Software tools for combinatorial algorithms

Joost Winne

Promotor: Prof. Dr. Veerle Fack

Academiejaar 2006-2007

Vakgroep Toegepaste Wiskunde
en Informatica

Proefschrift voorgelegd aan de Faculteit Wetenschappen tot het behalen van de graad van Doctor in de Wetenschappen: Informatica

Voor Sofie, onze kleine deugniet Bram en zijn toekomstige zusje.

Preface

The title of this work – Software tools for combinatorial algorithms – was chosen because we provide a software library in which combinatorial generation algorithms can be implemented effectively. The choice of the programming language is Java. Why Java? True, Java is slower than C. However, when solving a certain generation problem once, needing more running time than you would need with an equivalent C program, is not really an issue. Java gives us the opportunity to develop algorithms in a clean object-oriented way, is portable and also provides a graphical framework (Java Swing).

The developed software tools form the basis of this thesis. However, case studies, which led to some new results in design and coding theory, also form a substantial part of this work. Chapter 1, the introduction, states some combinatorial definitions related to this work. It also summarizes some software principles, which will be used throughout the text.

Following the introduction, Chapter 2 introduces the reader to the generation and enumeration of integer matrices meeting some constraints, by use of an exhaustive backtracking algorithm. We describe the functionality and implementation of the developed backtrack framework, and discuss the usage and implementation of a visualization tool. The visualization tool visualizes a backtracking algorithm in an interactive way without any extra programming effort. So the visualization tool is built on top of the backtrack framework. We give a brief overview of the other packages in Appendix C. There is no place in this thesis to describe all developed software components. Chapter 2 ends with a detailed example of how the backtrack and visualizer framework can be used. This example, together with the *javadoc* [1] documentation, should suffice to be able to use the package. We intend to use this backtrack framework for educational purposes in the near future.

Chapter 3 handles equivalence testing of (rectangular) matrices with non-negative small integer entries, and describes a small library which uses the popular graph isomorphism testing software *nauty* (written in C) in Java through JNI (Java Native Interface) [2]. The main purpose of this implementation is the ability to call *nauty* from Java programs in the same way that you call it from C programs. This implementation also provides some extra classes which makes using *nauty* easier. In particular, testing integer matrices for equivalence is easy. The usage of this library is illustrated by a small example.

Many combinatorial generation algorithms can be solved by splitting the problem into

[1] Javadoc is a tool for generating library documentation in HTML from comments in Java source code.

[2] Java Native Interface (JNI) allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly.

pieces which are run on a cluster of machines. Since our department has a cluster of 24 dual processors available, we wrote farming software, based on Java RMI (Remote Method Invocation), which makes it easier to split up the search and collect the results. This farming software is described in Appendix A. This appendix serves as a tutorial to use the farming package. This package is also used for educational purposes.

Chapter 4 describes the generation of 2-(v, k, λ) designs with non-trivial automorphisms following the local approach method. This chapter forms the basis for the three subsequent chapters.

Chapter 5 presents the classification of 2-(31,15,7) and 2-(35,17,8) Hadamard designs and 2-(36,15,6) Menon designs with automorphisms of odd prime order. The main interest in these particular designs is their relation to Hadamard matrices of order 32 and 36, and their relation to self-dual codes. We found 21879 Hadamard matrices of order 32 and 24920 Hadamard matrices of order 36, arising from the classified designs. Remarkably, all constructed Hadamard matrices of order 36 are Hadamard equivalent to a regular Hadamard matrix. To check the correctness of the results, parts of the classification results were double-checked by a second independent implementation.

The local approach method is not limited to designs. Chapter 6 uses this local approach method in a search for the existence of the partial geometry $pg(6, 6, 4)$ with an automorphism of order 3 with 7 fixed points. Unfortunately, it turns out that no such partial geometry exists. The existence of $pg(6, 6, 4)$ in general remains open.

Chapter 7 presents the enumeration of the doubles of the projective plane of order 4. Crucial in this enumeration was the (computer-assisted) proof of the unique reducibility of any reducible 2-(21,5,2) design. Again, most of the computer results are obtained by two different approaches and implementations.

Chapter 8 presents the results on small weight codewords in the codes arising from Desarguesian projective planes of prime order. This chapter discusses how a computer approach helped to characterize the small weight codewords in the codes arising from Desarguesian projective planes of prime order. We improve the results of K. Chouinard on codewords of small weight in the codes arising from $PG(2, p)$, p prime. Using a particular basis for this code, described by Moorhouse, we characterize all the codewords of weight up to $2p + (p - 1)/2$ if $p \geq 19$. Furthermore, we present some related additional results.

A summary, in Dutch, can be found in Appendix B. For a brief overview of other packages of the CAAGT library and a link to the software and its documentation, we refer to Appendix C.

Since my contribution to the article "Projective two-weight codes with small parameters and their corresponding graphs" [5, 16] was limited to the graph classification results, I do not discuss these results.

Acknowledgements - Dankwoord

First of all I thank Svetlana Topalova and Iliya Bouyukliev for working together and for their unlimited hospitality during my stay in Bulgaria. The remainder of this section is in Dutch...

U bevindt zich in de meest gelezen sectie. Het is ook meestal de laatst geschreven sectie. Hier zal ik dus mijn uiterste best doen.

Vooreerst dank ik het FWO - Vlaanderen om mij een beurs te geven. Mijn promotor Veerle Fack was een grote hulp voor uiteenlopende zaken. Dit gaat van het aanbrengen van enkele mogelijke onderzoeksonderwerpen en contacten leggen met (buitenlandse) onderzoekers, tot het corrigeren van Engelstalige teksten. Ook als persoon is Veerle bijzonder aangenaam.

Ik dank Kris Coolsaet voor vruchtbare en soms vurige discussies over software ontwikkeling. Jan Degraer dank ik eveneens voor de minder vurige, maar even interessante bijeenkomsten. Ik dank Leo Storme voor de aangename samenwerking en zijn geduld om mij elementaire zaken duidelijk uit te leggen.

Natuurlijk zijn er vele anderen die ik op een of andere wijze dankbaar ben. Mijn ouders, die mij veel levenswijsheden bijbrachten en mij alle mogelijkheden hebben gegeven. Mijn schoonouders, die ik er stilaan van verdenk om verbouwen als een hobby te zien. Mijn broer Johan, die mij nog net voor is met zijn doctoraat. Mijn broer Jelle, die straks ook Gent onveilig komt maken. Cimo, Mirka en kleine lieve Alica, straks zullen we jullie missen. Mijn Sofietje, al bijna 9 jaar mijn lief klein meisje, en de mama van super-Brammetje. Ons Brammetje, een lief eigenwijs manneke. Zijn toekomstige speelkameraadje, dat groeit zonder dat we het beseffen.

Aan alle anderen, ook bedankt.

Contents

1 Introduction

In Sections 1.1 and 1.2 we give some definitions regarding designs and graphs, respectively. In Section 1.3 we briefly explain some software principles and tools which will be vital to this work.

1.1 Designs

Definition 1.1.1 (t-design) *Let $V = \{P_i\}_{i=1}^{v}$ be a finite set of points, and $\mathcal{B} = \{B_j\}_{j=1}^{b}$ a finite collection of k-element subsets of V, called blocks. $D = (V, \mathcal{B})$ is a design with parameters t-(v, k, λ) if any t-subset of V is contained in exactly λ blocks of $\mathcal{B}$.*

Definition 1.1.2 (BIBD) *A balanced incomplete block design (BIBD) is a pair (V,$\mathcal{B}$) where V is a v-set (points) and $\mathcal{B}$ is a collection of b k-subsets of V (blocks) such that each point is contained in exactly r blocks and each pair of points is contained in exactly λ blocks. The numbers v,b,r,k,λ are the parameters of the BIBD.*

BIBD's are *t-designs* with $t = 2$. Trivial necessary conditions for the existence of a BIBD (v, b, r, k, λ) are $vr = bk$ and $r(k-1) = \lambda(v-1)$. In short we write 2-(v, k, λ). We will often use the term *design* instead of BIBD.

Definition 1.1.3 (Incidence matrix) *The incidence matrix of a design is a (0,1) matrix with v rows and b columns, where the element of the i-th row and j-th column is 1 if $P_i \in B_j$ ($i = 1, 2, ..., v$; $j = 1, 2, ..., b$) and 0 otherwise. A design is completely determined by its incidence matrix.*

Definition 1.1.4 (Isomorphic BIBD) *Isomorphism of two designs $D_1 = (V_1, \mathcal{B}_1)$ and $D_2 = (V_2, \mathcal{B}_2)$ is a bijection between their point sets V_1 and V_2 and their block collections $\mathcal{B}_1$ and $\mathcal{B}_2$, such that the point-block incidence is preserved.*

In terms of the incidence matrices, two designs are isomorphic if their incidence matrices are equivalent, i.e. if the incidence matrix of the second design can be obtained from the incidence matrix of the first design by a permutation of the rows and columns. We will often write a permutation in cycle notation. An element in the i-th position of a cycle is replaced by the $i + 1$-th element and the last element of the cycle is replaced by the first,

e.g. the row permutation (1) (2 3) (4 5 6) (7) fixes rows 1 and 7, swaps rows 2 and 3, row 5 is replaced by row 4, row 6 is replaced by row 5 and row 4 is replaced by row 6. We will sometimes omit the fixed elements from the notation, e.g. (2 3) (4 5 6) in the example.

Definition 1.1.5 (Automorphism of a design) *An automorphism of a design is an isomorphism of the design to itself, i.e. a permutation of the points that preserves the block collection. The set of all automorphisms of a design forms a group called its full automorphism group. Each subgroup of this group is an automorphism group of the design.*

For the basic concepts and notations concerning combinatorial designs refer for instance to [3], [10], [13], [45].

1.2 Graphs

Since we will often use graph equivalence testing to solve the problem of design equivalence testing, we also give some definitions related to graphs.

Definition 1.2.1 (Graph) *An undirected graph or graph is an ordered pair $G = (V, E)$ with a vertex set V and a set E of unordered pairs of distinct vertices, called edges.*

The *order* of a graph G is the number of its vertices, the *size* of G is the number of its edges. The vertices u and v of an edge $e = \{u, v\}$ are called *adjacent*.

Definition 1.2.2 (Adjacency matrix) *The adjacency matrix of a graph is a symmetric square (0,1) matrix of order the order of the graph, where the element of the i-th row and j-th column is 1 if vertices i and j are adjacent, and 0 otherwise. A graph is completely determined by its adjacency matrix.*

Definition 1.2.3 (Isomorphic graphs) *Isomorphism of two graphs is a bijection between their vertex sets such that edges are mapped to edges and non-edges are mapped to non-edges.*

Definition 1.2.4 (Automorphism of a graph) *An automorphism of a graph is an isomorphism of the graph to itself. The set of all automorphisms of a graph forms a group called its full automorphism group. Each subgroup of this group is an automorphism group of the graph.*

1.3 Software principles

In Section 1.3.1 we give the key concepts of Object-Oriented (OO) Programming, related to Java. This survey is by no means complete. The software of this thesis will be described using the Unified Modeling Language (UML). Our use of UML is described in Section 1.3.2. Section 1.3.3 explains some concepts which will be used in Chapter 2.

1.3.1 Object-Oriented Programming

In Java an object has *members* and *methods*, corresponding to the *attributes* and the *behavior* of objects in the "real world". A *class* can be seen as the blueprint of a certain type. An *object* is an instance of a class. As an example, a *Person* class might represent a human by its name, date of birth, ... The objects *joe* and *kelly* might be instances of this class. Besides classes, Java also has primitive types such as *int* (integer numbers), *boolean* (true or false) and *char* (a literal).

A member is specified by the type (a class or a primitive type) and an identifying name. A method has two parts: the header and the body. The *method header* specifies the return type, the method name and an optional list of arguments. The *method body* contains the implementation of the method. An *abstract method* contains only the method header. A Java *interface* contains only abstract methods. An interface does not have members. *Classes* contain both methods (header + body) and members. An *abstract class* is a class which may also contain abstract methods.

Interfaces and abstract classes can not be instantiated. The creation of an object occurs by calling the *constructor* of its class. A constructor is a special method which has the same name as its class. Its purpose is to initialize the members. Two important concepts are *class inheritance* and *interface implementation*, whereby classes and interfaces inherit certain characteristics and behaviors from another (abstract) class or interface. The *data hiding* concept is provided in Java through the concept of *private*, *protected* and *public* methods and members. Private methods and members from a class are accessible by all instances of that class. Protected methods and members from a class are accessible by all instances of that class or any subclass of that class. Public methods and members enforce no restrictions. In interfaces all methods are public by definition. If class S *extends* class A, then S is a *subclass* of class A. Class A is the *superclass* of class S. An extension of a class inherits all members and methods from its superclass, but only the non-private members and methods are accessible. A subclass may even *override* a method by providing a different implementation. If class S *implements* interface I, then S provides the implementation for all methods of I. If interface J *extends* interface I, then J also contains all methods from I. Interfaces, abstract classes and classes form a *class hierarchy*. In Java, every class is a subclass of exactly one (abstract) class. An (abstract) class may *implement* an unlimited amount of interfaces. Implementing an interface means providing the method bodies for all interface methods. In Java, class inheritance starts from the top class *java.lang.Object*. All classes are subclasses of *Object* or some subclass of *Object*, or some subclass of a subclass of *Object*, ... In Java, *class A extends B* indicates that class A is a subclass of class B, whereas *class A implements I* indicates that class A implements interface I.

Besides object methods, there can also be *static* methods in a class. Static methods are associated with the class (*Person* in our example), whereas object methods are associated with instances of the class. Static methods of a class can be called without creating any instances of the class. A static method resembles a method in a non-object-oriented language.

1.3.2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML[1]) is the most-used specification of the Object Management Group (OMG[2]). UML is the way the world models not only application structure, behavior, and architecture, but also business process and data structure [3].

Sometimes we will not show all methods, and certainly omit or limit the specification of data members. An interface or class diagram contains three parts:

On top, we either have the interface name or the class name. Interfaces are in *slanted* font, (abstract) classes in normal font.
The second part specifies the types (and names) of the data members. This part is always empty for interfaces and we also limit its use for classes. Sometimes we omit the attribute names, so each specification looks like: Type [memberName]
The third and final part is the most important: it contains the most relevant method specifications. We use a Java-like specification, but the *void* keyword is omitted when the method has no return type: [*static*] [*abstract*] [returnType] method([Type id,]*)

Interface methods are abstract by definition. The *static* keyword indicates that the method is static. The *abstract* keyword is used in abstract classes, which may contain both implemented methods (default) and abstract methods. A closed lock icon is shown for private methods (or members), a half open lock for protected methods and an open lock for public methods. An example class diagram is given in Figure 1.1, in which all members are private. Method *update()* is protected and has no return type. All other methods are public. The constructor has the same name as the class and has no return type.

Figure 1.2 shows an example of inheritance. The *GenericGenerator* class is a subclass of the *Processor* class, so *GenericGenerator* extends *Processor*. Furthermore, the *GenericGenerator* class implements the *NonRecursiveGenerator* interface. Both inheritance relations are shown in the same way.

Besides class inheritance and interface implementation, UML also expresses associations and aggregations. An *association* expresses a *usage* relation through an arrow which goes from the *user* towards the *usee*. Sometimes *multiplicity indicators* are given on each side of the association to indicate how many objects use how many other objects. We use a star (*) to indicate zero or more objects. No indicator stands for *exactly* one object. An *aggregation*

[1] *http://www.uml.org*

[2] *http://www.omg.org*

[3] In this text typesetting UML diagrams was done with MetaUML: *http://metauml.sourceforge.net*. MetaUML is a GNU GPL MetaPost library for typesetting UML diagrams, using a human-friendly textual notation.

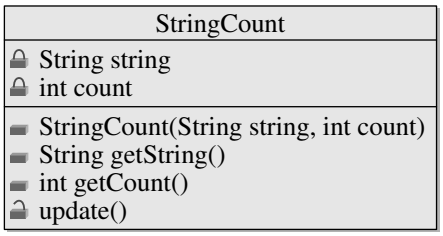


Figure 1.1: Example class diagram of a class named *StringCount* which contains a character *String* and an *int* as private data members. It contains one public constructor which initializes the two members. It contains two public methods to retrieve the members. It contains one protected *update()* method.

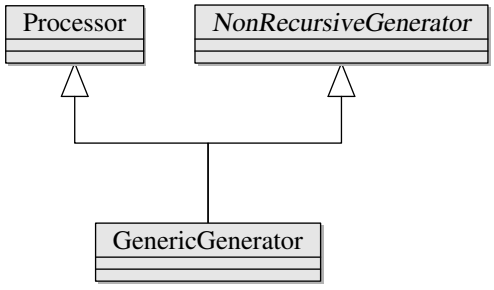


Figure 1.2: The *GenericGenerator* class extends the *Processor* class and implements the *NonRecursiveGenerator* interface. Note the slanted font of the interface.

is somewhat stronger than a usage relation: it is an association in which one object *contains* other objects. This is shown by drawing a diamond at the *owner* side. Figure 1.3 shows various examples of all described relations.

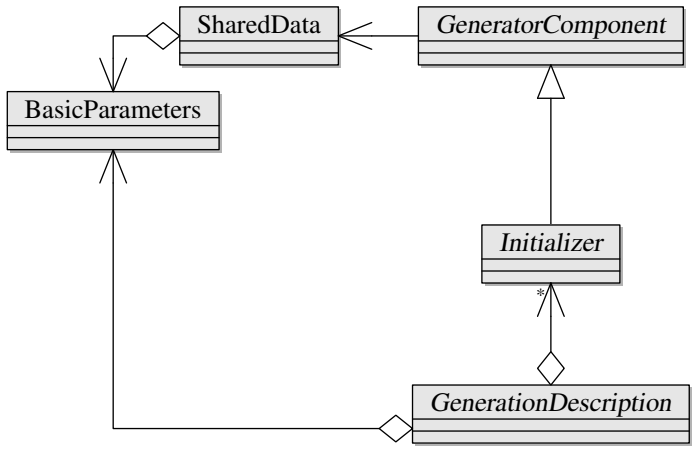


Figure 1.3: Example of interface implementation, association and aggregation relations. *Initializer* implements *GeneratorComponent*. *GenerationDescription* contains zero or more *Initializer*'s and exactly one *BasicParameters* object. *SharedData* contains one *BasicParameters* object. *GeneratorComponent* uses *SharedData*.

1.3.3 Related concepts

The MVC paradigm

The MVC (Model View Controller) paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. The model holds the *logic* of the application. A view presents the model's logic is some way, without changing it. Views are often visual components. A controller changes the model. Visual components often serve as both a controller and a view of the model.

Java Tiger

We will often give Java-like pseudocode for algorithms. Therefore we explain some basic concepts. There is a convention that the first letter of a class or interface name is a capital letter. Object and variable names start with a non-capital letter. Static constants are written in capital letters. In all user chosen names, every new word starts with a capital letter, e.g. *numberOfRows* might be the variable name of an integer. Besides library classes, Java also has built-in primitive types such as **int** (4-byte integer values), **boolean** (either true or false), **real** (for floating numbers), **char** (string characters).

In Java, all classes are organized into a package structure, which can be seen as a directory structure. The *fully quantified name* of a class is the package name together with

the class name, e.g. *be.ugent.caagt.backtrack.ValueMatrix* is the fully quantified name of the *ValueMatrix* class which resides in the *be.ugent.caagt.backtrack* package.

Java 1.5 (Java Tiger) introduced typed collections. E.g. *List*⟨*Checker*⟩ is a list in which all elements are of type *Checker*. *List* is an interface of the rich Java collections framework. In previous versions of Java the type could not be specified, it was always *Object*.

2 Software for backtracking algorithms

Generation algorithms are a superset of *enumeration algorithms*. *Enumeration algorithms* compute the number of different structures of a particular type. *Generation algorithms* explicitly construct all those different structures.

Note that the ambiguous notion of *different* must be made explicit. For designs and graphs, different often means *non-isomorphic*. Combinatorial objects such as graphs and designs can be represented by their adjacency matrix and incidence matrix, respectively. The exhaustive generation of such combinatorial objects is typically done with an exhaustive backtracking algorithm which generates all integer matrices meeting constraints derived from the objects' definition and parameters.

In this chapter we will describe our general integer matrix generation framework, which is implemented in the *Java* programming language. This *backtrack* framework is suited for a backtrack generation of rectangular integer matrices with small non-negative integral entries. Examples of combinatorial objects suitable for this framework are designs, graphs, association schemes and codes. The *backtrack* framework makes an abstraction of all components which are involved in a backtrack search.

We introduce the key concepts regarding backtracking in Section 2.1. The *backtrack* package, which forms the basis of the framework, is described in Section 2.2. Section 2.3 describes the *gentool* package: a *GUI* application which visualizes a backtracking algorithm without any extra programming effort. Finally, Section 2.4 contains a detailed example, which, together with the API, should suffice to use this package.

2.1 Exhaustive backtracking

We first introduce some notions regarding the backtracking algorithm. We refer to a certain position in the matrix by the term *entry*. The *domain* of an entry is a list of integer *values* which are possible for that entry. The binding of a certain entry to a value from the domain list is called *instantiating*. The domain also defines the order in which values are instantiated for a certain entry. The *path* determines the order in which the entries are instantiated by defining the next entry to instantiate after a successful instantiation.

The algorithm starts with an *empty* matrix, i.e. a matrix in which all entries are initially

undefined or *uninstantiated*. The backtracking algorithm (recursively) instantiates all entries of the matrix in turn with subsequent domain values. As said, the order in which this is done is determined by a *path* component. Whenever a partially instantiated matrix does not fulfill some integer matrix *constraints*, the algorithm uninstantiates the entry, *backtracks* to the previously instantiated entry and tries the next possible domain value for that entry. As long as all constraints are still satisfied, the algorithm continues to instantiate entries with values. We use the notion of a *checker* for the component which incrementally "checks" a certain constraint. When the matrix is totally instantiated and all constraints are satisfied, a solution is found. Solutions are handled by *leaf* components.

Recursive backtracking pseudocode is shown in Algorithm 2.1. The non-recursive version is given in Algorithm 2.2. The main difference is that we need to store the stack of instantiated entries in the non-recursive version. We assume the usual *push()* method to add an element to the stack, *top()* to get the top element and *pop()* to remove the top element. One execution of the *repeat until* loop of Algorithm 2.2 can be seen as a single "step". We keep track of the current matrix *entry*, defined at line 2. A step is either a forward step (lines 8–15) or a backward step (lines 17–22). A forward step binds the current entry to the next domain value and checks all constraints. If all checks passed, then we either move to the next entry or handle a solution. A backward step uninstantiates the current entry and backtracks to the most recent instantiated entry, if possible.

Algorithm 2.1 Recursive backtracking pseudocode

```
function generate()
 1  if allMatrixEntriesInstantiated() then
 2      handleSolution()
 3  else
 4      entry ← selectUndefinedMatrixEntry()
 5      domainList ← getDomainListForEntry(entry)
 6      for all value in domainList do
 7          matrix[entry] ← value
 8          if allConstraintsSatisfied() then
 9              generate()
10          matrix[entry] ← UNDEFINED
```

2.2 The backtrack package

The main *engine* of this framework is a so-called *generator* which implements a standard backtracking search algorithm in a recursive or non-recursive way. The generator works with various *generator components*. *GeneratorComponent* is the top interface of the five important interfaces *Checker*, *Domain*, *Initializer*, *LeafNode* and *Path*. Figure 2.1 expresses their relation in UML, showing only the interface names. We will sometimes use the terms *generator component*, *checker*, *domain*, *initializer*, *leaf* and *path* instead of the interface names. The top interface *GeneratorComponent* has two methods. One method is used to share data with other generator components, and another method resets the component to its initial state.

Algorithm 2.2 Non-recursive backtracking pseudocode

```
function generate()
 1  entryStack ← new Stack()
 2  entry ← selectUndefinedMatrixEntry()
 3  entryStack.push(entry)
 4  resetDomainValuesFor(entry)
 5  done ← false
 6  repeat
 7      if hasNextDomainValueFor(entry) then
 8          matrix[entry] ← getNextDomainValueFor(entry)
 9          if allConstraintsSatisfied() then
10              if allMatrixEntriesInstantiated() then
11                  handleSolution()
12              else
13                  entry ← selectUndefinedMatrixEntry()
14                  entryStack.push(entry)
15                  resetDomainValuesFor(entry)
16      else
17          matrix[entry] ← UNDEFINED
18          entryStack.pop()
19          if entryStack.isEmpty() then
20              done ← true
21          else
22              entry ← entryStack.top()
23  until done
```

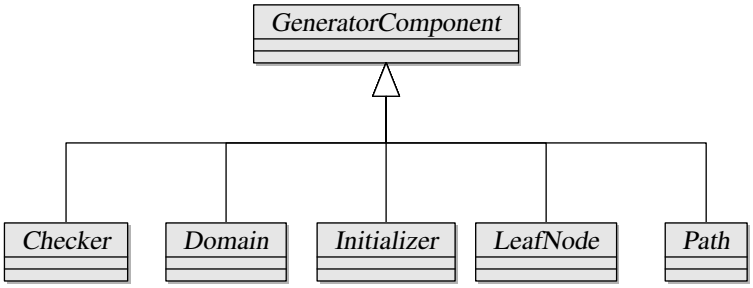


Figure 2.1: The top interface *GeneratorComponent* and its five subinterfaces.

Each *Checker* checks wether a certain constraint is still satisfied after instantiating a certain entry of the matrix, so all checkers are consulted in the *allConstraintsSatisfied()* method at line 8 of Algorithm 2.1 or line 9 of Algorithm 2.2. Hence constraint checkers determine whether a partially instantiated matrix could possibly be extended to a full matrix which satisfies the problem parameters.

Domain defines an ordered list of possible values for each matrix entry. It also provides a way to (repeatedly) traverse that list for each entry. Refer to lines 5 and 6 of Algorithm 2.1, or to lines 4, 7, 8 and 15 of Algorithm 2.2.

The *initialize()* method of each *Initializer* is called just before generation starts. An initializer is typically used to initialize the matrix in some way or to create and initialize data which needs to be shared between various generator components. This is not shown yet in the non-detailed algorithms.

The *ship()* method of each *LeafNode* is called whenever a solution is reached, i.e. when all matrix entries are instantiated. This will be done in the *handleSolution()* method, which is called at line 2 of Algorithm 2.1, and at line 11 of Algorithm 2.2. These leaves determine what will be done with the fully instantiated matrices that are encountered during the search (write to a file in some format, count number of structures, ...).

Path determines the search path to be taken during the generation process, i.e. it selects the next undefined matrix entry to instantiate. Hence it is used in method *selectUndefinedMatrixEntry()* at line 4 of Algorithm 2.1 or at lines 2 and 13 of Algorithm 2.2.

The description of a generation is packed into an implementation of the *GenerationDescription* interface, whose default implementation *DefaultDescription* is shown in Figure 2.2. A subclass of *DefaultDescription* is a description of a particular generation. Note that this description does not provide a list of leaves, since the way solutions are handled has nothing to do with the generation per se. Instead, leaves are provided to the generator. We will refer to a particular implementation of the *GenerationDescription* interface by the term *description*.

2.2.1 Sharing data

A mechanism to share data between different generator components is needed. As an example, consider the matrix which is being generated. This matrix needs to be shared between all checkers and leaves. A solution would be to use long arguments lists in the constructor, but this is a bad choice since these may change during the implementation cycle. A better solution is to write all components as a bean in which the necessary data must be set through various methods. The disadvantage of the latter option is that a lot of properties may need to be set, making the generation description class rather long. Another solution is to centralize data into a single class. This way each generator component retrieves data from this class, or creates data into this class. The standard generator components are shared through this last mechanism.

First of all, we defined a *BasicParameters* class which holds the dimensions of the matrix, the lower bound and upper bound value of all possible matrix entries, and if the matrix is symmetric or not. Note that this class, shown in Figure 2.3, forms a part of the description.

The *SharedData* class is the place where all data is centralized. This class, shown in Figure 2.4, is actually not much more than a *HashMap* which maps *String* keys to objects.

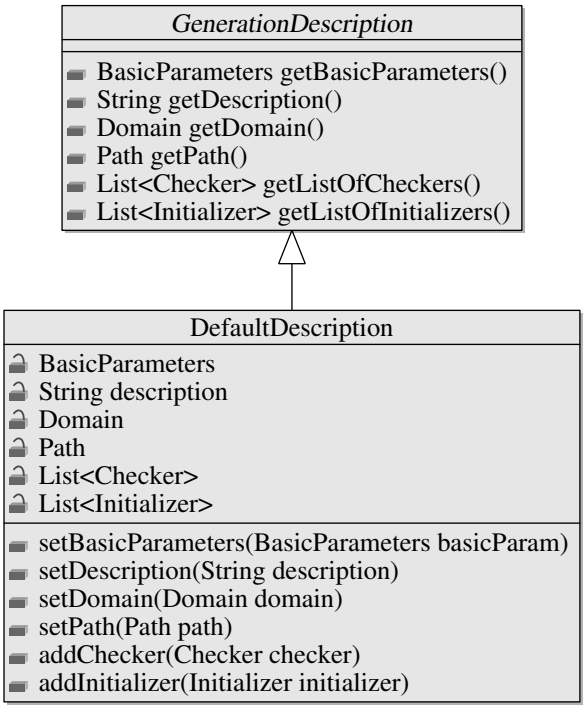


Figure 2.2: The *GenerationDescription* interface and its default implementation *DefaultDescription*. A subclass of *DefaultDescription* is a description of a particular generation.

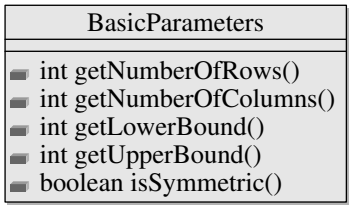


Figure 2.3: The *BasicParameters* class contains the basic problem parameters. It forms a part of the description.

The generator will create this shared data object. Before generation starts, it provides this object to all generator components through the *init(SharedData shared)* method of the *GeneratorComponent* interface, which is shown in Figure 2.6. The top interface *GeneratorComponent* has two methods:

- *init(SharedData shared)* should initialize the component: it retrieves and/or creates all internal data structures, which could be shared with other objects through *SharedData*.
- *reset()* should reset the component to its initial state so that it can be reused in a new generation process. This is useful since recreating all objects might not be very efficient.

This way, each generator component has the possibility to request the object which is mapped to some key, or to map a new key to an object. In the easiest case, the fully quantified name of the class (or interface) could be used as a key. This is not possible when multiple instances of the same class are needed. In that case, one should think carefully about the key scheme. A factory which creates *shared data* is defined in the interface *SharedDataFactory* (Figure 2.5), whose single method *createItem()* creates an object for a certain key. Note that this *createItem()* method receives the *SharedData* object, which may help to create new data.

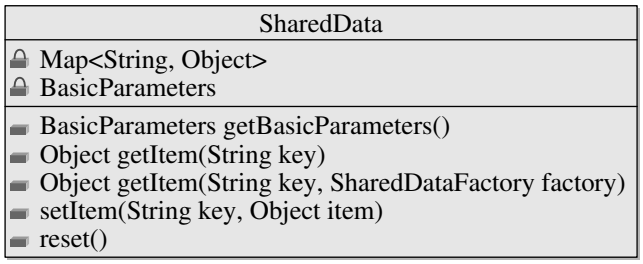


Figure 2.4: All data shared between all generator components is centralized in the *SharedData* class.

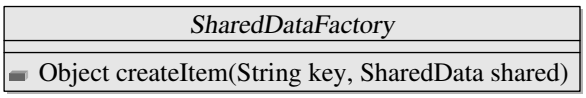


Figure 2.5: *SharedDataFactory* implementations create a shared item for a certain key.

So each shared item stored in a *SharedData* object is identified by a certain key. Generator components will use the following strategy:

- Explicitly store data through *setItem(String key, Object item)*.
- Retrieve data through *Object getItem(String key)*.

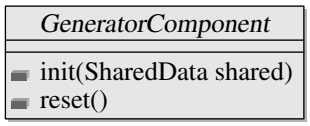


Figure 2.6: The top interface *GeneratorComponent* contains only two methods, one to initialize the component by use of the shared data object and one to reset the generation process.

- Retrieve (and store) data through
  Object getItem(String key, SharedDataFactory factory), in which *factory* creates and stores the data item if there is no item present for the given *key*.

To help in creating these objects, *SharedData* gets a registered *BasicParameters* object at construction time. The generator retrieves the *BasicParameters* from the description, and creates the *SharedData* object.

2.2.2 Generation

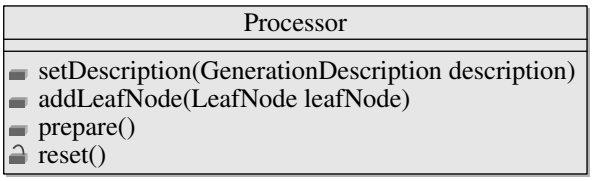


Figure 2.7: Generators are subclasses of this *Processor* class.

Generators can be implemented by subclassing the *Processor* class, which is shown in Figure 2.7. Leaves and a description are provided to the processor through methods *addLeafNode(LeafNode leafNode)* and *setDescription(GenerationDescription description)*, respectively. The *prepare()* method of *Processor* is called prior to generation and performs the following:

- It calls the *init(SharedData shared)* methods in the following order: initializers (in the order of the list), path, checkers (in the order of the list), domain, leaves (again in the order of the list). You could run into trouble with this order. Maybe your path needs something your domain creates. As a solution to this problem, you could write an initializer which creates this shared data.
- It calls the *initialize()* method of each initializer, again in the order of the list.
- It selects the first undefined matrix entry to instantiate by consulting the path and initializes the domain of that entry.

The *reset()* method of *Processor* calls the *reset()* method of all generator components. This method should only be called when you want to restart the generation.

A general non-recursive generator implementation is provided in class *GenericGenerator*, which implements the interface *NonRecursiveGenerator* and extends class *Processor*, as shown in Figure 2.8. Such a non-recursive version is needed to be able to write an application which visualizes the backtracking process step by step, as will be explained in Section 2.3.

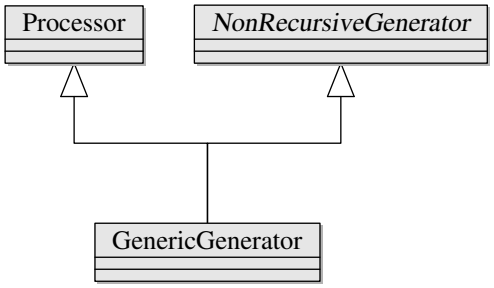


Figure 2.8: The *GenericGenerator* class extends *Processor* and is a generic non-recursive generator implementation.

The *GenericGenerator* class uses a *GeneratorStack* to store the stack of instantiated entries. It creates and puts a *GeneratorStack* instance into *SharedData* with its fully quantified name "be.ugent.caagt.backtrack.GeneratorStack". You can retrieve this object from the shared data object and use its information in other generator components. The *GeneratorStack* class is shown in Figure 2.9.

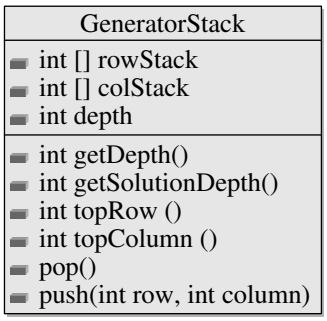


Figure 2.9: The *GeneratorStack* class stores the stack of instantiated entries.

2.2.3 Domain

Each entry of the rectangular integer matrix always has a list of values which are still possible for that entry, hence the notion of a domain. The *Domain* interface (Figure 2.10) hides the details of different kinds of domains (fixed range, dynamic adjustable, order in which domain values for an entry are tried, ...). Its most important methods are:

- *reset(int row, int column)* resets the domain which corresponds to entry (row, column), so it can be traversed by the following two methods.

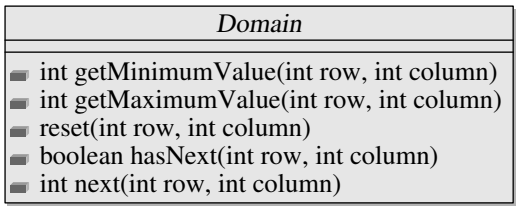


Figure 2.10: The *Domain* interface defines the way to repeatedly traverse the values of each entry.

- *boolean hasNext(int row, int column)* determines if the domain which corresponds to entry (row, column) has another value available.
- *int next(int row, int column)* gets and removes the next domain value to instantiate entry (row, column) with.

Standard implementations are available in the *be.ugent.caagt.backtrack.domain* package.

2.2.4 Integer matrix

Figure 2.11 shows the *IntMatrixView* and *IntMatrix* interfaces which reside in the *be.ugent.caagt.im* package. It also shows the standard implementation *ValueMatrix* which exposes the underlying *int [][]* array. For reasons of efficiency, most implementations will work directly on the two-dimensional array of *ValueMatrix* instead of going through the interface calls of *IntMatrix*. The integer matrix class (usually *ValueMatrix*) is stored in shared data with key *be.ugent.caagt.im.IntMatrix*. A -1 value in the matrix means *undefined*.

2.2.5 Generation path

The *Path* interface (Figure 2.12) determines the search path to be taken during the generation process. The *boolean prepare(int depth)* method of the path determines the next matrix entry to instantiate. This method is called by the generator. The *depth* parameter holds the current depth in the search tree, i.e. the number of entries the backtracking process instantiated. The method returns *false* when there is no next entry, otherwise the generator retrieves the next entry through the two methods *int getRow()* and *int getColumn()*.

In the easiest case, the path is *fixed* prior to generation. An example is a path which fills the matrix row by row, starting from the first column entry upto the last column entry in each row. A more advanced path is a *dynamic* path which is altered during generation. As an example, consider a generation strategy in which we use some forward checking method after each instantiation of an entry. Suppose this forward checking method reduces the domain list of one or more uninstantiated entries, i.e. it forbids some domain values for a certain uninstantiated entry. This way we can make the path choose the next entry to instantiate as the one with the least number of possible domain values left. In order to implement this forward checking method, we provided an abstract *DomainMatrix* class, which is basically a wrapper around two arrays:

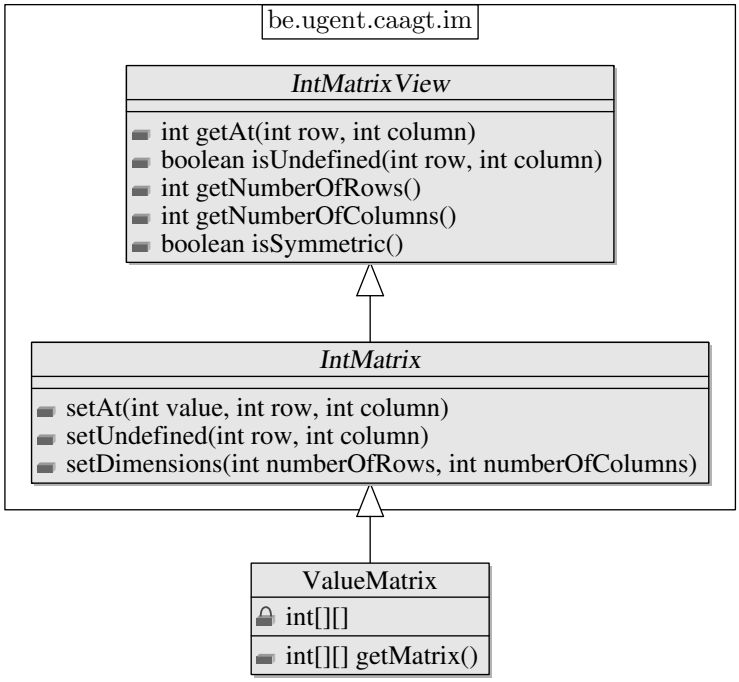


Figure 2.11: An implementation of the *IntMatrix* interface holds the matrix to generate. We usually use *ValueMatrix* which is a wrapper around a two-dimensional integer array. The *IntMatrixView* and *IntMatrix* interfaces reside in the *be.ugent.caagt.im* package.

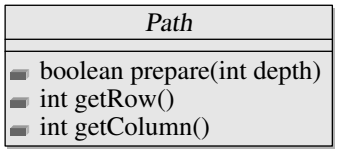


Figure 2.12: The *Path* interface determines the order in which entries are instantiated.

- *int [][][][] forbidden*: The value of *forbidden[row][column][v]* is an integer which counts the number of constraints that forbid value v at entry (*row*, *column*) and hence must be zero for this domain value to be allowed.
- *int [][][] possible*: The value of *possible[row][column]* holds the current size of the domain of entry (*row*, *column*), i.e. the number of remaining values to try.

The visualizer, which will be described in Section 2.3, visualizes the *DomainMatrix* if one can be retrieved from *SharedData* through key *be.ugent.caagt.backtrack.DomainMatrix*. If a *DomainMatrix* class can be found through this key, then the standard *Domain* implementations of the *be.ugent.caagt.backtrack.domain* package will also use this matrix (and thus skip forbidden values). To implement a forward checking scheme, the path could determine the next entry based on the *DomainMatrix*.

2.2.6 Checkers

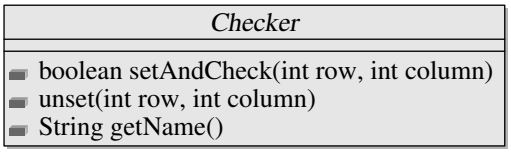


Figure 2.13: The *Checker* interface incrementally checks constraints, thereby updating internal information.

Figure 2.13 shows the *Checker* interface. A *Checker* implementation is used to check whether a certain constraint is still satisfied after instantiating a certain entry. After instantiating matrix entry (row, column) with a certain domain value, the generator calls *boolean setAndCheck(int row, int column)* for all the checkers. This allows each checker to determine whether the associated constraint is still valid for the current partially instantiated matrix and if so, it updates internal information regarding that instantiation of entry (row, column). In the process of backtracking, the generator calls *unset(row, column)* for every instantiated entry (in reverse order) for which the corresponding *setAndCheck(row, column)* returned **true**. This enables the checker to undo all changes made by *setAndCheck*. Note that *unset* is not called when *setAndCheck* returned **false**, as is clear from the two methods of Algorithm 2.3, which implement a "setAndCheck" and "unset" of all checkers. As an example, consider a degree constraint which needs to check that there are at most x ones in row r. An obvious implementation of the two methods for this example is shown in Algorithm 2.4.

Algorithm 2.3 The "setAndCheck" and "unset" paradigm the generator applies for all checkers. Only if all checks pass in *setAndCheck()*, the generator can continue towards the next entry, otherwise all internal updates must be undone. In the process of backtracking, all updates are undone in *unset()*.

```
function boolean setAndCheck(int row, int column)
  for i from 0 upto checkerList.size() −1 do
      if not checkerList.get(i).setAndCheck(row, column) then
          for j from i − 1 downto 0 do
              checkerList.get(j).unset(row, column)
          return false
  return true

function unset(int row, int column)
  for j from checkerList.size() −1 downto 0 do
      checkerList.get(j).unset(row, column)
```

Algorithm 2.4 An example checker implementation which checks whether there are at most x ones in row r. Each time a "1" is instantiated in row r, x is decremented.

```
function boolean setAndCheck(int row, int column)
  if (row = r) ∧ (matrix[row][column] = 1) then
      if x = 0 then
          return false
      x ← x − 1
  return true

function unset(int row, int column)
  if (row = r) ∧ (matrix[row][column] = 1) then
      x ← x + 1
```

Algorithm 2.5 Solutions are handled by leaves.

```
function handleSolution()
  for all leafNode in leafNodeList do
      leafNode.ship()
```

2.2.7 Initializers

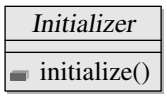


Figure 2.14: The *initialize()* method of each *Initializer* is called just before generation starts.

The *Initializer* interface is shown in Figure 2.14. Besides the two methods of the *GeneratorComponent* interface, it contains an *initialize()* method. The *initialize()* method of each *Initializer* is called just before generation starts. An initializer is typically used to initialize the matrix in some way. Another valuable use is to create and initialize data which needs to be shared between various generator components. This data could be put into the shared data class, or maybe you prefer to provide the initializer itself, whose data is needed, to the component.

2.2.8 Leaves

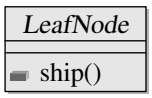


Figure 2.15: Each *LeafNode* handles a solution in some way. Writing them to a file in some format is an obvious example.

