 19\}$ can be constructed for any $n \geq 1$. Now, we will try to prove this.

Similarly as with the KTS, the PBD construction method also uses smaller PBDs to construct the larger ones. With the KTS construction we only required small KTSs of orders $9, 15, 21, 39$. Here, however, we need 'small' PBDs up to the order 46 to be able to construct larger ones.

First, we demonstrate a few of the smaller PBD construction methods. Then we introduce Wilson's group divisible design construction and we show how it can be extended to obtain PBDs of larger orders.

5.2.1 Smaller order PBDs

There are several special construction methods for PBD($v = 3n + 1$) up to the order $v = 46$ with the addition of orders $v = 79, 82$. Only a few of these methods are shown bellow. All of the methods, however, are implement in `SmallOrderPBDConstruction` and neatly summarized in [2].

PBD($v = 4, 7, 10, 19$). Design with a single block $b = \{1, 2 \ldots, v\}$.

PBD($v = 13$). Two mutually orthogonal Latin squares of order 3 can be used as orthogonal quasi-groups $(\{1, 2, 3\}, \circ_1)$, $(\{1, 2, 3\}, \circ_2)$ (see 3.3) to construct PBD(13) with elements $S = \{\infty\} \cup (\{1, 2, 3\} \times \{1, 2, 3, 4\})$ and blocks

$$\begin{aligned} B = \{&\{(x, 1), (y, 2), (x \circ_1 y, 3), (x \circ_2 y, 4) \mid 1 \leq x, y \leq 3\}\} \\ &\cup \{\{\infty, (1, l), (2, l), (3, l)\} \mid 1 \leq l \leq 4\}. \end{aligned}$$

PBD(16) and PBD(28) can be constructed in a very similar manner.

PBD($v = 37$) Use cyclic construction to obtain blocks $B = \{\{i, i+1, i+3, i+24\}, \{i, i+4, i+9, i+15\}, \{i, i+7, i+17, i+25\} \mid 1 \leq i \leq 37\}$ reducing all sums module 37. PBD(40) can be constructed similarly.

5.2.2 Wilson's GDD construction

Theorem 5.2 (Wilson's Fundamental Construction [15]). *Let there be a GDD with elements S, groups G and blocks B. Let ω be a positive integer called weight, and let $W = \{1, 2, \ldots, \omega\}$. Suppose that for each $b \in B$, there exists a (auxiliary) GDD with elements $W \times b$, groups $\{W \times \{p\} \mid p \in b\}$ and blocks $B(b)$. Then, there exists a GDD with elements $S' = W \times S$, groups $G' = \{W \times g \mid g \in G\}$ and blocks $B' = \bigcup_{b \in B} B(b)$.*

Wilson's construction is somewhat similar to the KTS construction described in the previous section. Instead of auxiliary KTSs transformed into block induced KTSs, it uses auxiliary GDDs transformed into block induced GDDs. It turns out, that the only auxiliary GDDs we require to use this construction efficiently are GDD(v, g, k) with uniform group and block sizes.

Example 5.2 (GDD$(42, 6, 4)$ construction). ***GDD***$(12, 3, 4)$ *can be obtained from PBD*(13) *above by removing element* ∞ *from its blocks and considering the modified blocks* $\{(1, l), (2, l), (3, l) \mid 1 \leq l \leq 4\}$ *as groups of the GDD.*

GDD$(14, 2, 4)$ *with elements* $\{1, 2, \ldots, 14\}$ *can be constructed cyclically by defining groups as* $G = \{\{i, i + 7 \ \mid 1 \leq i \leq 7\}\}$ *and blocks as* $B = \{\{i, i + 1, i + 4, i + 6\} \mid 1 \leq i \leq 14\}$ *reducing all sums modulo 14.*

Now, apply Wilson's construction to GDD$(14, 2, 4)$ *using weight* $w = 3$*. All its blocks* b *have size 4, so we can use auxiliary GDD*$(12, 3, 4)$ *to obtain* b *induced GDDs for each of the blocks. Following Wilson's construction, we end up with a GDD*$(42, 6, 4)$ *with elements* $\{\{1, 2, 3\} \times \{1, 2, \ldots, 14\}\}$*, groups* $\{\{1, 2, 3\} \times \{i, i + 7\} \mid 1 \leq i \leq 7\}$ *and block* $\bigcup_{b \in B} B(b)$*.*

Wilson's construction is implemented in the `WilsonGDDConstruction` class.

5.2.3 Larger order PBDs

It was stated that we are able to construct PBD$(v = 3n + 1)$ up to the order 46 with the addition of orders 79, 82.

Now, we show how to constructs PBDs for $v = 3n + 1 \geq 49, v \notin \{79, 82\}$, again with block sizes in $\{4, 7, 10, 19\}$. We will use Wilson's Fundamental Construction in 5.2 and the following theorem, the proof of which can be found in [17].

Theorem 5.3. *For all* $n \geq 4, n \neq 6$ *and possibly* $n \neq 10$*, there exist 3 MOLS*(n)*.*

The PBD order v can be expressed as

$$v = 12m + 3t + 1, \quad \text{where } 0 \leq t \leq m, m \geq 4 \text{ and } m \notin \{4, 6\}. \tag{5.1}$$

By theorem 5.3, there exist 3 MOLS of order m forming 3 orthogonal quasi-groups $(Q, \circ_1)$, $(Q, \circ_2)$ and $(Q, \circ_3)$ on the set $Q = \{1, 2, \ldots, m\}$. Using them, we define a GDD$(5m, m, 5)$ with elements $S' = Q \times \{1, 5, 3, 4, 5\}$, groups $G' = \{Q \times \{i\} \mid 1 \leq i \leq 5\}$, and blocks $B' = \{\{(x, 1), (y, 2), (x \circ_1 y, 3), (x \circ_2 y, 4), (x \circ_3 y, 5)\} \mid x, y \in Q\}$.

Next, we modify this GDD by deleting $m - t$ elements that have the same second coordinate, for example 4. This leaves a GDD (S, G, B) with $|S| = 4m + t$ elements, 4 groups of size m and 1 group of size t, and blocks of size 4 and 5.

To this modified GDD, we apply Wilson's Fundamental Construction with weight $w = 3$. For its blocks b of size 4, we use auxiliary GDD$(12, 3, 4)$ from example 5.2 to get $B(b)$. For blocks of size 5, we use GDD$(15, 3, 4)$ formed by deleting one point from PBD(16) so that all its blocks have size 4.

The newly constructed GDD has $12m + 3t$ elements. All its blocks are of size 4 and there are 4 groups of size $3m$ and one block of size $3t$.

Finally, we add a new element ∞ to each of the groups, producing 4 subsets of size $3m + 1$ and 1 subset of size $3t + 1$. As the sizes of these subsets fit the PBD$(3n + 1)$ from, we can recursively replace them with the correctly renamed blocks of PBD$(3m + 1)$ and PBD$(3t + 1)$ with sizes in $\{4, 7, 10, 19\}$.

This construction is implemented in `LargerOrderPBDConstruction`.

5.3 Partially balanced tournaments

The KTS approach to the three player tournaments is only useful, when the number of players is $6n + 3$, $n \in \mathbf{N}$. However, as was suggested in the introduction of this chapter, we are able to plan a partially balanced tournament for any number of players divisible by 3: $3n$, $n \in \mathbf{N}$ with the exception of $n = 2, 6$.

For the scheduling of partially balanced tournaments, we will only require resolvable GDDs with blocks of size 3 and 3 groups of size n. Fortunately, this class of GDDs - denoted as RGDD$(3n, n, 3)$ - can be constructed simply using Mutually orthogonal Latin squares and adjusted orthogonal arrays. The resolvable classes of the design can be then directly interpreted as the rounds of the tournament.

The steps for the construction of RGDD$(3n, 3, n)$ are:

- construct 2 MOLS of order n;
- construct an orthogonal array equivalent to the 2 MOLS;
- adjust the orthogonal array by leaving out its first row, adding n to the elements of its 3rd row and adding $2n$ to the elements of its 4th row;
- form the blocks of the RGDD taking the columns of the orthogonal array in the previous step;
- form the 3 groups of the RGDD as $\{1, 2, \ldots, n\}$, $\{n+1, n+2, \ldots, 2n\}$ and $\{2n+1, 2n+2, \ldots, 3n\}$.

In a RGDD constructed in this way, the first n blocks form the fist parallel class π_1, the following n blocks form the second parallel class π_2 and so on. There will be n parallel classes.

Because RGDD$(3n, n, 3)$ is structure equivalent to 2 MOLS of order n and because there are no MOLS of order 2 and 6 (see theorem 3.1), there are no such RGDDs for $n = 2, 6$.

Example 5.3 (RGDD$(9, 3, 3)$)**.** *Two mutually orthogonal Latin squares of order 3 can be constructed using the method described at 3.6.1.*

$$\begin{array}{ccc} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{array} \qquad \begin{array}{ccc} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{array}$$

The equivalent orthogonal array obtained from the 2 MOLS:

$$\begin{array}{ccccccccc} 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 2 & 3 & 1 & 3 & 1 & 2 \\ 1 & 2 & 3 & 3 & 1 & 2 & 2 & 3 & 1 \end{array}$$

The adjusted orthogonal array:

$$\underbrace{\begin{matrix}1&2&3\\4&5&6\\7&8&9\end{matrix}}_{\pi_1}\;\underbrace{\begin{matrix}1&2&3\\5&6&4\\9&7&8\end{matrix}}_{\pi_2}\;\underbrace{\begin{matrix}1&2&3\\6&4&5\\8&9&7\end{matrix}}_{\pi_3}$$

The groups of the constructed RGDD are $\{1,2,3\}$, $\{4,5,6\}$, $\{7,8,9\}$. *The parallel classes of the design are indicated by curly braces. The resulting tournament is captured in the table bellow. Notice, that the players from the same group never play a game together.*

Round #	Game 1	Game 2	Game 3
Round 1	1,4,7	2,5,8	3,6,9
Round 2	1,5,9	2,6,7	3,4,8
Round 3	1,6,8	2,4,9	3,5,7

Table 5.2: Partially balanced tournament with 9 players.

For the same number of players $v = 6n + 3$, there are $2n + 1$ rounds in a partially balanced tournament and $3n + 1$ rounds in a completely balanced one. So partially balanced tournaments can also be used, when we want to reduce the number of rounds.

Chapter 6

Whist tournaments

Classic whist tournaments consist of games, where two players play against other two players. The name *Whist* refers to an old English card game[16], but the tournament schedule can be applied to many other games such as doubles tennis, table football or many video games where two players compete against other two players.

In this chapter, we only discuss Whist tournaments for $4n$ and $4n+1$ players, for which they are know to exist. In case of $4n$ players, we provide construction methods up to $n = 10494$.

Construction of Whist tournaments has been studied for more than a century. The first ground-breaking work was Moore's 1896 [20] publication, in which several construction methods have been described. Anderson and Finizio expanded upon his work in [21]. Anderson later summarized his and Moore's work in [1], which is the main source for this chapter.

Definition 6.1. *A whist tournament* $Wh(4n)$ $(Wh(4n+1))$ *with* $4n$ $(4n+1)$ *players is a schedule of games, where each game involves two players playing against other two players, such that:*

- *for* $4n$ *players, the games are arranged in* $4n-1$ *rounds, each consisting of* n *games and each player plays in exactly one game in each round;*
- *for* $4n+1$ *players, the games are arranged in* $4n+1$ *rounds, each consisting of* n *games and each player plays in one game in all but one of the rounds;*
- *each player partners every other player exactly once;*
- *each player opposes every other player exactly twice.*

Example 6.1 (Whist tournament with 8 players)**.** $Wh(8)$ *with 7 round, each consisting of 2 games.*

Round #	Game 1	Game 2
Round 1	8,1 v 5,6	2,4 v 3,7
Round 2	8,2 v 6,7	3,5 v 4,1
Round 3	8,3 v 7,1	4,6 v 5,2
Round 4	8,4 v 1,2	5,7 v 6,3
Round 5	8,5 v 2,3	6,1 v 7,4
Round 6	8,6 v 3,4	7,2 v 1,5
Round 7	8,7 v 4,5	1,3 v 2,6

It might seem strange at first that a player is supposed to play only once with others in a team but twice against others as an opponent. When we look at a single game, however, say

8,1 v 5,6

it becomes clear, that the game yields 2 pairs of partners $(8,1),(5,6)$ and 4 pairs of opponents $(8,5),(8,6),(1,5),(1,6)$.

There are several construction methods used to obtain Whist tournaments. A lot of them are using the properties of difference sets and finite fields. We give a brief introduction to these concepts.

6.1 Difference sets for Whist tournaments

To be able to work with difference sets (and finite fields), the concept of a (commutative) *group* needs to be introduced.

Definition 6.2 (Group [19]). *A group is a set G with a binary operation $\circ$, such that:*

- *the associative law $(x \circ y) \circ z = x \circ (y \circ z)$ holds for any $x, y, z \in G$;*
- *there is an **identity element** $e \in G$ such that for all $x \in G$, $e \circ x = x \circ e = x$;*
- *for all $x \in G$, there exists an **inverse element** $y \in G$ such that $x \circ y = y \circ x = e$.*

*Furthermore, if the commutative law $x \circ y = y \circ x$ holds for any $x, y \in G$, the group is said to be **commutative** or **abelian**.*

Now, the difference sets can be introduced.

Definition 6.3 (Difference set [1]). *A (v, k, λ) difference set in an additive abelian group G of order v is a set $D = \{d_1, \ldots, d_k\}$ of distinct elements from G such that each non-zero element $g \in G$ can be expressed as $g = d_i - d_j$ exactly λ times.*

A so called *translate* of a difference set D can be obtained by $D+g,\ g \in G$. A translate of a difference set is also a difference set. Taking all the translates of a difference set as blocks forms a $\mathrm{BIBD}(v, k, \lambda)$.

Often, a single difference set D is not enough to generate a valid design.

Definition 6.4 (Difference system). *Let $D_1, \dots, D_n$ be sets of size k in an abelian group G of order v such that the differences of elements in D_i, $1 \leq i \leq n$ give all the non-zero elements of G exactly λ times.*

Difference systems generate block designs in the same way as difference sets.

For the construction of Whist tournament, we need to extend the concept of difference sets.

Definition 6.5 (Starter and 2-fold starter). *A starter in an abelian group G of order $2n-1$ is a set of $n-1$ unordered pairs $\{x_1, y_1\}$, ..., $\{x_{n-1}, y_{n-1}\}$ of elements in G such that x_1, y_1, ..., x_{n-1}, y_{n-1} are exactly all the non-zero elements of G and $\pm(x_1 - y_1)$, ..., $\pm(x_{n-1} - y_{n-1})$ are also exactly all the non-zero elements of G.*

The pairs $\{x_1, y_1\}$, ..., $\{x_{2n-2}, y_{2n-2}\}$ form a 2-fold starter in G if x_1, y_1, ..., x_{2n-2}, y_{2n-2} are the non-zero elements of G, each occurring twice and $\pm(x_1 - y_1)$, ..., $\pm(x_{2n-2} - y_{2n-2})$ are also all the non-zero elements of G, each occurring twice.

The following theorem shows, how starters can be used to construct some Whist tournaments.

Theorem 6.1. *Games $(\infty, 0 \text{ v } b_1, d_1)$, $(a_2, c_2 \text{ v } b_2, d_2)$, ..., $(a_n, c_n \text{ v } b_n, d_n)$ where a_i, b_i, c_i, d_i are the non-zero elements of an additive group G, can be taken as the first round games of a cyclic $Wh(4n)$ if*

- *the pairs $\{a_i, c_i\}$, $\{b_i, d_i\}$ form a starter;*
- *the pairs $\{a_i, b_i\}$, $\{b_i, c_i\}$, $\{c_i, d_i\}$, $\{d_i, a_i\}$ form a 2-fold starter.*

Basically, the pairs of partners should form a single starter and the pairs of opponents should form a 2-fold starter because every game yields twice as many opponents than partners.

Example 6.2. *Take the first round of $Wh(8)$*

$$\infty,\ 0 \text{ v } 4,\ 5 \qquad 1,\ 3 \text{ v } 2,\ 6.$$

The additive abelian group is $G = \{0, 1, 2, 3, 4, 5, 6\}$ with operation $x \circ y = x + y$ modulo 7. The pairs of partners $\{4, 5\}$, $\{1, 3\}$, $\{2, 6\}$ form a starter. The pairs of opponents $\{0, 4\}$, $\{0, 5\}$, $\{1, 2\}$, $\{1, 6\}$, $\{3, 2\}$, $\{3, 6\}$ form a 2-fold starter. Cyclically developing the first round yields a tournament equivalent to example 6.1.