The *ship()* method of each *LeafNode* will be called whenever a leaf node of the search tree is reached, i.e. for every fully instantiated matrix. The *LeafNode* interface is shown in Figure 2.15. An example of a leaf is one which writes all matrices to a file in some format. Note that the leaf can retrieve the generated matrix from shared data. The implementation of the *handleSolution()* method is shown in Algorithm 2.5.

2.2.9 Metadata

A lot of interesting *metadata* can be collected during the search, this data is stored in a *MetaData* object. The following metadata can be collected:

- The number of recursive calls.

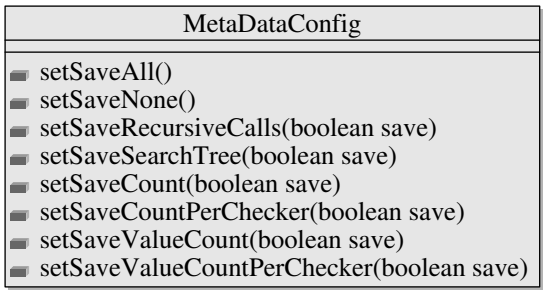


Figure 2.16: The *MetaDataConfig* class defines the metadata to be stored during generation.

- The number of nodes at each *depth* of the search tree, i.e. the number of times the generator instantiated *depth* entries.
- For each entry: the number of value bindings and the number of times all constraint checks passed.
- For each entry and each checker: the number of checks and the number of times that check passed.
- For each entry and each value: the number of value bindings and the number of times all checks passed for the value.
- For each entry, each checker and each value: the number of checks with the value and the number of times that check passed for that value.

Because storing metadata can be expensive in terms of memory and running time, a *MetaDataConfig* object (Figure 2.16) defines what data is to be stored or not. The class *MetaData* is of particular importance for the visualizer application, which will visualize all this metadata.

2.2.10 Summary

The backtrack package is summarized in Figure 2.17. Basic parameters, a domain, a path and a list of initializers and checkers define the generation description. Leaves are added to a generator implementation. The shown *GenericGenerator* class is an extension of the *Processor* class and an implementation of the *NonRecursiveGenerator* interface. The *GenericGenerator* class assumes you have written a generator component (e.g., an initializer) which puts some implementation class of the *IntMatrix* interface into the *SharedData* class with key *be.ugent.caagt.im.IntMatrix*. This implementation class of *IntMatrix* will be used as the matrix which is being generated. This generic generator implementation puts the following components into the *SharedData* class with the given key:

- *be.ugent.caagt.backtrack.Domain* : The provided *Domain* implementation class, retrieved from the *GenerationDescription* instance.

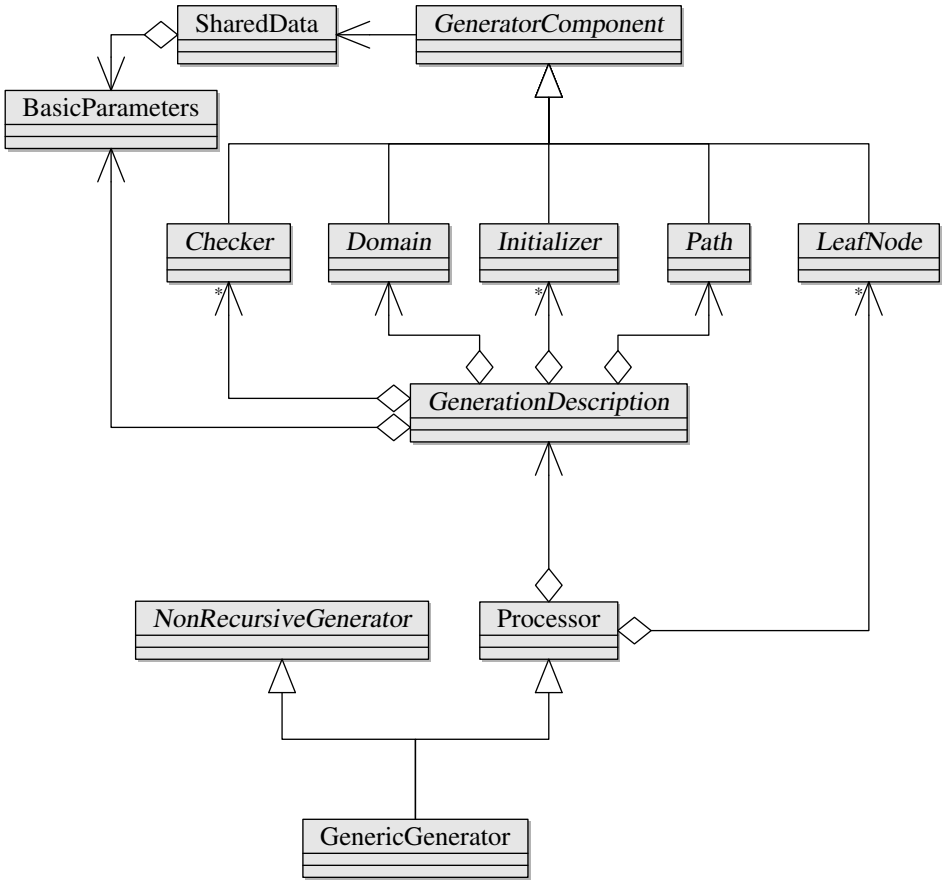


Figure 2.17: Summary of the *backtrack* package.

- *be.ugent.caagt.backtrack.DomainMatrix* : An instance of the *DomainMatrix* class, if used by the provided *Domain* implementation class.
- *be.ugent.caagt.backtrack.GeneratorStack* : An instance of the *GeneratorStack* class. This way you can use the data of this component in some other generator components.
- *be.ugent.caagt.backtrack.MetaDataConfig* : An instance of the *MetaDataConfig* class. By default *GenericGenerator* assumes all metadata is collected. You can change this by providing a *MetaDataConfig* object to *GenericGenerator* through its *setMetaDataConfig()* method.
- *be.ugent.caagt.backtrack.MetaData* : An instance of the *MetaData* class. The *GenericGenerator* class creates and puts this object into *SharedData* based on *MetaDataConfig*.
- *be.ugent.caagt.backtrack.Path* : The provided *Path* implementation class, retrieved from the *GenerationDescription* instance.

Algorithm 2.6 gives a detailed object oriented recursive version of the backtracking algorithm. The *generate()* method (lines 23–28) is the starting point of the algorithm. It creates a shared data object, calls the *init()* method of each generator component and the *initialize()* method of each initializer. Recursive generation starts at root *depth* 0. In this recursive generation (lines 11–22) the path selects the next undefined matrix entry to instantiate. Note the way in which the domain of a certain entry is reset, such that repeated traversal is possible. Note the use of the "setAndCheck" and "unset" in the *while* loop.

Algorithm 2.7 gives a detailed object oriented non-recursive version of the backtracking algorithm. In the non-recursive version, we store the visited entries on a stack (defined at line 2) before continuing towards the next matrix entry such that when backtracking, we can restore the current matrix entry. Variables *row*, *column* hold the current entry.

The interface calls of a generic generator implementation involves some overhead. To solve this, we can write a more dedicated generator for a specific problem. As an experiment, we wrote a small class which, based on a generation description, creates a generator class which optimizes the generation code:

- Loop unrolling: the "setAndCheck" and "unset" loop is unrolled.
- Hardcoding of a fixed path and a fixed domain into the generator, if possible.
- Using references to the implementation class of each generator component instead of using interface references. This eliminates the overhead of interface calls.
- Work directly on a two-dimensional array.

Experiments on some of our problems shows a speed gain of at most 3 percent compared with the generic generator implementation. When trying to complete a classification, a difference of 3 percent in running time is not going to make the difference. More importantly, a generic generator implementation helps in creating an error-free classification.

Algorithm 2.6 Detailed recursive backtracking pseudocode.

```
function boolean setAndCheck(int row, int column)
 1 for i from 0 upto checkerList.size() −1 do
 2     if ! checkerList.get(i).setAndCheck(row, column) then
 3         for j from i − 1 downto 0 do
 4             checkerList.get(j).unset(row, column)
 5         return false
 6 return true

function unset(int row, int column)
 7 for j from checkerList.size() −1 downto 0 do
 8     checkerList.get(j).unset(row, column)

function handleSolution()
 9 for all leafNode in leafNodeList do
10     leafNode.ship()

function generate(int depth)
11 if depth = numberOfEntriesToInstantiate then
12     handleSolution()
13 else if path.prepare(depth) then
14     row ← path.getRow()
15     column ← path.getColumn()
16     domain.reset(row, column)
17     while domain.hasNext(row, column) do
18         matrix[row][column] ← domain.next(row, column)
19         if setAndCheck(row, column) then
20             generate(depth +1)
21             unset(row, column)
22         matrix[row][column] ← UNDEFINED

function generate()
23 shared ← newSharedData()
24 for all generatorComponent do
25     generatorComponent.init(shared)
26 for all initializer do
27     initializer.initialize()
28 generate(0)
```

Algorithm 2.7 Detailed non-recursive backtracking pseudocode. The *setAndCheck()*, *unset()* and *handleSolution()* implementations from Algorithm 2.6 are not repeated.

```
function generate()
 1  shared ← newSharedData()
 2  entryStack ← newStack()
 3  for all generatorComponent do
 4      generatorComponent.init(shared)
 5  for all initializer do
 6      initializer.initialize()
 7  path.prepare(entryStack.size())
 8  row ← path.getRow()
 9  column ← path.getColumn()
10  entryStack.push(row, column)
11  domain.reset(row, column)
12  done ← false
13  repeat
14      if domain.hasNext(row, column) then
15          matrix[row][column] ← domain.next(row, column)
16          if setAndCheck(row, column) then
17              if entryStack.size() = numberOfEntriesToInstantiate then
18                  handleSolution()
19                  unset(row, column)
20              else if path.prepare(entryStack.size()) then
21                  row ← path.getRow()
22                  column ← path.getColumn()
23                  entryStack.push(row, column)
24                  domain.reset(row, column)
25              else
26                  unset(row, column)
27      else
28          matrix[row][column] ← UNDEFINED
29          entryStack.pop()
30          if entryStack.isEmpty() then
31              done ← true
32          else
33              row ← entryStack.topRow()
34              column ← entryStack.topColumn()
35              unset(row, column)
36  until done
```

2.3 The visualization tool

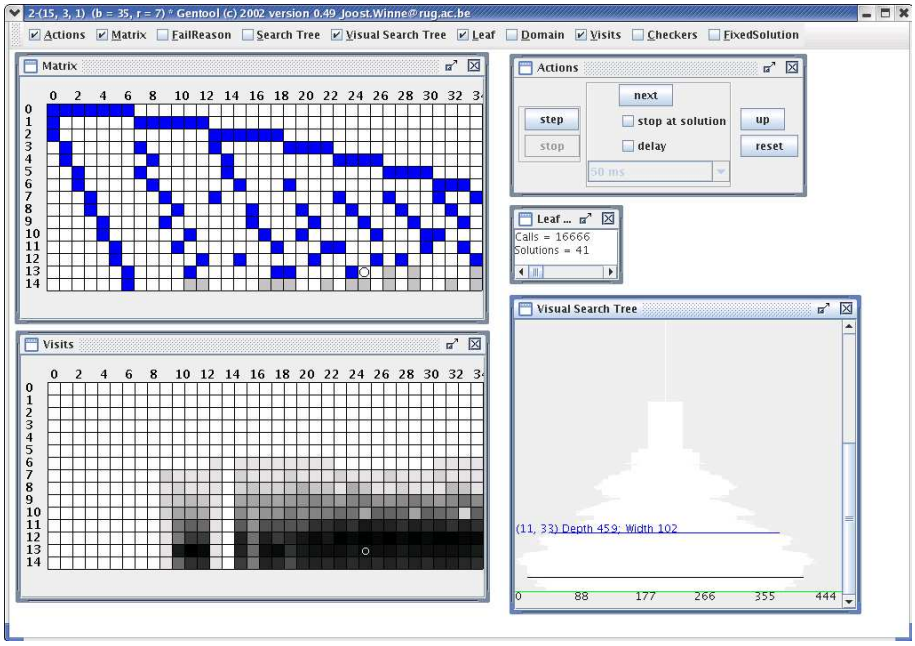


Figure 2.18: The visualizer window is organized into hidable internal frames.

The *be.ugent.caagt.gui.gentool* package is written on top of the *be.ugent.caagt.backtrack* package. It lets you visualize your backtracking algorithm in a step by step interactive fashion. Some benefits of such a visualization tool are:

- **debugging**: Locate where your algorithm does something wrong.
- **optimization**: Observe that your algorithm possibly does not prune where it could.
- **bottleneck detection**: Find out where your algorithm spends most of its time.
- **comparing**: Compare constraint checkers step by step.
- **constraints**: By looking at the effect of each single checker, you might come up with new constraints or ways to implement them better.

It is easy to start the visualizer. First a *GenericGenerator* object is created based on a *GenerationDescription* implementation class. Then this *GenericGenerator* object is given to the constructor of the *Gentool* class of this *be.ugent.caagt.gui.gentool* package.

```
GenerationDescription description = new MyDescription();
GenericGenerator generator = new GenericGenerator(description);
Gentool gentool = new Gentool(generator);
```

When the *Gentool* application is up and running, the backtrack generation has not started yet. It sits idle waiting for your actions.

2.3.1 Functionality

The application is organized into hidable internal frames which are accessible through the top toolbar, as shown in Figure 2.18. With the leftmost button of the toolbar, you can (re)define the colors to use for some value. On most frames you can use the mouse scroller (or + and -) to zoom in and out. In matrices, the left mouse button usually selects a certain entry and the right mouse button shows a popup menu with a number of visual options (show relation colors, show relation numbers, auto fit window, ...). The most important interactive *actions* are:

- **step**: Performs a single step. A step does one execution of the body of the *repeat until* loop of Algorithm 2.7 on page 26, i.e. a forward step or a backward step.
- **next**: Repeatedly performs steps with a specified delay in *milliseconds*. This action can be interrupted in a number of ways, as described below.
- **stop**: Stops the *next* action.
- **up**: Leaves the current branch, i.e. uninstantiates the current entry and backtracks to the previous instantiated entry. Consider carefully wether this action behaves properly with your implementation. Maybe your checker doesn't work if not all domain values are tried.
- **reset**: Resets the generation so it can be restarted once again.

The *next* action might be interrupted when:

1. *stop* is pressed,
2. a solution is reached,
3. a number of steps is performed,
4. a number of recursive calls is made,
5. some checker fails,
6. (un)instantiating some entry with (from) some value,
7. some checker fails when instantiating some entry with some value,
8. some depth in the search tree is reached,
9. some user written *InterruptCondition* is violated.

2.3.2 Overview of frames

The **Actions** frame (Figure 2.19) holds all described interactive actions, together with interrupt conditions 1 to 4 from the above list.

The **Matrix** frame (Figure 2.20) visualizes the integer matrix which is being generated. When you click on an entry, a breakpoint dialog such as the one depicted in Figure 2.21

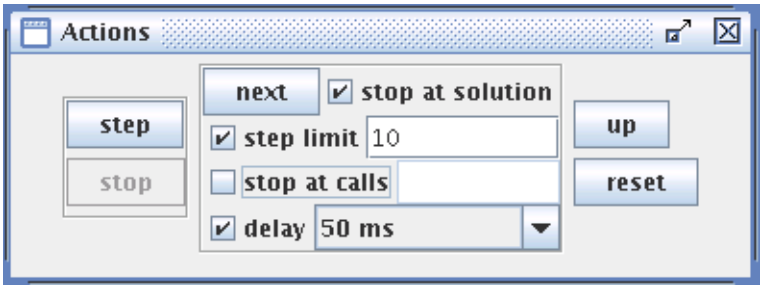


Figure 2.19: The **Actions** frame contains all interactive actions.

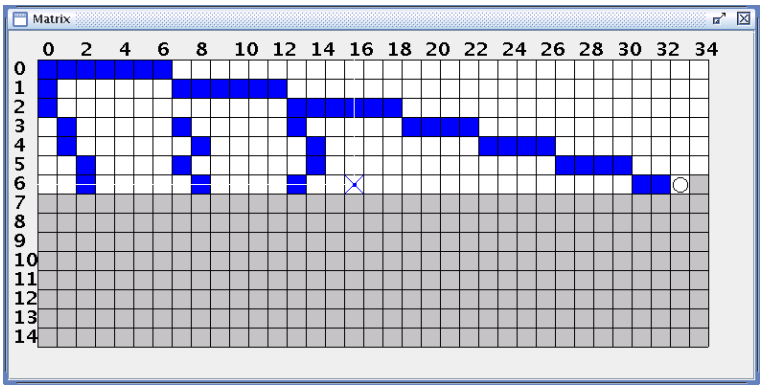


Figure 2.20: The **Matrix** frame visualizes the integer matrix. Note the blue cross which indicates that a breakpoint is set at that entry. Also note the circle which marks the current entry of the algorithm.

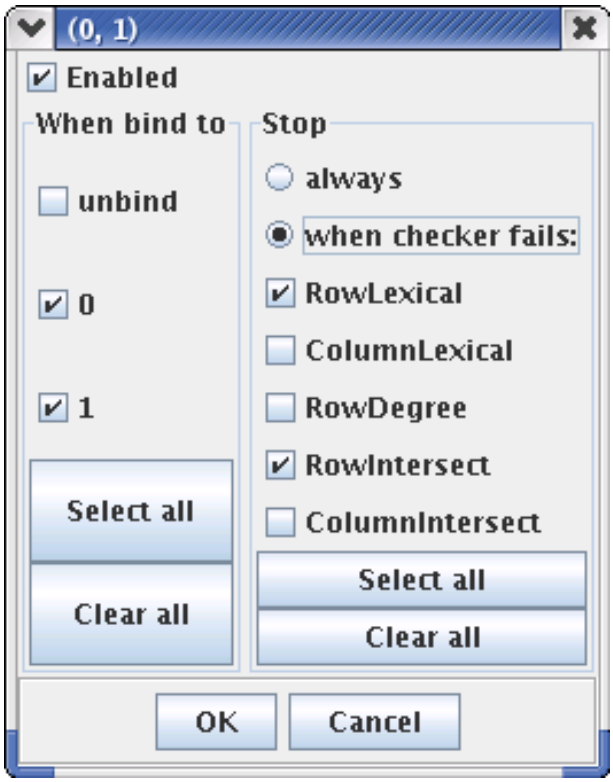


Figure 2.21: The breakpoint dialog lets you specify when to stop the process at a certain entry. It pops up when you click on an entry of the matrix frame.

Figure 2.22: The **Fail Reason** frame shows which checker failed.

pops up, in which you have a lot of options to interrupt the process at that entry. The next entry to instantiate is marked with a circle. Breakpoints are marked with blue crosses.

The **Fail Reason** frame (Figure 2.22) shows which checker failed. It shows the *toString()* representation of the failed checker.

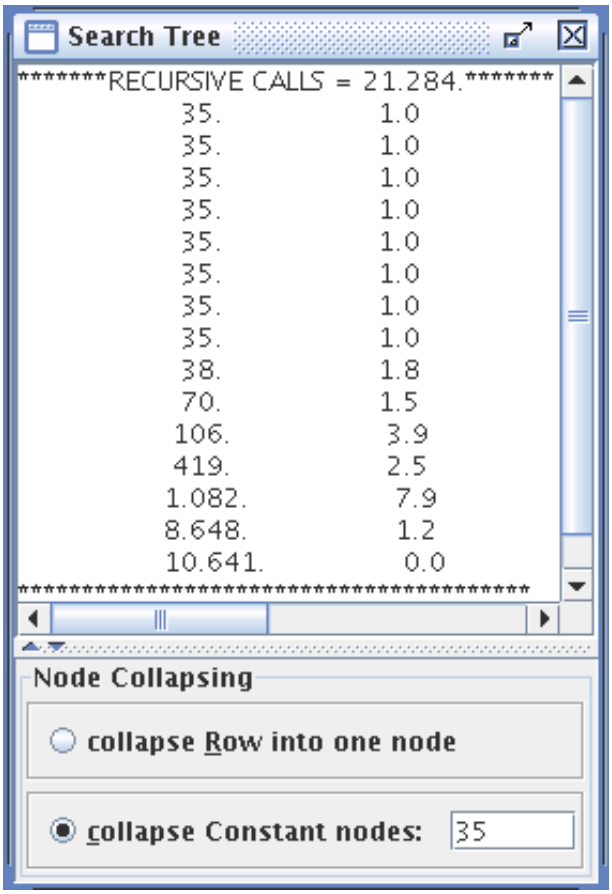


Figure 2.23: The **Search Tree** frame shows the search tree in textual format. You can narrow the view by considering nodes at subsequent depths as one node ("collapse").

The **Search Tree** frame (Figure 2.23) shows the search tree in a textual format, i.e. for each depth the number of bindings tried at that depth, together with the average branching factor. You can specify that nodes at subsequent depths are considered as one node, in order to have a condensed view of the search tree. We call this "collapsing".

The **Visual Search Tree** frame (Figure 2.24) draws the search tree in a logarithmic or linear scale. Clicking on a certain depth shows the number of bindings at that depth. Using the popup menu, you can also specify that the process should be interrupted when that depth is reached. The current depth in the search tree is indicated by a thick black line.

The **Leaf Output** frame (Figure 2.25) shows the number of solutions and recursive

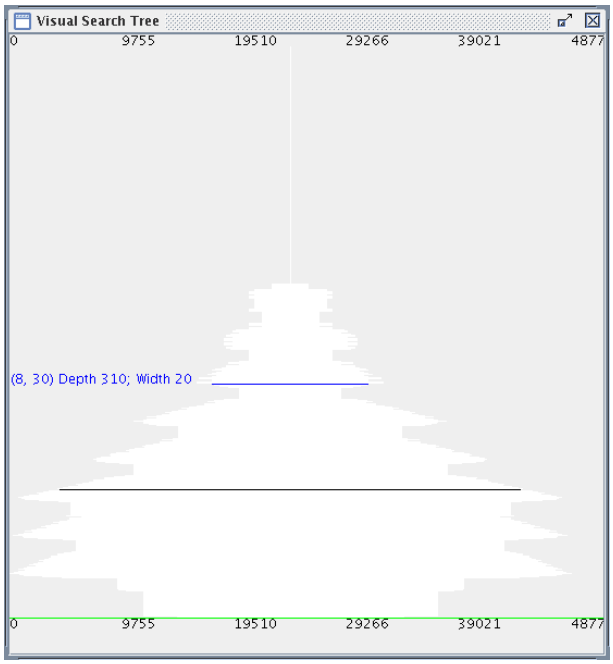


Figure 2.24: The **Visual Search Tree** frame draws the search tree. A logarithmic tree shape is shown. Clicking on a depth shows the tree width. The thick black line shows the current depth in the search tree.

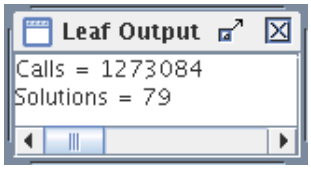


Figure 2.25: The **Leaf Output** frame shows the number of solutions and recursive calls made so far.

calls made so far. The number of recursive calls is the same as the number of forward steps performed in the non-recursive backtracking code.

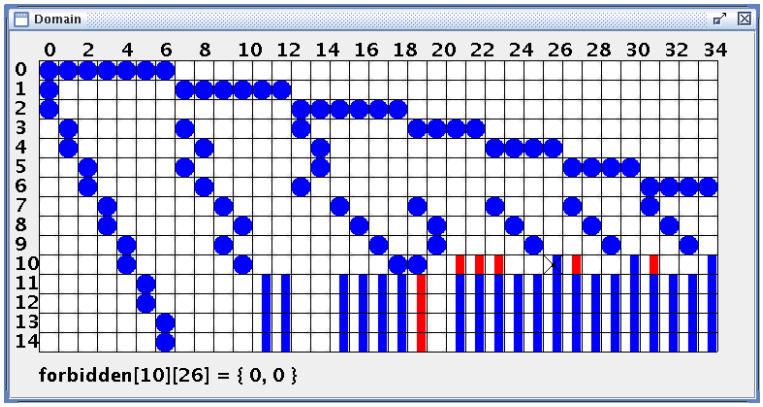


Figure 2.26: The **Domain** frame visualizes the *DomainMatrix* class. Circles are shown for instantiated entries. For uninstantiated entries the colors of the remaining possible values are shown. The contents of *forbidden[10][26]* is selected in this screenshot.

The **Domain** frame (Figure 2.26) visualizes the *DomainMatrix* class (if used) of the *backtrack* package, i.e. it shows the remaining possible domain values for each entry. Colored circles are shown for instantiated entries. For uninstantiated entries the colors of the remaining possible values are shown. Also, clicking on entry (row, column) shows the contents of *forbidden[row][column]* or *possible[row][column]*, depending on your selection, which you can set using the popup menu.

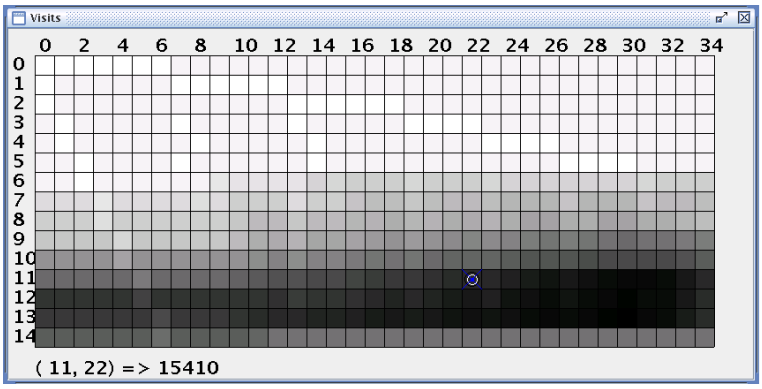


Figure 2.27: The **Visits** frame shows the number of bindings to some value at each entry. The darkest entries have had the most visits so far. It is shown that entry (11,22) was instantiated 15410 times.

The **Visits** frame shows the number of bindings to some value at each entry. The

darkest entries have had the most visits so far. You can click on each entry to see the visits of that entry. You can specify which value bindings should be counted through the popup menu. An example is shown in Figure 2.27.

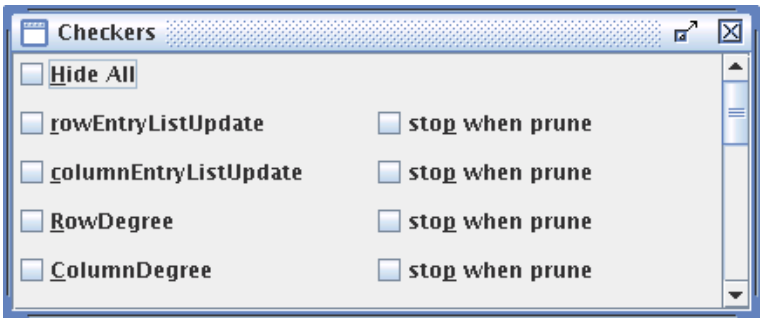


Figure 2.28: The **Checkers** frame shows a list of all your checkers. For each checker you can specify to interrupt the process whenever it prunes the search.

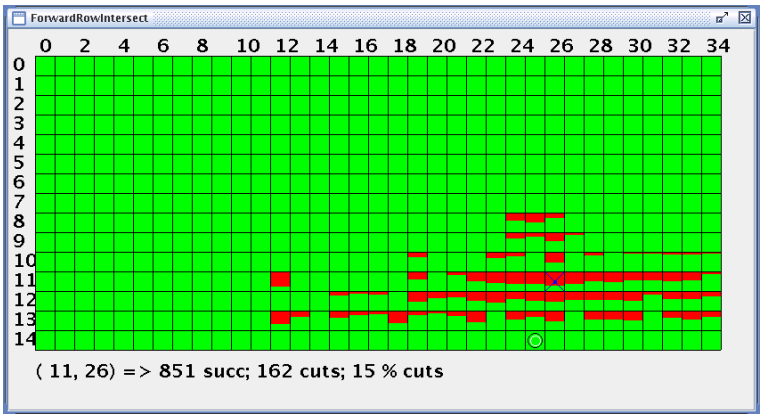


Figure 2.29: When you click on a checker of the list of Figure 2.28, its pruning statistics are shown in a frame like this one. The red part of each entry shows the pruning rate of the checker at that entry. Entry (11,26) is selected: The check passed 851 times and pruned 162 times.

The **Checkers** frame (Figure 2.28) contains a list of all checkers for which you can set a breakpoint, i.e. you specify to interrupt the process whenever that checker prunes the search. When you click on a checker of the list of Figure 2.28, its pruning statistics are visualized in a frame: An example is shown in Figure 2.29. The red part of each entry shows the pruning rate of the checker at that entry.

The **FixedSolution** frame (Figure 2.30) shows the entries which are the same in all solutions found so far. An empty entry (-) indicates that different values have been found in solutions.

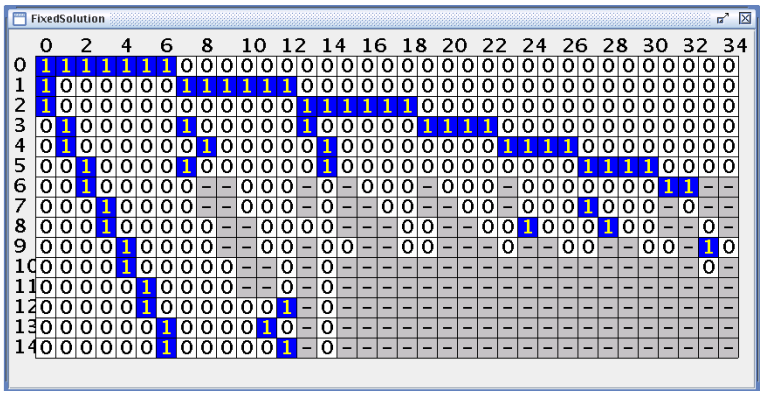


Figure 2.30: The **FixedSolution** frame shows the entries which are the same in all solutions found so far. In this example the first 8 columns are "fixed".

2.3.3 Implementation

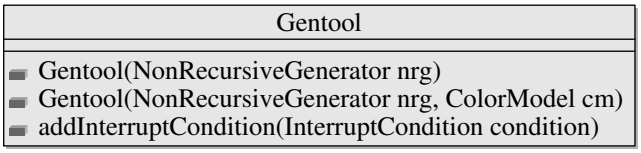


Figure 2.31: The *Gentool* class is the class to instantiate to start the visualizer. A non-recursive generator must be provided. Optionally, you can define the colors. You can add additional interrupt conditions for which the search should be interrupted.

The *be.ugent.caagt.gui.gentool.Gentool* class, shown in Figure 2.31, must be instantiated to visualize your generation process. Its constructors have the following arguments:

- A *NonRecursiveGenerator*, usually *GenericGenerator*.
- Optionally, a *ColorModel* which defines a color for each relation number. Colors can be altered in the application by clicking on the leftmost button in the top toolbar.

Hence visualization only takes one more line of code.

However, you might want to interrupt the process for some specific reason. For this purpose, we introduce the *InterruptCondition* interface, as shown in Figure 2.32. With the *addInterruptCondition(InterruptCondition ic)* method of *Gentool* you can add your own user written conditions for which the visualizer should stop. The *InterruptCondition* interface has only one method which takes the current entry as an argument. This method is called after each step, and the *next* action is stopped when the *shouldInterrupt()* method returns *true*.

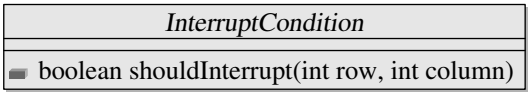


Figure 2.32: The *InterruptCondition* interface provides a way to interrupt the process when a user-written condition is met. Note that *shouldInterrupt()* takes the current entry as an argument.

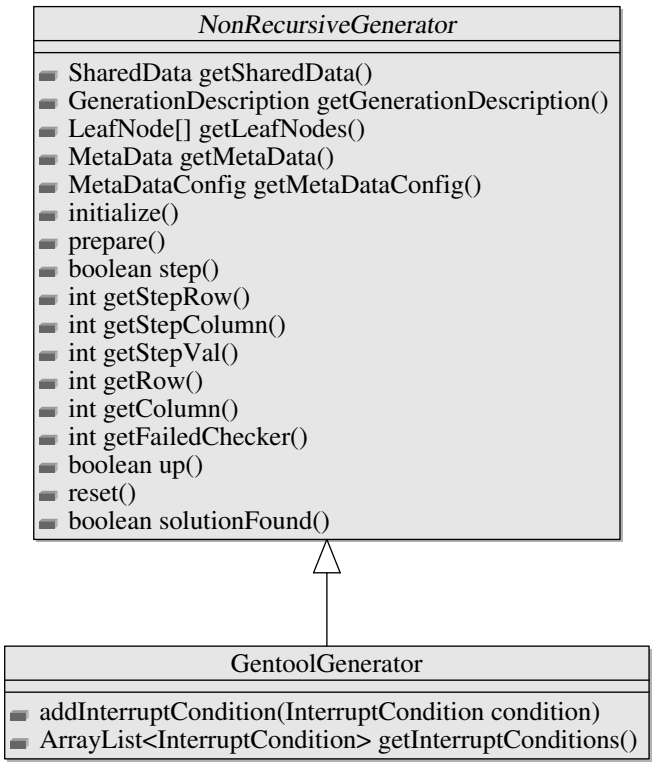


Figure 2.33: *GentoolGenerator* is the non-recursive generator implementation suitable for visualization. Actually, it is just a wrapper class around another non-recursive generator, usually *GenericGenerator*.

In order to visualize the generation process through user interaction, a non-recursive generator is needed to form the glue between the *backtrack* and the *gentool* package. Figure 2.33 shows the most important methods of the *NonRecursiveGenerator* interface and its implementation *GentoolGenerator*. Most methods are obvious, but some need more explanation. The *prepare()* method has the same meaning as the *prepare()* method of *Processor*. The *getStepXxx()* methods give information about the last step: Which entry was (un)instantiated to which value. The *getFailedChecker()* method gives the list index of the checker which pruned in the last step (-1 if none pruned). The *solutionFound()* method return **true** if the last step reached a solution. The *GentoolGenerator* class is a wrapper class around your *NonRecursiveGenerator* implementation which usually is the *GenericGenerator* class from the *backtrack* package. This wrapper contains methods, which are not shown, to expose generation metadata. Metadata is stored in the *MetaData* class of the *backtrack* package.

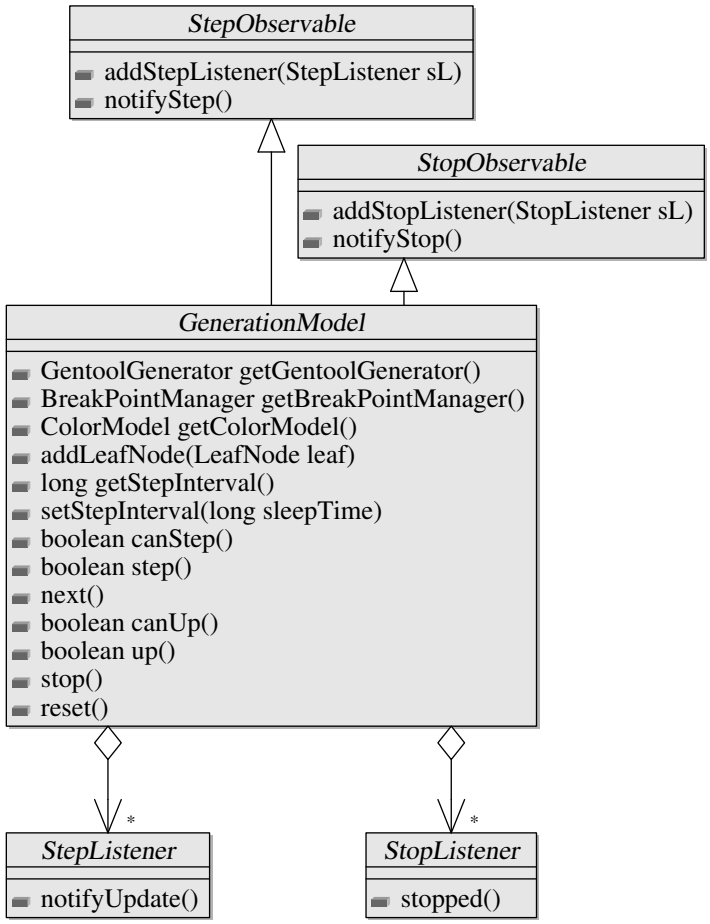


Figure 2.34: *GenerationModel* can be considered as the model in the Model-View-Controller paradigm. It is observed by listeners interested in the occurrence of a *step* or *stop* action.

GentoolGenerator is a part of *GenerationModel*, which can be considered as the *model* in the MVC (Model-View-Controller) paradigm, as shown in Figure 2.34. Frames are *viewers* of the model. The *actions* frame also serves as the *controller*. The model is observed by listeners interested in the occurrence of a *step* or *stop* action. Viewers which visualize each step, implement the *StepListener* interface and register themselves with the model through the *addStepListener(StepListener sL)* method. The model notifies its *StepListener*'s in the *notifyStep()* method by calling their *notifyUpdate()* method. Most parts of the internal frames implement the *StepListener* interface (matrix, search tree, domain, visits and checkers frame). However, when step by step visualization is disabled (no delay is set), viewers only update themselves after a *next* action finished. Therefore most viewers also implement *StopListener* and register themselves through *addStopListener(StopListener sL)*. Stop listeners are informed when a *stop* action occurred by calling their *stopped()* method in the *notifyStop()* method.

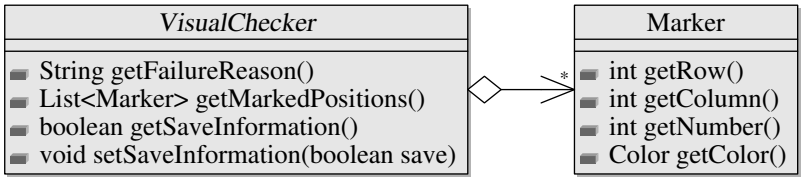


Figure 2.35: *VisualChecker* contains *Marker*'s which highlight a matrix entry. The highlighted entries could illustrate the constraint failure.

Sometimes it is useful to visualize why a constraint checker prunes, e.g. you could highlight the entries which constitute the violation of the constraint. Therefore, a *Checker* implementation might also implement the *VisualChecker* interface. *VisualChecker* contains a list of *Marker*'s, as shown in Figure 2.35. A *Marker* holds a matrix entry, together with an optional color and number to mark the entry with. Note that this behavior requires an extra programming effort. The *gentool* package is summarized in Figure 2.36.

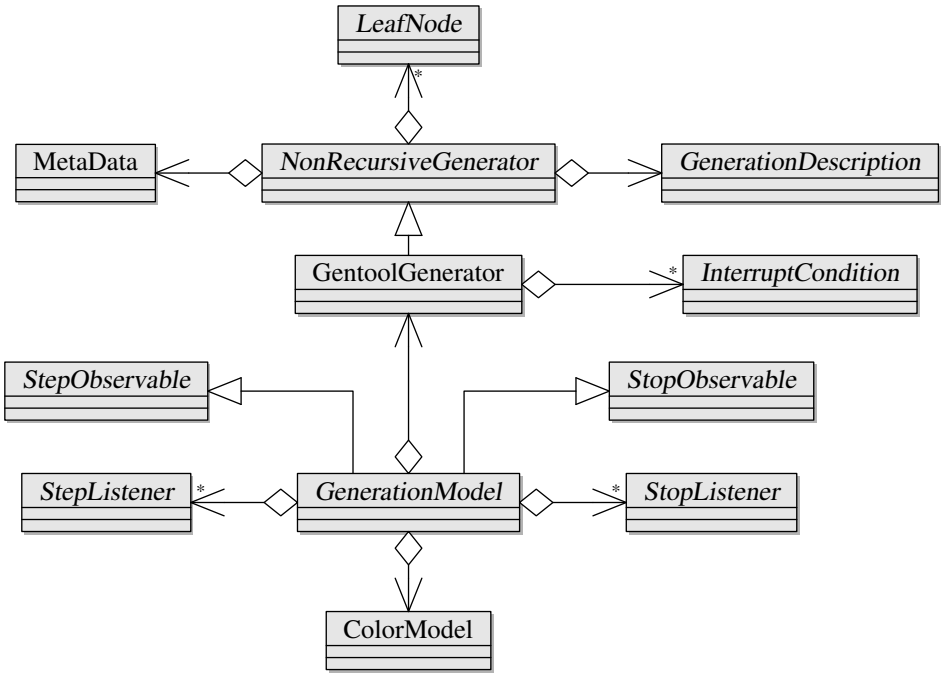


Figure 2.36: Summary of the *gentool* package. *LeafNode*, *MetaData*, *GenerationDescription* and *NonRecursiveGenerator* are part of the *backtrack* package.

2.4 Example

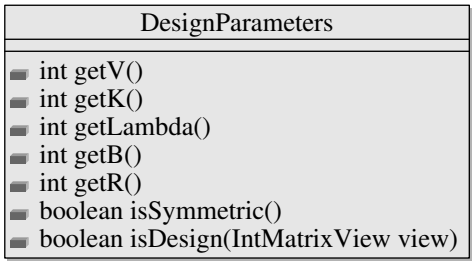


Figure 2.37: The *DesignParameters* class holds the design parameters.

This section contains a detailed example of how the backtracking and visualizer framework can be used. We give the implementation of the general generation of non-symmetric 2-designs. The design parameters are collected in the *DesignParameters* class, whose obvious methods are shown in Figure 2.37. We generate all non-symmetric $v \times b$ matrices which have r ones per row and k ones per column. Furthermore, the scalar product of any two rows is λ. We do not perform any isomorph rejection in this small example. This example can be found in the *be.ugent.caagt.design.example* package.

First of all, we choose an implementation of *IntMatrix* for our incidence matrix. We use the *ValueMatrix* class from the *backtrack* package. Recall that *ValueMatrix* is just a wrapper around a two-dimensional integer array. General purpose checkers (or other generator components) should use the *IntMatrix* interface, while dedicated checkers work directly on the array of *ValueMatrix*. The *SharedDataFactory* of Listing 2.1 is provided by the static *getFactory()* method of the *ValueMatrix* class. Note the use of *BasicParameters* to construct this matrix. The initializer of Listing 2.2 puts *any* implementation of *IntMatrix* into the shared data object with key *IntMatrix.class.getName()*, i.e. the fully quantified name of the *IntMatrix* interface. Furthermore, it initializes all entries to undefined (-1) in the *initialize()* method.

We use a standard row order path which fills the entries row by row. A possible path implementation is given by Listing 2.3. Recall that the generator will call *prepare()* to determine the next entry to instantiate. Furthermore, the generator stores the instantiated entries on a stack in the *GeneratorStack* class. The given implementation derives the row and column position from the *depth* argument of the *prepare()* method. Another possible implementation is to precalculate all positions and store them into an array.

A possible domain implementation is given by Listing 2.4. The domain list of each entry is $[1, 0]$. We could have used the standard implementation *FixedRangeDomain* from the *be.ugent.caagt.backtrack.domain* package instead.

This only leaves us three checkers to write. Each row must have r ones, each column must have k ones. To implement this, we first define an abstract class *DegreeInitializer*, which is shown in Listing 2.5. This abstract class just holds a two-dimensional integer array *degreeLeft*. Its use for rows is as follows: *degreeLeft[row][value]* holds the number of *value*'s which are still needed in *row* (*value* being either 1 or 0). Its use for columns is likewise: the

first index refers to a column. Two extensions of *DegreeInitializer* are given in Listings 2.6 and 2.7. Listing 2.6 contains the *DesignRowDegreeInitializer* class which initializes the *degreeLeft* array to hold the remaining row degrees. The *DesignColumnDegreeInitializer* class of Listing 2.7 does the same for the remaining column degrees.

The abstract *GenericDegreeChecker* class of Listing 2.8 is the superclass of the checkers which check the row and column degrees. This abstract base class expects a *DegreeInitializer* implementation which will be either *DesignRowDegreeInitializer* or *DesignColumnDegreeInitializer*. The *setAndCheck()* and *unset()* methods of *GenericDegreeChecker* are abstract. The *GenericRowDegreeChecker* class, shown in Listing 2.10, is the extension of *GenericDegreeChecker* which will check and hold the remaining row degrees. Suppose entry (row,column) has been instantiated to *value* and *setAndCheck(row, column)* is called for this *GenericRowDegreeChecker*: *setAndCheck(row, column)* will check wether *degree[row][value]* is non-zero and if so, then *degree[row][value]* is decremented and **true** is returned. Otherwise, **false** is returned. When backtracking, *unset(row, column)* increments *degree[row][value]*. The abstract *DegreeChecker* class, shown in Listing 2.9, is the subclass of *GenericDegreeChecker* which assumes *ValueMatrix* is used. The *ColumnDegreeChecker* checker, shown in Listing 2.11, is an extension of *DegreeChecker*, so it works directly with the integer array.