6.2 Finite fields

Definition 6.6 (Finite field [19]). *Finite field, also known as Galois fields and denoted as $GF(q)$, is a finite set T of size q with two binary operations addition $+$ and multiplication $\times$, such that:*

- *$(T, +)$ is a commutative group with identity element 0;*
- *$(T \setminus \{0\}, \times)$ is a commutative group with identity element 1;*
- *The distributive law $a \times (b + c) = a \times b + a \times c$ holds for any $a, b, c \in T$.*

The set of real numbers with the classic addition and multiplication is a (non-finite) field. The simplest example of a finite field is $\mathbf{Z}_p$ where p is a prime and the additions and multiplications are reduced modulo p.

Example 6.3. *$GF(5)$ with elements $\{0, 1, 2, 3, 4\}$.*

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\times$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Theorem 6.2. *If q is a prime or a prime power, there exists a $GF(q)$.*

The construction of prime power finite fields is more complicated and will not be addressed here.

Finite fields are extremely useful in cryptography, error correction and combinatorics. For the construction of Whist tournaments, the generating property of the *primitive element* of the finite field is utilized.

Definition 6.7. *A non-zero element θ of a $GF(q)$ is called a primitive element, if $\theta, \theta^2, \dots, \theta^{q-1} = 1$ are exactly all the non-zero elements of $GF(q)$.*

Example 6.4. *The primitive element of $GF(5)$ is 2:*

$$2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 3, \quad 2^4 = 1$$

It is fairly complicated to find the primitive element of a finite field and to perform calculation in the finite field arithmetic. For these purposes, we use the `AppliedAlgebra`[1] library.

[1] Open source .NET library developed by D. Litichevskiy. `https://github.com/litichevskiydv/GfPolynoms`.

6.3 Whist tournaments with $4n + 1$ players

The theorem 6.1 showing the cyclic construction of $\mathrm{Wh}(4n)$ can also be applied for the construction of $\mathrm{Wh}(4n+1)$. Only instead of an abelian group of order $4n$, we consider an abelian group of order $4n+1$ (or a GF in our specific case).

We only show the construction of $\mathrm{Wh}(v = 4n+1)$ where v is a prime or prime power. In such cases, the first round of the tournament can be written using the primitive element of $\mathrm{GF}(v)$ as

$$\theta^{i},\, \theta^{2n+i} \text{ v } \theta^{n+i},\, \theta^{3n+i}.$$

It can be shown that the ± differences between the partners yield all the non-zero elements of $\mathrm{GF}(v)$ and the ± differences between the opponents yield all the non-zero elements of $\mathrm{GF}(v)$ twice.

Example 6.5 (Wh(5))**.** *Take the GF(5) in example 6.3 with its primitive element 2. Then, the first round of Wh(5) is*

1, 4 v 2, 3.

The following 4 rounds can be obtained by cyclically developing the first round in GF(5):

2, 0	v	*3, 4*
3, 1	v	*4, 0*
4, 2	v	*0, 1*
0, 3	v	*1, 2*

This method is implemented in `PrimePowerWhistPlanner`. $\mathrm{Wh}(5)$ and $\mathrm{Wh}(17)$ are required for the construction of larger $\mathrm{Wh}(4n)$.

6.4 Whist tournaments with $4n$ players

As with the PBD construction, there is no unified method to construct Whist tournaments for smaller number of players. We show, however, that if we are able to construct $\mathrm{Wh}(4n)$ up to $n \leq 80$, then we can utilize transversal designs and GDDs to construct Whist tournaments of any order.

Theorem 6.3. *Suppose that a $Wh(4n)$ exists for all $n \leq 80$. Then a $Wh(4n)$ exists for all $n \geq 1$.*

The goal of this section is to show the construction methods for $n \leq 80$ and then introduce the technique, which will allow the construction of any $\mathrm{Wh}(4n)$.

6.4.1 Cyclic Wh(4n)

Tournaments $\mathrm{Wh}(4)$, $\mathrm{Wh}(8)$, $\mathrm{Wh}(12)$, $\mathrm{Wh}(16)$, $\mathrm{Wh}(20)$ and $\mathrm{Wh}(24)$ can be formed by cyclically developing their first rounds, because these round meet the requirements in 6.1. The first rounds are listed bellow.

Wh(4)

∞, 0 v 1, 2

Wh(8)

∞, 0 v 4, 5   1, 3 v 2, 6.

Wh(12)

∞, 0 v 4, 5   1, 10 v 2, 8   3, 7 v 6, 9

Wh(16)

∞, 0 v 5, 10   1, 2 v 4, 8   3, 11 v 12, 14   6, 9 v 7, 13

Wh(20)

∞, 0 v 11, 12   3, 9 v 16, 1   4, 14 v 13, 18   6, 8 v 5, 2   7, 15 v 10, 17

Wh(24)

∞, 0 v 7, 10   1, 8 v 12, 22   5, 6 v 2, 11   3, 9 v 14, 18   17, 19 v 4, 16
15, 20 v 13, 21

The cyclic construction is implemented in the `CyclicWhistPlanner`.

6.4.2 Moore's construction

The following method has first been described by E. H. Moore in 1896 [20]. It allows the construction of Wh$(v = 3q + 1)$ where q is a prime power and can be written as $q = 4m + 1,\ m \in \mathbf{N}$.

First, construct a resolvable BIBD$(3q + 1, 4, 1)$ by cyclically developing its first parallel class

$$\{0_1, 0_2, 0_3, \infty\};$$
$$\{\theta_j^i, \theta_j^{i+2m}, \theta_{j+1}^{i+m}, \theta_{j+1}^{i+3m}\} \quad 1 \leq i \leq m - 1,\ 1 \leq j \leq 3$$

reducing all sums containing j modulo 3. This parallel class is defined using *mixed difference sets* which are basically just 3 normal difference sets on GF(q) distinguished by the subscript. θ is naturally the primitive element of GF(q). The design contains q elements for each of the difference sets and one extra element ∞ - that is $3q + 1$ in total.

This construction of a resolvable BIBD is implemented in the `MooreRBIBDConstruction` class.

Next, construct a Wh(4) on each block of a given parallel class. Taken together, these yield 3 rounds of a Wh$(3q+1)$. Repeat for each of the parallel classes to obtain all the rounds of the tournament.

This part of the Wh$(3q + 1)$ construction is implemented in `MooreWhistPlanner`.

6.4.3 Product construction

Whist tournaments can be obtained as products of smaller whist tournaments.

Theorem 6.4. *Suppose that* $Wh(v)$ *and* $Wh(w)$ *exist. Then a* $Wh(vw)$ *also exists.*

There are actually 2 different constructions we require. One is used when both v and w are even and the other when one of them is odd. We only describe the even-even method, the other one is very similar.

Suppose that $v \equiv w \equiv 0 \pmod 4$. Given Wh($v$) on set S_v and Wh(w) on set S_w, the resulting Wh(vw) is constructed on $S = S_v \times S_w$. The first $w-1$ rounds can be obtained simply. For each $i \in S_v$, form tournaments Wh(w) on $i \times S_w$. The unions of the parallel rounds of these Wh(w) form the first $w-1$ rounds of Wh(vw).

Next, take one round of Wh(v) and for all its games i, j v k, l, form a resolvable TD$(4, w)$ on $\{i, j, k, l\} \times S_w$ with its groups determined by the first component. A block $\{(i, p), (j, q), (k, r), (l, s)\}$ of the TD can be interpreted as game $(i, p), (j, q)$ v $(k, r), (l, s)$. Take all these games formed by the games of one round of Wh(v) belonging to the same parallel class of TD to obtain a new round of Wh(vw). A single round of Wh(v) with the TDs constructed on its games yields w rounds of Wh(vw). Repeat this for all the $v-1$ round of Wh(v) to obtain $w(v-1)$ rounds of Wh(vw). Together with the first $w-1$, we have obtained all the $vw-1$ rounds of Wh(vw).

This construction method is implemented in the `EvenEvenProductWhistPlanner` class. `EvenOddProductWhistPlanner` implements the construction for cases when one of the whist orders is odd.

6.4.4 Construction utilizing SAMDRRs

Definition 6.8. *Spouse-avoiding mixed doubles round robin tournament for* n *couples or SAMDRR*(n) *is a tournament involving* n *husband and wife pairs. Each game of the tournament involves two pairs of the opposite sex competing against each other. The games are scheduled in such way, that every player plays exactly one game against every other player of the same sex and every player plays with each member of the opposite sex (except his or her spouse) once as a partner and once as an opponent. A game is denoted as* H_iW_k v H_jW_l.

Resolvable SAMDRRs can be used to schedule whist tournaments. The construction of resolvable SAMDRR(n), where GCD$(n, 6) = 1$ was described (in disguise as construction of self orthogonal Latin square) by Mendelsohn [22] and is very simple: For round k, $1 \le k \le n$, construct games $1 \le h \le \frac{n-1}{2}$ as

$$H_{k-h}W_{k-3h} \text{ v } H_{k+h}W_{k+3h}$$

reducing all sums modulo n. Notice that $\frac{n-1}{2}$ is an integer because GCD$(n, 6) =$ 1. This construction is implemented in the `SimpleSAMDRRPlanner` class.

A resolvable SAMDRR(n) where n is odd has n rounds and can be transformed into a Wh($4n$) on set $S = \{1, \dots, n\} \times \{1, 2, 3, 4\}$. For simplicity, we denote element (i, j) as i_j. Each round of the SAMDRR yields 3 rounds of the whist tournament. For every game $H_i W_l$ v $H_j W_k$ of a round t of the SAMDRR, construct 3 games for each of the 3 rounds of the whist tournament as follows:

- i_1, j_1 v l_3, k_3, i_2, j_2 v l_4, k_4, t_1, t_3 v t_2, t_4;
- i_4, l_1 v j_4, k_1, i_3, l_2 v j_3, k_2, t_1, t_4 v t_3, t_2;
- i_1, l_2 v k_2, j_1, i_3, l_4 v k_4, j_3, t_1, t_2 v t_4, t_3.

This gives $3n$ rounds. The last $n-1$ rounds are obtained using a TD$(4, n)$ constructed on set S in such way that all the elements in a single group have the same subscript and with one parallel class containing all the blocks $\{t_1, t_2, t_3, t_4\}$, $1 \leq t \leq n$. From the other remaining $n-1$ parallel classes of the TD, form $n-1$ rounds of the tournament by interpreting blocks $\{i_1, j_2, k_3, l_4\}$ as games i_1, k_3 v j_2, l_4.

6.4.5 Construction of larger Whist tournaments

The previously described methods - the cyclic construction, Moore's, product and SAMMDR - taken together allow for the construction of Wh($4n$), $n \leq 80$. Now we show how to obtain larger Wh($4n$). The described method is yet again recursive in nature and is based on the following theorem.

Theorem 6.5. *Let $v = 4n$ and suppose that there exists a GDD of order $v-1$ such that for all its blocks sizes $k \in K$, there exists a Wh(k) and for all its group sizes $g \in G$ there exists a Wh($g+1$) and $g \equiv 3 \pmod 4$. Then a Wh(v) exists.*

First, construct a transversal design TD$(17, g)$ with $17g$ elements, blocks of size 17 and 17 groups of size g. Valid choices of g will be shown later. `TDConstruction` can be used to obtain this TD.

Next, take $0 \leq u \leq g$, $0 \leq v \leq g$ and select u elements from the first group and v elements from the second group of the transversal design. Replace each of the selected elements with 5 new ones and place them in the same group as the selected element. Now, there should be $17g + 4(u+v)$ elements, 15 groups of size g, 1 group of size $g + 4u$ and one group of size $g + 4v$.

Any block of the of the TD contains at most 2 replaced elements - according to the definition 3.6, each element of a block is from a different group. If a block B has 1 replaced element, form a set containing the 16 unchanged elements of the block and the 5 new elements the original was replaced by. On this set of size 21, construct a BIBD$(21, 5, 1)$ such that the 5 new elements form a single block of this design. Replace the original TD block B with the 20 other blocks of the BIBD (leave out the block with the new elements).

If a block contains 2 replaced elements, take the 15 unchanged elements and the 10 new ones. On this set of size 25, form a BIBD$(25, 5, 1)$ such that the 2 sets of 5 new elements form 2 disjoint blocks of the new design. Replace the original block B by the other 28 blocks (leaving out the 2 blocks with the new elements).

If a block of the TD does not contain any replaced element, leave it unchanged.

With these adjustments to the groups and blocks of the original TD, a GDD with $17g{+}4(u{+}v)$ elements, blocks of sizes 5 and 17 and groups of sizes g, $g{+}4u$, $g{+}4v$ is created. This is exactly the GDD from theorem 6.5. This construction is implemented in `GDDForWhistConstruction`. The auxiliary BIBD$(21, 5, 5)$, $(25, 5, 1)$ are constructed in `SmallBIBDForWhistConstruction`. Valid choices of g for some n are listed bellow.

Range of n	g
81 - 119	19
120 - 169	27
170 - 194	31
195 - 269	43
270 - 294	47
295 - 419	67

Range of n	g
420 - 519	83
520 - 669	107
670 - 869	139
870 - 1244	199
1245 - 1519	243
1520 - 2019	323

Range of n	g
2020 - 2769	443
2770 - 3869	619
3870 - 5394	863
5395 - 7544	1207
7545 - 10494	1679

Example 6.6. *For Wh*$(4n = 348)$, $n = 87$, *a GDD*(347) *is required. It can be constructed with* $g = 19$, $u = 3$, $v = 3$. *Indeed,* $17 \times 19 + 4(3 + 3) = 347$. *Other choices of* u, v *are also possible.*

The last step is converting the GDD$(v-1)$ to Wh(v). Take the set of elements X of the GDD and add the last element ∞. On each block of the GDD of size $k \in \{5, 17\}$, form a Wh(k). On the elements of each group of size g with ∞ adjoined, form a Wh$(g+1)$ and label all its round by the g elements of the group (arbitrarily).

Wh(v) is going to have $|X| = v - 1$ rounds. For each $x \in X$ take as round x all the games from the rounds of Wh$(g+1)$ (constructed on a group containing x) labelled as x and add the games from the rounds omitting x of Wh(k) constructed on blocks containing x (each such Wh(k) has exactly one round omitting x). This last step is implemented in the `LargerWhistPlanner` class.

Chapter 7

Constraint programming approach to the scheduling of tournaments

In the previous two chapters, we only focused on very specific tournament types - be it three player game tournaments or Whist tournaments. For their constructions, we mostly applied combinatorial designs.

Unfortunately, the construction of even the simpler design structures turned out to be fairly difficult. Forming a KTS for example, which is just a resolvable BIBD$(v, 3, 1)$, requires specific MOLS, group divisible designs and pairwise balanced designs.

For the construction of other tournament types such as four player game tournament, we would require a whole new suite of design structures and algorithms. And even with these new algorithms, we would only be able to construct tournaments in highly specific cases.

With three player games and 9 players for example, we can only construct a balanced tournament that has exactly 4 rounds. It is not possible to adjust the number of rounds, nor is it possible to plan tournaments for arbitrary number of players.

In this chapter, we propose a new, less rigid approach. Instead of using balanced designs, we formulate the task of tournament construction as an constraint programming problem and use a solver to optimize it.

We use the `CP-SAT`[1] solver that can be applied for both constraint programming and integer programming problems. We also use it for the construction of BIBDs as described in sections 2.5.1 and 2.5.2.

7.1 Problem formalization

In this section, one possible formalization of the tournament planning problem is given.

[1] `CP-SAT` solver is part of the Google's OR-Tools bundle. It is free and open source. `https://developers.google.com/optimization/cp/cp_solver`

7.1.1 Variables of the model

For simplicity, we use similar terms as we used when working with block designs. We denote the number of players in the tournament as v, the set of players as S, the number of players in a single game as k and the number of rounds as r. Variables v, k, r are the input of the model.

First, compute the number of games played in a single round

$$m = \lfloor \frac{v}{k} \rfloor.$$

The number of players v does not have to be divisible by k - that is why a floor function is used. In such cases, some players will not take part in some of the rounds of the tournament.

Now we introduce the main binary decision variables we will work with:

$$b_{t,p,g} \text{ where } b_{t,p,g} \in \{0,1\} \text{ and } 1 \leq t \leq r,\ 1 \leq p \leq v,\ 1 \leq g \leq m$$

We say that $b_{t,p,g} = 1$ if and only if in round t the player p plays in the game g. Indices t, p, g are obviously used to index the rounds, players and games.

7.1.2 Constraints of the model

There are only 2 constraints that need to be ensured:

1. a player can play at most in one game in each round;
2. all the games in the tournament have k players.