The scalar product between any two rows is λ. So each two rows have λ $(1,1)$, $r-\lambda$ $(1,0)$, $r-\lambda$ $(0,1)$ and $b-2r+\lambda$ $(0,0)$ combinations. Listing 2.12 gives the abstract intersection base class *IntersectionInitializer* which holds a four-dimensional *intersection* integer array. I.e. *intersection[rowOrColumn1][rowOrColumn2][value1][value2]* holds the number of (*value1*,*value2*) combinations which are still needed between rows/columns *rowOrColumn1* and *rowOrColumn2*. Listing 2.13 gives the extension of *IntersectionInitializer* to hold the row intersection numbers. Listing 2.14 gives the abstract intersection checker base class, while Listing 2.15 gives its extension to check the row intersection numbers. For symmetric designs we would need similar extensions to initialize and check the column intersection numbers.

Finally, Listing 2.16 gives the description class for this example. This *DesignDescription* class is an extension of the *DefaultDescription* class. First the basic parameters are initialized. Next the *IntMatrixInitializer* is created and added. The mentioned path and domain are set, followed by adding all the checkers and their related initializers.

This *DesignDescription* class also contains a main program which has v, k and λ as command-line arguments. The main program illustrates the use of a *LeafNode* which just counts the number of solutions. It also shows the use of the *MetaDataConfig* class to store the number of recursive calls. You can run this program with the command (for 2-(7,3,2) designs)

java be.ugent.caagt.design.example.DesignDescription 7 3 2

The *Visualize* class of Listing 2.17 contains a main program which visualizes *DesignDescription*. It has v, k and λ as command-line arguments. You can run this program with the command (for 2-(7,3,2) designs)

java be.ugent.caagt.design.example.Visualize 7 3 2

```
    private static final SharedDataFactory FACTORY = new SharedDataFactory() {
            public Object createItem (String key, SharedData shared) {
                BasicParameters basic = shared.getBasicParameters ();
                return new ValueMatrix(basic.getNumberOfRows (),
                                       basic.getNumberOfColumns ());
            }
        };
```

Listing 2.1: The factory which creates a *ValueMatrix* and puts it into *SharedData*. The *ValueMatrix* implementation provides this factory through its *getFactory()* method. Note the use of *BasicParameters* to construct this matrix.

```
public class IntMatrixInitializer implements Initializer {

    private SharedDataFactory factory;
    private IntMatrix matrix;

    /** Uses the factory to create an IntMatrix */
    public IntMatrixInitializer(SharedDataFactory factory) {
        this.factory = factory;
    }

    public void init(SharedData shared) {
        // create and put into shared data
        matrix
          = (IntMatrix) shared.getItem (IntMatrix.class.getName (),
                                        factory);
    }

    public void reset() {}

    public void initialize() {
        // set all values to undefined (-1)
        matrix.setDimensions(matrix.getNumberOfRows(),
                             matrix.getNumberOfColumns(), -1);
    }

    public IntMatrix getIntMatrix() {
        return matrix;
    }
}
```

Listing 2.2: *Initializer* which puts an *IntMatrix* into shared data with the fully quantified name *IntMatrix.class.getName()*. It undefines all entries (−1 value).

```
public final class RowOrderPath implements Path {

    // current (row, column) entry
    private int row = 0;
    private int column = 0;
    private int numberOfColumns;

    // Gets the current row
    public int getRow() { return row; }

    // Gets the current column
    public int getColumn() { return column; }

    public void init(SharedData shared) {
        BasicParameters param = shared.getBasicParameters();
        numberOfColumns = param.getNumberOfColumns();
    }

    public void reset() {}

    /** Determines the next matrix entry to instantiate.
     *  The depth parameter holds the current depth
     *  in the search tree,
     *  which is the same as the number of instantiated entries.
     *  The root node is at depth 0.
     */
    public boolean prepare(int depth) {
        this.row = depth / numberOfColumns;
        this.column = depth % numberOfColumns; // % is modulo operator
        return true;
    }
}
```

Listing 2.3: Example of a possible row order path implementation. The next (row, column) entry to instantiate is derived from the *depth* argument of the *prepare()* method.

```
public class DesignDomain
        implements Domain {

    // next[row][column] holds next domain value for entry (row,column)
    private int [][] next;

    // lowerbound value is 0
    private final int min = 0;

    // upperbound value is 1
    private final int max = 1;

    public void init(SharedData shared) {
        next = new int[shared.getBasicParameters().getNumberOfRows()]
                      [shared.getBasicParameters().getNumberOfColumns()];
    }

    public void reset() {}

    /** Resets the domain of entry (row, column) */
    public void reset(int row, int column) {
        next [row][column] = max;
    }

    /** Indicates if the domain of (row, column) has more values */
    public boolean hasNext(int row, int column) {
        return next [row][column] >= min;
    }

    /** Gets and removes the next domain value for entry (row, column) */
    public int next(int row, int column) {
        return next [row][column]--;
    }

    public boolean isDomainTraversalIncreasing() { return false; }
    public int getMinimumValue(int row, int column) { return min; }
    public int getMaximumValue(int row, int column) { return max; }
    public int getMinimumValue() { return min; }
    public int getMaximumValue() { return max; }
    public boolean isRangeDomain() { return true; }
    public boolean isUsingDomainMatrix() { return false; }
}
```

Listing 2.4: Possible domain implementation. The domain list of each entry is $[1, 0]$. The two-dimensional *next* integer array holds the next domain value for entry (row, column).

```
public abstract class DegreeInitializer implements Initializer {

    protected int [][] degreeLeft = null;

    /** Gets the degreeLeft array */
    public int[][] getDegreeLeft() {
        return degreeLeft;
    }
}
```

Listing 2.5: Abstract degree initializer base class which holds a two-dimensional *degreeLeft* integer array. I.e. *degreeLeft[rowOrColumn][value]* holds the number of *value* values which are still needed in row/column *rowOrColumn*. The methods of the *Initializer* interface are abstract.

```
public class DesignRowDegreeInitializer
        extends DegreeInitializer {

    private DesignParameters param;

    public DesignRowDegreeInitializer(DesignParameters param) {
        this.param = param;
        degreeLeft = new int[param.getV()][2];
    }

    public void initialize() {
        for (int i = 0 ; i < degreeLeft.length ; i++) {
            degreeLeft[i][1] = param.getR();
            degreeLeft[i][0] = param.getB() - param.getR();
        }
    }

    public void init(SharedData shared) {}
    public void reset() {}
}
```

Listing 2.6: Extension of *DegreeInitializer* which initializes the *degreeLeft* array to check the row degrees. Each row has r ones and $b - r$ zeroes.

```
public class DesignColumnDegreeInitializer
        extends DegreeInitializer {

    private DesignParameters param;

    public DesignColumnDegreeInitializer(DesignParameters param) {
        this.param = param;
        degreeLeft = new int[param.getB()][2];
    }

    public void initialize() {
        for (int i = 0 ; i < degreeLeft.length ; i++) {
            degreeLeft[i][1] = param.getK();
            degreeLeft[i][0] = param.getV() - param.getK();
        }
    }

    public void init(SharedData shared) {}
    public void reset() {}
}
```

Listing 2.7: Extension of *DegreeInitializer* which initializes the *degreeLeft* array to check the column degrees. Each column has k ones and $v - k$ zeroes.

```
public abstract class GenericDegreeChecker implements Checker {

    protected DegreeInitializer initializer;

    protected IntMatrix intMatrix = null;
    protected int numberOfRows;
    protected int numberOfColumns;
    protected int[][] degree = null;

    /** Sets the degree initializer */
    public void setInitializer(DegreeInitializer initializer) {
        this.initializer = initializer;
    }

    public void init(SharedData shared) {
        intMatrix = (IntMatrix) shared.getItem(IntMatrix.class.getName());
        numberOfRows = intMatrix.getNumberOfRows();
        numberOfColumns = intMatrix.getNumberOfColumns();
        degree = initializer.getDegreeLeft();
    }

    public void reset() {}
    public String toString() {  return "Abstract Generic Degree"; }
    public String getName() { return toString(); }
}
```

Listing 2.8: Abstract degree checker base class. The *setAndCheck()* and *unset()* methods are abstract.

```
public abstract class DegreeChecker extends GenericDegreeChecker {

    protected int[][] matrix = null;

    public void init(SharedData shared) {
        super.init(shared);
        matrix = ((ValueMatrix) intMatrix).getMatrix();
    }

    public String toString() {  return "Abstract Degree"; }
}
```

Listing 2.9: Abstract subclass of *GenericDegreeChecker* which assumes *ValueMatrix* is used. Subclasses can work directly with this integer matrix of *ValueMatrix*.

```
public final class GenericRowDegreeChecker extends GenericDegreeChecker {

    public boolean setAndCheck(int row, int column) {
        int value = intMatrix.getAt(row, column);
        if (degree[row][value] == 0)
            return false;
        --degree[row][value];
        return true;
    }

    public void unset(int row, int column) {
        int value = intMatrix.getAt(row, column);
        ++degree[row][value];
    }

    public String toString() { return "Generic Row Degree"; }
```

Listing 2.10: Extension of *GenericDegreeChecker* which checks the row degrees. This implementation works with the *IntMatrix* interface.

```
public final class ColumnDegreeChecker extends DegreeChecker {

    public boolean setAndCheck(int row, int column) {
        int value = matrix[row][column];
        if (degree[column][value] == 0)
            return false;
        --degree[column][value];
        return true;
    }

    public void unset(int row, int column) {
        int value = matrix[row][column];
        ++degree[column][value];
    }

    public String toString() { return "Column Degree"; }
}
```

Listing 2.11: Extension of *DegreeChecker* which checks the column degrees. Note that this implementation works directly with the integer array.

```
public abstract class IntersectionInitializer implements Initializer {

    protected int[][][][] intersection = null;

    public int[][][][] getIntersection () {
        return intersection;
    }
}
```

Listing 2.12: Abstract intersection initializer base class which holds a four-dimensional *intersection* integer array. I.e. *intersection[rowOrColumn1][rowOrColumn2][value1][value2]* holds the number of (*value1*,*value2*) combinations which are still needed between rows/-columns *rowOrColumn1* and *rowOrColumn2*. The methods of the initializer interface are abstract.

```
public class DesignIntersectionInitializer
        extends IntersectionInitializer {

    private DesignParameters param;

    public DesignIntersectionInitializer(DesignParameters param) {
        this.param = param;
    }

    public void initialize() {
        int lambda = param.getLambda();
        int r = param.getR();
        int b = param.getB();
        for (int i = 0 ; i < intersection.length ; i++)
            for (int j = 0 ; j < intersection[i].length ; j++) {
```

```
                intersection[i][j][1][1] = lambda;
                intersection[i][j][1][0] = r - lambda;
                intersection[i][j][0][1] = r - lambda;
                intersection[i][j][0][0] = b + lambda - 2 * r;
            }
    }

    public void init(SharedData shared) {
        intersection = new int [param.getV()][param.getV()-1][2][2];
    }

    public void reset() {}
}
```

Listing 2.13: Extension of *IntersectionInitializer* which initializes the *intersection* array to hold the row intersection numbers. Each two rows have λ $(1,1)$, $r-\lambda$ $(1,0)$, $r-\lambda$ $(0,1)$ and $b-2r+\lambda$ $(0,0)$ combinations.

```
public abstract class IntersectionChecker implements Checker {

    protected IntersectionInitializer initializer;
    protected int[][] matrix = null;
    protected int numberOfRows;
    protected int numberOfColumns;
    protected int[][][][] intersection = null;

    /** Sets the intersection array kept by initializer */
    public void setInitializer(IntersectionInitializer initializer) {
        this.initializer = initializer;
    }

    public void init(SharedData shared) {
        ValueMatrix vm = NonSymSharedData.createValueMatrix(shared);
        this.numberOfRows = vm.getNumberOfRows();
        this.numberOfColumns = vm.getNumberOfColumns();
        matrix = vm.getMatrix();
        intersection = initializer.getIntersection();
    }

    public void reset() {}

    public String toString() { return "AbstractIntersect"; }
    public String getName()  { return toString(); }
}
```

Listing 2.14: Abstract intersection checker base class. The *setAndCheck()* and *unset()* methods are abstract.

```
public final class RowIntersectionChecker extends IntersectionChecker {

    public boolean setAndCheck (int row, int column) {
        int value = matrix[row][column];
        for (int i = 0; i < row; i++) {
            int iRel = matrix[i][column];
            if (--intersection[row][i][value][iRel] < 0) {
                ++intersection[row][i][value][iRel];
                for (--i; i >= 0; i--)
                    ++intersection[row][i][value][matrix[i][column]];
                return false;
            }
        }
        return true;
    }

    public void unset (int row, int column) {
        int value = matrix[row][column];
        for (int i = 0; i < row; i++)
            ++intersection[row][i][value][matrix [i] [column]];
    }

    public String toString () { return "RowIntersect"; }
}
```

Listing 2.15: Extension of *IntersectionChecker* which checks the row intersection numbers. This implementation assumes the entries in the rows smaller than the row of the last instantiated entry are bound.

```
/**
 * Description for the generation of non-symmetric designs
 * without any isomorphic rejection.
 */
public class DesignDescription extends DefaultDescription {

    // Design parameters
    private DesignParameters designParam;

    /** Construct this description based on the design parameters */
    public DesignDescription(DesignParameters designParam) {
        this.designParam = designParam;

        initBasicParameters();
        initValueMatrix();
        setPathAndDomain();

        // Add checkers
        addRCheck();
        addKCheck();
        addRowLexicalCheck();
        addColumnLexicalCheck();
        addLambdaCheck();

        // Set description string
        setDescription(designParam.toString());
    }

    // Set BasicParameters
    private void initBasicParameters() {
        BasicParameters b
                = BasicParameters.createDefaultNonSymmetricBasicParameters
                (designParam.getV(), designParam.getB(), 0, 1);
        setBasicParameters(b);
    }

    // Put ValueMatrix and its initializer into SharedData
    private void initValueMatrix() {
        SharedDataFactory factory = ValueMatrix.getFactory();
        Initializer initializer = new IntMatrixInitializer(factory);
        addInitializer(initializer);
    }

    // Choose a default row order path and design domain
    private void setPathAndDomain() {
        Path path = new RowOrderPath();
        setPath(path);
        Domain domain = new DesignDomain();
        setDomain(domain);
    }
```

```
// Check the r parameter
private void addRCheck() {
    DesignRowDegreeInitializer initializer;
    initializer = new DesignRowDegreeInitializer(designParam);
    addInitializer(initializer);
    GenericDegreeChecker checker = new GenericRowDegreeChecker();
    checker.setInitializer(initializer);
    addChecker(checker);
}

// Check the k parameter
private void addKCheck() {
    DesignColumnDegreeInitializer initializer;
    initializer = new DesignColumnDegreeInitializer(designParam);
    addInitializer(initializer);
    DegreeChecker checker = new ColumnDegreeChecker();
    checker.setInitializer(initializer);
    addChecker(checker);
}

// Check the lambda parameter
private void addLambdaCheck() {
    DesignIntersectionInitializer initializer;
    initializer = new DesignIntersectionInitializer(designParam);
    addInitializer(initializer);
    RowIntersectionChecker checker = new RowIntersectionChecker();
    checker.setInitializer(initializer);
    addChecker(checker);
}

// Add a row lexical checker
private void addRowLexicalCheck() {
    LexicalInitializer rowInitializer
            = new DesignRowLexicalInitializer(designParam);
    RowLexicalChecker rowChecker = new RowLexicalChecker();
    rowChecker.setInitializer(rowInitializer);
    addInitializer(rowInitializer);
    addChecker(rowChecker);
}

// Add a column lexical checker
private void addColumnLexicalCheck() {
    LexicalInitializer colInitializer
            = new DesignColumnLexicalInitializer(designParam);
    ColumnLexicalChecker colChecker = new ColumnLexicalChecker();
    colChecker.setInitializer(colInitializer);
    addInitializer(colInitializer);
    addChecker(colChecker);
}

private static void printUsage() {
    System.out.println("java DesignDescription v k lambda");
}
```

```
    /** Main program which takes three arguments
     *  v, k and lambda.
     */
    public static void main(String [] args) {
        if (args.length != 3) {
            printUsage();
            return;
        }
        int v, k, lambda;
        try {
            v = Integer.parseInt(args[0]);
            k = Integer.parseInt(args[1]);
            lambda = Integer.parseInt(args[2]);
        } catch (NumberFormatException io) {
            printUsage();
            return;
        }
        long time = System.currentTimeMillis();
        DesignParameters designParam = new DesignParameters(v, k, lambda);
        DesignDescription desc = new DesignDescription(designParam);
        GenericGenerator generator = new GenericGenerator(desc);
        MetaDataConfig config = new MetaDataConfig();
        config.setSaveRecursiveCalls(true);
        generator.setMetaDataConfig(config);
        CountingLeafNode leaf = new CountingLeafNode();
        generator.addLeafNode(leaf);
        generator.generate();
        long calls = generator.getMetaData().getRecursiveCalls();
        long solutions = leaf.getCount();
        time = System.currentTimeMillis() - time;
        System.out.println("Calls = " + calls);
        System.out.println("Solutions = " + solutions);
        System.out.println("Time = " + time + " ms");
    }

}
```

Listing 2.16: Description class for the design generation.

```
/**
 * Visualizes the generation of DesignDescription.
 */
public class Visualize {

    private Visualize() {
    }

    private static void printUsage() {
        System.out.println("java Visualize v k lambda");
    }

    /** Main program which takes three arguments
     *  v, k and lambda.
     */
    public static void main(String [] args) {
        if (args.length != 3) {
            printUsage();
            return;
        }
        int v, k, lambda;
        try {
            v = Integer.parseInt(args[0]);
            k = Integer.parseInt(args[1]);
            lambda = Integer.parseInt(args[2]);
        } catch (NumberFormatException io) {
            printUsage();
            return;
        }
        long time = System.currentTimeMillis();
        DesignParameters designParam = new DesignParameters(v, k, lambda);
        DesignDescription desc = new DesignDescription(designParam);
        GenericGenerator generator = new GenericGenerator(desc);
        Gentool gentool = new Gentool(generator);
    }
}
```

Listing 2.17: Visualizes DesignDescription.

3 Equivalence testing

Equivalence testing is needed in any effective generation or enumeration algorithm. We will need general integer matrix equivalence testing in Chapters 4, 5, 6 and 7. We use the popular *nauty* [36] graph isomorphism testing software for practical isomorphism and equivalence testing computations. How we translate the problem of equivalence of integer matrices into a graph equivalence problem, is explained in Section 3.1. We wrote a small Java library which makes it possible to call *nauty* directly from our Java programs, as described in Section 3.2.

3.1 Equivalence of integer matrices

Testing designs for isomorphism can be done by testing their incidence matrices for equivalence. With the graph isomorphism testing software *nauty*, testing for equivalence between incidence matrices is done by converting them into bipartite graphs, and checking these for isomorphism:

- Define a graph vertex for each point and each block.
- A point vertex is connected to a block vertex if the corresponding point is incident with the corresponding block.
- Two-color the graph: all point vertices get color 0 and all block vertices get color 1.

Besides computing the full automorphism group, *nauty* can also compute the graph G_c of a (colored) graph G, i.e. the canonical form of this graph. *Nauty* guarantees that graphs G and H are isomorphic if and only if their canonical forms G_c and H_c are equal. So to test two (colored) graphs for isomorphism, we test their *nauty canonical form* for equality. These canonical forms are especially useful to filter isomorphic graphs from a set of graphs. For huge sets, the canonical forms can be stored into a file rather than in memory. Linux/Unix sort utilities can then be used to filter isomorphic graphs from the file.

More generally, we also consider equivalence between integer matrices (with non-negative integral entries), possibly with colored rows and columns.

Definition 3.1.1 (Equivalent integer matrices) *Two (colored) integer matrices M_1 and M_2 are equivalent if M_2 can be obtained from M_1 by a permutation of the rows and columns*

(which respects the coloring, i.e. only permutes rows/columns with rows/columns of the same color).

In order to check for equivalence between two integer matrices A and B we use the following approach. Assume that all matrix elements are smaller than 2^ℓ. This means that we can represent these integer elements in bitvectors of length ℓ, and that we can map the $m \times n$ integer matrices A and B to the $m \times n\ell$ binary matrices A_b and B_b. Extending such a binary matrix A_b to a colored $(m+n+\ell) \times (n\ell)$ matrix A'_b as follows, where 1_ℓ is the $1 \times n$ all-one vector, 0_ℓ the $1 \times n$ all-zero vector and I_ℓ the identity matrix of order l:

$$A'_b = \left(\begin{array}{cccc|c} \multicolumn{4}{c|}{A_b} & (\text{color } \ell+1) \\ \hline 1_\ell & 0_\ell & \dots & 0_\ell & (\text{color } 0) \\ 0_\ell & 1_\ell & \dots & 0_\ell & (\text{color } 0) \\ \dots & \dots & \dots & \dots & \\ 0_\ell & 0_\ell & \dots & 1_\ell & (\text{color } 0) \\ \hline & & & & (\text{color } 1) \\ I_\ell & I_\ell & \dots & I_\ell & \vdots \\ & & & & (\text{color } \ell) \end{array}\right),$$

it is easy to see that $A \cong B$ if and only if $A'_b \cong B'_b$ and that $Aut(A) \cong Aut(A'_b)$. Hence the integer matrix equivalence problem is translated to an incidence matrix equivalence problem.

3.2 Nauty package

The *be.ugent.caagt.nauty* package can be used to call B. D. McKay's graph isomorphism testing software *nauty* [1] from Java through Java Native Interface. Java Native Interface (JNI) allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. The prior purpose of this implementation is the ability to call *nauty* from Java programs in the same way you would call it from C programs. People who are familiar with calling *nauty* from C should have no problems to use this software. For people interested in *nauty*, but not familiar with C, this software may also be helpful, since we also provide some extra classes which makes using *nauty* easier.

There are only four classes: *NautyStats*, *NautyInvariant*, *NautyLib* and *Nauty*. Their relation is shown in Figure 3.1. We first look at each of these classes in subsequent sections. We end this section with an example application.

3.2.1 *NautyStats*

Java class which reflects *nauty*'s *statsblk* C struct, which holds the results of a call to *nauty*. See page 8 of the *nauty* manual for a detailed explanation of all members. *NautyStats* is shown in Figure 3.2.

[1] Refer to http://cs.anu.edu.au/~bdm/nauty/ for the *nauty* webpage.

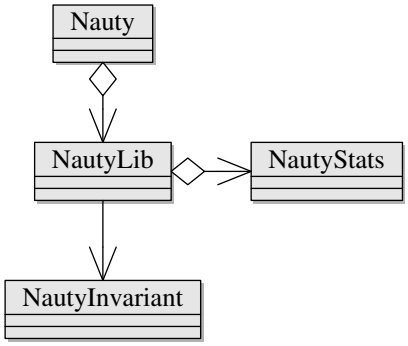


Figure 3.1: Overview of the *nauty* package.

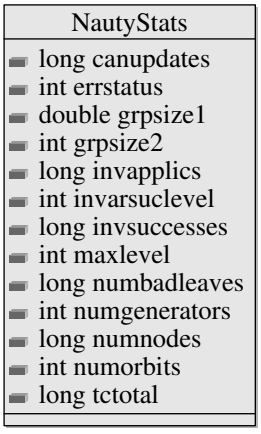


Figure 3.2: The *NautyStats* class reflects the *statsblk* struct.

3.2.2 *NautyInvariant*

The *NautyInvariant* class contains the vertex invariant constants, as shown in Figure 3.3. These invariants may influence the running time drastically. See pages 15-16 of the *nauty* manual for details.

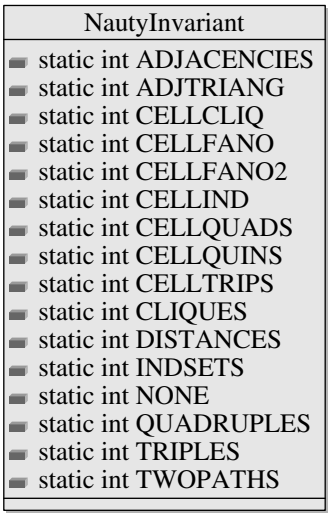


Figure 3.3: The *NautyInvariant* class contains the vertex invariants constants.

3.2.3 *NautyLib*

NautyLib is the class you need to instantiate if you want to call *nauty* in a way which reflects the *nauty* C call as close as possible. How you can repeatedly call *nauty* is shown in the activity diagram of Figure 3.4. The *NautyLib* class itself is shown in Figure 3.5.

Construction of *NautyLib*

Two constructors are available:

- *NautyLib (int maxn)*
- *NautyLib (int maxn, int worksize)*

At construction time, you must specify the maximum order *maxn* of the graphs you are going to feed to *nauty*. You can also specify the *worksize*, which is 50 by default [2], *nauty* uses

$$\frac{worksize * (maxn + WORDSIZE - 1)}{WORDSIZE}$$

integers, with WORDSIZE typically 32. The necessary memory is only allocated once on the native C side, and you can reuse the same instance of *NautyLib* for various graphs.

[2] nauty uses an additional parameter m, and states nauty should have a worksize of at least $50m$.

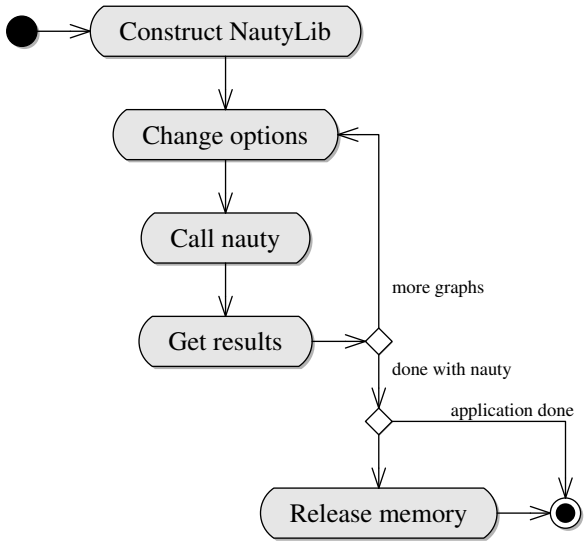


Figure 3.4: Calling nauty repeatedly from Java code.

However, each time *nauty* is called for a different graph, the graph needs to be copied from the Java environment to the native C side, so this involves some overhead.

Setting options

NautyLib contains a lot of *setXxx()* methods such as *setGetcanon(boolean jgetcanon)*, which allows to set the option *getcanon*. Options should be set before a call to *nauty*, by default the defaults of *nauty* apply. You can restore these defaults at any time by calling *restoreDefaultOptions()*. As already mentioned, you can call *nauty* repeatedly and alter the options between those calls.

Actual *nauty* call

The *nauty* C method is given by

```
nauty(graph *g, int *lab, int *ptn,
      set *active, int *orbits,
      optionblk *options, statsblk *stats,
      setword *workspace, setword *worksize,
      int m, int n, graph *canong)
```

Argument *active* is not supported in our implementation, since it is rarely used, as pointed out in the nauty manual. Arguments *options* and *stats* are omitted, they are set and retrieved through the *setXxx()* and *getXxx()* methods, respectively. Arguments *workspace*, *worksize* and *m* are already specified at construction time.

Figure 3.5: The *NautyLib* class.

The method *cnauty()* of *NautyLib* resembles this *nauty* C call. It has the following signature.

```
int[] cnauty(int[] jg, int[] jlab, int[] jptn,
             int[] jorbits, int jn, int[] jcanong,
             boolean returngenerators)
```

The first six arguments reflect the corresponding *g*, *lab*, *ptn*, *orbits*, *n* and *canong* arguments of the C *nauty()* call [3]. Some arguments are allowed to be *null*, with rules being more flexible than the rules of the original nauty call:

- *jg* and *jn* are obligatory. They contain the graph and its order. However, *jg* can be set to *null* when the graph of the last made call to *cnauty* can be reused.
- *jlab* can be set to *null* if *defaultptn* is *true* or to specify the default labelling 0, 1, ..., $jn-1$ (and of course only when you are not interested in the canonical labelling, since no output can be written).
- *jptn* can be set to *null* if *defaultptn* is *true*.
- *jorbits* and *jcanong* can be set to *null* if you are not interested in them. Otherwise, the orbits will be written into *jorbits*. If the option *getcanon* has been set and *jcanong* is not *null*, then the canonical form of the graph will be written into *jcanong*.

The extra **boolean** argument *returngenerators* specifies whether the group generators should be returned or not. *null* is returned when *returngenerators* is *false*, otherwise an array *gen* of exact length $p * jn$ is returned, where p is the number of generators. The generators are returned in cartesian format: generator i $(0 \leq i < jn)$ starts at $gen[i * jn]$ and ends at $gen[(i+1) * jn - 1]$. The number of generators can be determined from the array size of *gen*, i.e. the number of generators is $\frac{gen.length}{jn}$. The second method

```
int [] nauty (int[] jg, int[] jlab, int[] jcolors,
              int[] jcolorclasses, int jnrofcolors,
              int[] jorbits, int jn, int[] jcanong,
              boolean returngenerators)
```

has been deprecated, but remains here for backwards compatibility.

Getting results

The *NautyLib* class contains a lot of *getXxx()* methods such as *getNumorbits()*, which retrieves *numorbits* from the *NautyStats* object. These methods can be called after each call to *nauty*. There is also a *getNautyStats()* method which retrieves the whole *NautyStats* object.

[3] Adding a preceding "j" is a common JNI programming style guideline.

Releasing native memory

The method *releaseNautyMemory()* of *NautyLib* can be called when your application does no longer need *nauty*, thereby releasing the memory allocated at the native side. There is no need to call this method if your Java application ends (closely) after the last call to *nauty*.

Thread-safety

Thread-safe methods can be called from multiple programming threads without unwanted interaction between the threads. It is important to note that you can repeatedly call *cnauty()* from the same *NautyLib* instance, but these calls are *not* thread safe. So if you let two threads call *nauty* simultanuously, this will probably result in bad results or even a core dump. Different instances of *NautyLib* are also *not* thread safe, since *nauty* by itself is not thread safe. When you use multiple threads in an application which could call *nauty* simultanuously, you should provide the necessary synchronization yourself. In the backtrack framework, described in Chapter 2, you can easily share *NautyLib* through *SharedData*. To avoid problems, create only one instance of this class in your application.

3.2.4 *Nauty*

The *Nauty* class is a wrapper around *NautyLib*, containing useful methods. You can use *NautyLib* directly, the *Nauty* class is merely an extra layer around *NautyLib*. In particular, with the *Nauty* class, you can use *nauty* for incidence structures, square integer symmetric matrices and rectangular integer matrices with small integer non negative entries, without having to worry about the conversion of such matrices to graphs through some bijection. It makes it easier to define an initial coloring and you can traverse all permutations of the permutation group (row and/or column permutation group) by using the *be.ugent.caagt.perm* package. The typical usage cycle of the *Nauty* class is shown in the activity diagram of Figure 3.6.

In the UML diagram of the *Nauty* class, depicted in Figure 3.7, *IntMatrixView* is an interface of the *be.ugent.caagt.im* package, while *Perm* and *PermGroup* are classes of the *be.ugent.caagt.perm* package.

Construction of *Nauty*

Two constructors are available:

- *Nauty (int maxn, int worksize)*
- *Nauty (NautyLib lib, int maxn)*

The first constructor has the same meaning as the corresponding *NautyLib* constructor: it creates a *NautyLib* instance as a field member of the *Nauty* object. The second constructor is useful when you also need *nauty* in other parts of your application: you provide the *NautyLib* to use and specify the maximum graph order for which it can be used.

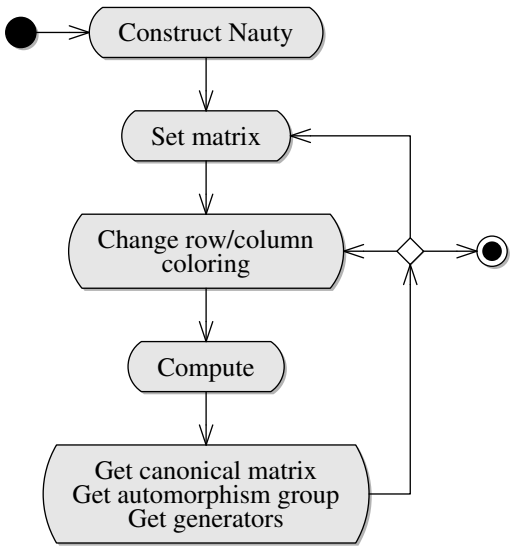


Figure 3.6: Typical usage cycle of the *Nauty* class.

Nauty
Nauty(int maxn, int worksize) -- CONSTRUCTORS -- Nauty(NautyLib lib, int maxn) NautyLib getNautyLib() int[] getGraph() int[] getLab() int[] getPtn() int[] getOrbits() setGetGenerators(boolean get) int setDefaultColors() int setColors(int[] colors) int setColors(int[] rowColors, int[] columnColors) setAdjacencyMatrix(IntMatrixView view) setIncidenceMatrix(IntMatrixView view) setRectangularIntegerMatrix(IntMatrixView view, int minValue, int maxValue) setSquareIntegerMatrix(IntMatrixView view, int minValue, int maxValue) compute() IntMatrixView getCanonicalMatrixView() PermGroup getAutomorphismGroup() PermGroup getRowAutomorphismGroup() PermGroup getColumnAutomorphismGroup() Perm[] getGenerators() Perm[] getRowGenerators() Perm[] getColumnGenerators()

Figure 3.7: *Nauty* is a wrapper class around *NautyLib*.

Providing the matrix

If you have a graph, then you use the *setAdjacencyMatrix(IntMatrixView view)* method. If you have a 0,1 non-symmetric matrix, then you use *setIncidenceMatrix(IntMatrixView view)*. If you have a square symmetric integer matrix with entries in the interval $[minValue, maxValue]$, then you use *setSquareIntegerMatrix(IntMatrixView view, int minValue, int maxValue)*. Finally, if you have a (rectangular) non-symmetric integer matrix with entries in the interval $[minValue, maxValue]$, then you use *setRectangularIntegerMatrix(IntMatrixView view, int minValue, int maxValue)*.

Changing the coloring

For adjacency (graphs) and square integer symmetric matrices, you define the coloring with the method *int setColors(int [] colors)*, so row (column) i has color *colors[i]*.

For incidence matrices and rectangular integer matrices, you define the coloring with the method *int setColors(int [] rowColors, int [] columnColors)*, so row i has color *rowColors[i]*, while column j has color *columnColors[j]*. Make sure you use different colors for rows and columns.

Calling nauty

Now that you defined the matrix and its coloring, you can actually call *nauty* with *compute()*.

Getting results

You can get the canonical form through a permuted view of your original matrix through the method *IntMatrixView getCanonicalMatrixView()*.

For symmetric matrices, you can get the automorphism group generators with *Perm [] getGenerators()*. You can get the automorphism group with *PermGroup getAutomorphismGroup()*.

For non-symmetric matrices, you can get the row automorphism group generators with *Perm [] getRowGenerators()* and the column automorphism group generators with *Perm [] getColumnGenerators()*. You can get the row automorphism group with *PermGroup getRowAutomorphismGroup()* and the column automorphism group with *PermGroup getColumnAutomorphismGroup()*.

3.2.5 Examples

We give an example of the use of both *NautyLib* and *Nauty* to determine group properties of the Petersen graph. Both programs produce the same result.

Example use of *NautyLib*

```
import be.ugent.caagt.nauty.NautyLib;
import be.ugent.caagt.perm.Perm;

/**
 * This program calculates the canonical form
 * of the Petersen graph.
 */
public final class NautyLibExample {
    public static void main(String[] args) {
        // Petersen graph as adjacency matrix:
        int[][] petersenGraph = new int[][] {
            { 0, 1, 0, 0, 1, 1, 0, 0, 0, 0 },  //0
            { 1, 0, 1, 0, 0, 0, 1, 0, 0, 0 },  //1
            { 0, 1, 0, 1, 0, 0, 0, 1, 0, 0 },  //2
            { 0, 0, 1, 0, 1, 0, 0, 0, 1, 0 },  //3
            { 1, 0, 0, 1, 0, 0, 0, 0, 0, 1 },  //4
            { 1, 0, 0, 0, 0, 0, 0, 1, 1, 0 },  //5
            { 0, 1, 0, 0, 0, 0, 0, 0, 1, 1 },  //6
            { 0, 0, 1, 0, 0, 1, 0, 0, 0, 1 },  //7
            { 0, 0, 0, 1, 0, 1, 1, 0, 0, 0 },  //8
            { 0, 0, 0, 0, 1, 0, 1, 1, 0, 0 }   //9
        };

        // order of petersen graph
        int jn = petersenGraph.length;

        // nauty lab
        int[] jlab = new int[jn];

        // defaultptn is true, jptn can be null
        int[] jptn = null;

        // nauty orbits will be written here
        int[] jorbits = new int[jn];

        // canonical labelled graph is written here
        int[] jcanong = new int[jn * jn];

        // generators will be returned
        boolean returngenerators = true;

        // convert 2-dimensional matrix to a
        // vector where all entries are listed row by row.
        int[] jg = new int[jn * jn];
        for (int i = 0, k = 0; i < jn; i++)
            for (int j = 0; j < jn; j++)
                jg [k++] = petersenGraph [i] [j];
```

```
        // create NautyLib for graphs of order <= jn
        NautyLib lib = new NautyLib(jn);

        // set options
        lib.setGetcanon(true);
        lib.setDefaultptn(true);

        // native call to nauty, returns generators
        int[] generators = lib.cnauty(jg, jlab, jptn,
                                      jorbits, jn, jcanong,
                                      returngenerators);

        System.out.println("** GENERATORS **");
        int nrOfGenerators = generators.length / jn;
        for (int i = 0 ; i < nrOfGenerators; i++) {
            int [] gen = new int[jn];
            System.arraycopy(generators, i*jn, gen, 0, jn);
            System.out.println("Generator: " + Perm.create(gen));
        }

        double grpsize1 = lib.getGrpsize1();
        int grpsize2 = lib.getGrpsize2();
        System.out.println("grpsize1 = " + grpsize1);
        System.out.println("grpsize2 = " + grpsize2);

        System.out.println("** ORBITS **");
        for (int i: jorbits)
            System.out.print(" " + i);

        System.out.println("\n** CANONICAL LABELLING **");
        for (int i: jlab)
            System.out.print(" " + i);

        System.out.println("\n** CANONICAL GRAPH **");
        for (int i = 0, k = 0 ; i < jn; i++) {
            for (int j = 0; j < jn; j++)
                System.out.print("" + jcanong[k++] + " ");
            System.out.println("");
        }
    }
}
```

Listing 3.1: Example use of *NautyLib*

Program Output:

```
** GENERATORS **
Generator:  0 1 2 7 5 4 6 3 9 8
Generator:  0 1 6 8 5 4 2 9 3 7
Generator:  0 4 3 2 1 5 9 8 7 6
Generator:  1 0 4 3 2 6 5 9 8 7
grpsize1 = 120.0
grpsize2 = 0
```

```
** ORBITS **
 0 0 0 0 0 0 0 0 0 0
** CANONICAL LABELLING **
 0 1 4 5 2 6 3 8 7 9
** CANONICAL GRAPH **
0111000000
1000110000
1000001001
1000000110
0100001010
0100000101
0010100100
0001011000
0001100001
0010010010
```

Example use of *Nauty*

```
import be.ugent.caagt.im.DefaultIM;
import be.ugent.caagt.im.IntMatrixView;
import be.ugent.caagt.nauty.Nauty;
import be.ugent.caagt.perm.Perm;

/** This program calculates the canonical form
 * and automorphism group of the Petersen graph. */
public final class NautyExample {

    public static void main(String[] args) {
        // Petersen graph as adjacency matrix:
        int[][] petersenGraph = new int[][] {
            { 0, 1, 0, 0, 1, 1, 0, 0, 0, 0 },  //0
            { 1, 0, 1, 0, 0, 0, 1, 0, 0, 0 },  //1
            { 0, 1, 0, 1, 0, 0, 0, 1, 0, 0 },  //2
            { 0, 0, 1, 0, 1, 0, 0, 0, 1, 0 },  //3
            { 1, 0, 0, 1, 0, 0, 0, 0, 0, 1 },  //4
            { 1, 0, 0, 0, 0, 0, 0, 1, 1, 0 },  //5
            { 0, 1, 0, 0, 0, 0, 0, 0, 1, 1 },  //6
            { 0, 0, 1, 0, 0, 1, 0, 0, 0, 1 },  //7
            { 0, 0, 0, 1, 0, 1, 1, 0, 0, 0 },  //8
            { 0, 0, 0, 0, 1, 0, 1, 1, 0, 0 }    //9
        };
        DefaultIM petersenIm = new DefaultIM(petersenGraph.length);
        petersenIm.fromArray(petersenGraph);

        Nauty nauty = new Nauty(10, 100);
        nauty.setAdjacencyMatrix(petersenIm);
        nauty.setDefaultColors();
        nauty.compute();
        IntMatrixView canonical = nauty.getCanonicalMatrixView();
        Perm [] gen = nauty.getGenerators();
        System.out.println("** GENERATORS **");
```

```
        for (Perm p: gen)
            System.out.println("Generator: " + p);

        double grpsize1 = nauty.getNautyLib().getGrpsize1();
        int grpsize2 = nauty.getNautyLib().getGrpsize2();
        System.out.println("grpsize1 = " + grpsize1);
        System.out.println("grpsize2 = " + grpsize2);

        System.out.println("** ORBITS **");
        for (int i: nauty.getOrbits())
            System.out.print(" " + i);

        System.out.println("\n** CANONICAL LABELLING **");
        for (int i: nauty.getLab())
            System.out.print(" " + i);

        System.out.println("\n** CANONICAL GRAPH **");
        System.out.println(canonical);
    }
}
```

Listing 3.2: Example use of *Nauty*

4 Generation of designs with non-trivial automorphisms

The classification of all combinatorial objects is often too hard for larger parameters, but it may still be possible to make classifications of such objects which have certain automorphisms. In this chapter we will focus on the classification of all 2-(v, k, λ) designs having an automorphism of prime order, following the well-known *local approach* method used in e.g. [25], [33] and [44].

We implemented a program which is suitable for the generation of 2-(v, k, λ) designs with automorphisms of small prime order (2, 3, 5 and 7). For larger orders the program still works, but it is not efficient. In particular, the degree and scalar product constraints are tested efficiently by calculating all possible row and column (intersection) patterns prior to the backtrack generation.

This chapter forms the basis for the three subsequent chapters. First, Chapter 5 presents the classification of 2-(31,15,7) and 2-(35,17,8) Hadamard designs and 2-(36,15,6) Menon designs with automorphisms of odd prime order. Second, Chapter 6 uses the local approach method for another combinatorial structure: a partial geometry. Third, Chapter 7, which presents the enumeration of the doubles of the projective plane of order 4, also relies on the local approach method.

4.1 Local approach method

Our generation algorithm for all 2-(v, k, λ) designs assumes an automorphism of small prime order p ($p \leq 7$) with f fixed points and f' fixed blocks. The following symbols are used:

$$
\begin{aligned}
v, b, r, k, \lambda &= \text{design parameters} && (4.1)\\
f &= \text{number of fixed points} && (4.2)\\
f' &= \text{number of fixed blocks} && (4.3)\\
h &= v - f = \text{number of non-fixed points} && (4.4)\\
g &= b - f' = \text{number of non-fixed blocks} && (4.5)\\
p &= \text{prime order of assumed automorphism} && (4.6)\\
n &= \frac{h}{p} && (4.7)\\
n' &= \frac{g}{p} && (4.8)
\end{aligned}
$$

Note that any automorphism of a symmetric 2-design ($v = b$, $k = r$) fixes the same number of points and blocks, see [14], so $f = f', n = n', h = g$ in that case. Throughout this chapter, as an example, we will illustrate the generation of all 2-$(9, 4, 3)$ designs with $p = 2$, $f = 3$ and $f' = 6$. For each of the 11 2-$(9, 4, 3)$ designs, we list their possible (p, f, f') values in Table 4.1. Note that we should generate 6 2-$(9, 4, 3)$ designs with an automorphism of order 2 and 3 fixed points and 6 fixed blocks. In this example, we have $v = 9$, $b = 18$, $r = 8$, $k = 4$, $\lambda = 3$, $f = 3$, $f' = 6$, $h = 6$, $g = 12$, $p = 2$, $n = 3$, $n' = 6$.