Let us focus on a single round and only consider variables $b_{p,g}$. These can be captured in a $v \times m$ binary matrix

$$\begin{matrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & & \vdots \\ b_{v,1} & b_{v,2} & \cdots & b_{v,m} \end{matrix}$$

in which the rows represent the players and the columns represent the games in a single round. To ensure constraint 1, the sum of the variables in a row has to be less or equal to 1. To ensure constraint 2, the sum of variable in a column has to be equal to k.

Considering all the rounds, the constraints can be formalized with the following equations:

$$\sum_{g=1}^{m} b_{t,p,g} \leq 1 \quad \text{for all } 1 \leq t \leq r,\ 1 \leq p \leq v; \tag{7.1}$$

$$\sum_{p=1}^{v} b_{t,p,g} = k \quad \text{for all } 1 \leq t \leq r, 1 \leq g \leq m. \tag{7.2}$$

7.1.3 Objective of the model

All the variables and constraints used so far can be expressed using integer programming. To express the objective function, however, it is easier to use some logical constraints in the framework of constraint programming directly.

The objective function itself is simple. We just try to minimize the maximum number of occurrences of pairs of players across all the games in the tournament. Denote the number of pair occurrences of players i, j where $i < j$ as $\lambda_{i,j}$. Then the objective function can be written as 7.3. As the maximal λ is being minimized, the other λ values should increase. This way, the differences between the pair occurrences should be minimal. Furthermore, if the values of $\lambda_{i,j}$ are the same for all i, j and the number of players is $v = km$, the constructed tournament should be equivalent to a resolvable BIBD(v, k, λ).

$$\text{Minimize } \max\{\lambda_{i,j} \mid 1 \leq i < j \leq v\} \tag{7.3}$$

A more difficult task is to express $\lambda_{i,j}$ using the decision variables $b_{t,p,g}$. This can not be done via a simple summation. For all pair of players $i, j|; i < j$, we need to iterate through all the round and games and increase $\lambda_{i,j}$ only if the $b_{t,i,g} = b_{t,j,g} = 1$ - meaning that the players i, j both take part in the game g of round t.

Let us define a double square bracket function $[\![\alpha]\!] \rightarrow \{0, 1\}$, which takes a logical expression α as an argument and returns 1 if the expression α is evaluated as true and 0 otherwise. Then, the $\lambda_{i,j}$ can be expressed as

$$\lambda_{i,j} = \sum_{t=1}^{r} \sum_{g=1}^{m} [\![b_{t,i,g} \wedge b_{t,j,g}]\!] \quad . \tag{7.4}$$

Unfortunately, the `CP-SAT` solver does not provide (as far as I know) such a convenient double bracket function. We can only use simple logical operators *and, or, not* and *implication*.

We introduce a new binary variable $\lambda_{t,g,i,j}$, which is equal to 1 if and only if the players i, j play together in the game g of round t.

$$\lambda_{t,g,i,j} \iff b_{t,i,g} \wedge b_{t,j,g} \tag{7.5}$$

However, equivalence can not be used either in this form so we need to replace the single expression 7.5 with the following 3 logical expressions, which can finally be used directly in the CP-SAT solver:

$$\lambda_{t,g,i,j} \implies b_{t,i,g} \tag{7.6}$$

$$\lambda_{t,g,i,j} \implies b_{t,j,g} \tag{7.7}$$

$$\neg b_{t,i,g} \vee \neg b_{t,j,g} \vee \lambda_{t,g,i,j} \tag{7.8}$$

The first two expressions 7.6, 7.7 are equivalent to $\lambda_{t,g,i,j} \implies b_{t,i,g} \wedge b_{t,j,g}$ and the third one 7.8 is equivalent to $b_{t,i,g} \wedge b_{t,j,g} \implies \lambda_{t,g,i,j}$.

Finally, we can rewrite equation 7.4 as

$$\lambda_{i,j} = \sum_{t=1}^{r} \sum_{g=1}^{m} \lambda_{t,g,i,j} \tag{7.9}$$

and use the solver to minimize the objective function 7.3.

7.2 Evaluation of the CP approach

With three player game tournaments, we can select the number of players and rounds so that there exists a perfectly balanced tournament. However, proving that the found solution is optimal can take a long time for the CP-SAT solver. So we create a simple callback function, that stops the search when the objective value is 1. Then, the solution representing a perfectly balanced tournament has clearly been found. This way we can compare the run time of the CP scheduling method with the run time of the constructive method using KTSs for the scheduling of the same tournament.

# Players	CP approach	KTS approach
9	141	1
15	14748	1
21	8900657	2
27	-	2

Table 7.1: The run times of the 2 methods in milliseconds.

The method utilizing KTSs schedules the tournaments almost instantly, whereas the CP method can take up to several minutes, even on the smallest instances. For 27 players, the optimal solution was not found even after 30 minutes. This result is not surprising but it nicely illustrates how difficult it is to "brute-force" this problem. The CP approach should only be used, when we do not require an optimal solution.

To evaluate the quality of the solution, we use the difference between the maximal and minimal pair occurrences $\lambda_{max} - \lambda_{min}$. The smaller this difference is, the better the solution. This value is more telling than the objective function. However, when the difference was used as an objective function, the results were the same but the model was more complicated. That is because minimizing the objective value also minimizes this difference.

For the experiment, we yet again consider three player games, we fix the number of rounds to 5 and set the time limit of the solver to 30 seconds.

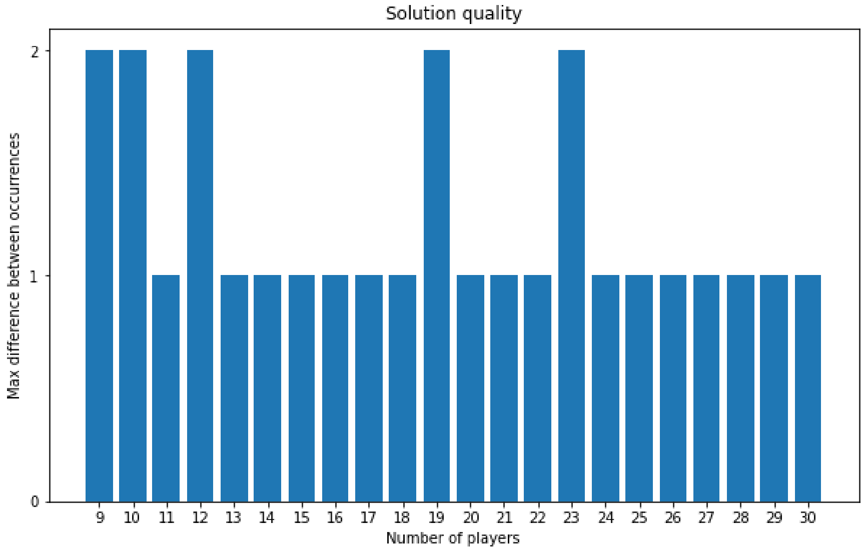


Figure 7.1: The quality of the solution measured as $\lambda_{max} - \lambda_{min}$ on the construction of three player game tournaments with 5 rounds. Time limit set to 30 seconds

The difference between the pairs that occur the most and the pairs that occur the least in the tournament ranges from 1 to 2. These values seem acceptable.

Chapter 8

Pairwise Testing

In the previous chapters, we have only been discussing the utilization of design structures for the scheduling of tournaments. Naturally, there are many other applications, where design theory turns out to be useful. In this chapter, we show one such application in the field of software testing.

8.1 Application of pairwise testing

Suppose there is a website selling public transportation tickets and fares, similar to PID Lítačka[1]. When buying a ticket, the customer has to specify the type of the ticket he wants to buy, its duration, the payment method and whether he is eligible to some sort of discount.

Discount	Tariff	Duration	Payment
Child	Type 1	1 Day	PayPal
Student	Type 2	1 Month	Bank transfer
Senior	Type 3	1 Year	Credit card

Table 8.1: Variables and the values they can take.

To exhaustively test the system with all combinations of values, we would require $3^4 = 81$ test cases. That is still manageable, but the number of test cases increases rapidly with the growing number of variables and values. With only 5 variables, where each of the variable can take 5 values, we would requite $5^5 = 3125$ test cases to test the system exhaustively.

With pairwise testing we can decrease the number of test cases significantly while still testing the system quite thoroughly. The rationale is, that the most common bugs in a piece of software are triggered either by a single input parameter or an interaction between pairs of parameters [23]. The number of bugs involving interactions between three or more parameters are progressively less common [24].

[1] Prague's public transportation website at `https://pidlitacka.cz` where customers can buy fares on-line.

Instead of creating 81 test cases to test the fare selling system, we can cover all the possible variable pair interactions with only 9 test cases.

TC #	Discount	Tariff	Duration	Payment
TC 1	Child	Type 1	1 Day	PayPal
TC 2	Child	Type 2	1 Month	Bank transfer
TC 3	Child	Type 3	1 Year	Credit card
TC 4	Student	Type 1	1 Month	Credit card
TC 5	Student	Type 2	1 Year	PayPal
TC 6	Student	Type 3	1 Day	Bank transfer
TC 7	Senior	Type 1	1 Year	Bank transfer
TC 8	Senior	Type 2	1 Day	Credit card
TC 9	Senior	Type 3	1 Month	PayPal

Table 8.2: The pairwise test cases that test all the possible variable pair interactions.

8.2 Simple construction of Pairwise test cases

The property of the pairwise test cases that all the possible pairs of parameters have to occur together in at least one test case is close to the $\lambda = 1$ property we discussed in the block designs - there clearly is some connection. The test cases, however, can not be represented directly by block designs. For this, we need *Orthogonal arrays.*

To reiterate, an orthogonal array $\mathrm{OA}(k, m)$ is a $k \times m^2$ matrix on m elements, such that any 2 rows contain each possible ordered pairs of elements in their vertical pairs. This is exactly the property we require for the pairwise tests.

Consider the following $\mathrm{OA}(4, 3)$:

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
Discount	1	1	1	2	2	2	3	3	3
Tariff	1	2	3	1	2	3	1	2	3
Duration	1	2	3	2	3	1	3	1	2
Payment	1	2	3	3	1	2	2	3	1

The 4 rows of the orthogonal array represent the 4 variables in table 8.1. Each of the rows only contains integers $1, 2, 3$. These numbers represent the values any of the variables can take. If we consider the mapping suggested in table 8.3, the columns of the orthogonal array provide exactly the 9 test cases in table 8.2.

Discount (row 1)	Tariff (row 2)	Duration (row 3)	Payment (row 4)
Child → 1	Type 1 → 1	1 Day → 1	PayPal → 1
Student → 2	Type 2 → 2	1 Month → 2	Bank transfer → 2
Senior → 3	Type 3 → 3	1 Year → 3	Credit card → 3

Table 8.3: Mapping of the possible variable values to the numbers in the rows of the orthogonal array.

In this example, the array fits the variable values perfectly. Generally, the factor k of the orthogonal array is chosen as the number of variables and the level m is chosen as the maximum number of values that each variable will take on. More often than not, the orthogonal array will not fit the variables and values perfectly.

8.3 Problems with the construction of test cases

There are two problems that need to be addressed when using orthogonal arrays to construct pairwise test cases.

8.3.1 Left over variable values

The number of values that a variable can take on might differ as in the following table.

Variable 1	Variable 2	Variable 3
$v_{1,1}$	$v_{2,1}$	$v_{3,1}$
$v_{1,2}$	$v_{2,2}$	$v_{3,2}$
-	$v_{2,3}$	-

The minimal orthogonal array required to construct the test cases is $\mathrm{OA}(3,3)$. However, there is one left over value in both its 1st and 3rd row that can not be mapped to any value of Variable 1 and Variable 3. The test cases constructed using this OA are captured in the table bellow.

TC #	Variable 1	Variable 2	Variable 3
TC 1	$v_{1,1}$	$v_{2,1}$	$v_{3,1}$
TC 2	$v_{1,1}$	$v_{2,2}$	$v_{3,2}$
TC 3	$v_{1,1}$	$v_{2,3}$	"left over"
TC 4	$v_{1,2}$	$v_{2,1}$	$v_{3,2}$
TC 5	$v_{1,2}$	$v_{2,2}$	"left over"
TC 6	$v_{1,2}$	$v_{2,3}$	$v_{3,1}$
TC 7	"left over"	$v_{2,1}$	"left over"
TC 8	"left over"	$v_{2,2}$	$v_{3,1}$
TC 9	"left over"	$v_{2,3}$	$v_{3,2}$

Table 8.4: Test cases with some left over values.

The left over values in the test cases can be replaced by arbitrary values the variable can take on. A better way is to cycle through the possible values when filling in the left overs so that the number of occurrences of each of the values is similar. Some of the test cases might become redundant (in the sense of pairwise testing).

8.3.2 Non existent orthogonal array

For the given factor k and level m, there might not exist an $\mathrm{OA}(k, m)$. For 6 variables $k = 6$, where each of the variables could take on 4 values $m = 4$, we would require an $\mathrm{OA}(6, 4)$. In the `TournamentPlanner` application, MOLS are used to obtain orthogonal arrays as in example 3.5. For the construction of $\mathrm{OA}(6, 4)$, we would require 4 MOLS $(k - 2)$ of order 4. However, there are only 3 such MOLS. Fortunately, we can utilize the following theorem.

Theorem 8.1. *Let $N(n)$ denote the maximum number of MOLS of order n. Then $\lim_{n\to\infty} N(n) = \infty$ [17].*

Basically, we should be able to increase the level of the OA (which is the order of the MOLS required for its construction) until we can obtain 4 MOLS of the given order. Indeed, it turns out that there are 4 MOLS of order 5 so $\mathrm{OA}(6, 5)$ can be constructed. In the test cases obtained from this OA, there will be some left over values, but they can be dealt with in the same way as in the previous subsection.

8.4 Construction of test cases in `TournamentPlanner`

In the `TournamentPlanner` application, the goal is to minimize the number of test cases while still testing all the possible variable pair interaction. The creation of the test cases are summarized in the steps bellow

1. Determine the factor k as the number of variables and the level m as the maximum number of values each variable can take on.

2. Construct $\mathrm{OA}(k, m)$ using MOLS. If $\mathrm{OA}(k, m)$ does not exist, keep incrementing m by 1 until an $\mathrm{OA}(k, m)$ can be obtained.

3. Transform the columns of the OA into test cases. Keep track of the test cases that do not contain any left over values and label them as *final test cases*. Label the other test cases containing left over values as *test case candidates*.

4. For the *test case candidates*, choose the values for the "left overs" cyclically.

5. If the group of *final test cases* does not cover all the variable pair interactions, choose a test case from the *test case candidates* that covers the largest number of interactions that have not yet been covered by the *final test cases* and label it as *final test case*. Repeat this until all the pair interactions are covered.

8.5 Comparison with other pairwise testing tools

There are many tools on-line for the construction of pairwise test cases. We compared the number of test cases constructed by 3 of these tools to the number of test cases constructed by the `TournamentPlanner`. The size of the inputs is denoted as an ordered list $(v_1, v_2, \ldots, v_n)$ where n is the number of variables and v_i is the number of values the ith variable can take on. The results are captured in the table bellow.

Input	`TournamentPlanner`	POT[2]	PG[3]	SQA[4]
$(2,3,2)$	6	7	6	6
$(2,4,2)$	9	8	8	8
$(3,5,5,3)$	25	25	25	26
$(5,5,6,4)$	38	35	36	35
$(3,3,4,7,5)$	39	41	36	36
$(7,8,9,7,9)$	81	81	89	93

Table 8.5: Number of pairwise test cases generated for different inputs.

It seems that each of the tools uses different heuristic to minimize the number of generated test cases. From the given inputs, it can not be determined which of the tools is the best (in minimizing the number of test cases) but the `TournamentPlanner` seems at least comparable to the other tools.

[2] Pairwise Online Tool at `https://pairwise.teremokgames.com/` developed by Victor Dementiev.

[3] Pairwise Generator at `https://slothman.dev/pairwise-generator/` developed by Pavel Kuptcov.

[4] SQA Pairwise at `https://sqamate.com/tools/pairwise` developed by Alexey Sotskov.

Chapter 9

Testing and benchmarking of the TournamentPlanner

The methods used to construct designs and tournaments described in the previous chapters are complex and require a lot of code. With one misplaced index or operator, the whole constructed structure is rendered invalid. It is therefore crucial to test these methods thoroughly.

In the first part of this chapter, we discuss the testing of the `TournamentPlanner` application. Then we show several runtime measurements and benchmarks of the tournament scheduling methods.

9.1 `TournamentPlannerTests`

As is the custom, the unit tests for the `TournamentPlanner` application are placed in a separate unit test project called `TournamentPlannerTests`. The namespace structure of the project copies the structure of the `TournamentPlanner`. The majority of the classes of the application have a corresponding test classes.