(p, f, f')					
$(3,0,0)$					
$(3,0,0)$	$(2,1,2)$		$(2,3,6)$		
$(3,0,0)$			$(2,3,6)$		
		$(2,3,4)$		$(2,5,6)$	$(2,7,8)$
	$(2,1,2)$	$(2,1,6)$		$(2,5,6)$	
			$(2,3,6)$		
			$(2,3,6)$		
	$(2,1,2)$		$(2,3,6)$		
	$(2,1,2)$		$(2,3,6)$		

Table 4.1: For each of the 11 2-$(9, 4, 3)$ designs we list all possible (p,f,f') values from its automorphisms.

Definition 4.1.1 (Circulant) *A square matrix C of order p is a circulant if it has a cyclic permutation φ such that $C_{\varphi i,\varphi j} = C_{i,j}$, $1 \leq i, j \leq p$.*

So a circulant can be defined by its first row. Let A be the incidence matrix of a 2-(v,k,λ) design. Assume A has an automorphism φ of prime order p with f fixed points and f' fixed blocks which works on A's rows as

$$(1)(2)\cdots(f)(f+1\ \cdots\ f+p)(f+p+1\ \cdots\ f+2p)\cdots(v-p+1\ \cdots\ v)$$

and on A's columns as

$$(1)(2)\cdots(f')(f'+1\ \cdots\ f'+p)(f'+p+1\ \cdots\ f'+2p)\cdots(b-p+1\ \cdots\ b)$$

The first f points and the first f' blocks are *fixed* and the last $h = v - f$ points and the last $g = b - f'$ blocks *non-fixed*. We want to generate all non-isomorphic incidence matrices A which have the automorphism φ. In the local approach method, the incidence matrices are considered to consist of 4 sub-matrices, reflecting fixed and non-fixed parts:

$$A = \begin{pmatrix} F & G \\ H & X \end{pmatrix}$$

The *fixed part* is formed by the $f \times f'$ matrix $F = (f_{i,j})$, the $f \times g$ matrix $G = (g_{i,j})$ and the $h \times f'$ matrix $H = (h_{i,j})$. Due to the assumed automorphism φ, the number of ones in each row of G (and in each column of H) is a multiple of p. Indeed, G's rows are fixed and its columns are in consecutive orbits of order p, hence the same value (1 or 0) occurs p times in every column which is in the same orbit of φ. The $h \times g$ matrix $X = (x_{i,j})$ forms the *non-fixed part*. It contains nn' circulants of order p, i.e. $X = (C_{i,j})$ with circulants

$$C_{i,j} = \begin{pmatrix} x_{(i-1)p+1,(j-1)p+1} & \cdots & x_{(i-1)p+1,jp} \\ \vdots & & \vdots \\ x_{ip,(j-1)p+1} & \cdots & x_{ip,jp} \end{pmatrix}, \qquad 1 \le i \le n; 1 \le j \le n'.$$

We refer to $(C_{i,1} \cdots C_{i,n'})$ as the i-th *row of circulants*, and to $(C_{1,j} \cdots C_{n,j})^T$ as the j-th *column of circulants*.

Definition 4.1.2 (Starting configuration) *The starting configuration A_s is the matrix A in which the fixed parts are determined and X is undefined, therefore we consider X to be the $h \times g$ all-zero matrix:*

$$A_s = \begin{pmatrix} F & G \\ H & 0 \end{pmatrix}$$

Let $\hat{G}$ be the $f \times n'$ matrix obtained from G with elements $\hat{g}_{i,j} = g_{i,jp}, 1 \le i \le f, 1 \le j \le n'$; this mapping is a bijection. Likewise, let $\hat{H}$ be the $n \times f'$ matrix obtained from H with elements $\hat{h}_{i,j} = h_{ip,j}, 1 \le i \le n, 1 \le j \le f'$.

Definition 4.1.3 (Starting orbit configuration) *The starting orbit configuration $\hat{A}_s$ is obtained from the starting configuration A_s by replacing G with $\hat{G}$, H with $\hat{H}$ and the $h \times g$ all-zero matrix by the $n \times n'$ all-zero matrix:*

$$\hat{A}_s = \begin{pmatrix} F & \hat{G} \\ \hat{H} & 0 \end{pmatrix}$$

$$\hat{A}_s = \left(\begin{array}{c|c} 111100 & 110000 \\ 111010 & 001100 \\ 000110 & 101010 \\ \hline 100101 & 000000 \\ 010011 & 000000 \\ 001000 & 000000 \end{array}\right) \quad \hat{A} = \left(\begin{array}{c|c} 111100 & 110000 \\ 111010 & 001100 \\ 000110 & 101010 \\ \hline 100101 & 011111 \\ 010011 & 110111 \\ 001000 & 111112 \end{array}\right)$$

$$A = \left(\begin{array}{c|cccccc} 111100 & 11 & 11 & 00 & 00 & 00 & 00 \\ 111010 & 00 & 00 & 11 & 11 & 00 & 00 \\ 000110 & 11 & 00 & 11 & 00 & 11 & 00 \\ \hline 100101 & 00 & 10 & 10 & 10 & 10 & 10 \\ 100101 & 00 & 01 & 01 & 01 & 01 & 01 \\ & & & & & & \\ 010011 & 10 & 10 & 00 & 10 & 01 & 01 \\ 010011 & 01 & 01 & 00 & 01 & 10 & 10 \\ & & & & & & \\ 001000 & 10 & 10 & 01 & 01 & 10 & 11 \\ 001000 & 01 & 01 & 10 & 10 & 01 & 11 \end{array}\right)$$

Figure 4.1: Possible $\hat{A}_s$, $\hat{A}$ and A of the running example.

Definition 4.1.4 (Orbit matrix) *The orbit matrix $\hat{X}$ is a $n \times n'$ matrix, in which entry $\hat{x}_{i,j}$ denotes the number of ones in a row of the circulant $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq n'$.*

Definition 4.1.5 (Extended orbit matrix) *The extended orbit matrix $\hat{A}$ is defined as*

$$\hat{A} = \begin{pmatrix} F & \hat{G} \\ \hat{H} & \hat{X} \end{pmatrix}.$$

Figure 4.1 shows $\hat{A}_s$, an extended orbit matrix $\hat{A}$ and its possible extension A of our running example.

The generation process consists of several phases, each of which performs an exhaustive backtracking search. Here we sketch the general ideas of the algorithm; more details on each phase are given in the following sections. First we generate all non-equivalent *starting (orbit) configurations*, normally this is the easiest part of the search. Only the $\hat{X}$ part of $\hat{A}$ remains to be generated. For every starting orbit configuration, we generate all orbit matrices $\hat{X}$. Constraints for this generation can be derived from the parameters of the 2-(v,k,λ) design. The final phase expands each (non-equivalent) solution for $\hat{A}$ to a full incidence matrix A by replacing each integer entry $\hat{x}_{i,j}$ of $\hat{X}$ by all the possible circulants for that entry, and by replacing $\hat{G}$ with G and $\hat{H}$ with H.

Generation algorithm

For each of the generation phases we will use, unless indicated otherwise, an orderly exhaustive backtracking algorithm which fills the matrix entry by entry (first the highest possible entry), row by row. We call this *row order generation*.

Definition 4.1.6 (Lexically ordered rows) *Consider an integer matrix M with entries $m_{i,j}$. We say that rows r and s of M are* **lexically ordered**, *or equivalently row r is* **lexically larger** *than row s (we write $M(r) > M(s)$) if and only if*

$$\exists j, \forall i < j : m_{r,i} = m_{s,i} \wedge m_{r,j} > m_{s,j}.$$

Definition 4.1.7 (Lexically ordered matrices) *Consider two integer matrices M_1 and M_2, we say that M_1 and M_2 are* **lexically ordered** *if and only if*

$$\exists s, \forall r < s : M_1(r) = M_2(r) \wedge M_1(s) > M_2(s).$$

Cheap partial isomorph rejection is possible in this algorithm by generating all matrices from the lexicografic largest one to the lexicografic smallest one. The specific techniques will be explained in each phase.

4.2 Generation of the fixed parts

The following properties are used in the backtracking algorithms which generate the fixed parts:

- A fixed point is incident with u_p non-fixed blocks and $r - u_p$ fixed blocks, u_p a multiple of p. Similarly, a fixed block is incident with u_p non-fixed points and $k - u_p$ fixed points, u_p a multiple of p.
- Each pair of fixed points is incident with w_p non-fixed blocks and $\lambda - w_p$ fixed blocks, w_p a multiple of p. For symmetric designs each pair of fixed blocks is incident with w_p non-fixed points and $\lambda - w_p$ fixed points.

We first generate all non-equivalent matrices F. In this generation, we lexically order the rows and columns of F, and perform a final full equivalence test on all obtained F. In our example, the 3×6 matrix F has the following constraints:

- Each row contains 0, 2, 4 or 6 ones.
- Each column contains 0 or 2 ones.
- Each pair of rows has scalar product 1 or 3.

This gives two non-equivalent solutions for F:

$$F_1 = \begin{pmatrix} 111100 \\ 111010 \\ 000110 \end{pmatrix} ; F_2 = \begin{pmatrix} 110000 \\ 101000 \\ 011000 \end{pmatrix}$$

For each F, we generate all non-equivalent $\hat{G}$ and $\hat{H}$. Note that the degree and scalar product constraints used in this generation depend on F. We can lexically order the columns of $\hat{G}$ and the rows of $\hat{H}$. If F has a set of consecutive equal rows, we can also lexically order the same rows in $\hat{G}$. The same applies for the columns in $\hat{H}$. In our example we can lexically order the first 3 columns of $\hat{H}$ given F_1, and the last 3 columns of $\hat{H}$ given F_2. The

obtained $\hat{G}$ and $\hat{H}$ matrices are then combined in all possible ways, and a final equivalence test is performed on the colored starting orbit configuration $\hat{A}_s$:

$$\left(\begin{array}{cc|c} F & \hat{G} & \text{color } 0 \\ \hat{H} & ? & \text{color } 1 \\ \hline \text{color } 2 & \text{color } 3 & \end{array}\right).$$

We respect φ by assigning a different color to the rows of F and $\hat{H}$, as well as to the columns of F and $\hat{G}$. This way we obtain a set of non-equivalent starting orbit configurations. In our example, given F_1, there is one possibility for $\hat{G}$ and there are 51 possibilities for $\hat{H}$, but only 14 starting configurations are non-equivalent. Given F_2, there is one possibility for $\hat{G}$ and there are 5 for $\hat{H}$, which leads to 3 non-equivalent starting configurations. In total, we have 17 starting configurations.

4.3 Generation of the orbit matrices

Let h_y^* be the number of ones in row y of $\hat{H}$, $g_{y'}^*$ the number of ones in column y' of $\hat{G}$, h_{y_1,y_2}^* the scalar product between rows y_1 and y_2 of $\hat{H}$ ($1 \le y, y_1, y_2 \le n$, $1 \le y' \le n'$), i.e.

$$h_{y_1,y_2}^* = \sum_{j=1}^{f'} \hat{h}_{y_1 j} \hat{h}_{y_2 j}$$

For symmetric designs, let $g_{y_1',y_2'}^*$ be the scalar product between columns y_1' and y_2' of $\hat{G}$ ($1 \le y_1', y_2' \le n'$). Double counting arguments on the number of ones in each row (resp. column) and the number of one-one intersections between two rows (and also columns for symmetric designs), lead to the following constraints for the orbit matrix $\hat{X}$, of which constraints (4.13) and (4.14) only apply for symmetric designs:

$$\sum_{j=1}^{n'} \hat{x}_{y,j} = r - h_y^*, \qquad 1 \le y \le n; \tag{4.9}$$

$$\sum_{i=1}^{n} \hat{x}_{i,y'} = k - g_{y'}^* \ ; \qquad 1 \le y' \le n'; \tag{4.10}$$

$$\sum_{j=1}^{n'} \hat{x}_{y,j}^2 = (p-1)\lambda + r - p h_y^*, \qquad 1 \le y \le n; \tag{4.11}$$

$$\sum_{j=1}^{n'} \hat{x}_{y_1,j} \hat{x}_{y_2,j} = p(\lambda - h_{y_1,y_2}^*), \qquad 1 \le y_1 < y_2 \le n; \tag{4.12}$$

$$\sum_{i=1}^{n} \hat{x}_{i,y'}^2 = (p-1)\lambda + k - p g_{y'}^*, \qquad 1 \le y' \le n'; \tag{4.13}$$

$$\sum_{i=1}^{n} \hat{x}_{i,y_1'} \hat{x}_{i,y_2'} = p(\lambda - g_{y_1',y_2'}^*), \qquad 1 \le y_1' < y_2' \le n'. \tag{4.14}$$

In order to speed up this generation process, we calculate and store some information prior to the backtracking search. We first determine all possible row patterns meeting constraints (4.9) and (4.11), all possible column patterns meeting constraints (4.10) (and (4.13) for symmetric designs), all possible row-row intersection patterns meeting (4.12) (and all possible column-column intersection patterns meeting (4.14) for symmetric designs) . The backtracking algorithm then checks after each binding of an entry to one of the p possible values, whether at least one pattern meeting all constraints remains. This technique can be used when p is small enough (typically $p \leq 7$), while for higher p memory issues arise. In our example, for 15 out of the 17 starting configurations, there are either no row patterns which meet both constraints (4.9) and (4.11), or for some combination of two rows, (4.12) is never possible. There are only two starting configurations which each lead to one orbit matrix, their extended orbit matrices are

$$\left(\begin{array}{c|c} 111100 & 110000 \\ 111010 & 001100 \\ 000110 & 101010 \\ \hline 100101 & 011111 \\ 010011 & 110111 \\ 001000 & 111112 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c|c} 110000 & 111000 \\ 101000 & 100110 \\ 011000 & 010101 \\ \hline 100110 & 011111 \\ 010101 & 101111 \\ 001011 & 111011 \end{array}\right)$$

Isomorph rejection

Prior to the orbit matrix generation phase, we calculate the full automorphism group of the colored starting orbit configuration $\hat{A}_s$:

$$\left(\begin{array}{cc|c} F & \hat{G} & \text{color } 0 \\ \hat{H} & 0 & \text{color } 1 \\ \hline \text{color } 2 & \text{color } 3 & \end{array}\right)$$

Two consecutive extended orbit matrix rows q and $q+1$ ($f+1 \leq q < f+n$) can be lexically ordered if there exists an automorphism of the colored *starting orbit configuration* $\hat{A}_s$ which swaps rows q and $q+1$ and permutes zero or more fixed columns (without permuting any other rows or columns). A similar argument holds for the columns. This lexical ordering technique is a special case of the following more general technique. If we filled i rows of $\hat{X}$ ($f+i$ rows of the extended orbit matrix $\hat{A}$), then we can apply any automorphism of the starting orbit configuration which fixes all rows different from the first $f+i$ rows. If this maps the partial matrix formed by the first $f+i$ rows of the extended orbit matrix to a partial matrix which is lexically larger than the current one, then we argue that the current partial matrix is equivalent to an already generated partial matrix, hence we can prune the search. Another special case of this argument is the ability to lexically order groups of rows with respect to each other. An example will be given in Chapter 6. To limit the amount of automorphisms which need to be checked, we omit automorphisms for which the above mentioned lexical ordering technique would prune. We also set a maximum number of automorphisms to try.

Another effective technique is the use of nauty canonical forms in the early stages of the search. Given a set of isomorphic graphs, *nauty* guarantees to produce the same graph as

000	00	111100	110000	
000	00	111010	001100	color 0
000	00	000110	101010	
000	00	100101	011111	color 1
000	00	010011	110111	
110	10	000000	000000	
110	01	000000	000000	
110	00	000000	000000	color 2
101	10	000000	000000	
011	01	000000	000000	
000	11	000000	000000	
101	01	000000	000000	
100	11	000000	000000	
011	10	000000	000000	color 3
010	11	000000	000000	
001	11	000000	000000	
000	11	000000	000000	

Figure 4.2: Adjacency matrix given $\hat{A}$'s first 5 rows of Figure 4.1.

their canonical form. A partial matrix of the first $f + i$ rows of the extended orbit matrix can be converted into a colored graph, as explained in Chapter 3. An easy way to test for equivalence is to store, for each row, all canonical forms of the graphs (corresponding to the partial matrices) in a hash set into main memory. If we encounter a partial matrix for which its corresponding graph is in the set, then we prune the search. Of course, due to memory limitations [1], we stop using this technique after a certain amount of rows are filled. Turning back to our example: If the first five rows of the extended orbit matrix are those of $\hat{A}$ in Figure 4.1, then the colored graph to give to *nauty* is presented in the colored adjacency matrix of Figure 4.2. So whenever we filled the first five rows, we call nauty to produce the canonical form. If the canonical form was already in our list, then the matrix is equivalent to one produced earlier. So we prune the search. To save memory and make the comparison fast, the graph is stored as a condensed ASCII string using some conversion scheme. So we actually store a hash set of strings.

4.4 Expansion of the orbit matrices

For each orbit matrix solution $\hat{X}$, each of its entries has to be replaced by all possible circulants. There are $\binom{p}{e}$ possible circulants for the orbit matrix entry e, $0 \leq e \leq p$. We do not actually replace each entry with a circulant, but replace it with a *circulant number* which stands for a circulant. We choose the numbers such that the lexically largest circulant (when comparing the first row of the circulants) gets the largest number. A possibility is to define the circulant number as the natural number when interpreting the first row of the

[1] On our department cluster, 1 GB of main memory is the limit

$0 = \begin{pmatrix} 000 \\ 000 \\ 000 \end{pmatrix}$	$1 = \begin{pmatrix} 001 \\ 100 \\ 010 \end{pmatrix}$	$2 = \begin{pmatrix} 010 \\ 001 \\ 100 \end{pmatrix}$	$4 = \begin{pmatrix} 100 \\ 010 \\ 001 \end{pmatrix}$
$7 = \begin{pmatrix} 111 \\ 111 \\ 111 \end{pmatrix}$	$3 = \begin{pmatrix} 011 \\ 101 \\ 110 \end{pmatrix}$	$5 = \begin{pmatrix} 101 \\ 110 \\ 011 \end{pmatrix}$	$6 = \begin{pmatrix} 110 \\ 011 \\ 101 \end{pmatrix}$

Figure 4.3: The circulant numbers for all possible 3×3 circulants.

circulant as an unsigned bit pattern. E.g. for all 3×3 circulants, we define the *circulant numbers* of Figure 4.3. Circulant number 0 is the only possibility for orbit matrix entry 0. Circulant numbers 1, 2 and 4 are the possibilities for orbit matrix entry 1. Circulant numbers 3, 5 and 6 are the possibilities for orbit matrix entry 2. Circulant number 7 is the only possibility for orbit matrix entry 3. We say orbit matrix entry e is extended to circulant c.

Definition 4.4.1 (Circulant matrix) *The circulant matrix $\hat{X}_e$ is a $n \times n'$ matrix, in which entry $(\hat{x_e})_{i,j}$ contains the circulant number of the circulant extension of orbit matrix entry $\hat{x}_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq n'$.*

Definition 4.4.2 (Extended circulant matrix) *The extended circulant matrix $\hat{A}_e$ is defined as*

$$\hat{A}_e = \begin{pmatrix} F & \hat{G} \\ \hat{H} & \hat{X}_e \end{pmatrix}.$$

The following extended circulant matrix corresponds to the incidence matrix A of Figure 4.1.

$$\hat{A}_e = \left(\begin{array}{c|c} 111100 & 110000 \\ 111010 & 001100 \\ 000110 & 101010 \\ \hline 100101 & 022222 \\ 010011 & 220211 \\ 001000 & 221123 \end{array}\right)$$

The constraints on row and column regularity, which arise from the parameter k of the design, are trivially satisfied for all possible circulant extensions. If $p \leq 3$, then the constraints on the scalar product of two rows located within the same row of circulants are also trivially satisfied. For larger p, the intersections between the first row of a row of circulants with half of the rows of another row of circulants have to be checked. For $p = 5$, we determine all possible row and column patterns which meet the λ condition. For small p, we determine all possible intersection patterns between each two rows (resp. columns). Since the row intersection pattern between circulant numbers 6 and 4 is the same as the row intersection pattern between 5 and 1, or between 3 and 2, we store only one of the p possibilities in the pattern list. If storing all these intersection patterns takes too much memory ($p \geq 7$), then we only store all intersections between each possible two circulants.

Isomorph rejection

A first isomorph rejection technique is the fixing of the first non-zero circulant in each row of circulants. We only try the lexically largest out of p possible circulants which can be obtained from one another via a cyclic shift, otherwise we would generate p isomorphic submatrices. E.g. for 3×3 circulants, we only try circulant number 4 for orbit matrix entry 1. Similarly, one circulant can be fixed in each column of circulants.

Prior to the orbit matrix expansion phase, we calculate the full automorphism group of the colored extended orbit matrix $\hat{A}$:

$$\left(\begin{array}{cc|c} F & \hat{G} & \text{color } 0 \\ \hat{H} & \hat{X} & \text{color } 1 \\ \hline \text{color } 2 & \text{color } 3 & \end{array}\right)$$

Two consecutive extended circulant matrix rows q and $q+1$ ($f+1 \leq q < f+n$) can be lexically ordered if there exists an automorphism of the colored *extended orbit matrix* $\hat{A}$ which swaps rows q and $q+1$ and permutes zero or more fixed columns (without permuting any other rows or columns). A similar argument goes for the columns. This lexical ordering technique is a special case of the following more general technique. If we filled i rows of $\hat{X}_e$ ($f+i$ rows of the extended circulant matrix $\hat{A}_e$), then we can apply any automorphism of the colored extended orbit matrix which fixes all rows different from the first $f+i$ rows. We can also apply permutations within a row/column of circulants which turns circulants into other circulants. After applying all these permutations to the partial matrix, we cyclic shift each row and column of circulants such that its first non-zero circulant is the lexically largest out of p possibilities. If this maps the partial matrix formed by the first $f+i$ rows of the extended circulant matrix to a partial matrix which is lexically larger than the current one, then we argue that the current partial matrix is equivalent to an already generated partial matrix, hence we can prune the search. To limit the amount of automorphisms which need to be checked, we omit automorphisms for which the above mentioned lexical ordering technique would prune. We also set a maximum number of automorphisms to try.

When the automorphism group of the colored extended orbit matrix $\hat{A}$ is trivial, the following generation strategy might be better than row order generation. In each step, the next element of the circulant matrix which is selected for binding, is one of those with the smallest number of possible circulant numbers left. So the exhaustive backtracking algorithm is combined with a forward checking method. After each expansion of an entry to a circulant, we determine all possible circulants for each unexpanded entry, thereby maybe reducing the number of possible circulants for that entry.

When the automorphism group of the colored extended orbit matrix $\hat{A}$ is non-trivial, the size of the search space can be reduced by reordering $\hat{A}$ based on its automorphism group. We reorder $\hat{A}$ in a greedy way, such that the submatrix containing the first rows of $\hat{A}$ has a lot of automorphisms. This combines well with the described partial isomorph rejection technique, although an optimal ordering is hard to find.

Once the extended circulant matrix $\hat{A}_e$ is extended, we convert it to the incidence matrix A and perform a full isomorphism test on the 2-(v,k,λ) design.

Of course, each orbit matrix leads to a set of designs. All these sets need to be put together to get the total set of non-isomorphic designs for the given parameters and assumed automorphism.

4.5 Implementation

The *be.ugent.caagt.design.orbit* package contains the implementation of the described technique for the exhaustive generation of designs with an assumed automorphism.

For full documentation we refer to the API documentation, available at *http://users.ugent.be/~jpwinne/phd*. The *GenerateDesigns* class from the *be.ugent.caagt.design.orbit* package contains the executable program.

5 2-(31,15,7), 2-(35,17,8) and 2-(36,15,6) designs with automorphisms of odd prime order, and their related Hadamard matrices and codes

This chapter presents the full classification of Hadamard 2-(31,15,7), Hadamard 2-(35,17,8) and Menon 2-(36,15,6) designs with automorphisms of odd prime order. The results of this chapter were presented at the *Eurocomb 2005* conference, held in Berlin [6] and are submitted to *Journal of Combinatorial Designs* in [7]. A preprint can be downloaded from *http://caagt.ugent.be/preprints*. This work is joint work with I. Bouyukliev.

We also give partial classifications of such designs with automorphisms of order 2. These classifications lead to related Hadamard matrices and self-dual codes. We found 21879 Hadamard matrices of order 32 and 24920 Hadamard matrices of order 36, arising from the classified designs. Remarkably, all constructed Hadamard matrices of order 36 are Hadamard equivalent to a regular Hadamard matrix. From our constructed designs, we obtained 786 doubly-even $[72, 36, 12]$ codes, which are the best known self-dual codes of this length until now.

5.1 Introduction

A *Hadamard matrix* H of order n is an $n \times n$ ± 1 matrix satisfying $HH^t = nI$. Two Hadamard matrices H_1 and H_2 are *Hadamard equivalent* if H_2 can be obtained from H_1 by a sequence of row permutations, column permutations, row negations and column negations. An automorphism of a Hadamard matrix is an equivalence with itself. A *normalized* Hadamard matrix has an all-ones first row and column. A *regular* Hadamard matrix has constant row and column sums.

Hadamard matrices have been completely classified up to order 28. For higher orders, only partial classifications are known. Lin, Wallis and Zhu [30] found 66104 inequivalent Hadamard matrices of order 32. Extensive results on order 32 appear in [31] and [32]. In the beginning of our work, at least 235 inequivalent Hadamard matrices of order 36 were known, see [20], [22] and [41]. However, during this work, astronomical bounds for the

number of Hadamard matrices of order 32 and 36 were obtained [38], in which the author also used our matrices to obtain these bounds. Full classification of Hadamard matrices of order 32 and 36 is improbable. A motivation for our research is to determine the numbers of Hadamard matrices of order 32 and 36 which have symmetry.

Hadamard matrices are related to self-dual codes, as described in [47] and [43]. The existence of an extremal self-dual $[72, 36, 16]$ code is an important open problem in coding theory [42]. As shown in [15], a code with such parameters can be obtained from Hadamard matrices of order 36 with a trivial automorphism group or with automorphisms of order 2, 3, 5 or 7. This is another motivation for our research.

To obtain the incidence matrix of a symmetric 2-$(4m-1, 2m-1, m-1)$ *Hadamard design*, delete the first row and column of a normalized Hadamard matrix of order $4m$ and replace -1 with 0. The choice of which row and column to normalize is not unique, i.e. non-isomorphic Hadamard designs can be obtained from one Hadamard matrix. However, only one Hadamard matrix can be obtained from a Hadamard design by reversing the above procedure.

A *Menon design* [14] is a 2-$(4u^2, 2u^2 \pm u, u^2 \pm u)$ design. A Menon 2-(36,15,6) design $(u = 3)$ exists if and only if a regular Hadamard matrix of order 36 exists. They are easily obtained from one another by replacing 0 with -1 (and vice versa). Furthermore, we can obtain 2-(35,17,8) designs from a regular Hadamard matrix of order 36 by the method described above. We use this property to check our classification results.

Our main result is the classification of Hadamard matrices of order 32 and 36 corresponding to all Hadamard and Menon designs with automorphisms of odd prime order. We also made a partial classification of Hadamard matrices of order 32 and 36 corresponding to a partial classification of Hadamard and Menon designs with automorphisms of order 2. From the Hadamard and Menon designs of order 36, we obtained doubly-even $[72, 36, 12]$ codes.

A Hadamard or Menon design is trivially converted to its corresponding Hadamard matrix H of order m. In order to test the obtained Hadamard matrices for Hadamard equivalence, we convert each Hadamard matrix H to an integer matrix H^* of order $2m$, defined as

$$H^* = \begin{pmatrix} H & -H \\ -H & H \end{pmatrix},$$

and use the property that two Hadamard matrices H_1 and H_2 are Hadamard equivalent if and only if the integer matrices H_1^* and H_2^* are isomorphic, see [35]. The order of the full automorphism group of a Hadamard matrix H is the same as the order of the full automorphism group of H^*. We test a set of H^* for equivalence by converting these integer matrices to bipartite graphs.

It is easy to see that any automorphism of a Hadamard design gives rise to an automorphism of the related Hadamard matrix which fixes the added all-one row and column. Also, any automorphism of a Menon design is an automorphism of the related Hadamard matrix. The non-existence of 2-(35,17,8) with an automorphism of order p and f fixed points/blocks implies the non-existence of 2-(36,15,6) with an automorphism of order p and $f+1$ fixed points/blocks. To see this, suppose a 2-(36,15,6) with an automorphism of order p and $f+1$ fixed points/blocks exists. Convert the related regular Hadamard matrix to a 2-(35,17,8) design by normalizing and removing one fixed point and one fixed block. The

obtained 2-(35,17,8) design has an automorphism of order p and f fixed points/blocks. An example of this procedure is shown in Table 5.1. The regular Hadamard matrices obtained from 2-(36,15,6) with an automorphism of order p and $f+1$ fixed points should be a subset of those obtained from 2-(35,17,8) with an automorphism of order p and f fixed points. This is a good check for the correctness of our results.

In Section 5.2, we present the classifications of 2-$(31, 15, 7)$, 2-$(35, 17, 8)$ and 2-$(36, 15, 6)$ designs. Sections 5.3 and 5.4 present the obtained results for Hadamard matrices and self-dual codes.

	2-(36,15,6)	$p = 3,\ f = 3$		2-(35,17,8)	$p = 3,\ f = 2$
111	111 111 111 111	000 000 000 000 000 000 000	+++	+++ +++ +++ +++	--- --- --- --- --- --- ---
111	111 000 000 000	111 111 111 000 000 000 000	+11	111 000 000 000	000 000 000 111 111 111 111
111	111 000 000 000	000 000 000 111 111 111 000	+11	111 000 000 000	111 111 111 000 000 000 111
111	000 100 100 100	100 100 100 100 100 100 111	+11	000 100 100 100	011 011 011 011 011 011 000
111	000 010 010 010	010 010 010 010 010 010 111	+11	000 010 010 010	101 101 101 101 101 101 000
111	000 001 001 001	001 001 001 001 001 001 111	+11	000 001 001 001	110 110 110 110 110 110 000
100	100 111 100 000	110 100 010 011 001 001 100	+00	100 111 100 000	001 011 101 100 110 110 011
100	010 111 010 000	011 010 001 101 100 100 010	+00	010 111 010 000	100 101 110 010 011 011 101
100	001 111 001 000	101 001 100 110 010 010 001	+00	001 111 001 000	010 110 011 001 101 101 110
				unchanged	negated
100	100 100 110 001	000 011 101 000 110 011 100	+00	100 100 110 001	111 100 010 111 001 100 011
100	010 010 011 100	000 101 110 000 011 101 010	+00	010 010 011 100	111 010 001 111 100 010 101
100	001 001 101 010	000 110 011 000 101 110 001	+00	001 001 101 010	111 001 100 111 010 001 110
100	100 000 001 111	101 010 010 101 010 100 100	+00	100 000 001 111	010 101 101 010 101 011 011
100	010 000 100 111	110 001 001 110 001 010 010	+00	010 000 100 111	001 110 110 001 110 101 101
100	001 000 010 111	011 100 100 011 100 001 001	+00	001 000 010 111	100 011 011 100 011 110 110
010	100 101 000 011	011 100 100 000 011 110 010	-01	011 010 111 100	011 100 100 000 011 110 010
010	010 110 000 101	101 010 010 000 101 011 001	-01	101 001 111 010	101 010 010 000 101 011 001
010	001 011 000 110	110 001 001 000 110 101 100	-01	110 100 111 001	110 001 001 000 110 101 100
010	100 010 101 010	001 111 000 110 100 001 010	-01	011 101 010 101	001 111 000 110 100 001 010
010	010 001 110 001	100 111 000 011 010 100 001	-01	101 110 001 110	100 111 000 011 010 100 001
010	001 100 011 100	010 111 000 101 001 010 100	-01	110 011 100 011	010 111 000 101 001 010 100
010	100 001 011 100	100 000 111 011 100 010 010	-01	011 110 100 011	100 000 111 011 100 010 010
010	010 100 101 010	010 000 111 101 010 001 001	-01	101 011 010 101	010 000 111 101 010 001 001
010	001 010 110 001	001 000 111 110 001 100 100	-01	110 101 001 110	001 000 111 110 001 100 100
				negated	unchanged
001	100 110 000 110	000 011 101 011 001 100 001	-10	011 001 111 001	000 011 101 011 001 100 001
001	010 011 000 011	000 101 110 101 100 010 100	-10	101 100 111 100	000 101 110 101 100 010 100
001	001 101 000 101	000 110 011 110 010 001 010	-10	110 010 111 010	000 110 011 110 010 001 010
001	100 010 011 001	110 100 001 100 111 000 001	-10	011 101 100 110	110 100 001 100 111 000 001
001	010 001 101 100	011 010 100 010 111 000 100	-10	101 110 010 011	011 010 100 010 111 000 100
001	001 100 110 010	101 001 010 001 111 000 010	-10	110 011 001 101	101 001 010 001 111 000 010
001	100 001 110 100	011 001 010 100 000 111 001	-10	011 110 001 011	011 001 010 100 000 111 001
001	010 100 011 010	101 100 001 010 000 111 100	-10	101 011 100 101	101 100 001 010 000 111 100
001	001 010 101 001	110 010 100 001 000 111 010	-10	110 101 010 110	110 010 100 001 000 111 010
000	111 100 001 001	010 001 010 010 100 100 111	-11	000 011 110 110	010 001 010 010 100 100 111
000	111 010 100 100	001 100 001 001 010 010 111	-11	000 101 011 011	001 100 001 001 010 010 111
000	111 001 010 010	100 010 100 100 001 001 111	-11	000 110 101 101	100 010 100 100 001 001 111

Table 5.1: On the left side is the incidence matrix of a 2-(36,15,6) design with an automorphism of order 3 and 3 fixed points/blocks. The first 3 rows/columns are fixed. The other rows/columns are structured into circulants. Replacing 0 with -1 and normalizing and removing the first row and column gives, after replacing -1 with 0, the matrix on the right side. The rows and columns which were negated are marked with a $-$, the others are marked with a $+$. The matrix on the right is the incidence matrix of the related 2-(35,17,8) design with an automorphism of order 3 and 2 fixed points/blocks. Note that two of the four non-fixed matrix parts are negated, i.e. all entries for which either its row or (exclusive) its column were negated.

p	f	# 2-(31,15,7)	Non-existence proof
2	1	0	By exhaustive generation: No orbit matrices for the unique fixed configuration were found.
	3	57536	
	≥ 5	?	Too many for $f = 5,7,9,11,13,15$.
3	1	16350	
	4	0	$U = \{4,5\}$. $W = \{1,2\}$. $X_4 = 126$. $X_5 = 126$. $u = 5$ impossible by (c), so $U = \{4\}$. $I_4 = \{\}$. By (d), $B_4 \leq 1$.
	7	205112	
5	1	274	
	6	0	$U = \{2,3\}$. $W = \{1\}$. $X_2 = 10$. $X_3 = 10$. $u = 3$ impossible by (c), so $U = \{2\}$. $I_2 = \{\}$. By (d), $B_2 \leq 1$.
	11	0	$U = \{1,2,3\}$. $W = \{0,1\}$. $X_1 = 4$. $X_2 = 6$. $X_3 = 4$. $u = 3$ impossible by (c), so $U = \{1,2\}$. $I_1 = \{\}$. $I_2 = \{2\}$. By (a), $B_2 \leq 6$. By (d), $B_1 \leq 1$.
7	3	98	
	10	0	$U = \{1,2\}$. $W = \{0,1\}$. $X_1 = 3$. $X_2 = 3$. $I_1 = \{7\}$. $I_2 = \{0\}$. By (a), $B_2 \leq 3$. By (b), $B_1 \leq 3$.
11	9	0	$U = \{1\}$. $W = \{0\}$. $X_1 = 2$. $u = 1$ impossible by (c), so $U = \{\}$.
13	5	0	$U = \{1\}$. $W = \{\}$. $X_1 = 2$. $u = 1$ impossible by (c), so $U = \{\}$.

Table 5.2: Number of 2-(31,15,7) designs with an automorphism of prime order p and f fixed points. Where appropriate, the counting argument proving the non-existence of such designs is given. Otherwise the results are obtained by exhaustive generation.

5.2 Partial classification of 2-(31,15,7), 2-(35,17,8) and 2-(36,15,6) designs

First we determine all possible prime orders p and all possible values for f for the cases of 2-(31,15,7), 2-(35,17,8) and 2-(36,15,6) designs. If a 2-(v,k,λ) design possesses an automorphism of prime order p, then $p \leq k$ or $p \mid v$, otherwise all points and blocks would be fixed. Clearly $(v - f) \mod p = 0$ and $f \leq v/2$, see [11]. Tonchev (Lemma 1.8.1 [46]) proved that an automorphism of order 3 of a 2-(v,k,λ) design fixes at most $b - 3(r - \lambda)$ blocks. Therefore if $p = 3$, then $f \leq 7$ for 2-(31,15,7), $f \leq 8$ for 2-(35,17,8) and $f \leq 9$ for 2-(36,15,6).

The possible prime divisors p for 2-(31,15,7) or 2-(36,15,6) are 2, 3, 5, 7, 11 and 13. The possible prime divisors p for 2-(35,17,8) are 2, 3, 5, 7, 11, 13 and 17. The first two columns of Tables 5.2, 5.3 and 5.4 summarize all possible prime orders p and corresponding possible numbers f of fixed points for 2-(31,15,7), 2-(35,17,8) and 2-(36,15,6), respectively.

p	f	# 2-(35,17,8)	Non-existence proof
2	1	0	By exhaustive generation: No orbit matrices for the unique fixed configuration were found.
	3	111098	
	5	237058	
	≥ 7	?	Too many for f = 7,9,11,13,15,17.
3	2	63635	
	5	3698	
	8	14692	
5	0	12	
	5	0	$U = \{3\}$. $W = \{1\}$. $X_3 = 20$. $u = 3$ impossible by (c), so $U = \{\}$.
	10	0	$U = \{2,3\}$. $W = \{0,1\}$. $X_2 = 10$. $X_3 = 10$. $u = 3$ impossible by (c), so $U = \{2\}$. $I_2 = \{\}$. By (d), $B_2 \leq 1$.
	15	0	$U = \{1,2,3\}$. $W = \{0,1\}$. $X_1 = 4$. $X_2 = 6$. $X_3 = 4$. $u = 3$ impossible by (c), so $U = \{1,2\}$. $I_1 = \{\}$. $I_2 = \{3\}$. By (a), $B_2 \leq 6$. By (d), $B_1 \leq 1$.
7	0	4	
	7	0	$U = \{2\}$. $W = \{1\}$. $X_2 = 6$. $I_2 = \{1\}$. By (a), $B_2 \leq 6$.
	14	0	$U = \{1,2\}$. $W = \{0,1\}$. $X_1 = 3$. $X_2 = 3$. $I_1 = \{8\}$. $I_2 = \{1\}$. By (a), $B_2 \leq 3$. By (b), $B_1 \leq 3$.
11	2	0	$U = \{\}$. $W = \{\}$.
	13	0	$U = \{1\}$. $W = \{0\}$. $X_1 = 2$. $u = 1$ impossible by (c), so $U = \{\}$.
13	9	0	$U = \{1\}$. $W = \{0\}$. $X_1 = 2$. $u = 1$ impossible by (c), so $U = \{\}$.
17	1	11	See [44]

Table 5.3: Number of 2-(35,17,8) designs with an automorphism of prime order p and f fixed points. Where appropriate, the counting argument proving the non-existence of such designs is given. Otherwise the results are obtained by exhaustive generation.

p	f	# 2-(36,15,6)	Non-existence proof
2	2	0	
	4	170648	
	≥ 6	?	Too many for f = 6,8,10,12,14,16.
3	0	58720	
	3	51824	
	6	2368	
	9	909	
5	1	38	
	6	0	
	11	0	
	16	0	
7	1	4	
	8	0	
	15	0	
11	3	0	$U = \{\}$. $W = \{\}$.
	14	0	$U = \{1\}$. $W = \{0\}$. $X_1 = 2$. $u = 1$ impossible by (c), so $U = \{\}$.
13	10	0	$U = \{1\}$. $W = \{0\}$. $X_1 = 2$. $u = 1$ impossible by (c), so $U = \{\}$.

Table 5.4: Number of 2-(36,15,6) designs with an automorphism of prime order p and f fixed points. Where appropriate, the counting argument proving the non-existence of such designs is given. Otherwise the results are obtained by exhaustive generation. The non-existence can also be derived from the non-existence of the corresponding 2-(35,17,8) design.

We will reject the existence of most cases by a simple counting argument. Refer to equations (4.1)-(4.8) on page 70 for an overview of the used symbols. For each possible value of p and f, we determine the sets U and W defined as follows:

- A fixed point is incident with pu non-fixed blocks and $k - pu$ fixed blocks, where the feasible u values are in the set

$$U = \{u \in \mathbb{N} \mid 0 \leq pu \leq \min(k, v - f) \ \wedge \ 0 \leq k - pu \leq \min(k, f)\}. \tag{5.1}$$

  Similarly, a fixed block is incident with pu non-fixed points and $k - pu$ fixed points.

- Each pair of fixed points is incident with pw non-fixed blocks and $\lambda - pw$ fixed blocks, where the feasible w values are in the set

$$W = \{w \in \mathbb{N} \mid 0 \leq pw \leq \min(\lambda, v - f) \ \wedge \ 0 \leq \lambda - pw \leq \min(\lambda, f)\}. \tag{5.2}$$

  Similarly, each pair of fixed blocks is incident with pw non-fixed points and $\lambda - pw$ fixed points.

Let X_u, $u \in U$, denote the number of different ways in which those pu non-fixed blocks can be incident with a fixed point, clearly

$$X_u = \binom{n}{u}. \tag{5.3}$$

Consider two fixed points which are both incident with $k - pu$ fixed blocks. The possible values for the number of fixed blocks that contain the point pair belong to

$$I_u = \{m \in \mathbb{N} \mid \max(0, 2(k - pu) - f) \leq m \leq k - pu \ \wedge \exists w \in W : m = \lambda - pw\}. \tag{5.4}$$

For each $u \in U$, we calculate X_u and I_u. Let B_u, $u \in U$, denote the maximum number of fixed points which could be incident with pu non-fixed blocks and $k - pu$ fixed blocks. The following properties hold:

(a) If $X_u < f$ and $pu > \lambda$, then $B_u \leq X_u$.
Indeed, if B_u would exceed X_u, there must exist two points whose scalar product is pu, so B_u exceeds λ.

(b) If $X_u < f$ and $2k - pu - \lambda > f$, then $B_u \leq X_u$.
Consider two fixed points which are incident with the same pu non-fixed blocks, then $\lambda - pu$ fixed blocks must be incident with both fixed points. Each of both fixed points is in $k - pu$ fixed blocks. So by a simple counting argument, if $2(k - pu) - f > \lambda - pu$, we will have too many intersections. Hence B_u cannot exceed X_u.

(c) If $f > 1$ and $\forall w \in W : \lambda - pw > k - pu$, then $B_u = 0$.
Clearly we cannot have scalar product $\lambda - pw$ if we only have $k - pu$ ones.

(d) If $\forall w \in W : \lambda - pw \notin I_u$, then $B_u \leq 1$.
Two such type u points cannot meet the intersection pattern, so there is at most one type u point.

If applying the above properties leads to the conclusion that $\sum_{u \in U} B_u < f$, the setup is impossible and no 2-(v,k,λ) designs with an automorphism of order p and f fixed points exist. Otherwise exhaustive generation is needed to find all such designs or prove their non-existence.

As an example we next give a detailed calculation for the case of 2-(31,15,7) designs with $p = 3$ and $f = 4$. The set $U = \{u \in \mathbb{N} \mid 0 \leq 3u \leq 15 \wedge 0 \leq 15-3u \leq 4\} = \{4, 5\}$, meaning a fixed point is either incident with 12 non-fixed blocks and 3 fixed blocks, or with 15 non-fixed blocks and 0 fixed blocks. The set $W = \{w \in \mathbb{N} \mid 0 \leq 3w \leq 7 \ \wedge \ 0 \leq 7 - 3w \leq 4\} = \{1, 2\}$, meaning each pair of fixed points is either incident with 3 non-fixed blocks and 4 fixed blocks, or with 6 non-fixed blocks and 1 fixed block. Applying (c) with $u = 5$ gives $B_5 = 0$, so $U = \{4\}$. $I_4 = \{m \in \mathbb{N} \mid 2 \leq m \leq 3 \wedge \exists w \in \{1, 2\} : m = 7 - 3w\} = \{\}$. Applying (d) with $u = 4$ gives $B_4 \leq 1$. This way $B_4 + B_5 \leq 1$. We conclude that no 2-(31,15,7) designs with $p = 3$ and $f = 4$ exist.