Unfortunately, most of the construction methods can not be easily split into simpler testable parts. The best way to efficiently test them is to validate the structures that have been constructed by them. This is done via *validators*.

9.1.1 Validators

Validators are static classes that validate of the constructed structures using the `NUnit`[1] framework. They are named after the structure, they are supposed to validate.

For example `BIBDValidator` validates the instances of the `BIBD` class. It checks the number of elements of the design, its block sizes and even all its element pair occurrences (which have to be equal to λ). Therefore if a `BIBD` instance passes through the validation without throwing any exception, it is reasonable to assume that it represents a valid BIBD.

There are dedicated validators for most of the design types and tournaments. There is also an additional validator for designs implementing the `IResolv-`

[1] `NUnit` is an open-source unit testing framework for `.NET`.

ableDesign interface that checks parallel classes of a given design. Using these validators, the unit tests for the construction methods and tournament planners become very simple.

```
[Test]
public void RecursiveKTSConstructionTest()
{
    var ns = Enumerable.Range(1, 20);
    var orders = ns.Select(n => (6 * n) + 3);
    foreach (var order in orders)
    {
        var alg = new RecursiveKTSConstruction(order);
        var kts = alg.ConstructDesign();

        BIBDValidator.Validate(kts);
        ResolvableDesignValidator.Validate(kts);
    }
}
```

Listing 9.1: Snippet of the the KTS construction test using validators.

The structures that do not have a dedicated validator can usually be checked using a combination of validators - like kts in listing ?? because a KTS is a resolvable BIBD$(v, 3, 1)$.

All of the validators are located in the TournamentPlannerTests.Model namespace. As the user interface is fairly simplistic, it was possible to test it manually.

9.2 Runtime measurements and benchmarks

The tournament scheduling methods combine several design construction algorithms. Some of the construction algorithms also use some external libraries. It is therefore difficult to determine their complexity exactly. Clearly, all the constructive scheduling methods (the ones that are not using constraint or integer programming) run in polynomial time. Unfortunately, I am not proficient enough in complexity analysis to determine the exact degree of the polynomial with certainty.

Nonetheless, it is still beneficial to measure the run times of the scheduling methods and to show how it increases with the growing number of players in the tournament. The experiments were performed on a modest machine with 16 GB of RAM and an Intel Core i5-8350U CPU.

All of the data depicted in the following figures were obtained by averaging 10 subsequent runs of the experiments.

9.2.1 Construction of tournaments with three players

There are 2 scheduling methods for tournaments with three player games. The more complicated one uses KTSs and yields completely balanced tournaments. The simpler one uses GDDs and only yields partially balanced tournaments. They are described in chapter 5. Figure 9.1 depicts the run times of the these methods up to 1005 player tournaments.

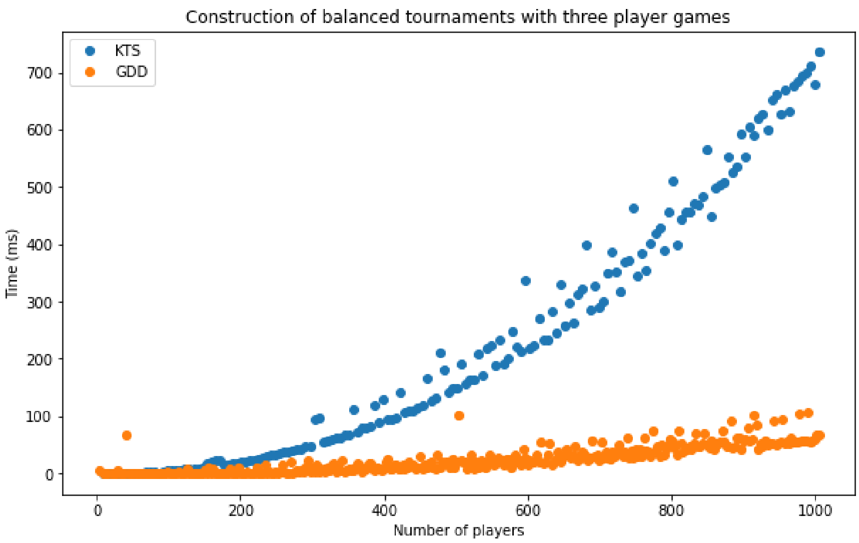


Figure 9.1: Runtime measurements of the 2 methods denoted as `KTS` and `GDD` according to the design they use to schedule tournaments.

The outliers in the figure tend to appear in the same places across all the experiments. The tournament scheduling methods use several design construction algorithms and these outliers actually correspond to a switch in the used construction algorithm. This is probably caused by the internal workings of the CLR[2] and just-in-time compilation.

9.2.2 Construction of whist tournaments

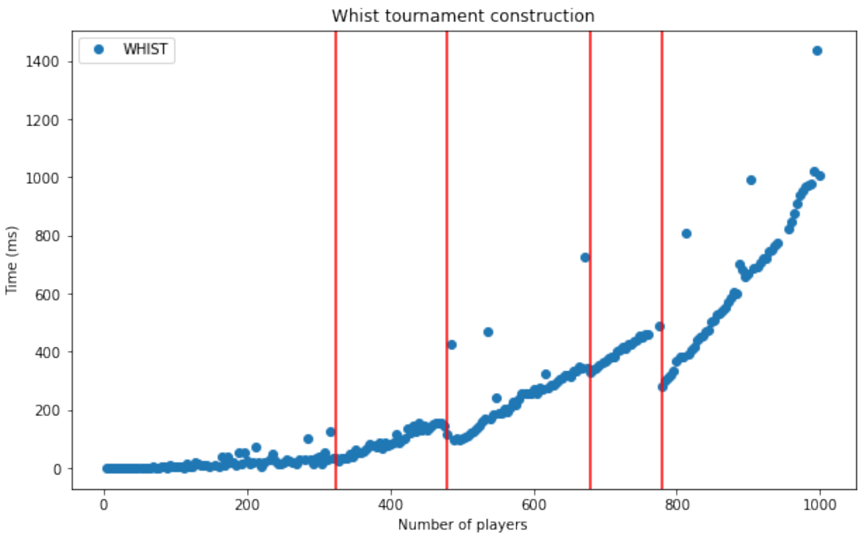


Figure 9.2: Run times of whist tournament construction up to 1000 players.

[2] The Common Language Runtime is the virtual machine component of the `.NET` framework.

There are noticeable jumps in the run times of the whist tournament construction when certain player number thresholds denoted by the vertical lines are reached. These actually correspond to the growth of the group sizes g of the auxiliary GDD used for the construction of larger whist tournaments as is described in 6.4.5.

9.2.3 Integer programming construction of BIBDs

There are 2 integer programming methods for the construction of BIBDs. The first is the naive quadratic formulation of the problem described in subsection 2.5.1. The second incremental block building method is implemented according to [7] and described in subsection 2.5.2.

First, we compare these methods on the problem of STS construction in figure 9.3. To reiterate, STS(v) exists if $v = 6n + 1$ or $v = 6n + 3$.

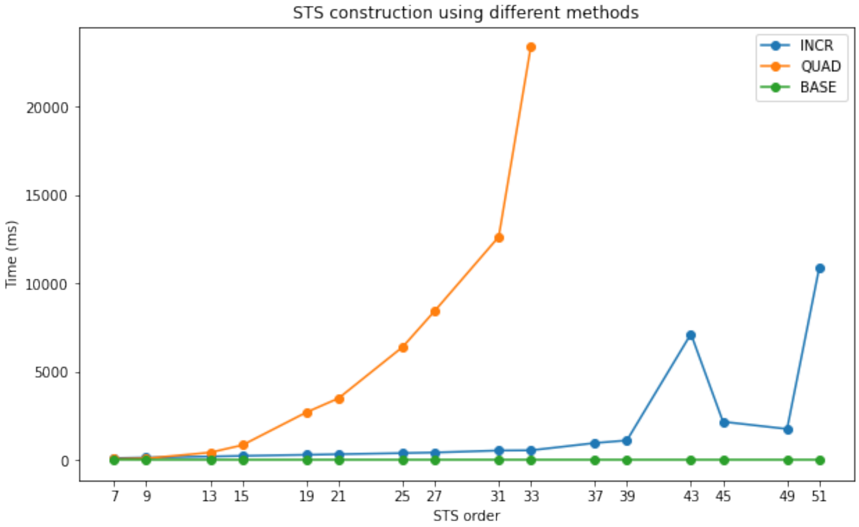


Figure 9.3: Comparison of the run times of methods on the STS construction. `INCR` is the incremental block building, `QUAD` is the quadratic programming method and `BASE` is the baseline constructive method.