Similar non-existence proofs for 2-(31,15,7), 2-(35,17,8) and 2-(36,15,6) designs for certain combinations of p and f are summarized in Tables 5.2, 5.3 and 5.4. For the cases where the counting argument does not lead to a conclusion, we present the number of non-isomorphic designs found by our generation program. The non-existence of cases of 2-(36,15,6) can also be derived from the non-existence of the corresponding 2-(35,17,8) design with one fixed point/block less, as explained in the introduction.

For 2-(31,15,7), 2-(35,17,8) and 2-(36,15,6) we generated all designs with an automorphism of order 3, 5 and 7. However, for $p = 2$ we only classified all 2-(31,15,7) with 1 or 3 fixed points, all 2-(35,17,8) with 1, 3 or 5 fixed points and all 2-(36,15,6) with 2 or 4 fixed points. For larger values of f, we failed to generate all designs with an automorphism of order 2 and f fixed points, since these isomorphism classes are too large to enumerate. The cases where $p = 11, 13$ can be eliminated by a counting argument, hence no exhaustive search is needed. For the case $p = 17$ for 2-(35,17,8), the 11 designs and corresponding 11 Hadamard matrices of order 36 (available in [40]) were constructed previously by Tonchev [44]. In order to be sure about our computer results, I. Bouyukliev made an independent implementation for parts of this classification. Tables 5.5, 5.6 and 5.7 give the running time and the number of recursive calls of each individual phase for all constructed designs [1].

[1] Using a 1.8 GHz AMD PC running Linux.

p	f	Fixed	Orbit	Expand	Iso	Total	
2	3	367	1.0e6	3.5e7	-	3.6e7	Calls
		0.4	49	44606	9447	54103	Seconds
3	1	36	96611	1.9e8	-	1.9e8	Calls
		0.3	4	6910	310	7226	Seconds
3	7	9257	18591	5.4e7	-	5.4e7	Calls
		115	270	21942	10090	32418	Seconds
5	1	16	271	3.0e6	-	3.0e6	Calls
		0.3	0.5	92	3	96	Seconds
7	3	34	50	97996	-	98080	Calls
		0.3	0.6	16	2	19	Seconds

Table 5.5: Number of recursive calls and running time (in seconds) of each individual phase for certain values of p and f for 2-(31,15,7) designs. "Fixed" refers to the generation of the fixed parts. "Orbit" refers to the orbit matrix generation phase. "Expand" refers to the orbit matrix expansion phase. "Iso" refers to the final isomorphism test on all obtained matrices. "Total" refers to the sum of all phases. Note that we do not list the number of recursive calls for "Iso".

p	f	Fixed	Orbit	Expand	Iso	Total	
2	3	253	1.4e8	2.2e9	-	2.3e9	Calls
		0.9 s	2134 s	56391 s	4652 s	63175 s	Seconds
2	5	31027	3.7e7	1.1e9	-	1.1e9	Calls
		1076 s	905 s	45130 s	10742 s	57855 s	Seconds
3	2	71	3.0e5	3.1e9	-	3.1e9	Calls
		0.3 s	13 s	72297 s	1744 s	74054 s	Seconds
3	5	553	6050	1.4e8	-	1.4e8	Calls
		0.5 s	12 s	3932 s	107 s	4051 s	Seconds
3	8	6403	208	1.1e7	-	1.1e7	Calls
		0.5 s	0.7 s	939 s	521 s	1461 s	Seconds
5	0	0	1471	1.1e7	-	1.1e7	Calls
		0 s	0.5 s	317 s	0.2 s	318 s	Seconds
7	0	0	101	1.3e6	-	1.3e6	Calls
		0 s	0.4 s	86 s	0.05 s	87 s	Seconds

Table 5.6: Number of recursive calls and running time (in seconds) of each individual phase for certain values of p and f for 2-(35,17,8) designs.

p	f	Fixed	Orbit	Expand	Iso	Total	
2	4	1710	8.4e7	2.7e9	-	2.8e9	Calls
		2	1461	55811	15735	73010	Seconds
3	0	0	2.0e8	1.1e10	1.8e6	1.1e10	Calls
		0	6174	270595	1781	278552	Seconds
3	3	227	1.4e5	2.8e9	-	2.8e9	Calls
		0.4	8	62359	3138	65506	Seconds
3	6	2181	3189	4.8e7	-	4.8e7	Calls
		0.8	8	1304	113	1426	Seconds
3	9	34538	190	1.5e6	-	1.5e6	Calls
		0.6	1	162	79	243	Seconds
5	1	20	668	5.1e6	-	5.1e6	Calls
		0.3	0.6	162	1	164	Seconds
7	1	12	45	1.3e6	-	1.3e6	Calls
		0.6	0.5	83	0.1	84	Seconds

Table 5.7: Number of recursive calls and running time (in seconds) of each individual phase for certain values of p and f for 2-(36,15,6) designs.

5.3 Results for Hadamard matrices

We look further at the connection between the automorphism group of a design and its corresponding Hadamard matrix. As mentioned, any automorphism of a Hadamard design gives rise to an automorphism of the related Hadamard matrix which fixes the added all-one row and column. But the opposite is not true.

Theorem 5.3.1 (Theorem 1.5.1 [46]) *Let H be a Hadamard matrix of order $n \geq 4$ and $p > 2$ a prime divisor of the order of the full automorphism group of H. Then we have at least one of the following cases: (a) p divides n; (b) p divides $n-1$; (c) $p \leq \frac{n}{2}-1$. Moreover, if p does not divide n then p is the order of an automorphism of the corresponding Hadamard 2-$(n-1, n/2-1, n/4-1)$ design.*

Using this theorem, we conclude that we can construct Hadamard matrices of orders 32 and 36, arising from all Hadamard designs having an automorphism of odd prime order, from the corresponding 2-designs, except Hadamard matrices of order 36 for which the only automorphisms of odd prime order are of order 3 without fixed points. However, we can obtain all regular Hadamard matrices of order 36 with an automorphism of order 3 without fixed points from the Menon designs.

Table 5.8 lists the number of non-isomorphic 2-(31,15,7) designs and the number of non-equivalent Hadamard matrices of order 32 for given p and f. Tables 5.9 and 5.10 list similar results for 2-(35,17,8) and 2-(36,15,6) designs and related Hadamard matrices of order 36, respectively. When several values of f are possible for a given p, we also give the total number of non-isomorphic Hadamard matrices which have an automorphism of order p. Note that, e.g., a 2-(31,15,7) design might have an automorphism of order 3

p	f	# 2-(31,15,7)	# HM order 32
2	3	57536	6960
3	1	16350	5478
	7	205112	9603
	all	220900	14824
5	1	274	125
7	3	98	32
Total		278744	21879

Table 5.8: Number of 2-(31,15,7) designs with an automorphism of order p and f fixed points and number of related Hadamard matrices of order 32.

p	f	# 2-(35,17,8)	# HM order 36
2	3	111098	8123
	5	237058	6719
	all	347407	14640
3	2	63635	7238
	5	3698	158
	8	14692	260
	all	81937	7631
5	0	12	12
7	0	4	4
17	1	11	11
Total		428502	21916

Table 5.9: Number of 2-(35,17,8) designs with an automorphism of order p and f fixed points and number of related Hadamard matrices of order 36.

p	f	# 2-(36,15,6)	# HM order 36
2	4	170648	8123
3	0	58720	3189
	3	51824	7238
	6	2368	158
	9	909	260
	all	113259	10700
5	1	38	12
7	1	4	4
Total		282931	18503

Table 5.10: Number of Menon 2-(36,15,6) designs with an automorphism of order p and f fixed points and number of related Hadamard matrices of order 36.

with both 1 and 7 fixed points. This explains why the total number of non-equivalent designs (and Hadamard matrices) of order p is typically less than the sum of the numbers of order p over all possible f values. The Hadamard matrices obtained from 2-(35,17,8) with an automorphism of order p with f fixed points are the same as the ones obtained from 2-(36,15,6) with an automorphism of order p with $f+1$ fixed points, except for the matrices obtained from 2-(36,15,6) with an automorphism of order 3 without fixed points. Of the 3189 matrices obtained from 2-(36,15,6) with $p = 3$ and $f = 0$, there are 3004 different from all other obtained Hadamard matrices. In total we have 24920 Hadamard matrices of order 36. All obtained Hadamard matrices from 2-(35,17,8) with an automorphism of odd prime order are equivalent to a regular Hadamard matrix. Of the 11 Hadamard matrices with an automorphism of order 17 (which where constructed in [44]), we found, by a greedy computer search, that these are also Hadamard equivalent to a regular Hadamard matrix. In Table 5.11 we give a Hadamard matrix of order 17 from [44] together with a Hadamard equivalent regular Hadamard matrix. Maybe this suggests all Hadamard matrices of order 36 are Hadamard equivalent to a regular Hadamard matrix.

Tables 5.12 and 5.13 list the automorphism group sizes for the Hadamard matrices corresponding to 2-(31,15,7) and 2-(35,17,8) Hadamard designs with an automorphism of odd prime order. Table 5.14 lists the automorphism group sizes for the 3189 Hadamard matrices corresponding to 2-(36,15,6) Menon designs with an automorphism of order 3 and 0 fixed points.

To check the correctness of our results, we performed the following tests. As an example: Consider the 158 Hadamard matrices of order 36 which were obtained from the 3698 2-(35,17,8) designs which have an automorphism of order 3 with 5 fixed points/blocks.

- We normalized all 158 Hadamard matrices in all 36∗36 possible ways, yielding 158∗36∗36 = 204768 (not all non-isomorphic) designs. This set of designs contains 3713 non-isomorphic designs with automorphisms of order 3, and 3698 non-isomorphic designs with automorphisms of order 3 with 5 fixed points/blocks.

- We converted all 3698 designs to 3698 Hadamard matrices, without performing a Hadamard equivalence test on the obtained matrices. These 3698 Hadamard matrices were converted back to designs by normalizing all possible combinations of the fixed points/blocks, thus $6 * 6 = 36$ normalizations for each Hadamard matrix (the 5 fixed points/blocks from the design and the added fixed point/block). So we get 36∗3698 = 133128 designs, yielding 3698 non-isomorphic designs.

Partial results of this work were announced at the *European Conference on Combinatorics* 2005 [6], this led to private communication with W. P. Orrick, who used our results for his switching operations for Hadamard matrices [38], which lead to millions of Hadamard matrices of order 32 and 36.

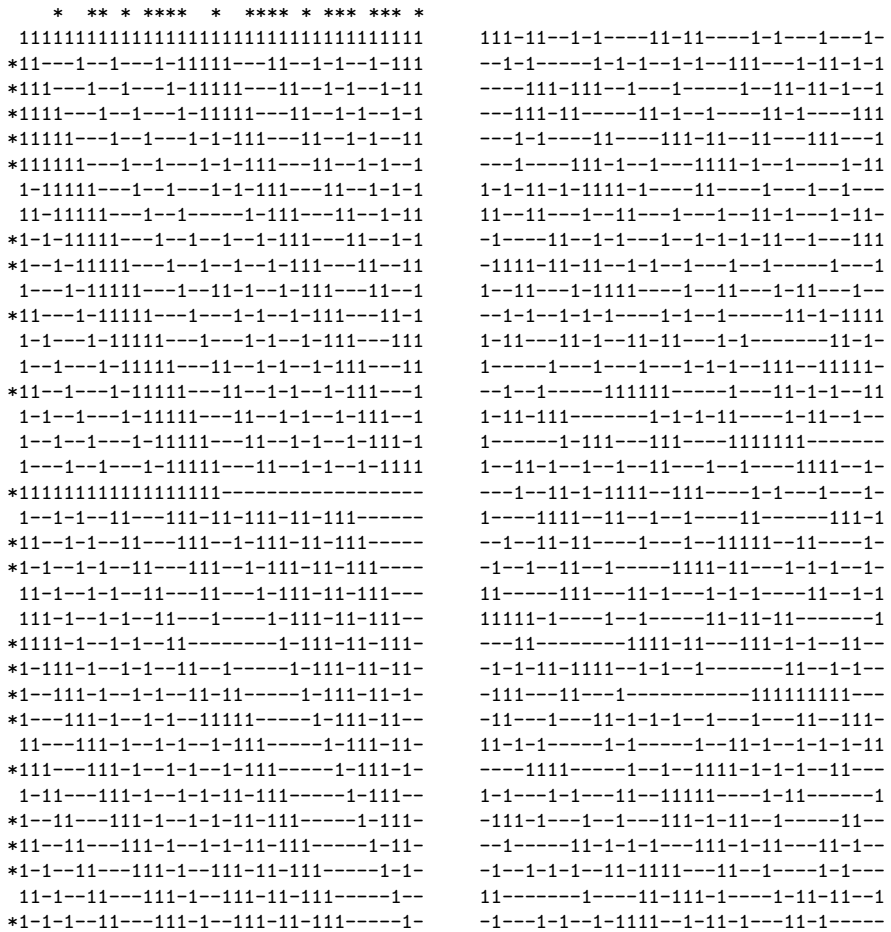


Table 5.11: On the left side we have a Hadamard matrix of order 17 from [44] and on the right side a Hadamard equivalent regular Hadamard matrix obtained by negating the rows and columns which are marked with a star (*).

$\|Aut(H)\|$	Total	$\|Aut(H)\|$	Total	$\|Aut(H)\|$	Total
6	3066	336	5	14336	2
10	19	384	619	24576	41
12	3315	448	4	29760	1
18	32	576	12	36864	2
20	44	768	267	49152	8
24	2433	1152	15	73728	2
36	82	1344	4	98304	8
40	9	1536	155	122880	3
42	6	2304	4	172032	1
48	2320	2688	6	196608	6
56	4	3072	134	294912	2
60	9	3840	2	393216	6
72	31	4608	8	516096	4
96	1141	6144	85	589824	5
112	6	7168	2	688128	4
120	20	8064	2	786432	5
144	44	9216	2	917504	1
192	850	10240	8	16515072	2
288	6	10752	4	18874368	1
320	9	12288	43	20478689280	1
				all	14932

Table 5.12: Order of the full automorphism group and number of non-isomorphic Hadamard matrices of order 32 arising from Hadamard 2-(31,15,7) designs with an automorphism of odd prime order.

$\|Aut(H)\|$	Total	$\|Aut(H)\|$	Total	$\|Aut(H)\|$	Total
6	6937	120	1	840	1
10	4	144	4	972	1
12	369	162	1	1152	1
18	25	192	7	1296	2
20	2	216	7	1728	1
24	130	288	4	1944	1
36	34	320	1	2304	1
42	1	324	5	3072	1
48	42	336	1	3456	1
54	15	384	4	3888	1
68	10	432	4	8640	1
72	7	480	1	19584	1
96	6	648	1	31104	1
108	9	768	3	2903040	1
				all	7650

Table 5.13: Order of the full automorphism group and number of non-isomorphic Hadamard matrices of order 36 arising from Hadamard 2-(35,17,8) designs with an automorphism of odd prime order.

$\|Aut(H)\|$	Total	$\|Aut(H)\|$	Total	$\|Aut(H)\|$	Total
6	2040	144	8	1152	2
12	636	162	1	1296	2
18	109	192	1	1728	1
24	190	216	7	1944	1
36	86	288	1	3456	1
48	27	324	5	3888	1
54	15	432	4	8640	1
72	31	480	1	19584	1
96	4	648	1	31104	1
108	9	972	1	2903040	1
				all	3189

Table 5.14: Order of the full automorphism group and number of non-isomorphic Hadamard matrices of order 36 arising from Menon 2-(36,15,6) designs with an automorphism of order 3 and 0 fixed points.

5.4 Results for codes

As mentioned before, an extremal self-dual $[72, 36, 16]$ code can be obtained from Hadamard matrices of order 36 with a trivial automorphism group or with automorphisms of order 2, 3, 5 or 7 [15].

We say that a permutation of prime order p is of type p-(c, f) if it has exactly c p-cycles and f fixed points in its factorization into disjoint cycles. The permutation $\sigma \in S_n$ is an automorphism of a binary linear code C if $C = \sigma(C)$. The set of all automorphisms of C forms its automorphism group $Aut(C)$. It has been proved [8, 9] that, if σ is an automorphism of a putative doubly-even $[72, 36, 16]$ code of prime order then σ is of type 7-$(10, 2)$, 5-$(14, 2)$, 3-$(24, 0)$, or 2-$(36, 0)$. We have the following theorem:

Theorem 5.4.1 (Theorem 2.2.1 [46]) *Let A be the incidence matrix of a symmetric 2-(v, k, λ) design with $k - \lambda$ odd. Then:*

- *if $k \equiv 3$ (mod 4), then the code with generator matrix (I, A) is a doubly-even self-dual $[2v, v]$ code.*
- *if $k \equiv 2$ (mod 4), then the code with generator matrix*

$$\begin{pmatrix} & 1 & \dots & 1 & 0 \\ & & & & 1 \\ I_{v+1} & & A & & \dots \\ & & & & 1 \end{pmatrix},$$

is a doubly-even self-dual $[2v + 2, v + 1]$ code.

We know that the automorphism group of such a symmetric design (2-$(35, 18, 9)$ in our case, which is complement to 2-$(35, 17, 8)$) is a subgroup of the automorphism group of the corresponding code (doubly-even self-dual $[72, 36]$ code in our case). As we consider 2-$(35, 17, 8)$ designs with automorphisms of order 7 with 10 cycles, and of order 5 with 14 cycles, it is possible that these designs yield to doubly-even self-dual $[72, 36, 16]$ codes. We checked all doubly-even codes obtained from these designs, but all of them have minimum distance at most 12. Actually, using all the constructed designs, we have obtained 786 doubly-even $[72, 36, 12]$ codes, which are the best known self-dual codes of this length until now. They have 26 different orders of their automorphism groups and 79 different weight enumerators. Menon 2-$(36, 15, 6)$ designs with automorphisms of order 7 with 10 cycles, of order 5 with 14 cycles, of order 3 with 24 cycles and of order 2 with 36 cycles could also yield to such codes. We managed to enumerate all such Menon designs for orders 3, 5 and 7. We checked all doubly-even codes obtained from all generated Menon designs, but all of them have minimum distance at most 12.

6 A search for $pg(6,6,4)$

In this chapter we apply the local approach method of Chapter 4 to search for the existence of the partial geometry $pg(6,6,4)$ which has an automorphism of order 3 with 7 fixed points and 7 fixed lines. Unfortunately, it turns out that no such partial geometry exists. The existence of $pg(6,6,4)$ in general remains open. This work was presented at Combinatorics 2004 [49]. This work is joint work with S. Topalova. We each made an independent implementation.

In Section 6.1 we give the definition of a partial geometry, and restate it in terms of the incidence matrix of $pg(6,6,4)$. In Section 6.2 we determine the possible starting configurations when we assume the mentioned automorphism. Section 6.3 describes the problem specific pruning techniques, which are derived from the starting configuration.

6.1 Partial Geometry

Definition 6.1.1 (Partial geometry) *Let P and B be disjoint (non-empty) sets of objects called points (P) and lines (B). Let I be a symmetric point-line incidence relation*

$$I \subseteq (P \times B) \cup (B \times P)$$

The partial geometry $S = (P,\ B,\ I)$ with parameters $pg(s,t,\alpha)$ satisfies:

1. *Each point is incident with $1+t$ $(t \geq 1)$ lines and two different points are incident with at most one line.*
2. *Each line is incident with $1+s$ $(s \geq 1)$ points and two different lines are incident with at most one point.*
3. *If x is a point not incident with line L, then exactly α $(\alpha \geq 1)$ points $y_1, y_2, \ldots, y_\alpha$ and α lines $M_1, M_2, \ldots, M_\alpha$ exist such that xIM_i, M_iIy_i and y_iIL $(1 \leq i \leq \alpha)$.*

Let $|P| = v$ and $|B| = b$. Then v and b can be determined from s, t and α:

$$v = \frac{(s+1)(st+\alpha)}{\alpha} \tag{6.1}$$

$$b = \frac{(t+1)(st+\alpha)}{\alpha} \tag{6.2}$$

The parameters of $pg(6,6,4)$ are

$$s = 6;\ t = 6;\ \alpha = 4;\ v = 70;\ b = 70$$

The incidence matrix is defined in a similar way as for designs, taking rows for points and columns for lines. The 70×70 incidence matrix of $pg(6,6,4)$ has the following properties:

1. There are 7 ones per row. The scalar product of two rows is 0 or 1.
2. There are 7 ones per column. The scalar product of two columns is 0 or 1.
3. If point x is not on line L, then exactly 4 points y_1, y_2, y_3, y_4 and 4 lines M_1, M_2, M_3, M_4 exist such that the incidences are:

	L	M_1	M_2	M_3	M_4
x	0	1	1	1	1
y_1	1	1	0	0	0
y_2	1	0	1	0	0
y_3	1	0	0	1	0
y_4	1	0	0	0	1

6.2 $pg(6,6,4)$ with automorphism of order 3 with 7 fixed points and 7 fixed lines

We assume an automorphism of order 3 with 7 fixed points and 7 fixed lines. In the notation of (6.1), (6.2) and (4.2)-(4.8), we have $v = b = 70$ and $f = f' = 7$, $h = g = 63$, $p = 3$, $n = n' = 21$, respectively. The generation of the fixed parts is done in the same way as described in Section 4.2. However, for the fixed parts it now holds that each pair of fixed points is incident with 0 non-fixed lines and at most 1 fixed line. The same applies for all pairs of fixed lines with respect to the points.

For the 7×7 fixed part F, the following restriction holds based on $\alpha = 4$. For each fixed point x and each fixed line L not incident with x, we must find another fixed point y_1 and a fixed line M_1, such that xIM_1, M_1Iy_1 and y_1IL. This is justified by the following observations. Consider any fixed point x which is not incident with a fixed line L. x is incident with either 1 or 4 fixed lines, and so with 6 or 3 non-fixed lines. L is incident with either 1 or 4 fixed points, and so with 6 or 3 non-fixed points. Consider three non-fixed points y_2, y_3 and y_4 incident with L and lying in one orbit of the assumed automorphism. Consider three non-fixed lines L_2, L_3 and L_4 incident with x and lying in one orbit of the assumed automorphism. A non-zero circulant of order 3 formed by the rows of points y_2, y_3 and y_4 and by the columns of lines L_2, L_3 and L_4 will contain three incidences $y_2IL_2, y_3IL_3, y_4IL_4$.

Exhaustive generation yields only two possible F configurations, each leading to exactly one non-equivalent *starting orbit configuration*, which we show in Figure 6.1 in the form of Definition 4.1.3. Note the use of a dot instead of a zero for reasons of readability.

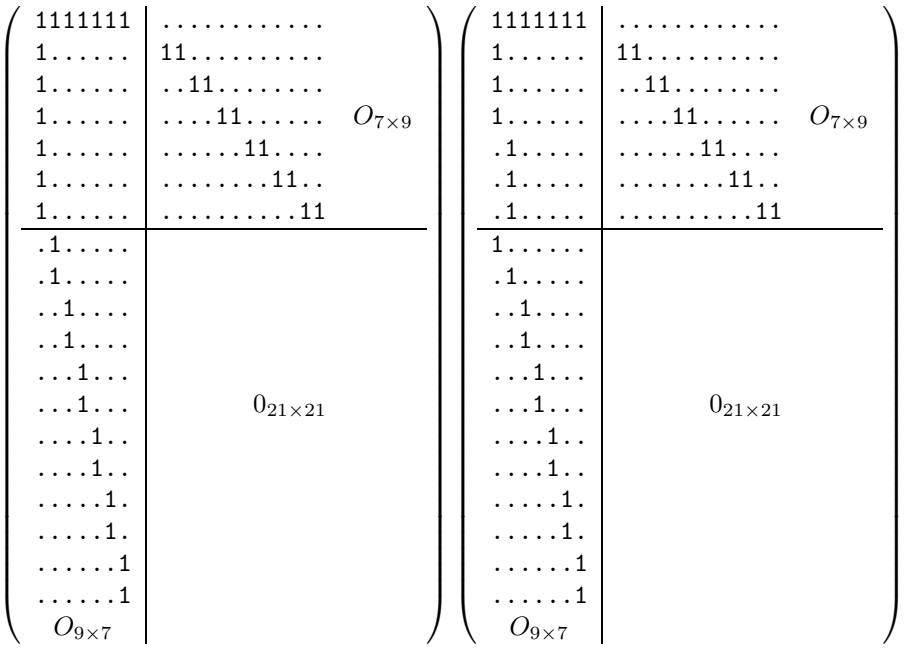


Figure 6.1: The two possible starting orbit configurations.

6.3 Pruning techniques

Given the first starting configuration of Figure 6.1, we will explain the pruning techniques which can be used for the orbit matrix generation phase. A lot of these techniques take advantage of the symmetry of the fixed part. The other starting configuration has less symmetry, leading to weaker yet still very usable properties.

Figure 6.2 shows the upper 19 rows of the extended orbit matrix. It shows the 7 fixed rows followed by the 12 rows of the extended orbit matrix which contain ones in the fixed columns part. Recall that the orbit matrix is the extended orbit matrix without the fixed rows and fixed columns. The first 12 rows of the orbit matrix are naturally split into two parts A and B, as shown in Figure 6.2. To explain the details, we consider the first 13 rows of the full incidence matrix, i.e. the extension of the gray rows of the matrix of Figure 6.2. Actually, by isomorph rejection arguments, we only need to consider one possibility for the first 13 rows, which is shown in Figure 6.3. This will be explained later.

6.3.1 Degree constraints

Applying the α condition on all non-fixed points x and the first fixed line L (note that x is not incident with L), we get the following constraints:

- A is a 12×12 matrix with exactly three 1's in each row. This is illustrated in bold type in Figure 6.3. The first column plays the role of line L which is not incident with

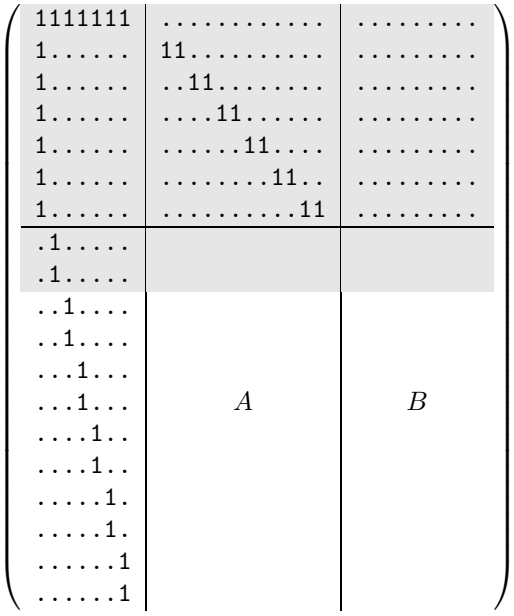


Figure 6.2: The upper 19 rows of the extended orbit matrix.

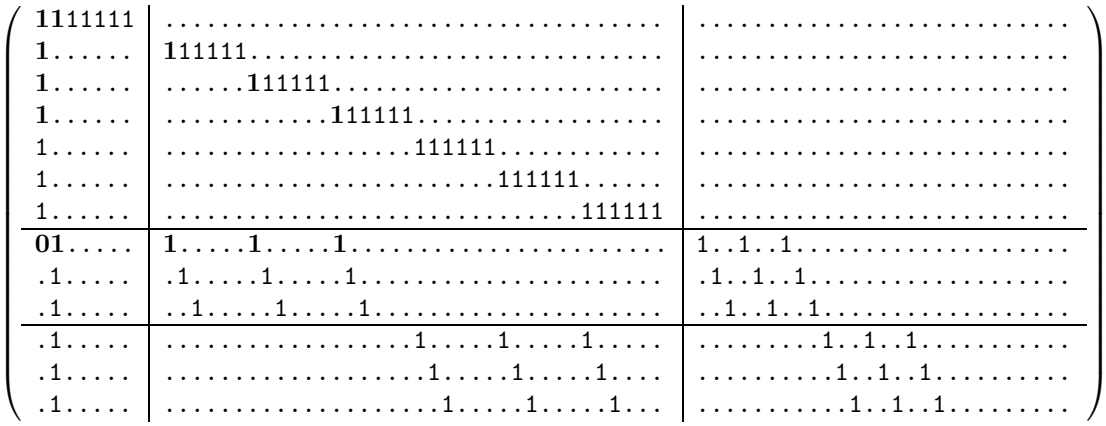


Figure 6.3: The first 13 rows of the incidence matrix. The bold **0** indicates the role of x and L to apply the α condition of the definition. The bold **1**'s illustrate the needed incidences for the α condition to be met.

point x, the latter being the 8'th row. The first 4 rows play the role of the 4 points y_i of the α condition. Columns 2, 8, 14 and 20 play the role of the 4 lines L_i.

- Consequently, B is a 12×9 matrix with exactly three 1's in each row.

Applying the α condition on the first fixed point x and all non-fixed lines L (note that x is not incident with L), we get the following constraints (by a similar argument):

- A is a 12×12 matrix with exactly three 1's in each column.
- B is a 12×9 matrix with exactly four 1's in each column.

Note that we cannot have any 2's in part A or B of the orbit matrix since the scalar product of any two rows (columns) can be at most 1.

For $1 \leq i \leq 6$, denote by S_i, as shown in Figure 6.4, the orbit matrix part containing both the $(2i-1)$-th and $2i$-th row of $(A|B)$. Denote by L_i the orbit matrix part containing both the $(2i-1)$-th and $2i$-th column of A. We have the following constraints:

- It is not possible that two 1's are above one another in some S_i. We already have scalar product one because of the fixed columns part H.
- It is not possible that two 1's are beside one another in some L_i. We already have scalar product one because of the fixed rows part G.
- Moreover, every group of four entries formed by the intersection of any S_i and L_j contains precisely one 1 and three 0's. This is easily shown by applying the α condition on all fixed points and all fixed lines.

Other starting configuration

Using similar arguments, the reader could verify that the following constraints hold for the second starting configuration of Figure 6.1:

- A is a 12×12 matrix with exactly three 1's in each column,
- B is a 12×9 matrix with exactly four 1's in each column,
- $(A|B)$ has exactly six 1's in each row.
- It is not possible that two 1's are above one another in S_i $(2 \leq i \leq 6)$. This also counts for S_1 in the A part only.
- It is not possible that two 1's are beside one another in some L_i. Every group of four entries formed by the intersection of any S_i and L_j contains precisely one 1 and three 0's.

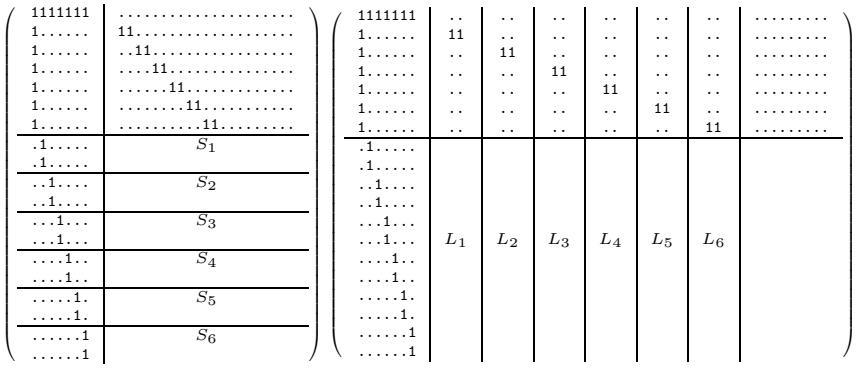


Figure 6.4: Definition of the orbit matrix parts S_i and L_j.

6.3.2 Row and column lexical ordering

We can lexically order all the columns of B. Within each S_i, both rows can be ordered with respect to each other. Within each L_i, both columns can be ordered with respect to each other.

We can lexically order the S_i groups with respect to each other, because there exists an automorphism of the starting orbit configuration which turns the S_i's into each other. I.e., to interchange S_i and S_j, it suffices to perform the row permutation $(7+2i \quad 7+2j)$ $(7+2i-1 \quad 7+2j-1)$ together with the column permutation $(i+1 \quad j+1)$ on the extented orbit matrix (so no fixed points or non-fixed lines are permuted). This *row group ordering* with respect to each other is defined as follows. Let us denote by S_i^k the k-th row of S_i (k is 1 or 2 in this case). Then the ordering of S_i and S_j is defined as

$$S_i < S_j \Longleftrightarrow \exists k \;\; \forall m < k \;\; S_i^m = S_j^m \wedge S_i^k < S_j^k$$

By a similar argument, we can lexically order the L_i groups with respect to each other (column ordering). This *column group ordering* with respect to each other is defined in the following way in order to be compatible with the row order generation. Let us denote by L_i^k the k-th *row* of L_i ($1 \leq k \leq 12$ in this case). Then the ordering of L_i and L_j is defined as

$$L_i < L_j \Longleftrightarrow \exists k \;\; \forall m < k \;\; L_i^m = L_j^m \wedge L_i^k < L_j^k$$

This is the technique which was mentioned in Chapter 4.

When we backtrack from the first valid placed one in the first row of some S_i, we don't need to try a zero for that position. This is in fact a consequence of all previous remarks. Also, when we backtrack from the first valid placed one in L_i (the first placed in the generation order), we don't need to try a zero for that position. Now you can see that we can fix the first two rows and the first two columns of $(A|B)$, as shown in the left extended orbit matrix of Figure 6.5.

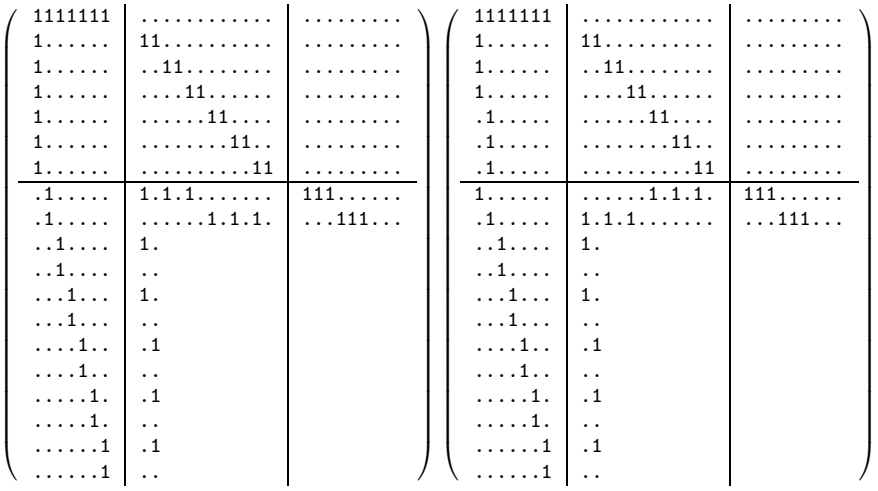


Figure 6.5: The first two rows and two columns of the orbit matrix can be fixed for the first and second starting configuration as shown in the left and right matrix, respectively.

Other starting configuration

We can lexically order all the columns of B. Within each S_i (except S_1), both rows can be ordered with respect to each other. Within each L_i, both columns can be ordered with respect to each other.

We can lexically order the S_i (except S_1) groups with respect to each other (row ordering). The L_2 and L_3 groups can be ordered with respect to each other (column ordering). The L_4, L_5 and L_6 groups can be ordered with respect to each other (column ordering).

When we backtrack from the first valid placed one in the first row of some S_i, we don't need to try a zero for that position. When we backtrack from the first valid placed one in L_i or in some column of B (the first placed in the generation order), we don't need to try a zero for that position. Now you can see that we can fix the first two rows of $(A|B)$, as shown in the right extended orbit matrix of Figure 6.5.

6.3.3 Sum of scalar products

This section will focus on the rows, although a similar reasoning applies to the columns as well. Consider the extension to circulants of some S_i and the extension of a row of some S_j ($j \neq i$). Figure 6.6 shows a possible extension of S_1 and a possible extension of the first row of S_2. To increase readability, we alternate the use of symbols . and * for a 0. In this matrix, x and L are shown to use the α condition of the definition: the sum of the scalar products of x and all the rows of S_i (S_1 in Figure 6.6) is exactly 3. Figure 6.7 shows the corresponding extended orbit matrix rows of Figure 6.6. The orbit matrix intersections of Figure 6.8 are needed between S_i and any row of S_j ($j \neq i$). The sum of the scalar products of a column y of (A) and all the columns of L_i is at most 3. The sum of the scalar products

	L		
	1111111	...***...***...***...***...***...***	...***...***...***...***...
	.1.....	1..***1..***1..***...***...***...***	1..1*.1..***...***...***...
	.1.....	.1.***.1.***.1.***...***...***...***	.1.*1*.1.***...***...***...
	.1.....	..1***..1***..1***...***...***...***	..1**1..1***...***...***...
S_1	.1.....	...***...***...***1..***1..***1..***	...***...1**1..1**...***...
	.1.....	...***...***...***.1.***.1.***.1.***	...***...*1*.1.*1*...***...
	.1.....	...***...***...***..1***..1***..1***	...***...**1..1**1...***...
x	.01....	1..***.1.***..1***...***...***...***	...***...***...***1..1**1..
S_2	..1....	.1.***..1***1..***...***...***...***	...***...***...***.1.*1*.1.
	..1....	..1***1..***.1.***...***...***...***	...***...***...***..1**1..1

Figure 6.6: An extension of S_1 and an extension of a row of S_2 is shown together with the first fixed row. Note that x and L are indicated to reason with the α condition of the definition.

	1111111	.*.*.*.*.*.*	.*.*.*.*.
	.1.....	1*1*1*.*.*.*	111*.*.*.
S_1	.1.....	.*.*.*1*1*1*	.*.111.*.
S_2	.01....	1*1*1*.*.*.*	.*.*.*111

Figure 6.7: The corresponding extended orbit matrix rows of Figure 6.6.

of a column y of (B) and all the columns of L_i is at most 4. These intersection numbers are tested easily while constructing a new orbit matrix row (or column).

Other starting configuration

The sum of the scalar products of a row x of $(A|B)$ and all the rows of S_i (except S_1) is exactly 3. The sum of the scalar products of a column y of (A) and all the columns of L_i is at most 3. The sum of the scalar products of a column y of (B) and all the columns of L_i is at most 4.

There is also a special case to consider. Define E_1 to be the extension of the first row of S_1 together with the 2-nd, 3-th and 4-th fixed rows. Define E_2 to be the extension of the second row of S_1 together with the 5-th, 6-th and 7-th fixed rows. Then the sum of scalar products of a row of some S_i ($i \neq 1$) and E_1 (or E_2) is 3.

S_i	$\binom{1}{0}$ or $\binom{0}{1}$	$\binom{1}{0}$ or $\binom{0}{1}$	$\binom{0}{0}$	$\binom{0}{0}$
row of S_j $(j \neq i)$	1	0	1	0
amount	3	9	3	6

Figure 6.8: Needed intersection pattern between S_i and a row of S_j.

6.3.4 Isomorph rejection

First starting configuration

As explained in Section 4.3, we can use nauty canonical forms to prune the search. We use this technique at the end of each row. The number of non-isomorphic partial configurations are given in the following table.

Filled orbit matrix rows	Non-equivalent
1	1
2	1
3	17
4	34
5	767
6	330
7	48.512
8	2.737
9	29.546
10	0

The 10-th row could not be made. There are no orbit matrices for the $(A|B)$ part. It took 55 seconds on a 1.8 GHz Pentium IV. 5.366.428 recursive calls where made, but this does not incorporate the "calls" made by *nauty*.

Second starting configuration

We use nauty canonical forms at the end of each row for the first 6 rows. The number of non-isomorphic partial configurations are given in the following table.

Filled orbit matrix rows	Non-equivalent
1	1
2	1
3	245
4	266
5	111.841
6	37.236
7	0

The 7-th row could not be made. There are no orbit matrices for the $(A|B)$ part. It took about 66 hours on a 1.8 GHz Pentium IV. 26e9 recursive calls where made, but this does not incorporate the "calls" made by *nauty*.

6.4 Conclusion

There does not exist a $pg(6,6,4)$ with a fixed automorphism of order 3 with 7 fixed points and lines. We also tried to check an automorphism of order 3 with 1 fixed point and block, but there the properties are to weak to make the enumeration possible. The existence of $pg(6,6,4)$ in general remains open.

7 Enumeration of the doubles of the projective plane of order 4

The results collected in this chapter have appeared in [18]. A classification of the doubles of the projective plane of order 4 with respect to the order of the automorphism group is presented and it is established that, up to isomorphism, there are 1 746 461 307 doubles. We start with the designs possessing non-trivial automorphisms. Since the designs with automorphisms of odd prime orders have been constructed previously, we are left with the construction of the designs with automorphisms of order 2. Moreover, we establish that a 2-$(21, 5, 2)$ design cannot be reducible in two inequivalent ways. This makes it possible to calculate the number of designs with only the trivial automorphism, and consequently the number of all double designs. Most of the computer results are obtained by two different approaches and implementations, one by myself and one by Svetlana Topalova.

7.1 Introduction

Each 2-(v, k, λ) design determines the existence of 2-$(v, k, m\lambda)$ designs (for any integer $m > 1$), which are called *quasimultiples* of a 2-(v, k, λ) design. A quasimultiple 2-$(v, k, m\lambda)$ is *reducible* into m 2-(v, k, λ) designs if there is a partition of its blocks into m subcollections each of which forms a 2-(v, k, λ) design. This partition is called a *reduction*. For $m = 2$ quasimultiple designs are called *quasidoubles*, and the reducible quasidouble designs are called *doubles*. We shall denote by $(D_1 \cup D_2)$ a double design which can be reduced to the two designs D_1 and D_2. A *reduction* of a double design D with parameters 2-$(v, k, 2\lambda)$ can be represented by a set of two collections of blocks, each containing half the blocks of D, such that each collection of blocks forms a 2-(v, k, λ) design. An obvious reduction of a double design $(D_1 \cup D_2)$ is $\{D_1, D_2\}$. The order in which the constituent designs are listed, is not relevant. We will often use the notation $D_2 = \mu D_1$, in which μ is a point permutation applied to the points of D_1 to obtain D_2. Doubles can be reducible in more than one way. Two reductions $\{D_1, D_2\}$ and $\{D_3, D_4\}$ of a double design are *equivalent* if and only if there exists some point permutation μ such that $D_3 = \mu D_1$ and $D_4 = \mu D_2$, or such that $D_4 = \mu D_1$ and $D_3 = \mu D_2$. A double which has, up to equivalence, only one reduction is *uniquely reducible*.

Reducible 2-$(21, 5, 2)$ designs are the subject of this chapter. We will show that they are

uniquely reducible. Up to equivalence there is a unique 2-$(21, 5, 1)$ design (the projective plane of order 4 $PG(2, 4)$) and the reducible 2-$(21, 5, 2)$ designs are its doubles. The first lower bound on the number of reducible 2-$(21, 5, 2)$ designs is derived in [34] and it is 10. Lower bounds on the number of doubles of projective planes in general are derived in [23] and [24]. These bounds are much more powerful for projective planes of bigger orders, but for the doubles of the projective plane of order 4 the bound is 24.

We enumerate the reducible 2-$(21, 5, 2)$ designs by constructing those which have non-trivial automorphisms, which allows us to calculate the number of all the reducible 2-$(21, 5, 2)$. This is possible, because these designs are made up of two 2-$(21, 5, 1)$ subdesigns. For other examples of enumerating designs which contain incidence structures see for instance [27], [28], [29], [50].

In [48] all 2-$(21, 5, 2)$ designs with automorphisms of odd prime orders were constructed, their number was determined to be 22 998 and 4 170 of them were found to be reducible. This leaves only the reducible 2-$(21, 5, 2)$ designs with automorphisms of order 2 to be constructed. There are two types of such automorphisms, namely those which transform each of the constituent 2-$(21, 5, 1)$ designs into itself and those which transform one of the 2-$(21, 5, 1)$ into the other (and vice versa). We construct 40 485 designs of the first type and 991 957 of the second. We study their automorphism groups. The results coincide with those obtained in [48]. Using this data we calculate that the number of all doubles of the projective plane of order 4 is 1 746 461 307.

Section 7.2 derives an equation to determine the number of doubles with only the trivial automorphism, given an enumeration of those with non-trivial automorphism. Section 7.3 shows all reducible 2-(21,5,2) are uniquely reducible. Section 7.4 presents the enumeration of all reducible 2-(21,5,2) with non-trivial automorphisms. Finally, Section 7.5 presents the classification of the doubles of the projective plane of order 4.

7.2 Doubles of a uniquely reducible design

Below we will consider doubles of designs for which, up to isomorphism, only one design of its parameter set exists. So instead of $(D_1 \cup D_2)$ we will often use the notation $(D \cup \varphi D)$, where the constituent design φD is obtained from D by a permutation φ of its points.

In the rest of this section, D will be a 2-(v, k, λ) design and $(D \cup \varphi D)$ will be a uniquely reducible double of D. By G we denote the full automorphism group of D. By G_φ we denote the intersection of the full automorphism groups of D and φD. By $\widehat{G}_\varphi$ we denote the full automorphism group of the double design $(D \cup \varphi D)$.

The set of all $v!$ doubles of D of the form $(D \cup \varphi D)$ (determined by all $v!$ permutations φ of the points of D) can be partitioned into isomorphism classes $\mathcal{C}_G(\varphi)$. Then obviously

$$v! = \sum_{\mathcal{C}_G(\varphi)} |\mathcal{C}_G(\varphi)| \tag{7.1}$$

In the following proposition we determine the size of an isomorphism class $\mathcal{C}_G(\varphi)$ with a given representative point permutation φ.

Proposition 7.2.1 *The set $\mathcal{C}_G(\varphi)$ of all doubles of the form $(D \cup \psi D)$ which are isomorphic to $(D \cup \varphi D)$ is obtained by choosing for ψ all permutations from the set*

$$G\varphi G \cup G\varphi^{-1}G.$$

Moreover, the number of such doubles is given by

$$|\mathcal{C}_G(\varphi)| = \begin{cases} |G|^2/|G_\varphi| & \text{if } G\varphi G = G\varphi^{-1}G, \\ 2|G|^2/|G_\varphi| & \text{otherwise, i.e. } G\varphi G \cap G\varphi^{-1}G = \emptyset\,. \end{cases} \tag{7.2}$$

Proof. Suppose $\psi \in G\varphi G$, then $\exists \alpha, \beta \in G : \psi = \beta\varphi\alpha$. Clearly $(D \cup \psi D)$ is isomorphic to $(D \cup \varphi D)$ since $\beta^{-1}(D \cup \beta\varphi\alpha D) = (D \cup \varphi D)$. Similarly, if $\psi \in G\varphi^{-1}G$, then $\exists \alpha, \beta \in G : \psi = \beta\varphi^{-1}\alpha$, and $(D \cup \psi D)$ is isomorphic to $(D \cup \varphi D)$ since $\varphi\beta^{-1}(D \cup \beta\varphi^{-1}\alpha D) = (\varphi D \cup D)$.

Conversely, we now suppose that $(D \cup \psi D)$ is isomorphic to $(D \cup \varphi D)$. Since $(D \cup \varphi D)$ is uniquely reducible, only two cases are possible. In the first case there exists a point permutation μ for which $\mu D = D$ and $\mu\varphi D = \psi D$, which implies that $\mu \in G$ and $\varphi^{-1}\mu^{-1}\psi \in G$, and thus that $\psi \in G\varphi G$. In the second case there exists a point permutation μ such that $\mu D = \psi D$ and $\mu\varphi D = D$, which implies that $\mu^{-1}\psi \in G$ and $\mu\varphi \in G$, and thus $\psi \in \mu G$ and $\mu \in G\varphi^{-1} \Rightarrow \psi \in G\varphi^{-1}G$.

From the theory of double cosets ([21], Theorem 2.19) it follows immediately that $|G\varphi G| = |G\varphi^{-1}G| = |G|^2/|G_\varphi|$. Moreover it is known that either $G\varphi G \cap G\varphi^{-1}G = \emptyset$ or $G\varphi G = G\varphi^{-1}G$. Hence $|G\varphi G \cup G\varphi^{-1}G| = 2|G|^2/|G_\varphi|$ when $G\varphi G \cap G\varphi^{-1}G = \emptyset$, and $|G\varphi G \cup G\varphi^{-1}G| = |G|^2/|G_\varphi|$ otherwise. □

Proposition 7.2.2 *If $G\varphi G = G\varphi^{-1}G$, then there exists $\omega \in \widehat{G}_\varphi$ such that $(D \cup \varphi D) = (D \cup \omega D)$. This ω transforms D into φD and vice versa. If $|G_\varphi| = 1$, then ω is of order 2.*

Proof. Since $G\varphi G = G\varphi^{-1}G$, it holds that $\varphi^{-1} \in G\varphi G$, hence $\exists \rho, \sigma \in G : \varphi^{-1} = \rho\varphi\sigma$. This means that $\varphi\rho\varphi\sigma = 1$, hence $\varphi\rho\varphi \in G$. Let $\omega = \varphi\rho$.

Then $\omega D = \varphi\rho D = \varphi D$. Since $\omega\varphi \in G$, $\omega\varphi D = D$. Hence $(D \cup \varphi D) = (D \cup \omega D)$ and ω transforms D into φD and vice versa.

Moreover it follows from $\omega^2 D = D$ and $\omega^2\varphi D = \varphi D$, that $\omega^2 \in G_\varphi$. In case $|G_\varphi| = 1$, this means that $\omega^2 = 1$. □

Corollary 7.2.3 *If $|\widehat{G}_\varphi| = 1$, then $|\mathcal{C}_G(\varphi)| = 2|G|^2$.*

Let N_i (resp. N_i') denote the number of isomorphism classes $\mathcal{C}_G(\varphi)$ for which $|G_\varphi| = i$ and $G\varphi G \cap G\varphi^{-1}G = \emptyset$ (resp. $G\varphi G = G\varphi^{-1}G$). Then, using equation (7.2), equation (7.1) can be rewritten as

$$v! = 2|G|^2 N_1 + |G|^2 N_1' + \sum_{i>1} \frac{2|G|^2}{i} N_i + \sum_{i>1} \frac{|G|^2}{i} N_i'\,. \tag{7.3}$$

Let N be the total number of non-isomorphic doubles of D, then

$$N = N_1 + N_1' + \sum_{i>1} N_i + \sum_{i>1} N_i'\,. \tag{7.4}$$

If we can enumerate the doubles of D with non-trivial automorphisms by means of some construction techniques, i.e. determine the numbers N_1' as well as N_i and N_i' for all $i > 1$, equation (7.3) can be used to obtain the number N_1 of doubles of D with trivial automorphisms. Equation (7.4) can be used to calculate the total number N of doubles of D.

Corollary 7.2.4 *A 2-(v,k,λ) design D of which all doubles are uniquely reducible, has at least $v!/(2|G|^2)$ non-isomorphic doubles (with G the full automorphism group of D).*

All quasidoubles of the projective planes of orders 2 and 3 have been known before this work. Up to isomorphism there are 4 doubles of the projective plane of order 2 and 184 doubles of the projective plane of order 3. We established that they are uniquely reducible, and then investigated their automorphisms and checked that they match equations (7.3) and (7.4).

We also checked equations (7.3) and (7.4) on the doubles of the affine planes of orders 2, 3 and 4. Most related to the main result of this work are the doubles of the affine plane of order 4, since their unique reducibility follows from the considerations in the next section. All the doubles of the affine plane of order 4 are among the resolvable 2-$(16, 4, 2)$ designs which are constructed in [26] and for which the designs with non-trivial automorphisms are available from the authors' web-page. We determined that 9102 among them are reducible and, using equations (7.3) and (7.4), we found that the number of doubles of the affine plane of order 4 is 320 061. We independently constructed all these doubles and obtained the same result.

It follows from Corollary 7.2.4 that the number of the non-isomorphic doubles of the projective plane of order 4 is at least 1 745 944 200. To determine their exact number by equations (7.3) and (7.4), we have to construct all designs with non-trivial automorphisms.

More precisely this means that we have to

- construct the double designs for which $|G_\varphi| \neq 1$ and determine the numbers N_i and N_i', $i > 1$ (cf. Section 7.4.1), and
- construct the double designs for which $|G_\varphi| = 1$ and $G\varphi G = G\varphi^{-1}G$, and thus determine the number N_1' (cf. Section 7.4.2).

7.3 On the unique reducibility of 2-(21,5,2)

In this section, two reductions $\{D_1, D_2\}$ and $\{D_3, D_4\}$ of a double design are considered *different* if the two sets of collections of blocks are not pairwise equal. In order to prove the unique reducibility of a double 2-(21,5,2) design, we will consider all different reductions and show that they are equivalent by a computer assisted proof.

Consider a double 2-$(v, k, 2\lambda)$ design $(D_1 \cup D_2)$, which is a double of a unique (for its parameter set) 2-(v, k, λ) design. We consider all reductions, different from the obvious reduction $\{D_1, D_2\}$, in the form $\{D_1^a \cup D_2^b, D_2^a \cup D_1^b\}$, where the collection of blocks D_1^a and D_1^b form D_1, the collection of blocks D_2^a and D_2^b form D_2, the collection of blocks D_1^a and D_2^b form D_3 and the collection of blocks D_1^b and D_2^a form D_4, with D_1, D_2, D_3 and D_4 all isomorphic designs:

Without loss of generality, we can restrict ourselves to reductions where the a parts have at least as many blocks as the b parts. Also, we only consider reductions where D_1^b and D_2^b have no common blocks (i.e. no two blocks, one of D_1^b and one of D_2^b are incident with the same set of points), since such a reduction is not different from the reduction where the equal blocks of the b parts are put in the a parts.

Proposition 7.3.1 *Let n be the number of blocks in D_1^b (D_2^b), and β_i the number of blocks in D_1^b (D_2^b) containing point i ($i = 1, 2, \ldots, v$). The following considerations can be made:*

(a) Any point is in the same number of blocks of D_1^a and D_2^a (D_1^b and D_2^b).

(b) Any pair of points is in the same number of blocks of D_1^a and D_2^a (D_1^b and D_2^b).

(c) If D_1 is a projective plane of order q (i.e. a 2-$(q^2+q+1, q+1, 1)$ design), the following holds:

$$\beta_i \neq 1 \quad ; \; i = 1, 2, \ldots, v \tag{7.5}$$

$$\sum_{i=1}^{v} \beta_i = n(q+1) \tag{7.6}$$

$$\sum_{i=1}^{v} \beta_i^2 = n(n+q) \tag{7.7}$$

$$n \leq \frac{q^2+q}{2} \tag{7.8}$$

Proof.

(a) D_1^b and D_1^a (or D_2^a) form a 2-(v, k, λ), so point i is in $r - \beta_i$ blocks of D_1^a (D_2^a).

(b) Let the pair of points (i,j) be in λ_{ij} blocks of D_1^b. Since D_1^b and D_1^a (or D_2^a) form a 2-(v, k, λ), the pair of points (i,j) is in $\lambda - \lambda_{ij}$ blocks of D_1^a (D_2^a).

(c) In a 2-$(q^2+q+1, q+1, 1)$ design two blocks have exactly one common point. Consider any point i. When we look at an arbitrary subset S_q of q blocks out of the $q+1$ blocks incident with point i, S_q forces the last block (which contains point i) to be incident with all q remaining points which are not in any of the blocks of S_q. So this last block containing point i is fixed by the other q blocks. $\beta_i = 1$ would force $D_1^a(D_2^a)$ to have q blocks containing point i. This will force the block of D_1^b containing point i to be the same as the block of D_2^b containing point i, but we supposed that D_1^b and D_2^b have no common blocks, so (7.5) follows.

(7.6) is obtained by counting the number of ones in the incidence matrix of D_1^b (D_2^b) in two ways.

A 2-$(q^2+q+1, q+1, 1)$ design is symmetric, and from the $\lambda = 1$ condition for the blocks of the D_1^b (D_2^b) part we obtain $\binom{n}{2} = \sum_{i=1}^{v} \binom{\beta_i}{2}$. Using also (7.6) we get (7.7).

D_1^b (D_2^b) has at most as many blocks as $D_1^a(D_2^a)$. That is why $n \leq \lfloor v/2 \rfloor$, so (7.8) follows.

□

Proposition 7.3.2 *A reducible 2-(21,5,2) design is uniquely reducible.*

Proof. For $q = 4$, the set of equations (5), (6), (7), (8) has solutions only for $n = 6, 8, 9, 10$.

For each case, exhaustive generation is performed in the following way, satisfying (a) and (b) from Proposition 7.3.1:

- We generate the set of all non-equivalent D_1^b.
- For each such D_1^b, we generate the set of all non-equivalent D_2^b, taking into account the limitation that D_1^b and D_2^b have no common blocks.
- For each such combination of D_1^b and D_2^b, we generate all non-equivalent a parts (D_1^a or D_2^a), and show that all obtained reductions are equivalent.

The unique solution for the values of β_i if $n = 6$ is $(2_{15}, 0_6)$, namely 15 twos and 6 zeroes. There is only one non-equivalent way to choose six blocks for D_1^b matching this pattern, but exhaustive generation shows we cannot construct D_2^b. So $n = 6$ is impossible.

The unique solution for the values of β_i if $n = 8$ is $(4_2, 2_{16}, 0_3)$. There is only one non-equivalent way to choose eight blocks for D_1^b matching this pattern. Given D_1^b, there is only one non-equivalent way to construct D_2^b. For the unique combination of D_1^b and D_2^b, we generated 12 non-equivalent D_1^a (D_2^a). For all obtained reductions, one of which is shown in Figure 7.1, there exist point permutations φ_a and φ_b such that:

- $D_2 = \varphi_b \varphi_a D_1$
- $D_2^b = \varphi_b D_1^b$, $D_1^a = \varphi_b D_1^a$, $D_2^a = \varphi_b D_2^a$, $(\varphi_b)^2 = 1$.
- $D_2^a = \varphi_a D_1^a$, $D_1^b = \varphi_a D_1^b$, $D_2^b = \varphi_a D_2^b$.

The reduction $\{D_1^a \cup D_2^b, D_2^a \cup D_1^b\}$ is equivalent to the reduction $\{D_1, D_2\}$ because

$$\varphi_b(D_1^a \cup D_2^b) = (D_1^a \cup D_1^b) = D_1 \quad ; \quad \varphi_b(D_2^a \cup D_1^b) = (D_2^a \cup D_2^b) = D_2.$$

There are 3 solutions for the values of β_i if $n = 9$:
$(5_1, 3_4, 2_{14}, 0_2)$, $(4_3, 3_1, 2_{15}, 0_2)$ and $(3_9, 2_9, 0_3)$. None of the subsets of nine blocks of a 2-$(21, 5, 1)$ design matches the first two patterns. There is only one non-equivalent way to choose nine blocks for D_1^b matching pattern $(3_9, 2_9, 0_3)$, but exhaustive generation shows we cannot construct D_2^b.

D_1^a				D_1^b		D_2^a				D_2^b	
1	1111	....	....	....	....	1	....	1111	....	....	....
1	....	1111	....	....	....	1	....	....	1111	....	....
1	....	....	1111	....	....	1	1111	....	....	....	....
1	....	....	....	1111	....	1	....	....	....	....	1111
1	....	....	....	....	1111	1	....	....	....	1111	....
.	1...	1...	1...	1...	...1	.	1...	1...	1...	1...	...1
.	.1..	.1..	.1..	.1..	...1	.	.1..	.1..	.1..	.1..	...1
.	..1.	..1.	..1.	..1.	...1	.	..1.	..1.	..1.	..1.	...1
.	...1	...1	...1	...1	...1	.	...1	...1	...1	...1	...1
.	.1..	..1.	...1	1...	..1.	.	.1..	..1.	...1	1...	..1.
.	1...	...1	..1.	.1..	..1.	.	1...	...1	..1.	.1..	..1.
.	...1	1...	.1..	..1.	..1.	.	...1	1...	.1..	..1.	..1.
.	..1.	.1..	1...	...1	..1.	.	..1.	.1..	1...	...1	..1.
.	..1.	...1	.1..	1...	.1..	.	..1.	...1	.1..	1...	.1..
.	...1	..1.	1...	.1..	.1..	.	...1	..1.	1...	.1..	.1..
.	1...	.1..	...1	..1.	.1..	.	1...	.1..	...1	..1.	.1..
.	.1..	1...	..1.	...1	.1..	.	.1..	1...	..1.	...1	.1..
.	...1	.1..	..1.	1...	1...	.	...1	.1..	..1.	1...	1...
.	..1.	1...	...1	.1..	1...	.	..1.	1...	...1	.1..	1...
.	.1..	...1	1...	..1.	1...	.	.1..	...1	1...	..1.	1...
.	1...	..1.	.1..	...1	1...	.	1...	..1.	.1..	...1	1...

Figure 7.1: One of the obtained reductions for the $n = 8$ case.

There are 3 solutions for the values of β_i if $n = 10$:
$(5_1, 4_2, 3_3, 2_{14}, 0_1)$, $(4_5, 2_{15}, 0_1)$ and $(4_2, 3_8, 2_9, 0_2)$. None of the subsets of ten blocks of a 2-$(21, 5, 1)$ design matches the first pattern. There is only one non-equivalent way to choose ten blocks for D_1^b matching pattern $(4_5, 2_{15}, 0_1)$ or $(4_2, 3_8, 2_9, 0_2)$, but exhaustive generation shows we cannot construct D_2^b.

So we conclude that a 2-$(21, 5, 2)$ design cannot have two inequivalent reductions, i.e. it is uniquely reducible. □

7.4 Reducible 2-$(21, 5, 2)$ with non-trivial automorphisms

7.4.1 Automorphisms for which $|G_\varphi| \neq 1$

All 2-$(21, 5, 2)$ designs with automorphisms of odd prime orders are constructed in [48]. It turns out that 4170 of them are reducible and we use these for our classification. So we only have to construct the designs with automorphisms of order 2.

Consider $(D_1 \cup D_2)$ with a full automorphism group of order 2^s $(s \geq 1)$, and $|G_\varphi| \neq 1$. Then D_1 and D_2 have common automorphisms of order 2. The 2-$(21, 5, 1)$ design is known to have automorphisms of order 2 with 5 fixed points and automorphisms of order 2 with 7 fixed points. Their action is illustrated in Figures 7.2 and 7.3. We construct all double designs with such automorphisms of order 2 which are automorphisms of both D_1 and D_2. We consider the two cases, namely

Automorphism of order 2 with 5 fixed points: Consider the incidence matrix of D_1 in the form presented in Figure 7.2 and suppose an automorphism γ which acts on the points of the double as
$(1)(2)\cdots(5)\,(6,7)(8,9)\cdots(20,21)$, and on the blocks as

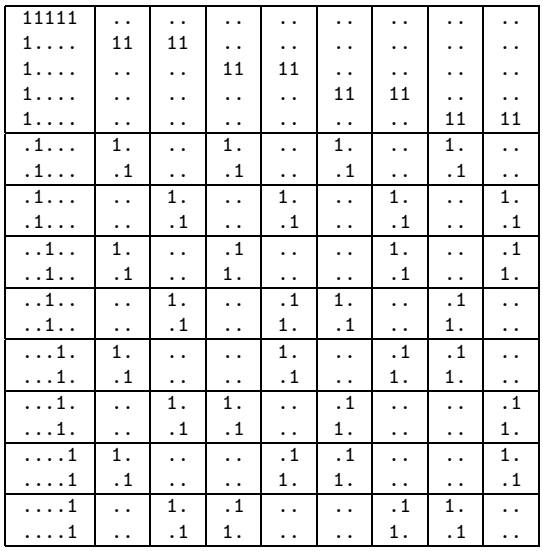


Figure 7.2: The design D_1 presented in a form to illustrate its automorphism of order 2 with 5 fixed points.

$$(1)(2)\cdots(5)\,(6,7)(8,9)\cdots(20,21)\,(22)(23)\cdots(26)\,(27,28)(29,30)\cdots(41,42)\,.$$

Automorphism of order 2 with 7 fixed points: Consider the incidence matrix of D_1 in the form presented in Figure 7.3 and suppose an automorphism δ which acts on the points of the double as
$(1)(2)\cdots(7)\,(8,9)(10,11)\cdots(20,21)$, and on the blocks as
$(1)(2)\cdots(7)\,(8,9)\cdots(20,21)\,(22)(23)\cdots(28)\,(29,30)\cdots(41,42)\,.$

Let $D_2 = \varphi D_1$. Then φ is a permutation of the points which should

(a) transform any fixed point (with respect to γ or δ) into a fixed point,

(b) transform two points of one and the same orbit (with respect to γ or δ) into points which are in one and the same orbit.

We have used two different approaches for the actual construction. The results are the same.

In the first approach we initially leave the fixed points aside and construct the non-trivial orbit part of the incidence matrix of the double design. We generate all possibilities for the non-trivial orbit part of D_2 by applying all possible permutations of whole point orbits of D_1 and filtering the equivalent solutions away. For each non-equivalent solution we then generate all possible permutations within the point orbits of D_2 and again filter the equivalent solutions away. Finally we add the fixed part in all possible ways (the fixed part of D_2 is a permutation of the fixed part of D_1) and check for isomorphism. However, when applying this approach, the equivalence checks in the first and second step should be carried out with great care since a large number of restrictions hold.

In the second approach we first find all automorphisms of D_1. Next we generate all point permutations meeting conditions (a) and (b) in lexicographic order. When generating the

11.1...	11	..	..	..	..	..	..
.11.1..	..	11	..	..	..	..	..
..11.1.	..	..	11	..	..	..	..
...11.1	..	..	..	11	..	..	..
1...11.	..	..	..	..	11	..	..
.1...11	..	..	..	..	..	11	..
1.1...1	..	..	..	..	..	..	11
1.....	..	.1	1.	1.	..	1.	..
1.....	..	1.	.1	.1	..	.1	..
.1.....	..	..	.1	1.	1.	..	1.
.1.....	..	..	1.	.1	.1	..	.1
..1....	1.	..	..	.1	1.	1.	..
..1....	.1	..	..	1.	.1	.1	..
...1...	..	1.	..	..	.1	1.	1.
...1...	..	.1	..	..	1.	.1	.1
....1..	1.	..	1.	..	..	.1	1.
....1..	.1	..	.1	..	..	1.	.1
.....1.	1.	1.	..	1.	..	..	.1
.....1.	.1	.1	..	.1	..	..	1.
......1	.1	1.	1.	..	1.	..	..
......1	1.	.1	.1	..	.1	..	..

Figure 7.3: The design D_1 presented in a form to illustrate its automorphism of order 2 with 7 fixed points.

current permutation φ, we search for $\alpha, \beta \in G$, such that $\beta\varphi\alpha$ or $\beta\varphi^{-1}\alpha$ is a permutation which is lexicographically smaller than φ (see Proposition 7.2.1) and meets conditions (a) and (b). The existence of such a pair $\alpha, \beta \in G$ means that the solution is equivalent to one we have already generated, so we can drop it. Note that conditions (a) and (b) are of such a form that they allow us to prune partial solutions for the permutations, which makes the programme much faster. Since the order of the automorphism group G of the 2-$(21, 5, 1)$ design is 120 960, considering all $120\,960^2$ combinations is too time-consuming, so we only consider α and β among a random part of the elements of the group G. We finally filter away the isomorphic solutions (only a limited number of which happen to turn up) by a full isomorphism test.

In this way we construct 9 564 non-isomorphic doubles with an automorphism of order 2 with 5 fixed points, and 31 094 with an automorphism of order 2 with 7 fixed points. This gives a total of 40 485 doubles for this case, because 173 have both an automorphism of order 2 with 5 or 7 fixed points. Of these doubles 305 have also an automorphism of odd prime order, so they were already counted among the 4 170 doubles found above.

7.4.2 Automorphisms of order 2 with $|G_\varphi| = 1$

We generate all designs $(D \cup \varphi D)$, where φ is a permutation of order 2 (see Proposition 7.2.2). We use a method similar to the second approach from the previous section. We first find all automorphisms of D and then generate all possible permutations of order 2 in lexicographic order. As the automorphism group of the projective plane of order 4 is doubly transitive, we can fix one non-trivial orbit.

Suppose we have constructed the current permutation φ. Suppose $\exists\alpha, \beta \in G$, such that $\beta\varphi\alpha$ is lexicographically smaller than φ (see Proposition 7.2.1, and mind that φ is of order

2, i.e. $\varphi = \varphi^{-1}$). If we have already constructed $\beta\varphi\alpha$, then it is of order 2, namely

$$\beta\varphi\alpha\beta\varphi\alpha = 1 \Rightarrow \varphi\alpha\beta\varphi = \beta^{-1}\alpha^{-1} \Rightarrow \varphi\alpha\beta\varphi \in G.$$

But $\varphi\alpha\beta\varphi$ is also an automorphism of φD. Hence $\varphi\alpha\beta\varphi \in G_\varphi$. If $|G_\varphi| = 1$, then $\alpha\beta = 1 \Rightarrow \beta = \alpha^{-1}$. Since $|G_\varphi| = 1$ for most of the designs constructed this way, for the currently constructed permutation φ, we only search for $\alpha \in G$, such that $\alpha^{-1}\varphi\alpha$ is lexicographically smaller than φ, and we drop the solution if such an α exists. This way most of the isomorphic copies are filtered, the final full isomorphism check does not filter much more. This simpler pruning condition makes the programme much faster, which is important because we cannot prune partial solutions for the permutations in this case.

We checked the results by two different implementations. In one of them we used McKay's program *nauty* [36] for the final isomorphism check. We construct 991 957 non-isomorphic designs which have an automorphism of order 2 transforming the constituent designs into one another. We establish that for 984 549 of them the order of the full group of automorphisms is 2, and $|G_\varphi| = 1$.

7.5 Classification results

Classification results are presented in Table 7.1. The classification is based on three properties:

- the order of the automorphism group of the doubles (column $|\widehat{G}_\varphi|$),
- the order of the common subgroup of the full automorphism groups of D and φD (column $|G_\varphi|$), and
- whether $G\varphi G = G\varphi^{-1}G$.

The column labeled $N'_{|G_\varphi|}$ gives the number of non-isomorphic doubles for the given values of the properties. The number[1] of designs isomorphic to one of these doubles among all the 21! possible $(D \cup \psi D)$ is presented in column $|\mathcal{C}_G(\varphi)|/|G|$. This number is determined using Proposition 7.2.1. Column $N'_{|G_\varphi|}$ multiplied by column $|\mathcal{C}_G(\varphi)|/|G|$ gives the last column $N'_{|G_\varphi|} \times |\mathcal{C}_G(\varphi)|/|G|$.

Having constructed all 1 028 899 doubles which possess non-trivial automorphisms, we use equations (7.3) and (7.4) to calculate the number of non-isomorphic designs which possess only the trivial automorphism which turns out to be 1 745 432 408. So the first row of Table 7.1, which is marked, is derived from the other rows.

The number of all 2-$(21, 5, 2)$ doubles is 1 746 461 307, which does not differ very much from the bound obtained by Corollary 7.2.4.

[1] Note that, to avoid big numbers, the values in the last two columns are divided by $|G| = 120\,960$. I.e. the order of the automorphism group of the projective plane of order 4.

$\lvert\widehat{G}_\varphi\rvert$	$\lvert G_\varphi\rvert$	$G\varphi G \stackrel{?}{=} G\varphi^{-1}G$	$N'_{\lvert G_\varphi\rvert}$	$\frac{\lvert\mathcal{C}_G(\varphi)\rvert}{\lvert G\rvert}$	$N'_{\lvert G_\varphi\rvert} \times \frac{\lvert\mathcal{C}_G(\varphi)\rvert}{\lvert G\rvert}$
1	1	no	1 745 432 408	241 920	422 255 008 143 360
2	1	yes	984 549	120 960	119 091 047 040
2	2	no	33 631	120 960	4 068 005 760
3	3	no	2 764	80 640	222 888 960
4	2	yes	5 709	60 480	345 280 320
4	4	no	389	60 480	23 526 720
5	5	no	26	48 384	1 257 984
6	3	yes	1 019	40 320	41 086 080
6	6	no	67	40 320	2 701 440
8	4	yes	345	30 240	10 432 800
8	8	no	17	30 240	514 080
9	9	no	1	26 880	26 880
10	5	yes	30	24 192	725 760
12	6	yes	167	20 160	3 366 720
12	12	no	2	20 160	40 320
14	7	yes	2	17 280	34 560
14	14	no	1	17 280	17 280
16	8	yes	55	15 120	831 600
16	16	no	3	15 120	45 360
18	9	yes	18	13 440	241 920
18	18	no	1	13 440	13 440
21	21	no	1	11 520	11 520
24	12	yes	24	10 080	241 920
28	14	yes	2	8 640	17 280
30	15	yes	1	8 064	8 064
32	16	yes	20	7 560	151 200
32	32	no	1	7 560	7 560
36	18	yes	15	6 720	100 800
40	20	yes	1	6 048	6 048
42	21	yes	2	5 760	11 520
48	24	yes	3	5 040	15 120
54	27	yes	1	4 480	4 480
64	32	yes	7	3 780	26 460
96	48	yes	4	2 520	10 080
96	96	no	1	2 520	2 520
108	54	yes	2	2 240	4 480
120	60	yes	1	2 016	2 016
128	64	yes	2	1890	3 780
192	96	yes	4	1 260	5 040
252	126	yes	1	960	960
256	128	yes	1	945	945
384	96	yes	1	1 260	1 260
384	192	yes	1	630	630
480	240	yes	1	504	504
576	288	yes	1	420	420
1152	288	yes	1	420	420
1152	576	yes	1	210	210
1536	384	yes	1	315	315
3840	1920	yes	1	63	63
120960	120960	yes	1	1	1
All			1 746 461 307		$21!/\lvert G\rvert =$ 422 378 820 864 000

Table 7.1: Classification of the doubles of the projective plane of order 4.

8 Small weight codewords in the codes arising from Desarguesian projective planes of prime order

In this chapter we discuss a problem in coding theory, in which a computer approach helped to characterize the small weight codewords in the codes arising from Desarguesian projective planes of prime order. This work is joint work with L. Storme.

We improve the results of K. Chouinard [12] on codewords of small weight in the codes arising from $PG(2, p)$, p prime. Chouinard characterized all the codewords up to weight $2p$ in these codes. Using a particular basis for this code, described by Moorhouse, we characterize all the codewords of weight up to $2p + (p - 1)/2$ if $p \geq 19$. Furthermore, we present some related additional results.

The results of this chapter form a substantial part of the article *Small weight codewords in the codes arising from Desarguesian projective planes*, which was submitted to *Designs, Codes and Cryptography* [17]. A preprint can be downloaded from *http://caagt.ugent.be/preprints*. This chapter will focus on the computer approach, since this is my main contribution to this work. For the omitted proofs we refer to the article.

8.1 Introduction

A projective plane is a set of points, a set of lines, and a set of incidences of points with lines, which satisfy the following axioms:

(a) Every pair of points is on a unique line.

(b) Every pair of lines intersects in a unique point.

(c) There exist four points of the plane, no three of which are collinear (so there are quadrangles).

The projective plane $PG(2, q)$ of order $q = p^h$ (p prime, integer $h \geq 1$) over the field $\mathbb{F}_q$ has $q^2 + q + 1$ points and lines, and is equivalent to the symmetric 2-$(q^2 + q + 1, q + 1, 1)$ design, which was defined in Definition 1.1.2.

q	Lines (Points)	Rank
2	7	4
3	13	7
4	21	10
5	31	16
7	57	29
8	73	28
9	91	37
11	133	66

Table 8.1: Lines and rank of incidence matrix of $PG(2,q)$.

We define the incidence matrix $A = (a_{ij})$ of the projective plane $PG(2,q)$, $q = p^h$, p prime, $h \geq 1$, as the matrix whose rows are indexed by lines of the plane and whose columns are indexed by points of the plane, and with entry

$$a_{ij} = \begin{cases} 1 & \text{if point } j \text{ belongs to line } i, \\ 0 & \text{otherwise.} \end{cases}$$

The rank of the incidence matrix A is

$$\binom{p+1}{2}^h + 1.$$

Table 8.1 gives the number of lines (points) and rank of $PG(2,q)$ for $q \leq 11$.

The p-ary code C of the projective plane $PG(2,q)$, $q = p^h$, p prime, $h \geq 1$, is the $\mathbb{F}_p$-span of the rows of the incidence matrix A. The references [1] and [39] contain a lot of information on codes from planes.

In particular, in [1], it is proven that the scalar multiples of the incidence vectors of the lines are the codewords of minimal weight $q+1$ in the code arising from $PG(2,q)$. Chouinard [12] proved that for the code arising from $PG(2,p)$, p prime, there are no codewords of weight in the interval $[p+2, 2p-1]$ and that the only codewords of weight $2p$ are the scalar multiples of the differences of the incidence vectors of two distinct lines.

We will improve the result of Chouinard by characterising the codewords up to weight $2p+\frac{p-1}{2}$, for $p \geq 19$. We show that the only possible non-zero weights are $p+1, 2p$, and $2p+1$, and prove that codewords of weight $2p+1$ are a linear combination of two incidence vectors of lines, with the linear combination non-zero in the intersection point of the two lines. To obtain these results, we will use a particular basis for the code C, found by E. Moorhouse, see [37]. This basis is described in Section 8.2. In Section 8.3, we describe how the computer approach helped us to obtain new results. Finally, Section 8.4 discusses the main result: the characterization of all the codewords of weight up to $2p + (p-1)/2$ if $p \geq 19$.

8.2 The Moorhouse basis for $AG(2,p)$, p prime

The rank of the p-ary linear code of the projective plane $PG(2,p)$, p prime, is $\binom{p+1}{2} + 1$ and the rank of the p-ary linear code of the affine plane $AG(2,p)$, p prime, is $\binom{p+1}{2}$. In

[37], Moorhouse gives an easy construction for a basis for the code of $AG(2,p)$, p prime. $AG(2,p)$ can be seen as the projective plane $PG(2,p)$, with one line M omitted.

Consider the $(p^2+p+1)\times(p^2+p+1)$ incidence matrix A of $PG(2,p)$ with the line M as the first row:

$$A = \begin{pmatrix} 1 & \dots & 1 & 0 & \dots & 0 \\ * & \dots & * & & & \\ \vdots & & \vdots & & B & \\ * & \dots & * & & & \end{pmatrix}.$$

The $(p^2+p)\times p^2$ matrix B, obtained by deleting the first row and the first $p+1$ columns of A, is the incidence matrix of $AG(2,p)$. Moorhouse gives the following basis for the row space of B, in which $r_0, r_1, \dots, r_p$ are the points of M:

for $i \in \mathbb{N},\ 0 \le i \le p-1$, take $p-i$ random affine lines through r_i.

These, in total, $\binom{p+1}{2}$ lines form a basis for the row space of B. When we also add the line M, we obtain a basis for the code C of $PG(2,p)$. The solid lines of Figure 8.1 give the Moorhouse basis for the code of $PG(2,p)$.

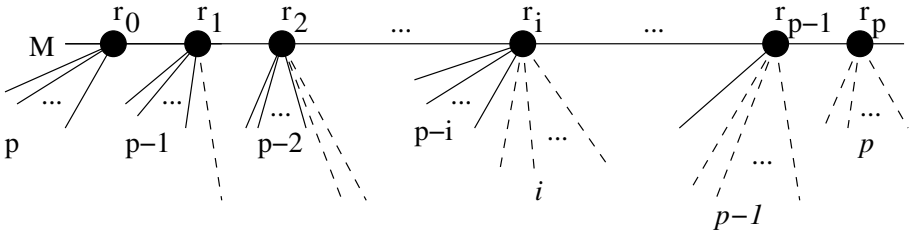


Figure 8.1: The solid lines give the Moorhouse basis for the code of $PG(2,p)$.

8.2.1 Coordinates towards the Moorhouse basis

Consider a vector space over the field $\mathbb{F}_p$ with vectors of length $n = p^2+p+1$. We need $k = \binom{p+1}{2}+1$ vectors to construct a Moorhouse basis of $PG(2,p)$. To determine the coordinates $\lambda\ (\lambda_1, \dots, \lambda_k)$ of a vector $\overline{w}(w_1, \dots, w_n)$ towards a set of basis vectors

$$\{\overline{v_1}(v_{11}, \dots, v_{1n})\ ,\ \dots\ ,\ \overline{v_k}(v_{k1}, \dots, v_{kn})\},$$

we must solve the set of equations

$$w_j = \sum_{i=1}^{k} \lambda_i v_{ij} \quad ; \quad 1 \le j \le n$$

The *be.ugent.caagt.algebra* package has a *VectorSpace* class which represents a vector space over a finite field $\mathbb{F}_q$. Its methods are shown in Figure 8.2. The vector space is

regarded as a subspace of the canonical vector space of a certain dimension, i.e. vectors in this space are represented as n-tuples. We create an instance of the *VectorSpace* class by providing the finite field $\mathbb{F}_q$ and the vector length n. We use this implementation by adding all non removed rows (lines) of our incidence matrix to a *VectorSpace* through the method *extendWith(int [] vector)*. This (time-consuming) method has no effect when the given *vector* already belongs to the vector space, otherwise an extra basis vector is stored, which is not necessarely the same as the added *vector*. As such each *extendWith(int [] vector)* method call increases the rank by one or zero. After each adding of a vector, we can consult the resulting rank through the method *int getDimension()*. So the dimension is k when the added vectors form a basis.

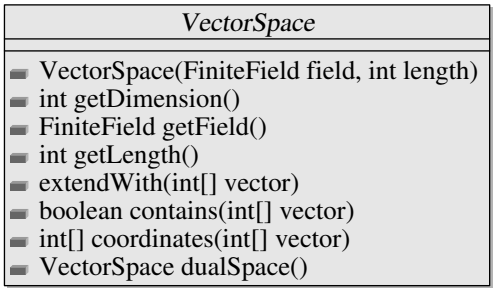


Figure 8.2: The *VectorSpace* class from the *algebra* package.

To determine the coordinates towards the set of basis vectors $\{\overline{v_1}, \ldots, \overline{v_k}\}$, we add the following list of vectors to a *VectorSpace* of length $k+1$ over the same field:

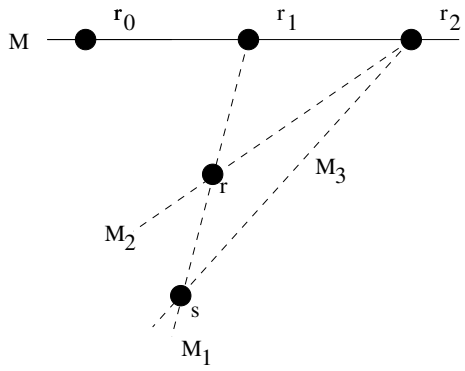
$$\left(\begin{array}{cccc|c} v_{11} & v_{21} & \ldots & v_{k1} & w_1 \\ v_{12} & v_{22} & \ldots & v_{k2} & w_2 \\ \vdots & \vdots & & \vdots & \vdots \\ v_{1n} & v_{2n} & \ldots & v_{kn} & w_n \end{array}\right) = \left(\begin{array}{cccc|c} \overline{v_1}^T & \overline{v_2}^T & \ldots & \overline{v_k}^T & \overline{w}^T \end{array}\right)$$

The *VectorSpace* class reduces these vectors to the set of k basis vectors $(I_k|\lambda^T)$, which contains the coordinates.

8.2.2 Slightly adjusted basis

By looking at the coordinates of the non-basis vectors towards the basis in $PG(2,5)$, $PG(2,7)$, $PG(2,11)$ and $PG(2,13)$, we observed the following.

Consider the Moorhouse basis B with the notation of this section. M_1 is the line through r_1 not in B. M_2 and M_3 are the lines through r_2 not in B. $r = M_1 \cap M_2$. $s = M_1 \cap M_3$. This is illustrated below.



The following holds:

- M_1 is the sum of all the lines of B through r_0 (this gives the all one vector), minus the sum of all the lines of B through r_1.
- We can write M_2 as a linear combination of all lines of B through r_0, r_1 and r_2 except line sr_0.
- We can write M_3 as a linear combination of all lines of B through r_0, r_1 and r_2 except line rr_0.

Based on this observation, the following variation of the Moorhouse basis was found. This variation of the Moorhouse basis, with $p-1$ lines through the points r_0, r_1 and r_2, is used in the characterization of all codewords of weight up to $2p+(p-1)/2$ if $p \geq 19$.

Theorem 8.2.1 *The space generated by the affine lines of the Moorhouse basis through r_0, r_1, and r_2, can also be generated by choosing $p-1$ affine lines through each of the points r_0, r_1, and r_2, with the restriction that the three non-selected affine lines are not concurrent.*

For the proof we refer to Theorem 1 of our article [17].

8.3 Computer results

Consider the lines L: $a_0X_0 + a_1X_1 + a_2X_2 = 0$ where $a_0, a_1, a_2 \in \mathbb{F}_q$, q square, such that

$$a_0^{\sqrt{q}+1} + a_1^{\sqrt{q}+1} + a_2^{\sqrt{q}+1} = 0$$

These $q\sqrt{q}+1$ lines construct a hermitian curve. An exhaustive computer search revealed the following result (checked by computer for q = 4,9,16,25,49,64,81). Subsequently, this result was proved by L. Storme.

Lemma 1 *Consider $PG(2,q)$, q square.*

1. *If we remove all $q\sqrt{q}+1$ lines of the hermitian curve, the rank of the incidence matrix of $PG(2,q)$ is* reduced by exactly one.
2. *If we remove $q\sqrt{q}$ lines of the hermitian curve (one less), the rank of the incidence matrix of $PG(2,q)$ is* not reduced.

Subsequently, the following more general result was proven. For the proof we refer to Theorem 2 of our article [17].

Theorem 8.3.1 *If a set of columns is deleted from A, then the rank of A decreases if and only if there is a codeword in C with its non-zero positions contained in the set of deleted columns of A.*

This result links nicely the non-zero positions in codewords to a rank problem regarding the incidence matrix A of $PG(2, q)$. Therefore it is interesting to investigate, by computer, which removal of columns reduces the rank. A standard backtracking algorithm is used in which m columns are removed recursively from the $(p^2 + p + 1) \times (p^2 + p + 1)$ incidence matrix A of $PG(2, p)$. For each set of m removed columns, we calculate the remaining rank of the incidence matrix A (without the removed columns) and investigate the properties of the set of removed columns. Removing a column from the incidence matrix A means removing a line from $PG(2, p)$, when we consider the columns to be the lines. Since we have a removed *set* of columns, the order in which columns are removed is not important. We number the columns from 1 upto p^2+p+1. We eliminate equivalent removals by removing columns from small to large. E.g., the removal $(1, 5, 9)$ is equal to the removal $(5, 1, 9)$, so we will not generate the last removal. More formally, when removing m columns exhaustively, we generate inequivalent

$$(c_1, c_2, \ldots, c_m)$$

removals with

$$c_1 < c_2 < \ldots < c_m.$$

In each recursive step, we never remove a column which is smaller than the last removed column. The straight forward exhaustive backtracking algorithm, which only incorporates an obvious bound check at line 2, is given in Algorithm 8.1. The method *calculateRank()* at line 8 creates the mentioned vector space from all non-removed column incidence vectors from the incidence matrix.

Algorithm 8.1 Trivial exhaustive rank calculation

```
function remove(int lastRemovedColumn, int columnsToRemove)
 1 if columnsToRemove > 0 then
 2     if columnsToRemove ≤ numberOfColumns - lastRemovedColumn then
 3         for column from lastRemovedColumn + 1 upto numberOfColumns do
 4             removeColumn(column)
 5             remove(column, columnsToRemove - 1)
 6             addColumn(column)
 7 else
 8     calculateRank()
 9
function main(int columnsToRemove)
10 remove(0, columnsToRemove)
```

We start from a full matrix and gradually remove columns. We know that with each removal of a column the rank is reduced by at most one. The used implementation from the

algebra package builds up a vector space by adding vectors to it one by one. Each vector add involves a diagonalization algorithm which adds the vector to the vector space and so it determines wether the rank increases by one or not. A way to reuse the rank calculation is possible, as illustrated by the following small example. Consider the 13×13 incidence matrix of $PG(2,3)$, with columns numbered from 1 upto 13. Suppose that we remove 4 columns exhaustively, so we generate all ordered removals (a,b,c,d) with $a < b < c < d$. Suppose the algorithm removed columns 1 and 4. Now we know that columns 2 and 3 will not be removed in this part of the search, therefore we create the vector space V_1 of columns 2 and 3. Suppose we remove 8 in the next recursive step. We make a copy of V_1 to V_2 and add columns 5, 6 and 7 to V_2. Finally 10 is the last removed column, therefore we copy V_2 to V_3 and add columns 9, 11, 12 and 13 to V_3. So the rank calculation is done incrementally instead of at the end, which was the case in Algorithm 8.1. The given description naturally implies the implicit use of a stack of vector spaces.