The results are not surprising. The incremental method performs much better than the naive quadratic programming method. Both of these methods however pale in comparison with the constructive baseline method that uses Bose's and Skolem's STS construction[3]. This should showcase how much more efficient it is to use constructive methods instead of solvers.

However, when comparing the incremental and quadratic methods on the construction of *finite projective planes*[4], the result are unexpected. As can be

[3] Bose's STS construction method is implemented in the `BoseSTSConstruction` class and Skolem's in `SkolemSTSConstruction`. As these methods were not used for the scheduling of any tournaments, they were not addressed in this thesis.

[4] A finite projective plane of order n is a BIBD$(n^2 + n + 1, n + 1, 1)$. They are known to exists for any $n \geq 1$[1].

seen in figure 9.4, the naive quadratic programming method performs better. The incremental method was not even able to construct a finite projective plane of order 5 in reasonable time. It seems that the incremental method is not generally faster and only works faster for certain subsets of BIBDs.

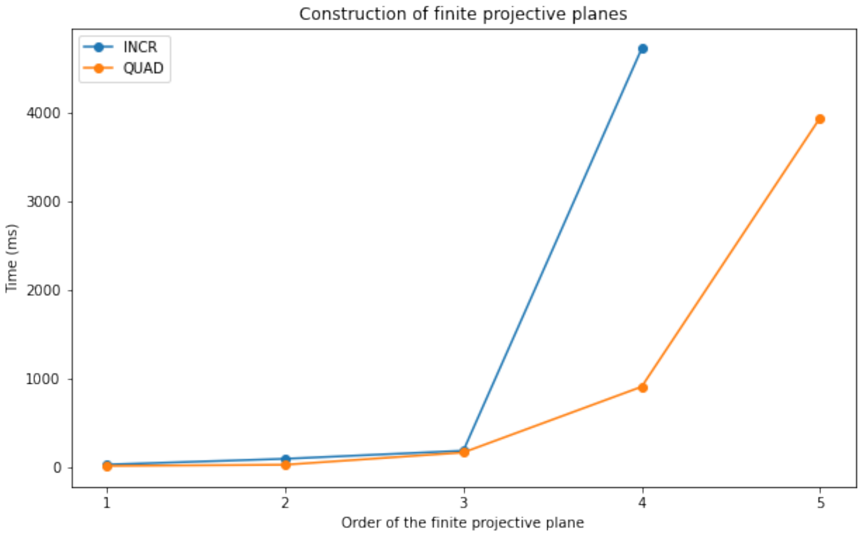


Figure 9.4: Comparison of integer programming construction methods on the construction of finite projective planes.

Chapter 10

Conclusion and future work

The goal of this thesis was to schedule tournaments with several opponents in one game. This has been achieved completely for balanced tournaments with three player games and for whist tournaments with $4n$ players. For the scheduling of these tournaments, multiple designs from the field of combinatorial design theory were required. The construction of such designs is difficult and they only allow for the scheduling of certain types of tournaments with a predetermined number of players or rounds. To this end, a more general constraint programming approach to the tournament scheduling has been proposed and implemented.

The practical part of this thesis - the `TournamentPlanner` application - implements all the described methods and provides a simple user interface for the construction of tournaments, several types of block designs and mutually orthogonal Latin squares.

The `TournamentPlanner` application could be extended by implementing the construction of further block designs and larger sets of MOLS which could then be used to schedule other types of balanced tournaments.

As has been shown in chapter 8, the application of combinatorial designs is not limited to tournament scheduling. Aside from software testing, designs can also be used in networking, mathematical chemistry or cryptography [26]. For this purpose, the part of the `TournamentPlanner` dedicated to the construction of combinatorial designs could be transformed into a `.NET` library similar to the combinatorial module of Sage Math in Python.

Bibliography

[1] I. Anderson. *Combinatorial Designs and Tournaments.* Clarendon Press; 1st edition, 1998.

[2] C.C. Lindner, C.A. Rodger. *Design Theory, 2nd Edition.* Chapman and Hall, 2017.

[3] F.D. Croce, D. Oliveri. *Scheduling the Italian Football League: an ILP-based approach.* Computers & Operations Research. 2006.

[4] D. Baumeister, T. Hogrebe. *Complexity of Scheduling and Predicting Round-Robin Tournaments.* Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems. 2021.

[5] C.C. Ribeiro. *Sports scheduling: Problems and applications.* International Transactions in Operational Research. 2012.

[6] D.K. Ray-Chaudhuri, R.M. Wilson. *Solution of Kirkman's schoolgirl problem.* Proceedings of Symposia in Pure Mathematics. 1971.

[7] D. Yokoya, T. Yamada. *A mathematical programming approach to the construction of BIBDs.* International Journal of Computer Mathematics. 2011.

[8] J. Ozanam. *Recreation mathematiques et physiques.* Berlin New York de Gruyter. 1725.

[9] T. Gaston. *Le Probléme de 36 Officiers.* Secrétariat de l'Association. 1901.

[10] R.C. Bose, S.S Shrikhande. *On the falsity of Euler's conjecture about the non-existence of two orthogonal Latin squares of order $4t + 2$.* Proceedings of the National Academy of Sciences USA. 1959.

[11] R.C. Bose, S.S Shrikhande. *Further results on the construction of mutually orthogonal Latin squares and the falsity of Euler's conjecture.* Canadian Journal of Mathematics. 1960.

[12] J.D.H. Smith. *An introduction to quasigroups and their representations.* Boca Raton, FL : Chapman & Hall/CRC. 2007.

[13] R.H.F. Denniston. *Double resolvability of some complete 3-designs.* Manuscripta mathematica. 1974.

[14] J.Lei. *On the large sets of Kirkman triple systems.* Discrete Mathematics. 2002.

[15] R. M. Wilson. *An existence theory for pairwise balanced designs: 1. Composition theorems and morphisms.* J. Combinatorial Th., 13. 1972.

[16] W.P. Courtney. *English whist and English whist players.* London : Richard Bentley and Son. 1894.

[17] C.J. Colbourn, J.H. Dinitz (eds.). *Handbook of Combinatorial Designs, Second Edition.* Chapman & Hall/CRC. 2006.

[18] D. Guichard. *An Introduction to Combinatorics and Graph Theory.* Self-published. 2017.

[19] Petr Olšák. *Úvod do algebry, zejména lineární.* Nakladatelství ČVUT. 2013.

[20] E.H. Moore. *Tactical Memoranda I-III.* American Journal of Mathematics. 1896.

[21] I. Anderson, N.J. Finizio. *A generalization of a construction of E. H. Moore.* Bulletin of the Institute of Combinatorics and its Applications. 1992.

[22] N.S. Mendelsohn. *Latin squares orthogonal to their transposes.* Journal of Combinatorial Theory. 1971.

[23] R. Black. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional.* John Wiley & Sons. 2016.

[24] D.R. Kuhn, D.R. Wallace and A.M. Gallo. *Software fault interactions and implications for software testing.* IEEE Transactions on Software Engineering. 2004.

[25] D.R. Stinson. *Combinatorial Designs: Constructions and Analysis.* New York: Springer. 2003.

[26] D. Raghavarao, L.V. Padgett. *Block Designs: Analysis, Combinatorics and Applications.* World Scientific. 2005.

Appendix A

List of abbreviations

BIBD	Balanced incomplete block design
(v, k, λ)	BIBD with parameters v, k, λ
RBIBD	Resolvable BIBD
KTS	Kirkman triple system
STS	Steiner triple system
PBD	Pairwise balanced design
GDD	Group divisible design
RGDD	Resolvable group divisible design
MOLS	Mutually orthogonal Latin squares
OA	Orthogonal array
TD	Transversal design
RTD	Resolvable transversal design
GF(q)	Finite (Galois) field with q elements
GCD	Greatest common divisor
SAMDRR	Spouse-avoiding mixed doubles round robin

Appendix B

Contents of included DVD

- `DP.pdf` - a pdf version of the thesis
- `DP.zip` - archive with the LaTeX sources
- `tournamentplanner-master.zip` - archive containing the source code of the `TournamentPlanner` application
- `documentation.pdf` - documentation of the source code generated using doxygen