What about isomorph rejection? We use the *nauty* software [36] to calculate the orbits of the set of non-removed columns with respect to the set of removed columns. Since *nauty* can only work with graphs, we create a bipartite graph of order $2(p^2+p+1)$ in which we assign color 0 to the removed columns, color 1 to the non-removed columns and color 2 to the rows. From each orbit in the set of non-removed columns, we choose only one column to remove in the recursive step. To be compatible with the generation method, we remove only the smallest column from each orbit which is larger than the last removed column. This prunes a lot at low depth in the search tree, calling *nauty* in those early stages does not introduce a bottleneck. The most time consuming part is the rank calculation.

Algorithm 8.2 incorporates isomorph rejection and the reuse of the rank calculation. Note that we can start with removing any two lines. The *clone()* method (used at lines 5 and 14) of *VectorSpace* produces an exact copy of the vector space instance. The *partitionNonRemovedColumns()* method (used at line 7) creates a set of integer numbers. This set contains the smallest column out of each orbit larger the last removed column.

Applying Algorithm 8.2 on the smallest $PG(2,p)$'s revealed some properties about the removed set of columns. From now on, we use the term "lines" instead of "columns". Tables 8.2 and 8.3 show the result for $PG(2,3)$ and $PG(2,5)$, respectively. These tables show what rank (table columns) is left when removing a certain amount (table rows) of lines. An empty entry indicates that no such line removal leads to the rank. Otherwise, a value denotes the size of the largest subset of concurrent lines of a certain removal, its subscript is the number of such subsets. We use dots when more possibilities than the listed ones are possible. A star (*) indicates that the subsets of concurrent lines are disjoint. From the tables we see that when removing less than $2p$ lines, the rank decreases if and only if we remove all lines through one point (indicated by 4_1 in Table 8.2 and 6_1 in Table 8.3). When removing $2p$ lines, the rank can also decrease by removing all lines through two points, but not the joining line (indicated by 3_{2*} in Table 8.2 and 5_{2*} in Table 8.3). The rank is reduced by two when removing all lines through two points (indicated by 4_2 in Table 8.2 and 6_2 in Table 8.3).

When removing all lines through three points, the rank sometimes decreases by 3 and sometimes by 4 (indicated by 6_3 in Table 8.3 in columns of rank 13 and 12). A closer look at all possibilities when removing all lines through three points revealed the following (confirmed by computer for p prime, $p \leq 23$).

Algorithm 8.2 Exhaustive rank calculation with rank reuse and isomorph rejection

```
 1 {returns sorted set with smallest column out of each orbit > lastRemovedColumn}
function SortedIntSet partitionNonRemovedColumns(int lastRemovedColumn)
 2
function remove(int lastRemovedColumn, int columnsToRemove, VectorSpace space)
 3 if columnsToRemove > 0 then
 4     if columnsToRemove ≤ numberOfColumns - lastRemovedColumn then
 5         VectorSpace superSpace ← space.clone()
 6         SortedIntSet columnSet
 7         ← partitionNonRemovedColumns(lastRemovedColumn)
 8         for all column in columnSet do
 9             removeColumn(column)
10             remove(column, columnsToRemove - 1, superSpace)
11             superSpace.extendWith(column)
12             addColumn(column)
13 else
14     VectorSpace superSpace ← space.clone()
15     for column from lastRemovedColumn + 1 upto numberOfColumns  do
16         superSpace.extendWith(column)
17     OUTPUT superSpace.getDimension()
18
function main(int columnsToRemove, FiniteField field, int n)
19 removeColumn(1)
20 removeColumn(2)
21 remove(2, columnsToRemove-2, new VectorSpace(field, n))
```

Removed \ Rank	7	6	5	4
3	2_3,3_1			
4 ($= p+1$)	2_6,3_1	4_1		
5	3_1,3_2	4_1		
6 ($= 2p$)	3_3,3_4	3_2*,4_1		
7 ($= 2p+1$)		3_5,3_6,4_1	4_2	
8			3_8,4_1,4_2	
9 ($= 3p$)				$4_2, 4_3, 3_{12}$

Table 8.2: Exhaustive line removal in $PG(2,3)$, showing what possible rank (table columns) is left when removing a certain amount (table rows) of lines. The meaning of the numbers is explained in the text.

Removed \ Rank	16	15	14	13	12
5	$5_1, \ldots$				
6 $(= p+1)$	$5_1, \ldots$	6_1			
7	$5_1, \ldots$	6_1			
8	$5_1, \ldots$	6_1			
9	$5_2, \ldots$	6_1			
10 $(= 2p)$	$5_2, \ldots$	$6_1, 5_2*$			
11 $(= 2p+1)$	$5_2, \ldots$	$6_1, 5_2*$	6_2		
12	$\ldots$	$6_1, 5_2*, 4_3*$	6_2		
13	$\ldots$	$6_1, 5_2, \ldots$	6_2		
14	$\ldots$	$\ldots$	$6_2, 6_1, 5_2*$		
15 $(3p)$	$\ldots$	$\ldots$	$6_2, 6_1, 4_9$	$6_3, 6_2, 5_3*$	
16		$\ldots$	$\ldots$	$6_3, 6_2, 5_3*, 4_{12}$	6_3

Table 8.3: Exhaustive line removal in $PG(2,5)$, showing what possible rank (table columns) is left when removing a certain amount (table rows) of lines. The meaning of the numbers is explained in the text.

Theorem 8.3.2 *If all lines of $PG(2,p)$, p prime, through three collinear points are deleted, then the rank of the incidence matrix decreases by four. If all lines of $PG(2,p)$, p prime, through three non-collinear points are deleted, then the rank of the incidence matrix decreases by three.*

Proof: We prove this by use of the Moorhouse basis. We use the notations of Section 8.2, i.e. $r_0, r_1, \ldots, r_p$ are the points of the line M defining the affine plane $AG(2,p)$.

Case 1: We delete all lines through the points r_{p-2}, r_{p-1}, r_p of M. For $i \in \mathbb{N}$, $0 \leq i \leq p-3$, take all lines (different from M) through r_i. These lines give a matrix of rank $\sum_{i=0}^{p-3}(p-i) = (\binom{p+1}{2}+1) - 4$. The rank decreased by four.

Case 2: We delete all lines through the points r_{p-1}, r_p and r (r not on M). For the point r_0, we only have $p-1$ lines available for the Moorhouse basis (not $r_0 r$). For $i \in \mathbb{N}$, $1 \leq i \leq p-2$, we have $p-i$ lines through r_i available for the Moorhouse basis. So the rank is at least $(p-1) + \sum_{i=2}^{p-1} i = \binom{p+1}{2} - 2$.

Suppose that we have rank $\binom{p+1}{2} - 1$, then by results of Moorhouse [37, Theorem 6.1], we have the net defined by the directions $r_0, \ldots, r_{p-2}$, including the line $r_0 r$. But it is impossible to have $r_0 r$ as a linear combination of the other chosen $\binom{p+1}{2} - 2$ lines, because r is not on any of those lines. So the rank is $(\binom{p+1}{2}+1) - 3$. The rank decreased by three. □

When removing $3(p-1)$ lines, the rank can also be reduced by removing $p-1$ lines through three points (indicated by 4_3* in Table 8.3). A closer look, confirmed by computer for $p \leq 29$, p prime, gave the following result.

Theorem 8.3.3 *If in $PG(2,p)$, p prime, $p-1$ lines through three collinear points a, b and*

c, but not their joining line, are deleted, then the rank decreases if the three non-removed lines M_1, M_2 and M_3 ($\neq ab$) through respectively a, b and c are concurrent.

The unique codeword which corresponds to the removal of these lines is, up to equivalence, given by

$$(\underbrace{1,2,\ldots,p-1}_{\text{lines through } a},\underbrace{1,2,\ldots,p-1}_{\text{lines through } b},\underbrace{1,2,\ldots,p-1}_{\text{lines through } c},0,\ldots,0).$$

Proof: Let $M = ab$ be the line at infinity of the corresponding affine plane $AG(2,p)$. Let a, b, and c be the points at infinity of respectively the vertical, horizontal, and diagonal lines. Suppose that M_1, M_2, M_3 all pass through the origin $(0,0)$.

We give the coordinate positions of the $p-1$ remaining affine lines through a, b and c the following values, and we prove that the constructed vector indeed is a codeword.

In the coordinate positions of the lines $X = \alpha$, we put the value α. In the coordinate positions of the lines $Y = \beta$, we put the value $-\beta$, and in the coordinate positions of the lines $Y = X + \beta - \alpha$, we put the value $\beta - \alpha$. All other coordinate positions are zero. Note that the coordinate values of the lines M_1, M_2 and M_3 are indeed zero.

Let the incidence matrix A of $PG(2,p)$ have rows corresponding to the points of $PG(2,p)$. We show first of all that the constructed vector c is orthogonal to all the rows of A.

The vector c is orthogonal to the rows of A corresponding to the points a, b and c, since $\sum_{i=1}^{p-1} i \equiv 0 \pmod{p}$. The vector c is also orthogonal to the rows of A corresponding to the other points at infinity since these points lie on none of the lines with non-zero coordinates.

An affine point (a,b) lies on the lines $X = a$, $Y = b$, and $Y = X + b - a$, so the sum of the corresponding coordinate values is $a - b + b - a = 0$.

We have shown that c is orthogonal to all the rows of A, hence $c \in C^{\perp}$.

But $C^{\perp} \subset C$. This is proven in the following way. The code C is a $[p^2+p+1, (p^2+p)/2+1]$-code, so $C^{\perp}$ is a $[p^2+p+1, (p^2+p)/2]$-code. But $\mathrm{Hull}(C) = C \cap C^{\perp}$ is a code of dimension $(p^2+p)/2$ [1]. So this shows that $C^{\perp} \subset C$. Hence, $c \in C^{\perp}$ also implies $c \in C$.

This shows that there is a codeword of C with its non-zero positions in the $3(p-1)$ positions of the deleted lines through a, b and c. So, by Theorem 8.3.1, the rank of A decreases when deleting these $3(p-1)$ columns from A. □

Theorem 8.3.4 *If in $PG(2,p)$, p prime, $p-1$ lines through three collinear points a, b and c, but not their joining line, are deleted, then the rank does not decrease if the three non-removed lines M_1, M_2 and M_3 ($\neq ab$) through respectively a, b and c are non-concurrent.*

Proof: Let r_0 be a point of the line ab, different from a, b and c. Let $\{r_1\} = M_1 \cap M_2$ and let $M = r_0 r_1$. Let $\{r_2\} = M \cap M_3$.

We construct a Moorhouse basis for the affine plane defined by the line M. Through r_0, we have the p necessary lines for the affine Moorhouse basis. Through r_1, we have the $p-1$ necessary affine lines for the Moorhouse basis since the only line through r_1 that cannot be used is the line $r_1 c$. Through r_2, we have the $p-2$ necessary affine lines for the Moorhouse basis since only the lines $r_2 a$ and $r_2 b$ cannot be used. Through all the remaining points of

M, we have $p-3$ affine lines available for the Moorhouse basis. Finally, M can be used to construct the final line for the basis of the code of $PG(2,p)$.

So the rank of the incidence matrix of $PG(2,p)$ does not decrease, the $3(p-1)$ deleted lines are not the non-zero positions of a codeword of the p-ary linear code defined by $PG(2,p)$. □

Corollary 8.3.5 *If in $PG(2,p)$, p prime, $p-1$ lines through three collinear points a, b and c, but not their joining line, are deleted, then the rank decreases if and only if the three non-removed lines M_1, M_2 and M_3 ($\neq ab$) through respectively a, b and c are concurrent.*

The proof of Theorem 8.3.3 gives us an algorithm to construct the codeword which corresponds to the removal of $p-1$ lines through three collinear points a, b and c. Consider:

- Select an arbitrary line M with three points a, b, c on it in $PG(2,p)$, p prime.
- Remove $p-1$ random lines through a ($\neq M$), let M_1 be the non-removed line ($\neq M$).
- Remove $p-1$ random lines through b ($\neq M$), let M_2 be the non-removed line ($\neq M$).
- Let point $r = M_1 \cap M_2$.
- Remove the $p-1$ lines through c different from line M and line $M_3 = rc$.
- The removed lines through a are refered to as $A_i (1 \leq i \leq p-1)$.
- The removed lines through b are refered to as $B_i (1 \leq i \leq p-1)$.
- The removed lines through c are refered to as $C_i (1 \leq i \leq p-1)$.
- Let $\overline{A_i}$ be the incidence vector of line A_i (equivalent for $\overline{B_i}$ and $\overline{C_i}$).

Assume the codeword which corresponds to this removal of lines is written as

$$(\dot{A}_1, \dot{A}_2, \ldots, \dot{A}_{p-1}, \dot{B}_1, \dot{B}_2, \ldots, \dot{B}_{p-1}, \dot{C}_1, \dot{C}_2, \ldots, \dot{C}_{p-1}, 0, \ldots, 0)$$

in which, e.g., the dotted $\dot{A}_1$ is the non-zero coordinate corresponding to line A_1. Assume the codeword is such that the linear combination of the incidence vectors of the corresponding removed lines is the $\overline{0}$ vector, i.e.

$$\sum_{i=1}^{p-1} (\dot{A}_i \overline{A_i} + \dot{B}_i \overline{B_i} + \dot{C}_i \overline{C_i}) = \overline{0}$$

Algorithm 8.3 constructs this codeword. It is easy to see that Algorithm 8.3 assigns:

- $\dot{A}_i = i$ $(1 \leq i \leq p-1)$
- $\dot{B}_i = \dot{C}_i = p - i (1 \leq i \leq p-1)$

Algorithm 8.3 Codeword construction of removing $3(p-1)$ lines.

function constructCodeword()
  $A_1 \leftarrow$ Any A_i line
  $\dot{A}_1 \leftarrow 1$
  $B_1 \leftarrow$ Line through b and $A_1 \cap M_3$
  $\dot{B}_1 \leftarrow p-1$
  $C_1 \leftarrow$ Line through c and $A_1 \cap M_2$
  $\dot{C}_1 \leftarrow p-1$
  for i **from** 2 **upto** $p-2$ **do**
    $A_i \leftarrow$ Line through a and $B_{i-1} \cap C_1$
    $\dot{A}_i \leftarrow (2p - \dot{B}_{i-1} - \dot{C}_1) \bmod p$
    $B_i \leftarrow$ Line through b and $A_i \cap M_3$
    $\dot{B}_i \leftarrow p - \dot{A}_i$
    $C_i \leftarrow$ Line through c and $A_i \cap M_2$
    $\dot{C}_i \leftarrow p - \dot{A}_i$

By a similar construction, we can show that the codeword corresponding to the removal of p lines through two points a and b, but not their joining line is

$$(\underbrace{1,1,\ldots,1}_{p \text{ lines through } a}, \underbrace{-1,-1,\ldots,-1}_{p \text{ lines through } b}, 0,\ldots,0)$$

We also considered the removal of $4(p-2)$ lines through 4 collinear points a, b, c and d (but not their joining line), in which the 8 non-removed lines can be partitioned into two disjoint sets of concurrent lines. We used a similar construction as in Algorithm 8.3. The unique codeword of weight $4(p-2)$ was found by an exhaustive computer search for $PG(2,p)$, p prime, $p \leq 23$. The only remarkable thing about these codewords is that, for every $p-2$ lines through a point, we have $(p-3)/2$ times twice the same value, and then once some other value. As an example, a codeword of the code of $PG(2,5)$ corresponding to such a removal is given by

$$(\underbrace{1,1,3}_{3 \text{ lines through } a}, \underbrace{3,3,4}_{3 \text{ lines through } b}, \underbrace{2,4,4}_{3 \text{ lines through } c}, \underbrace{1,2,2}_{3 \text{ lines through } d}, 0,\ldots,0)$$

And a codeword of the code of $PG(2,11)$ is

$$(\underbrace{1,1,3,3,4,6,6,10,10}_{9 \text{ lines through } a}, \underbrace{1,1,2,2,4,4,5,7,7}_{9 \text{ lines through } b}, \underbrace{4,4,5,5,6,6,8,8,9}_{9 \text{ lines through } c}, \underbrace{1,1,3,3,4,6,6,10,10}_{9 \text{ lines through } d}, 0,\ldots,0)$$

The following two theorems can be proved using the Moorhouse basis. These theorems show Chouinard's result in another way.

Theorem 8.3.6 *If we remove less than $2p$ lines in $PG(2,p)$, p prime, and the rank decreases, then we remove all lines through one point. The incidence vectors of all lines through one point are the codewords of weight $p+1$.*

Theorem 8.3.7 *If you throw away $2p$ lines in $PG(2,p)$, p prime, and the rank decreases, then either you throw away all lines through one point, or you throw away p lines through two points r_{p-1} and r_p, but not the line $r_{p-1}r_p$. The difference of the incidence vectors of all lines through two points are the codewords of weight $2p$.*

8.4 Improved results for $PG(2,p)$, p prime

We will use Theorem 8.3.1 to improve the results of Chouinard who characterized all the codewords in the code of $PG(2,p)$, p prime, of weight at most $2p$ [12].

Since we let the columns of the incidence matrix A correspond to the lines of $PG(2,p)$ and the rows to the points of $PG(2,p)$, deleting columns from the incidence matrix A then corresponds to deleting a set B of lines of $PG(2,p)$. The rank of A only decreases when it is not possible to reconstruct a basis for the column space of A by using the non-deleted lines of $PG(2,p)$.

A possible way for constructing a basis for the column space of A is by trying to construct a Moorhouse basis for an affine space contained in $PG(2,p)$ by using the lines not in B, and then by finding a final $(p^2+p+2)/2$-th line which extends this basis of $AG(2,p)$ to a basis of $PG(2,p)$. This is the method we will apply.

All codewords of weight up to $2p$ in the code arising from $PG(2,p)$, p prime, are known by the results of Assmus and Key [1], and Chouinard [12]. We characterize all codewords c, with $2p+1 \leq wt(c) \leq 2p+\frac{p-1}{2}$, by induction on the weight of the codewords.

In the induction hypothesis, we assume that the codewords of weight smaller than $wt(c)$ are already classified as being either:

1. a codeword of weight $p+1$ which is, up to a scalar multiple, the incidence vector of all lines through one point r,

2. a codeword of weight $2p$ which is, up to a scalar multiple, the difference of the incidence vectors of all lines through two points r and r',

3. a codeword of weight $2p+1$ which is a linear combination $\alpha c_1 + \beta c_2$ of the incidence vectors c_1 and c_2 of all lines through two points r and r', with $\alpha+\beta \neq 0$.

We also rely on a result of Ball on dual double blocking sets.

Definition 8.4.1 *A* dual double blocking set *of $PG(2,q)$ is a set B of lines such that each point of $PG(2,q)$ belongs to at least two lines of B.*

Theorem 8.4.2 (Ball [2]) *A double blocking set in $PG(2,p)$, p prime, has at least size $(5p+5)/2$.*

Suppose now that c is a codeword with $wt(c) = 2p+i$, with $i \in [1, \frac{p-1}{2}]$, where we assume that there are no codewords of weight in the interval $[2p+2, 2p+i-1]$. The non-zero positions in such a codeword define a set B of lines such that if the columns in A corresponding to these lines are deleted, the rank of A decreases (Theorem 8.3.1).

We now study all cases in which we delete at most $2p + \frac{p-1}{2}$ lines corresponding to the set of non-zero positions of a codeword c of C. The set of deleted lines is denoted by B.

Case 1: Suppose that there is a point r_0 on zero lines of B.

If at most $2p + \frac{p-1}{2}$ lines are deleted, we can select and delete two lines through r_0, then at most $2p + \frac{p+3}{2}$ lines are deleted. So there remains a point r_1 on at most one deleted line since a dual blocking set in $PG(2,p)$ has at least $(5p+5)/2$ lines (Theorem 8.4.2).

Let $M = r_0 r_1$ and let M be the line at infinity of the corresponding affine plane $AG(2,p)$ of $PG(2,p)$. Note that $M \notin B$. Let $r_0, \ldots, r_p$ be the points of M. We check whether we can reconstruct the Moorhouse basis for $AG(2,p)$. Using the notations of the beginning of Section 8.2, through the point r_i, there pass $p-i$ affine lines of the Moorhouse basis.

By induction on the index i for r_i, we can select $p-i$ affine lines through a point r_i, $2 \leq i \leq p$, of M for the Moorhouse basis if $\frac{2p+\frac{p-1}{2}}{p-i+1} < i+1$ since then there is a point in the set $\{r_i, \ldots, r_p\}$ lying on less than $i+1$ lines in B. The previous condition is equivalent to $i + 1 + \frac{p-1}{2(i-1)} < p$.

This is satisfied for all $i \leq p-2$ when $p > 5$.

Problems arise when all lines through r_{p-1} and r_p, different from the line $r_{p-1}r_p$, belong to B since we need one affine line through r_{p-1} for the Moorhouse basis.

If all affine lines through r_{p-1} and r_p are deleted, then this means that in the corresponding codeword c, the coordinates corresponding to these $2p$ lines all have non-zero entries. So two out of the p deleted lines through r_{p-1} have the same non-zero entry. We rescale c so that at least these two entries are one, i.e.

$$c = (\underbrace{0}_{\text{line } r_{p-1}r_p}, \underbrace{1,1,*,\ldots,*}_{p \text{ affine lines through } r_{p-1}}, \underbrace{*,\ldots,*}_{p \text{ affine lines through } r_p}, *,\ldots,*).$$

The codeword c' of weight $2p$ defined by the $2p$ affine lines through r_{p-1} and r_p is, up to a scalar multiple,

$$c' = (\underbrace{0}_{\text{line } r_{p-1}r_p}, \underbrace{1,\ldots,1}_{p \text{ affine lines through } r_{p-1}}, \underbrace{-1,\ldots,-1}_{p \text{ affine lines through } r_p}, 0,\ldots,0).$$

Then

$$c - c' = (\underbrace{0}_{\text{line } r_{p-1}r_p}, \underbrace{0,0,*,\ldots,*}_{p \text{ affine lines through } r_{p-1}}, \underbrace{*,\ldots,*}_{p \text{ affine lines through } r_p}, *,\ldots,*).$$

So $wt(c-c') < wt(c)$. By induction on $wt(c)$, $2p+1 \leq wt(c) \leq 2p + \frac{p-1}{2}$, we can assume that $c - c'$ is already classified as being either:

1. a codeword of weight $p+1$ which is, up to a scalar multiple, the incidence vector of all lines through one point r,

2. a codeword of weight $2p$ which is, up to a scalar multiple, the difference of the incidence vectors of all lines through two points r and r',

3. a codeword of weight $2p+1$ which is a linear combination $\alpha c_1 + \beta c_2$ of the incidence vectors c_1 and c_2 of all lines through two points r and r', with $\alpha + \beta \neq 0$.

All three possibilities show that c can be written as a linear combination of at most three codewords of weight $p+1$, so a linear combination of at most three incidence vectors of all lines through points r, r', and r''. Since $wt(c) \leq 2p + \frac{p-1}{2}$, we deduce that c is a linear combination of at most two such codewords of weight $p+1$. Hence, c is described as written in one of the three possibilities above.

Now we can assume that not all lines through r_{p-1}, different from $r_{p-1}r_p$, are deleted. We use one of them for the Moorhouse basis. Then select the line r_0r_1 through r_p to obtain a basis of size $\frac{p^2+p}{2}+1$ for the code of $PG(2,p)$.

In this latter case, we have reconstructed a basis for the column space of A. The rank of A has not decreased, so the set B of deleted lines cannot correspond to a codeword of the code of $PG(2,p)$ (Theorem 8.3.1).

Case 2: Suppose that every point of $PG(2,p)$ lies on at least one line of B.

Then there is a point on exactly one deleted line, since a double blocking set in $PG(2,p)$, p prime, has size at least $2p + \frac{p+5}{2}$, see Theorem 8.4.2.

Case 2.1: Suppose that there is a line $L \in B$ containing two points only lying on the line L of B.

Let r_0, r_1 be two points lying on exactly one line L of B, thus $L = r_0r_1$.

We try to reconstruct the Moorhouse basis for the affine plane defined by L. Like in Case 1, problems only start to arise when all lines through r_{p-1} and r_p belong to B, now including the line L. As in Case 1, we can reduce the codeword c by the codeword c', which corresponds to all affine lines through r_{p-1} and r_p, to a codeword $c-c'$ of lower weight. So these codewords $c-c'$ are classified, leading to the same characterization for c as in Case 1.

So we can assume that at least one affine line through r_{p-1} is not deleted.

Suppose that all lines through r_p belong to B, then, the $p+1$ positions in c corresponding to the lines through r_p are non-zero. At least two of those positions have the same non-zero value; assume this value is equal to one.

Consider the codeword $c' = (\underbrace{1,\ldots,1}_{p+1 \text{ times}},0,\ldots,0)$ with a one in the positions corresponding to the lines through r_p. Then $c-c'$ is a codeword of weight at most $wt(c)-2$. By induction on the weight, we can assume that the codeword c is already characterized. So either we get a basis for the code C, or $c-c'$ is a codeword already characterized as being a linear combination of at most two codewords of minimal weight $p+1$. Then c is a codeword which is a linear combination of at most three codewords of minimal weight. In fact, since $wt(c) \leq 2p + (p-1)/2$, c is a linear combination of at most two codewords of minimal weight.

If *not* all lines through r_p belong to B, we can select a line through r_p, not in B, as the $(p^2+p+2)/2$-th line for a basis of the code of $PG(2,p)$, p prime. But this then implies

that the set B of deleted lines does not correspond to a codeword (Theorem 8.3.1).

Case 2.2: Suppose that there is a line $L \in B$ containing at least one point r_0 only lying on the line L of B and at least one point r_1 lying on exactly two lines of B.

This case is discussed in the same way as Case 2.1.

Case 2.3: Suppose that there is a line $L \in B$ such that all points of L belong to at least two lines of B, and containing three points r_0, r_1, r_2 lying on exactly two lines of B.

Let M_0, M_1, M_2 be the lines, different from L, lying in B and passing through respectively r_0, r_1, r_2.

Let L be the line at infinity of the corresponding affine plane for which we try to construct the Moorhouse basis.

Case 2.3.1: Suppose that M_0, M_1, M_2 are not concurrent.

From Theorem 8.2.1, we know that the affine lines through r_0, r_1, and r_2, not belonging to B, generate the same vector space as the lines of the Moorhouse basis through these points generate. We can find enough lines through the points $r_i,\ i \in \mathbb{N},\ 3 \leq i \leq p$, of L if $\frac{2p+\frac{p-9}{2}}{p-i+1} < i+1$. Note that M_0, M_1, M_2 and L are not considered in this inequality.

As before, problems only start to arise if all affine lines through r_{p-1} and r_p belong to B. But then it is impossible that all points of L lie on at least two lines of B. Hence, there are no problems to select an affine line through r_{p-1} for constructing the Moorhouse basis for $AG(2,p)$.

If all lines through r_p are deleted, as in Case 2.1, we can again reduce c to a codeword of lower weight (known by induction on the weight).

If not all lines through r_p are deleted, as in Case 2.1, we reconstruct a basis for the code C to obtain the same contradiction.

Case 2.3.2: Suppose that M_0, M_1, M_2 are concurrent in a point r.

Let c be the codeword corresponding to the set B of deleted lines. Let c' be the codeword corresponding to the $p+1$ lines through r. Let c and c' have the same non-zero symbol in the coordinate position corresponding to the line $r_0 r$. Then $c - c'$ is a new codeword of weight at most

$$\underbrace{2p + \frac{p-1}{2}}_{wt(c)} \quad + \quad \underbrace{(p-2)}_{\text{lines } r_i r \ ;\ i=3,\ldots,p} \qquad \underbrace{-1.}_{\text{line } r_0 r \text{ is zero}}$$

So $wt(c-c') \leq 3p + \frac{p-7}{2}$. When we remove the lines corresponding to $c - c'$, we know that the point r_0 is not on any deleted affine line, and that the points r_1 and r_2 are on at most one deleted line.

A point r_i, $i > 2$, of L is on at most i deleted lines if

$$\frac{3p + \frac{p-7}{2}}{p-i+1} < i+1 \iff i+2+\frac{p-1}{2(i-2)} < p.$$

For $i = p-3$, this inequality reduces to $p > 9$. So if $p > 9$, all necessary affine lines for the Moorhouse basis of the affine plane with L as line at infinity can be selected through the points r_i of L for $i \in \mathbb{N}, 3 \leq i \leq p-3$.

We still need two affine lines through one of the points r_{p-2}, r_{p-1}, and r_p, and one affine line through one of the other points among r_{p-2}, r_{p-1}, and r_p. Problems arise when at least $p-1$ affine lines are deleted through each of the points r_{p-2}, r_{p-1}, and r_p, so at least $3(p-1)$ affine lines are deleted through these three points.

Since subtracting the codeword c' from c only affects one line through each of the points r_{p-2}, r_{p-1}, and r_p, at least $3p-6$ lines of B would necessarily pass through r_{p-2}, r_{p-1}, and r_p. This is false since $|B| \leq 2p + (p-1)/2$.

So it is possible to find a point r_{p-2} still lying on at least two affine lines not in B, which then can be selected as lines through r_{p-2} for the Moorhouse basis.

We also need at least one affine line through r_{p-1} or r_p for the Moorhouse basis. Assume that all affine lines through r_{p-1} and r_p have non-zero positions in the codeword $c-c'$. Then at least $2p-2$ of the affine lines through r_{p-1} and r_p have non-zero positions in c, so are lines of B. But then at most $2p + (p-1)/2 - 1 - (2p-2) = (p+1)/2$ other affine lines in B remain. This then contradicts the assumption that every point of L lies on a second line in B.

So we find the requested affine line through r_{p-1} for the construction of the Moorhouse basis for $AG(2,p)$.

If at least one line through r_p has a zero position in $c-c'$, then this line can be used as the $(p^2+p+2)/2$-th line for the basis of $PG(2,p)$, but then $c-c'$ does not define a codeword of the code of $PG(2,p)$, so also c does not define a codeword of the code of $PG(2,p)$.

So assume that all lines through r_p have non-zero coordinate values in $c-c'$. Add a suitable scalar multiple of the codeword c'' of weight $p+1$ defined by the lines through r_p to $c-c'$ so that some line through r_p has a zero position in $c-c'+c''$. We have a new codeword of C. But at the same time, we can construct a basis for the column space of A by using lines with zero positions in $c-c'+c''$. For, we still can use the previously determined $(p^2+p)/2$ lines of the Moorhouse basis since none of those lines passes through r_p. We now can select a line through r_p having a zero position in $c-c'+c''$ to construct a basis of the code of $PG(2,p)$. This is however impossible since $c-c'+c''$ is a codeword of C.

Summary: The preceding cases imply the following assumptions on the lines in the set B, for the cases not yet discussed.

- Every point of $PG(2,p)$ belongs to at least one line of B (consequence of Case 1).
- If a line $L \in B$ contains a point r_0 only lying on the line L of B, then all other points of L lie on at least three lines of B (consequence of Cases 2.1 and 2.2).

- If all points of a line $L \in B$ lie on at least two lines of B, and there is a point $r_0 \in L$ on exactly two lines of B, then there is at most one other point $r_1 \in L$ on exactly two lines of B. All other points lie on at least three lines of B (consequence of Case 2.3).

The preceding cases imply that a line L of B has at most two points on at most two lines of B. At most $2|B| \leq 4p + p - 1 = 5p - 1$ points lie on at most two lines of B. All the other points of $PG(2, p)$, so at least $p^2 + p + 1 - (5p - 1) = p^2 - 4p + 2$ points, lie on at least three lines of B, since we assume that every point of $PG(2, p)$ lies on at least one line of B. So the number of incidences of the points of $PG(2, p)$ with the lines of B is at least $3(p^2 - 4p + 2) + 5p - 1 = 3p^2 - 7p + 5$. But the exact number of incidences is

$$(p + 1)|B| \leq (p + 1)(2p + \frac{p - 1}{2}) = \frac{5p^2}{2} + 2p - 1.$$

So $3p^2 - 7p + 5 \leq \frac{5p^2}{2} + 2p - 1$. This is false for $p \geq 19$.

This brings us to the following theorem. We state the theorem in the original setting where the rows of A correspond to the incidence vectors of the lines of $PG(2, p)$.

Theorem 8.4.3 *The only codewords c, with $0 < wt(c) \leq 2p + \frac{p-1}{2}$, in the p-ary linear code C arising from $PG(2, p)$, p prime, $p \geq 19$, are:*

- *codewords with weight $p + 1$: the scalar multiples of the incidence vectors of the lines of $PG(2, p)$,*
- *codewords with weight $2p$: $\alpha(c_1 - c_2)$, c_1 and c_2 the incidence vectors of two distinct lines of $PG(2, p)$,*
- *codewords with weight $2p+1$: $\alpha c_1 + \beta c_2$, $\beta \neq -\alpha$, with c_1 and c_2 the incidence vectors of two distinct lines of $PG(2, p)$.*

Based on the computer results, we expect the following conjecture to be true.

Conjecture 8.4.4 *The only codewords c, with $0 < wt(c) \leq 3p - 4$, in the p-ary linear code C arising from $PG(2, p)$, p prime, $p \geq 7$, are the ones of Theorem 8.4.3. So we conjecture that there are no codewords in the interval $[2p + 2, 3p - 4]$ ($p \geq 7$, p prime). The codewords of weight $3p - 3$ are the ones of Theorem 8.3.3.*

A Farming package

A.1 Introduction

Most exhaustive backtrack search algorithms are easily split into independent pieces which generate a different part of the search tree. As an example, consider a search space which has n_d nodes at depth d of the search tree. The task of generating all nodes at depth greater than d is dividable into m independent subtasks which each generate the search trees with one of the $\frac{n_d}{m}$ nodes at depth d as the root node.

Our research group *CAAGT* has a cluster of 24 dual processors available, so the best case scenario gives a speed gain of 48. It is very improbable that all subtasks have the same execution time, so the real speed gain will be worse.

In order to make it easier to write such parallel programs, a *farming* package was written in *Java* based on *Java RMI* [1] [19]. Section A.2 explains the main ideas of a farming application. Section A.3 explains all implementation issues you need to know to write your own farming application. Finally, Section A.4 gives a basic example. This chapter is written as a tutorial to use the farming package, it does not explain all implementation issues. This farming package is being used for the course *Parallel Algorithms*.

A.2 Conceptual

A farming application consists of one master process and some slave processes, which preferably all run on separate processors. In short, we write *master* and *slave* for master and slave process, respectively. The master manages the tasks which are to be executed by the slaves. Slaves are identified by a unique name. In most cases it is not of any significance which slave executes which task.

The communication between master and slave is illustrated in Figure A.1, on the left we have all slave actions, on the right all master actions. The master is started first (1), waiting for slaves to register themselves. Slaves are started (2) on some processor and register (3) themselves with the master, by use of a remote method call. When a slave has

[1] http://java.sun.com/products/jdk/rmi/
Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

been disallowed (4), it exits (5), otherwise it asks a task to the master (6), which creates and returns the next task (7). The slave exits (8) if it received an end marker *null* task, otherwise the slave executes its task (9) and gives the result of this task to the master (10), which handles the task result (11). Then the slave (12) repeats the procedure from (6). The master terminates when all task results have been received (13). It is also possible for tasks to write intermediate results directly to the master process through standard Java writers, this is not shown in the figures. The developed *farming* package is only suited for problems which can be chopped into coarse grained subtasks (a task should take at least a couple of seconds), this way the overhead of the network communication is negligible. Since remote calls are slow, one should not overuse them.

The activity diagram of the master and slave is shown in Figure A.2 and A.3, respectively. Note that task results may not arrive in the same order as they were handed out, it depends on the problem at hand wether this is important, if so the implementation becomes less straight forward. For our problems this is not an issue. On our department cluster, we would typically start two slaves on each dual processor, and also run the master on a separate processor.

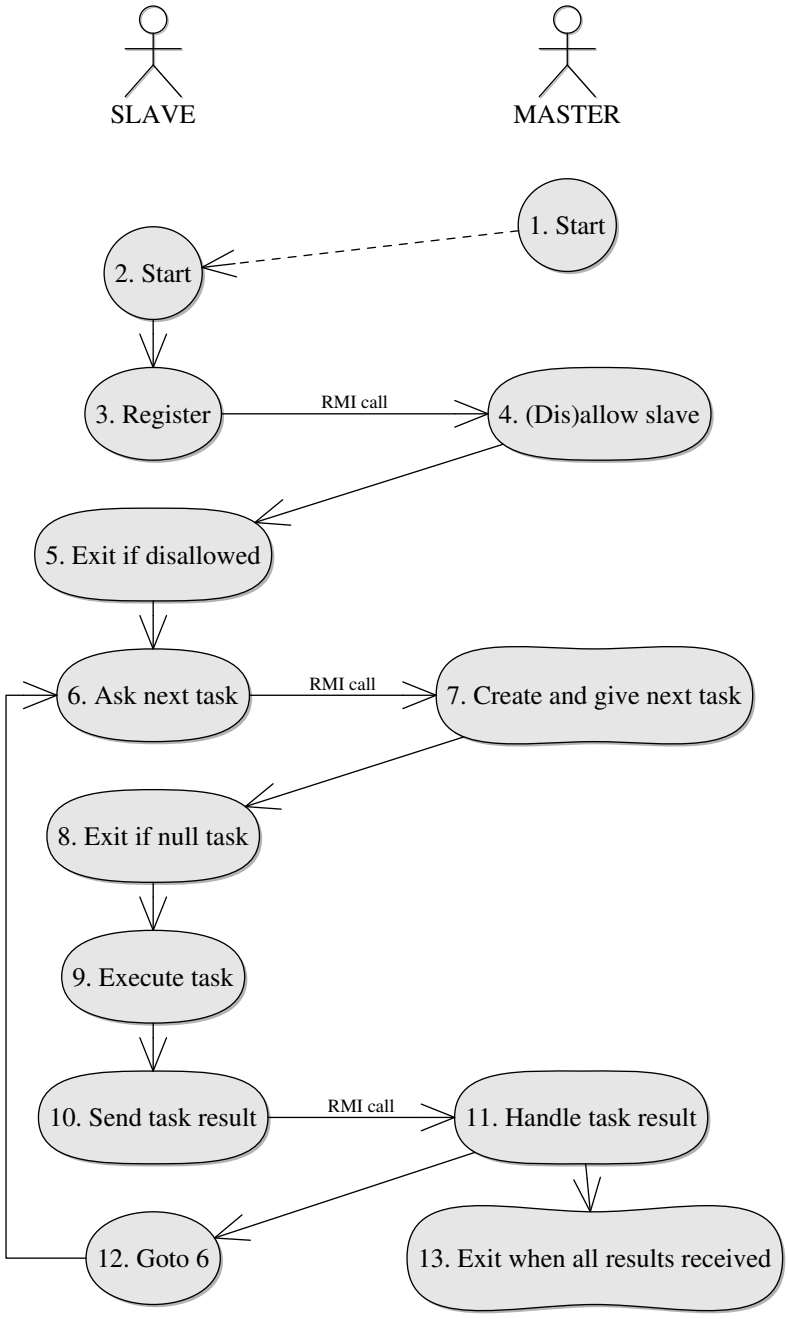


Figure A.1: Farming mechanism using remote calls

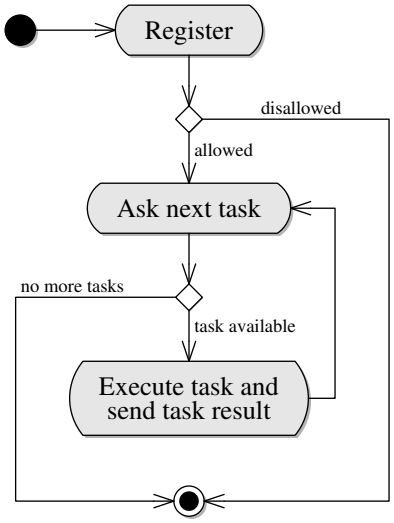


Figure A.2: Slave Activity Diagram

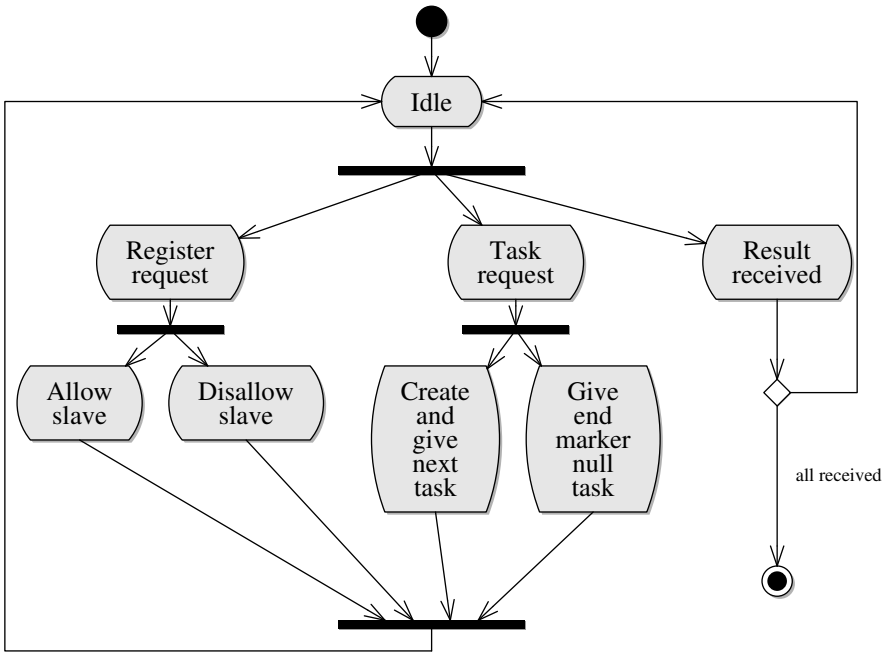


Figure A.3: Master Activity Diagram

A.3 Implementation

All classes reside in the package *be.ugent.caagt.farming*, a part of the CAAGT library. In Table A.1 we give a brief summary of all the classes, you don't need to know the details of the classes which are not marked in gray to be able to use this package.

A.3.1 The master

The master object is added to the RMI registry [2], such that slaves can call its methods remotely. The interface *RemoteMaster* contains all the methods which can be called remotely. The abstract subclass *AbstractRemoteMaster* implements most of the behaviour of *RemoteMaster*. This master interface and its subclass are shown in Figure A.4. We only show the constants of the interface *RemoteMaster*, and the methods of *AbstractRemoteMaster* you may have to call in your implementation of *AbstractRemoteMaster*. *AbstractRemoteMaster*

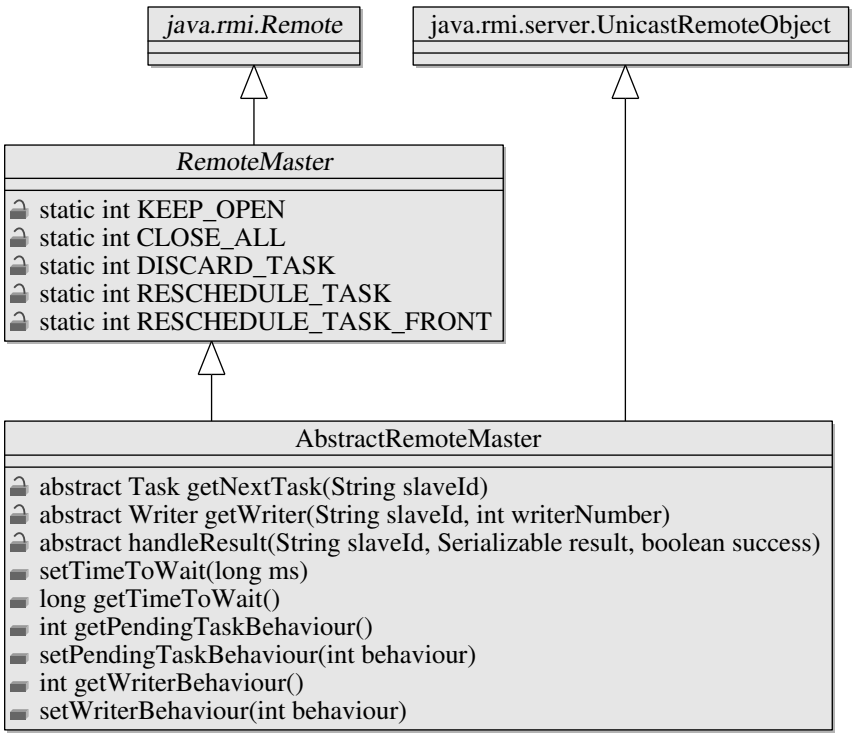


Figure A.4: Master interface and abstract subclass

has three abstract methods which remain to be implemented by your master implementation:

[2] A registry is a remote object that maps names to remote objects. A server registers its remote objects with the registry so that they can be looked up. When an object wants to invoke a method on a remote object, it must first lookup the remote object using its name. The registry returns to the calling object a reference to the remote object, using which a remote method can be invoked.

Interface	**Summary**
RemoteMaster	Top master interface which holds methods which can be called remotely.
Task	Top Task interface, a Task can be run by Slave.

Class	**Summary**
AbstractBlockingMaster	Extension of AbstractRemoteMaster which lets the slaves block when a task is not available yet, you need to subclass this.
AbstractRemoteMaster	Abstract implementation of RemoteMaster, you need to subclass this.
AbstractTask	Abstract implementation of Task, you need to subclass this.
KillSlaveWhenFinished	This class connects to the master and it instructs the master to stop giving new tasks to a certain slave.
MakeScript	This class writes a script to STD_OUT, this makes it easier to use the farming package.
MakeSecurityFile	This class writes a security file to STD_OUT.
MasterConsole	This class connects to the master through RMI and asks and shows the status of the slaves.
MasterInfo	Holds information about a master and his slaves.
RMIWriter	Writer which performs RMI calls to forward its output to the master.
Slave	Generic slave class which runs tasks handed to it by a remote master.
SlaveInfo	Holds information about a slave.

Table A.1: Summary of farming interfaces and classes

- Create and provide the next task through the method
  Task getNextTask(String slaveId):
  This returns the next *Task* to perform by slave *slaveId*.

- Handle a task result through the method
  handleResult(String slaveId, Serializable result, boolean success):
  This handles the *result* from a task performed by slave *slaveId*, *success* indicates wether the task was performed correctly.

- Provide *Writer*'s where the slave output will be send to through the method
  Writer getWriter(String slaveId, int writerNumber):
  This returns a *Writer* object to which the master redirects *Writer writerNumber* of a task executed by slave *slaveId*. Hence writers of the slave process seem to write directly to writers of the master process. Argument *slaveId* could be used when every slave has its own *Writer*, but will probably be ignored in most cases. You could use one *Writer* for solutions, one for logging, one for errors, etcetera.

Note that all arguments of a remote call must either be primitive (e.g. *int*, *boolean*, *double*, ...), or implement the interface *Serializable*. All parameters are copied and transferred ("serialized") over the network. This means that all arguments of a remote method are passed *by value*, rather than *by reference*, which would be the normal case in Java. The necessary synchronization must be provided by your implementation for these three methods. If you don't synchronize, a task could be handed out twice, or output could get scrambled. *AbstractRemoteMaster* does not synchronize these methods, since these methods are so-called *alien* methods [4] [3]. One should synchronize the parts which are really necessary.

There are only a couple of things you need to *specify* for the master process:

- The slave time-out value.

- The action the master should take when a slave has crashed or failed (reschedule the task or drop it).

- The *Writer* behaviour of the master process (specify when to close the *Writer*).

We will now give the details of these three specifications. *AbstractRemoteMaster* provides a mechanism to check wether the slaves are still alive or not. The master expects a *slaveAlive(String slaveId)* call from every slave every x seconds. So if the slaves call *slaveAlive* every $3 * x/4$ seconds, where x is sufficiently large (1000 is good for practical purposes), this protocol works fine. If the slave calls *slaveAlive* too late, its results are discarded, but it is given a new task again. You can set the slave time-out with the method *setTimeToWait(long ms)*. A slave gets the appropriate time-out value when registering. You can specify the action the master should take when a slave failed to execute its task with the method *setPendingTaskBehaviour(int behaviour)*. The available *behaviour* constants are

- *RESCHEDULE_TASK*: Task should be rescheduled after all other tasks are executed (default).

[3] Alien methods are methods implemented by the user of the package.

- *RESCHEDULE_TASK_FRONT*: Task should be rescheduled immediately.
- *DISCARD_TASK*: Task should be discarded.

Two constants specify what the master should do with the *Writer*'s of the master process when a task closes its *Writer*'s which are redirected to these *Writer*'s of the master process:

- *KEEP_OPEN*: Keep *Writer* of the master process open until all tasks are finished (default).
- *CLOSE_ALL*: Close *Writer* of the master process.

A.3.2 The slaves

There is nothing to be implemented for the slave process. A slave just continuously asks a task and executes it. You should just start an appropriate amount of *Slave*'s on each node of your cluster. Every *Slave* must have a unique name and needs to know the server name, the name of its master and the port the *rmiregistry* is listening to, so it can connect to the master. The command line arguments of *Slave* are

1. *slaveId* : The unique name of this slave.
2. *rmi://server[:port]/name* : The location of the master object by specifying the RMI *server*, the *port* to which the rmiregistry deamon listens and the *name* of the master.
3. *[library*]* : A list of libraries which the slave should load.

A.3.3 The tasks

The interface *Task* contains all the methods a slave needs to execute it properly. *Task* implements *Serializable*, therefore the whole *Task* object is serialized over the network when it is passed as a parameter. As already mentioned, the parameter is passed *by value* instead of *by reference*. The *Task* object of the slave process is *not* the same as the *Task* object created in the master process, it is merely a copy. The relevant methods are in Figure A.5.

The abstract class *AbstractTask* implements most of the behaviour of *Task*. There remain two methods to be implemented:

- The only method of the interface *Runnable*:
  run()
  This method should execute the task and save the result. To limit network traffic, you should try to create the data your task needs here, not when the task is created by the master. You can add the not well-known Java keyword **transient** for *Task* data members with should not be serialized (so they don't need to be "serializable").
- The method which retrieves the result:
  Serializable getResult()
  Gets the result produced by the last *run()* execution of this *Task*. This object must implement *Serializable*. If you don't want to use a result, then just return *null;*.

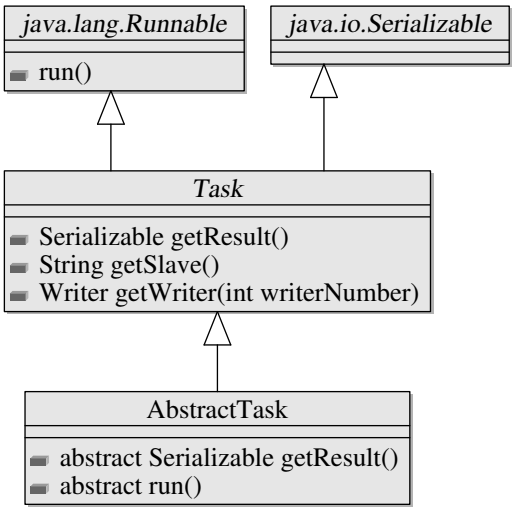


Figure A.5: Task interface and abstract subclass

In your *AbstractTask* extension class, you can get a *Writer* by using the method *Writer getWriter(int writerNumber)*. This gets a *Writer* where your task can write to, and what you write will be redirected to the *Writer* you specified in *AbstractRemoteMaster.getWriter(String slaveId, int writerNumber)*. In your task code, for performance reasons, you should wrap other *Writer*'s around the *RMIWriter* you get. Note that *AbstractTask* knows which slave is executing it, so the appropriate writer for the combination of *slaveId* and *writerNumber* is found.

A.3.4 Overview of other classes

AbstractBlockingMaster

The abstract class *AbstractBlockingMaster* extends the abstract class *AbstractRemoteMaster*, as shown in Figure A.6. *AbstractBlockingMaster* implements *Task getNextTask(String slaveId)*, an abstract method of *AbstractRemoteMaster*. Tasks should be created by a background thread which runs in the master process, this thread can queue tasks to a list with the method *queueTask(Task task)*. This master lets the slaves block when no task is available yet. This master may be of use if new tasks are created based on previous task results, and so the list of tasks is not known at the start. Or maybe the creation of all tasks takes some time, and you don't want to wait to start your slaves.

MasterConsole

The executable class *MasterConsole* can connect, at any time, to the master through a remote call and ask the master the current execution status, which is encapsulated in a *MasterInfo* object. This information includes all the current running slaves and the current running time of their current task.

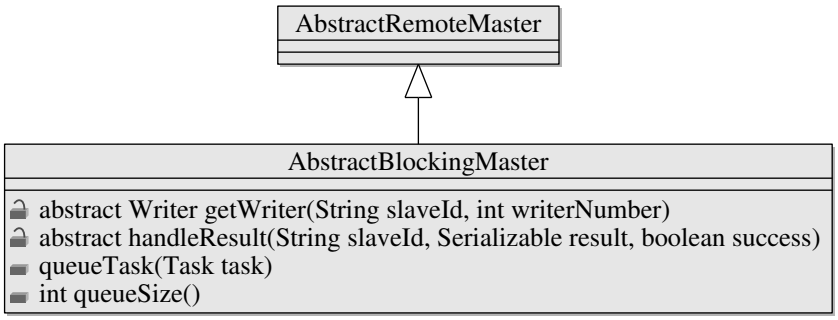


Figure A.6: A blocking master

```
Use:
 java MasterConsole //rmiserver[:port]/mastername [pollInterval]
```

where

1. *//rmiserver[:port]/mastername* identifies the master, refer to Section A.4 for details,
2. *pollInterval* can be specified such that the details are asked and shown every *pollInterval* seconds.

KillSlaveWhenFinished

The executable class *KillSlaveWhenFinished* can connect, at any time, to the master through a remote call and instructs the master to stop giving new tasks to a certain slave. Killing a slave instantly can only be done by killing the slave process (*kill -9* in Linux). Because it is impossible to stop a Java thread without the use of busy waiting, it is impossible to implement the instant killing of a slave by the master efficiently, even if the slave would be remotely accessible.

```
Use:
 java KillSlaveWhenFinished
        //rmiserver[:port]/mastername
        slaveId
```

where

1. *//rmiserver[:port]/mastername* identifies the master,
2. *slaveId* identifies the slave to kill.

MakeSecurityFile, MakeScript

The executable classes *MakeSecurityFile* and *MakeScript* can be used to make a security file and a script to execute your application onto a cluster of machines, respectively. Again refer to Section A.4 for details.

A.4 Example farming application

This section lists the code for a generalization of the well known *8 queens problem*: Place n chess queens on an $n \times n$ board such that no queen can attack another. This problem can be solved with a standard backtracking algorithm, as shown in Algorithm A.1. Note that this algorithm does not perform an isomorphism test.

Algorithm A.1 n queens problem pseudocode

```
function queens (int column)
  for all row do
      if positionIsSafe(row, column) then
          placeQueenAt(row, column)
          if notAllQueensPlaced() then
              queens(column + 1)
          else
              handleSolution()
          removeQueenFrom(row, column)
```

We can easily farm this problem by using this algorithm on the first i columns only ($i < n$). So we split up the search at depth i of the search tree. Any valid configuration with i columns filled can then be used as the start configuration for a task.

A.4.1 Code listing

```
import be.ugent.caagt.farming.AbstractTask;
import java.io.PrintWriter;
import java.io.Serializable;

/**
 * Task which solves the n queens problem recursively
 * by use of a standard backtracking algorithm.
 */
public class QueensTask
        extends AbstractTask {

    /** board dimension */
    private final int dim;

    /** board[column] holds row position of
     *  the queen in column,
     *  counting starts from 0.
     */
    private final int[] board;

    /** number of initially placed queens */
    private int initialQueens;

    /** number of solutions */
    private int solutions = 0;
```

```
/** solution writer */
private PrintWriter out;

/** Creates a QueensTask.
 * @param initial contains initial positions
 *                of 0 or more queens,
 *                initial[i] contains the
 *                initial row position of
 *                the queen in column i
 *                for all i in [0,initial.length[
 * @param dim     dimension of the board
 */
public QueensTask(int[] initial, int dim) {
    this.dim = dim;
    this.board = new int[dim];
    for (int i = 0; i < initial.length; i++)
        board[i] = initial[i];
    this.initialQueens = initial.length;
}

/** Solves the 8 queens problem recursively */
public void run() {
    solutions = 0;
    // check initial placement
    for (int i = 1; i < initialQueens; i++)
        if (!isSafe(board[i], i))
            return;
    // gets a Writer
    out = new PrintWriter(getWriter(0), true);
    if (initialQueens < board.length)
        put(initialQueens);
}

/** Returns the number of solutions as an Integer */
public Serializable getResult() {
    return new Integer(solutions);
}

/** Is it possible to put a queen at (row, column).
 *  Is called when queens have been put
 *  in all columns [0,column[.
 * @return true if the position is safe
 */
private boolean isSafe(int row, int column) {
    for (int j = 0; j < column; j++)
        if (board[j] == row
            || Math.abs(row - board[j]) == column - j)
            return false;
    return true;
}

/** Puts one queen in each of the
 *  columns [column, dim-1] recursively */
private void put(int column) {
```

```
        for (int row = 0 ; row < dim ; row++)
            if (isSafe(row, column)) {
                board [column] = row;
                if (column == dim - 1)
                    shipSolution();
                else
                    put(column + 1);
            }
    }

    /** Increment solutions and print board */
    private void shipSolution() {
        solutions++;
        printBoard();
    }

    /** print board */
    private void printBoard() {
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < dim; i++) {
            for (int j = 0; j < dim; j++)
                buf.append(board [j] == i ? 'Q':'#');
            buf.append("\n");
        }
        buf.append("\n");
        out.print(buf.toString());
    }
}
```

Listing A.1: QueensTask class

```
import be.ugent.caagt.farming.AbstractRemoteMaster;
import be.ugent.caagt.farming.Task;
import java.io.BufferedOutputStream;
import java.io.PrintWriter;
import java.io.Serializable;
import java.io.Writer;
import java.rmi.Naming;
import java.rmi.RemoteException;

/** Master of the 8 queens problem. */
public class QueensMaster
        extends AbstractRemoteMaster {

    /** chess board dimension */
    private final int dim;

    /** initial positions of the queens */
    private int[] initial;

    /** number of queens to place before farming */
    private final int initialQueens;
```

```
    /** number of solutions */
    private int solutions = 0;

    /** solution writer */
    private PrintWriter out;

    /** Creates a QueensMaster where the problem
     *  is split into QueensTask 's
     *  when initialQueens have been placed
     *  in the first initialQueens columns.
     *  All output is send to standard output.
     *
     * @param dim dimension of the chessboard
     * @param initialQueens number of initially placed
     *                      queens before the problem
     *                      is farmed.
     */
    public QueensMaster(int dim, int initialQueens)
    throws RemoteException {
        super(null);  // no master output
        this.dim = dim;
        this.initialQueens = initialQueens;
        initial = new int[initialQueens];

        // send output to STD_OUT
        out = new PrintWriter(
                new BufferedOutputStream(System.out),
                false);

        // set 15 minutes time out
        setTimeToWait(15 * 60 * 1000);
    }

    /** Gets unique Writer which redirects
     *  to standard output */
    protected Writer getWriter(String slaveId,
                               int slaveStreamNo) {
        return out;
    }

    /** Creates and returns the next QueensTask */
    protected synchronized
            Task getNextTask(String slaveId) {
        for (int i = initial.length - 1; i >= 0; i--) {
            if (++initial[i] < dim)
                return new QueensTask(initial, dim);
            else
                initial[i] = 0;
        }
        return null;
    }

    /** Handles the result. The result type is just
     * an Integer which is added to the
     * number of solutions so far.
```

```
     */
    protected void handleResult(String slaveId,
                                Serializable result,
                                boolean success) {
        if (success) {
            int count = ((Integer)result).intValue();
            synchronized (this) {
                solutions += count;
            }
        }
    }


    /** Usage: java QueensMaster port dimension */
    public static void main(String[] args) {
        try {
            if (args.length != 4) {
                System.out.println(
                  "Usage: " +
                  "java QueensMaster " +
                  "masterHost port " +
                  "dimension initialQueens");
                return;
            }
            String masterHost = args[0];
            String port = args[1];
            int dimension = Integer.parseInt(args[2]);
            int initialQueens = Integer.parseInt(args[3]);
            QueensMaster obj
                    = new QueensMaster(dimension, initialQueens);
            // Rebind
            Naming.rebind("rmi://" + args[0] + ":" + args[1]
                          + "/QueensMaster", obj);
        } catch (Exception e) {
            System.err.println(e);
            e.printStackTrace();
        }
    }
}
```

Listing A.2: QueensMaster class

A.4.2 Running the example

Here we will explain what is needed to actually run the example of the previous section.

Creating a security policy file

The command line program *be.ugent.caagt.farming.MakeSecurityFile* can be used to create a security file. It has a single argument which should be either 0, 1 or 2. Running with argument 0 will produce the following file, (only use this for testing purposes):

```
grant {
    permission java.security.AllPermission;
};
```

Argument 1 gives the file:

```
grant {
    permission java.net.SocketPermission
                "*:1024-65535",
                "connect,accept,resolve";
};
```

Creating a script

[4]

The command line program *be.ugent.caagt.farming.MakeScript* can be used to make a script to execute your farming application onto a cluster of machines. This way you do not have to make scripts by hand, which can be a time consuming job. A lot of details such as classpath, server codebase, ... are taken care of. The script is used as follows:

```
Use:
java be.ugent.caagt.farming.MakeScript
       port securityFile masterClass
       masterRMIName classpath masterHost
       megaBytes [slaveHost*]
```

in which

1. *port* is the port number of the rmi registry process,
2. *securityFile* is the security policy file which you can create with *be.ugent.caagt.farming.MakeSecurityFile*,
3. *masterClass* is the fully quantified name (package and class name) of your master class,
4. *masterRMIName* is the name you have chosen for the binding of your master,

[4] UNIX/LINUX expected

5. *classpath* is your CLASSPATH, the base of your class files,
6. *masterHost* is the IP or hostname of the master,
7. *megaBytes* is the memory to allocate for JVM (in MB),
8. finally you give a list of *slaveHost*'s to use, you can reuse the same host.

Example use:

```
java be.ugent.caagt.farming.MakeScript
       4001 slave.policy QueensMaster
       QueensMaster /home/jpwinne/codebase/ beo1
       512 beo5 beo5 beo7
```

meaning

1. Start rmiregistry at port *4001* of host *beo1*
2. Your security file is "slave.policy"
3. Your master class is *QueensMaster*
4. You have done *Naming.rebind("//twiwulf:4001/QueensMaster", queensMaster)*, where *queensMaster* is an instance of *QueensMaster*.
5. Your CLASSPATH is */home/jpwinne/codebase/*
6. The master will run at host *beo1*
7. The JVM can use up to *512* MB for each process
8. You want to use three slaves: two at beo5 (names "slave1_at_beo5", "slave2_at_beo5"), one at beo7 (name "slave3_at_beo7").

This produces the following script.

```
echo "First 2 steps are out commented, execute (1) once!"
echo "***1. STARTING RMI SERVER AT PORT 4001 "
# CLASSPATH= nohup rmiregistry 4001 &

echo "***2. WAITING 5 SECONDS"
# sleep 5

echo "***3. STARTING MASTER APPLICATION"
java -Xmx512m                                          \
 -classpath /home/jpwinne/codebase/                    \
 -Djava.rmi.server.codebase=file:/home/jpwinne/codebase/ \
 -Djava.rmi.server.hostname=beo1                       \
```

```
 QueensMaster 4001 &

echo "***4. WAITING 5 SECONDS"
sleep 5

echo "***5. STARTING SLAVES"
ssh beo5 java -Xmx512m                                    \
 -classpath /home/jpwinne/codebase/                       \
 -Djava.rmi.server.codebase=file:/home/jpwinne/codebase/ \
 -Djava.security.policy=slave.policy                      \
 be.ugent.caagt.farming.Slave                             \
 slave_1_at_beo5 //beo1:4001/QueensMaster &

ssh beo5 java -Xmx512m                                    \
 -classpath /home/jpwinne/codebase/                       \
 -Djava.rmi.server.codebase=file:/home/jpwinne/codebase/ \
 -Djava.security.policy=slave.policy                      \
 be.ugent.caagt.farming.Slave                             \
 slave_2_at_beo5 //beo1:4001/QueensMaster &

ssh beo7 java -Xmx512m                                    \
 -classpath /home/jpwinne/codebase/                       \
 -Djava.rmi.server.codebase=file:/home/jpwinne/codebase/ \
 -Djava.security.policy=slave.policy                      \
 be.ugent.caagt.farming.Slave                             \
 slave_3_at_beo7 //beo1:4001/QueensMaster &
```

This script consists of five steps:

1. Start the RMI registry at a port. This has to be done only once, therefore it is outcommented in the script.
2. Wait 5 seconds so the RMI registry is up and running (outcommented).
3. Start the user written master application at some host, note that this master application is supposed to have one argument: the *port* number.
4. Wait 5 seconds so the master is up and running.
5. Start the slaves at the desired hosts.

B Nederlandstalige samenvatting

In deze Nederlandstalige samenvatting geven we een kort overzicht van deze scriptie. We volgen de structuur van de Engelstalige tekst.

B.1 Inleiding

Definitie B.1.1 (t-design) *Gegeven een eindige verzameling van punten $V = \{P_i\}_{i=1}^{v}$ en een eindige collectie $\mathcal{B} = \{B_j\}_{j=1}^{b}$ van deelverzamelingen, genaamd blokken, die k elementen van V bevatten. Dan is $D = (V, \mathcal{B})$ een design met parameters t-(v, k, λ) als elke deelverzameling die t elementen van V bevat, volledig bevat is in precies λ blokken van $\mathcal{B}$.*

Definitie B.1.2 (BIBD) *Een gebalanceerde incomplete blokdesign (BIBD) is een paar (V,$\mathcal{B}$): V is een verzameling van v punten en $\mathcal{B}$ een collectie van b deelverzamelingen, genaamd blokken, die k elementen van V bevatten. Hierbij behoort elk punt tot precies r blokken, en elk puntenpaar behoort tot precies λ blokken. De getallen v,b,r,k,λ zijn de parameters van de BIBD.*

Definitie B.1.3 (Incidentiematrix) *De incidentiematrix van een design is een $v \times b$ (0,1) matrix waarbij het element van rij i en kolom j 1 is als $P_i \in B_j$ ($i = 1, 2, ..., v$; $j = 1, 2, ..., b$), en anders 0. Een design kan gedefinieerd worden aan de hand van zijn incidentiematrix.*

Definitie B.1.4 (Isomorfe BIBD) *Een isomorfie van twee designs $D_1 = (V_1, \mathcal{B}_1)$ en $D_2 = (V_2, \mathcal{B}_2)$ is een bijectie tussen hun puntenverzamelingen V_1 en V_2, en hun blokkencollecties $\mathcal{B}_1$ en $\mathcal{B}_2$, waarbij de punt-blok incidenties behouden blijven.*

Een permutatie wordt vaak in cykelnotatie geschreven. Voorbeeld: (1) (2 3) (4 5 6) (7) verwisselt 2 en 3; 4 wordt op 5 afgebeeld, 5 op 6 en 6 op 4. Merk op dat 1 en 7 ongewijzigd blijven, daarom wordt de permutatie vaak verkort weergegeven als (2 3) (4 5 6).

Definitie B.1.5 (Automorfisme van een design) *Een automorfisme van een design is een isomorfie van de design met zichzelf, dus een puntpermutatie die de blokkencollectie onveranderd laat. De verzameling van alle automorfismen van een design vormt een groep: de volledige automorfismegroep. Elke deelgroep van deze groep is een automorfismegroep van de design.*

Voor verdere informatie over designs verwijzen we naar [3], [10], [13], [45].

B.2 Software voor backtrackalgoritmen

In dit hoofdstuk beschrijven we een algemeen framework, geschreven in Java, dat we in dit werk ontwikkelden. Dit framework is geschikt is voor een exhaustieve generatie van rechthoekige matrices die uitsluitend kleine niet-negatieve gehele getallen bevatten. Deze matrices worden vaak gebruikt om combinatorische objecten voor te stellen. Meest voorkomend zijn incidentiematrices en adjacentiematrices, die bijvoorbeeld designs of grafen voorstellen. Dergelijke matrix voldoet aan bepaalde voorwaarden die triviaal volgen uit de definitie van het combinatorisch object. Het nagaan van het bestaan van dergelijke matrix is echter meestal niet triviaal. Het genereren van alle matrices die voldoen aan bepaalde voorwaarden, gebeurt aan de hand van een exhaustief backtrackalgoritme, dat alle oplossingen systematisch genereert. Dergelijk algoritme werd generisch geïmplementeerd door een abstractie te maken van alle componenten waaruit het algoritme bestaat. We geven eerst een informele beschrijving van deze componenten in Sectie B.2.1. In Sectie B.2.2 bespreken we hoe deze voorgesteld worden in onze implementatie: het *backtrack* pakket. Tevens werd een grafische gebruikersinterface ontwikkeld die toelaat om een backtrackalgoritme, dat geïmplementeerd werd volgens het ontwerp van het *backtrack* pakket, te visualizeren zonder extra programmeerinspanning. Dit wordt besproken in Sectie B.2.3.

B.2.1 Exhaustief backtrackalgoritme

Aan elke (rij, kolom) *positie* van de matrix wordt een *domein* geassocieerd. Het *domein* van een positie is een geordende lijst van waarden (getallen) die nog mogelijk zijn voor die positie. Met *nog mogelijk zijn* bedoelen we dat we die waarde nog niet kunnen uitsluiten. Bij incidentiematrices is het domein van elke positie initieel $(0, 1)$. Het binden van een positie aan een zekere domeinwaarde wordt *instantiëren* genoemd. Een instantiatie is *geslaagd* indien na het instantiëren nog aan alle voorwaarden voldaan is. Het *generatiepad* bepaalt de volgorde waarin posities geïnstantieerd worden: het definieert de volgende te instantiëren positie na een geslaagde instantiatie. Een eenvoudig generatiepad ligt vooraf vast. Een voorbeeld hiervan is een pad dat alle posities, rij per rij, in de volgorde van de kolommen, instantieert. Dit vaak voorkomend pad noemen we een *rijpad*. Meer formeel: Gegeven een $n \times m$ matrix met rijen genummerd van 1 tot n en kolommen van 1 tot m. Wanneer (i, j) de laatst geslaagde geïnstantieerde positie is, dan is de volgende positie van het rijpad $(i, j+1)$ indien $j+1 \leq m$, anders $(i+1, 1)$ indien $i+1 \leq n$, en anders is de matrix volledig geïnstantieerd.

Het backtrackalgoritme start vanaf een *lege* matrix, d.i. een matrix waarbij geen enkele positie geïnstantieerd is. De posities worden systematisch geïnstantieerd in de volgorde bepaald door het generatiepad. Wanneer een instantiatie niet slaagt of wanneer alle domeinwaarden geprobeerd zijn voor een bepaalde positie, wordt de laatste instantiatie ongedaan gemaakt en wordt er teruggekeerd naar de laatst geïnstantieerde positie en de volgende domeinwaarde geprobeerd voor die positie. Algoritme B.1 beschrijft dit algoritme in pseudocode.

Algoritme B.1 Recursief backtrackalgoritme

```
functie genereer()
1  als matrixVolledigGeïnstantieerd() dan
2      verwerkOplossing()
3  anders
4      positie ← generatiePad.geefVolgendePositie()
5      domeinLijst ← geefDomeinLijstVanPositie(positie)
6      voor alle waarde in domeinLijst doe
7          matrix[positie] ← waarde
8          als alleVoorwaardenVoldaan() dan
9              genereer()
10         matrix[positie] ← ONGEDEFINIEERD
```

B.2.2 Het backtrack pakket

De kern van het *be.ugent.caagt.backtrack* pakket bestaat uit een *generator* die het standaard backtrackalgoritme al dan niet recursief implementeert. Hierbij gebruikt de generator diverse *generatorcomponenten*. Elke *generatorcomponent* implementeert de *GeneratorComponent* interface. Deze interface bevat twee methoden. Enerzijds een methode om deze generatorcomponent te initialiseren aan de hand van data die gedeeld wordt met andere generatorcomponenten. Anderzijds een methode om deze generatorcomponent te herinitialiseren, d.w.z. terug in zijn initiële toestand brengen, zodat hergebruik mogelijk is. *GeneratorComponent* heeft vijf belangrijke subinterfaces *Checker*, *Domain*, *Initializer*, *LeafNode* en *Path*.

Elke *Checker* controleert, na het instantiëren van een zekere positie, of er nog steeds aan een bepaalde voorwaarde voldaan is. Wanneer dit het geval is, zal de *Checker* zijn toestand aanpassen aan deze instantiatie. Veronderstel bijvoorbeeld dat elke rij exact r keer 1 moet bevatten. Intern houdt de *Checker*, voor elke rij, bij hoeveel keer 1 er nog mag gebruikt worden voor de resterende niet geïnstantieerde posities van de rij. Na het geslaagde instantiëren van positie (r, k) met 1, verlaagt de checker het aantal keer dat 1 nog mag gebruikt worden in rij r. Wanneer alle eentjes opgebruikt zijn voor rij r, faalt de instantiatie van positie (r, k) met 1, en wordt er dus ook niets veranderd aan de interne toestand van de *Checker*. Alle *Checker*'s samen controleren dus alle voorwaarden waaraan de matrix moet voldoen.

Domain definieert voor elke positie een geordende lijst van domeinwaarden. Het voorziet ook de mogelijkheid om deze lijst herhaaldelijk te overlopen. Standaard implementaties voorzien een (vast) domein van (aaneengesloten) waarden voor elke positie en een manier om bepaalde waarden uit het domein te verwijderen.

De *initialize()* methode van elke *Initializer* wordt opgeroepen net vóór het starten van het generatieproces. Deze methode kan bijvoorbeeld gebruikt worden om sommige matrixposities reeds te instantiëren. Een andere toepassing is het initialiseren van data die verschillende generatorcomponenten delen.

De *ship()* methode van elke *LeafNode* wordt opgeroepen voor elke oplossing, d.w.z. telkens wanneer de matrix volledig geïnstantieerd is. Het uitschrijven of opslaan van deze

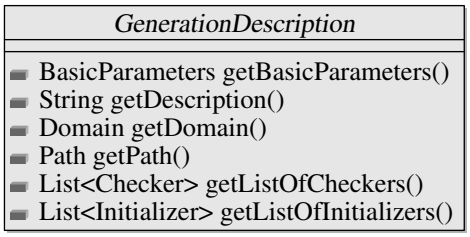
oplossingen in een bepaald formaat zijn evidente voorbeelden.

Het generatiepad wordt gedefinieerd door een implementatie van de *Path* interface. De generator roept diens *boolean prepare(int depth)* methode op na een geslaagde instantiatie. Hierbij is de *depth* parameter het huidig aantal geïnstantieerde posities. De *prepare* methode bepaalt de volgende te instantiëren positie. Deze positie wordt opgevraagd aan de hand van de methoden *int getRow()* en *int getColumn()* van de *Path* interface.

De voorstelling van de matrix moet een implementatie van de interface *IntMatrix* zijn. Deze interface bevat voor de hand liggende methoden zoals het instellen en opvragen van de dimensies en de waarden. Om efficiëntieredenen kan je er echter voor kiezen om je generatorcomponenten rechtstreeks met een 2−dimensionale array te laten werken. Hiervoor kan je de standaard implementatieklasse *ValueMatrix* gebruiken. De *ValueMatrix* klasse bevat een methode die de array teruggeeft, dus deze klasse is niet veel meer dan een wrapper rond een 2−dimensionale array.

De *SharedData* klasse kan gebruikt worden om data te delen tussen verschillende generatorcomponenten. De *SharedData* klasse is niet veel meer dan een verzameling (sleutel, waarde) paren, waarbij de sleutel een tekenreeks is (Java type *String*), en de waarde het te delen object. Een aantal (sleutel, waarde) paren zijn voorgedefinieerd, zoals later wordt besproken.

Een volledige beschrijving van een bepaalde generatie wordt gebundeld in een implementatie van de *GenerationDescription* interface, zoals hieronder weergegeven.



Merk op dat een *GenerationDescription* implementatie geen *LeafNode* objecten bijhoudt. De wijze waarop oplossingen behandeld worden, heeft niets te maken met de generatie op zich. *LeafNode* objecten zullen dan ook gegeven worden aan de generator. Wel bevat *GenerationDescription* een extra object *BasicParameters*. De *BasicParameters* klasse bevat de dimensie van de matrix, de minimaal en maximaal mogelijke domeinwaarden en het al dan niet symmetrisch zijn van de te genereren matrix.

Voor het visualiseren van een backtrackalgoritme, wat het onderwerp is van de volgende sectie, hebben we behoefte aan een niet-recursieve implementatie van een generator. Hiertoe werd een interface *NonRecursiveGenerator* voorzien, tesamen met een standaard implementatieklasse *GenericGenerator*. Deze *GenericGenerator* klasse is ook in staat om zogenaamde metadata bij te houden. Deze metadata wordt gebundeld in een *MetaData* klasse. De volgende metadata *kan* verzameld worden:

- Het aantal recursieve oproepen.
- Het aantal keer dat de generator een bepaald aantal posities geïnstantieerd heeft. Dit stelt de zogenaamde zoekboom voor.

- Voor elke positie: het aantal instantiaties en het aantal geslaagde instantiaties.
- Voor elke positie en elke *checker*: het aantal controles en het aantal geslaagde controles.
- Voor elke positie en elke waarde: het aantal instantiaties met die waarde en het aantal geslaagde instantiaties daarvan.
- Voor elke positie, *checker* en waarde: het aantal controles met die waarde op die positie en het aantal geslaagde controles daarvan.

Omdat metadata verzamelen duur is in termen van geheugen en uitvoeringstijd, definieert een *MetaDataConfig* object welke metadata er al dan niet moet bijgehouden worden.

Het *SharedData* object bevat een aantal voorgedefinieerde sleutels die gebruikt worden voor volgende doeleinden door de generische implementatie.

- *be.ugent.caagt.im.IntMatrix* : Bij deze sleutel hoort een instantie van een implementatieklasse van de *IntMatrix* interface. Deze instantie wordt gebruikt als de te genereren matrix. Zorg zelf voor een initializer die deze matrix in *SharedData* stopt.
- *be.ugent.caagt.backtrack.Domain* : Een instantie van een implementatieklasse van de *Domain* interface, gespecifieerd door *GenerationDescription*. De *GenericGenerator* klasse stopt deze klasse in het *SharedData* object.
- *be.ugent.caagt.backtrack.DomainMatrix* : Een instantie van de *DomainMatrix* klasse (wordt optioneel gebruikt door *Domain*, zie documentatie).
- *be.ugent.caagt.backtrack.GeneratorStack* : Een instantie van de *GeneratorStack* klasse (niet besproken, wordt intern gebruikt door *GenericGenerator*, die deze klasse in het *SharedData* object stopt).
- *be.ugent.caagt.backtrack.MetaDataConfig* : Een instantie van de *MetaDataConfig* klasse. De *GenericGenerator* klasse stopt dit object in het *SharedData* object. Dit object kan je instellen aan de hand van de methode *setMetaDataConfig()* van *GenericGenerator*.
- *be.ugent.caagt.backtrack.MetaData* : Een instantie van de *MetaData* klasse. De *GenericGenerator* klasse maakt dit object aan op basis van *MetaDataConfig*, en stopt dit object in het *SharedData* object.
- *be.ugent.caagt.backtrack.Path* : Een instantie van een implementatieklasse van de *Path* interface, gespecifieerd door *GenerationDescription*. De *GenericGenerator* klasse stopt deze klasse in het *SharedData* object.

B.2.3 Visualisatie van een algoritme

Het *be.ugent.caagt.gui.gentool* pakket laat toe om algoritmen, geschreven volgens de principes van het *be.ugent.caagt.backtrack* pakket, stapsgewijs te visualiseren. Dergelijke tool is handig om programmafouten op te sporen, nieuwe voorwaarden te ontdekken, logische fouten op te sporen, het algoritme te optimaliseren, verschillende implementaties te vergelijken, ...

De visualizer starten is eenvoudig. U maakt een *GenericGenerator* object aan de hand van een *GenerationDescription* implementatieklasse. Dit *GenericGenerator* object geef je dan mee bij de constructie van de *Gentool* klasse van dit *be.ugent.caagt.gui.gentool* pakket.

```
GenerationDescription description = new MyDescription();
GenericGenerator generator = new GenericGenerator(description);
Gentool gentool = new Gentool(generator);
```

De grafische gebruikersinterface start op in de toestand net voor het starten van het generatieproces. Het sturen van de generatie gebeurt aan de hand van volgende gebruikersacties.

- *stap*: Het uitvoeren van een enkele generatiestap. Een enkele stap doet het volgende: Als er nog domeinwaarden zijn voor de huidige positie (voorwaartse stap):
  - Dan wordt de volgende domeinwaarde geprobeerd. Indien aan alle constraints voldaan is, wordt de huidige positie ingesteld op de volgende positie, bepaald door het generatiepad (tenzij er een oplossing werd gevonden).

  Als er geen domeinwaarden meer waren voor de huidige positie (achterwaartse stap):
  - Dan wordt de huidige positie ongeïnstantieerd en wordt teruggekeerd naar de laatst geïnstantieerde positie. Als er geen dergelijke positie meer is, dan is het generatieproces voltooid.
- *herhaal*: Het herhaaldelijk uitvoeren van stappen waarbij er een zekere tijd gepauzeerd wordt tussen elke *stap*. Deze actie kan onderbroken worden, zoals verder wordt besproken.
- *stop*: Onderbreekt de *herhaal* actie.
- *omhoog*: Verlaat de huidige tak, d.w.z. maak de instantiatie van de huidige positie ongedaan en keer terug naar de laatst geïnstantieerde positie.
- *reset*: Herstart het generatieproces.

Behalve door *stop*, kan de *herhaal* actie ook onderbroken worden als:

1. er een oplossing gevonden wordt,
2. er een bepaald aantal stappen zijn uitgevoerd,
3. er een bepaald aantal recursieve oproepen gebeurd zijn,
4. er een bepaalde checker snoeit,
5. er een bepaalde positie (on)geïnstantieerd wordt,
6. er een bepaald aantal posities geïnstantieerd zijn,

7. er aan een, door de gebruiker geschreven, onderbrekingsconditie voldaan is. Dit kan door de interface *InterruptCondition* te implementeren. Diens methode *boolean shouldInterrupt(int row, int column)* wordt opgeroepen bij elke stap. Wanneer **true** wordt teruggegeven, wordt de *herhaal* actie onderbroken.

De applicatie is opgebouwd uit verbergbare interne vensters. In het **Kleuren** venster (her)definieer je de kleuren. Het **Acties** venster herbergt enerzijds alle beschreven interactieve acties en anderzijds ook punten 1, 2 en 3 van bovenstaande lijst. Het **Matrix** venster toont de huidige toestand van de matrix. Breekpunten (punten 4 en 5 uit bovenstaande lijst) kunnen ingesteld worden door op een matrixpositie te klikken, en worden aangeduid met een blauw kruis. Het **Reden** venster toont de reden waarom de laatste checker snoeide, d.i. de *toString()* representatie van de snoeiende checker. Het **Zoekboom** venster toont de zoekboom in een textueel formaat, d.w.z. voor elke diepte wordt het aantal bindingen op die diepte weergegeven, tesamen met de verbredingsfactor. Het **Visuele zoekboom** venster tekent de zoekboom grafisch. Het **Leaf** venster toont het aantal oplossingen en het aantal recursieve oproepen. Het **Domein** venster visualiseert de *DomainMatrix* klasse indien deze gebruikt wordt. Het **Bindingen** venster visualiseert het aantal bindingen per positie aan de hand van grijswaarden. De meest bezochte posities zijn het donkerst. In het **Checkers** venster kan je enerzijds specifiëren dat het proces onderbroken moet worden wanneer een bepaalde checker snoeit, en anderzijds kan je ieders individuele snoeistatistieken visualiseren. Voor elke positie wordt de snoeiverhouding weergegeven: Het rode deel duidt de verhouding aan tussen het aantal keer dat er gesnoeid werd en het totaal aantal controles. Tenslotte toont het **Vaste oplossing** venster de waarden van de posities die hetzelfde zijn in elke gevonden oplossing.

B.3 Equivalentie van matrices

Het equivalentietesten van matrices van gehele getallen zullen we nodig hebben in de volgende vier hoofdstukken. We vertalen dit probleem naar een graafequivalentieprobleem, hetgeen opgelost wordt door het softwarepakket *nauty* [36] te gebruiken.

Om na te gaan of twee designs isomorf zijn, vormen we elke design om tot een bipartiete graaf. We definiëren een top voor elk punt en voor elk blok, en een boog tussen een "punt"-top en "blok"-top voor elke punt-blok incidentie. Alle "punt"-toppen krijgen kleur 0 en alle "blok"-toppen krijgen kleur 1. *Nauty* kan, naast het bepalen van de volledige automorfismegroep, ook een canonische vorm produceren van een graaf. *Nauty* garandeert dat grafen G en H isomorf zijn als en slechts als hun canonische vormen gelijk zijn. Vooral voor het bepalen van het aantal niet-isomorfe grafen, gegeven een set van grafen, zijn deze canonische vormen bijzonder bruikbaar.

Definitie B.3.1 (Equivalentie van matrices van gehele getallen) *Twee (gekleurde) matrices M_1 and M_2 zijn equivalent als M_2 gelijk is aan M_1 na het toepassen van een rij -en kolompermutatie (die de kleuring respecteert, d.w.z. enkel rijen/kolommen permuteert met rijen/kolommen van hetzelfde kleur).*

Matrices waarin alle gehele getallen tot het interval $[0, 2^\ell[$ behoren, kunnen vertaald worden

naar een incidentiematrix equivalentieprobleem door elementen voor te stellen als bitvectoren van lengte ℓ.

Het *be.ugent.caagt.nauty* pakket kan gebruikt worden om B. D. McKay's graaf- isomorfietest software *nauty* te gebruiken [1] vanuit Java d.m.v. Java Native Interface. Java Native Interface (JNI) laat toe om Java code te laten samenwerken met applicaties die geschreven zijn in andere talen, zoals C en assembleertaal. Deze implementatie laat toe om *nauty* op te roepen op dezelfde manier zoals dit mogelijk is vanuit C. Er werd echter ook een klasse ontwikkeld die het gebruik van *nauty* gemakkelijker maakt.

De *NautyStats* klasse weerspiegelt nauty's *statsblk* C struct, een structuur die de resultaten van de laatste oproep van de *nauty()* methode bijhoudt. De *NautyInvariant* klasse bevat constanten die invarianten voorstellen. Het gebruik van invarianten kan de uitvoeringstijd aanzienlijk beïnvloeden. De *NautyLib* klasse kan je instantiëren als je *nauty* wil gebruiken zoals je dit zou doen vanuit C. Je kan herhaaldelijk de opties instellen, nauty oproepen en de resultaten opvragen. De *Nauty* klasse is een wrapperklasse rond *NautyLib*, die toelaat om nauty te gebruiken voor incidentiestructuren, symmetrische matrices van gehele getallen en rechthoekige matrices van gehele getallen. Deze klasse voorziet de nodige conversie tussen de corresponderende graaf en deze structuren.

B.4 Het genereren van designs met niet-triviale automorfismen

De classificatie van alle combinatorische objecten is vaak te moeilijk voor de grotere parameters. Toch kan het mogelijk zijn om partiële classificaties te maken van dergelijke objecten door enkele deze te genereren die bepaalde automorfismen bevatten. In dit hoofdstuk wordt de welgekende *local approach* methode [25] [33] [44] beschreven in het kader van de classificatie van alle 2-(v, k, λ) designs die een automorfisme van priemorde bevatten.

We implementeerden een programma dat bruikbaar is om alle 2-(v, k, λ) designs met een automorfisme van kleine priemorde (2, 3, 5 en 7) te genereren. Voor grotere ordes werkt het programma wel, maar is het niet efficiënt. In het bijzonder worden de voorwaarden voor de mogelijke rijen/kolommen en het scalair product van twee rijen/kolommen efficiënt gecontroleerd door alle mogelijke rij -en kolom(intersectie)patronen vooraf te bepalen.

Dit hoofdstuk vormt de basis van de drie volgende hoofdstukken. Hoofdstuk B.5 behandelt de classificatie van alle 2-(31,15,7) en 2-(35,17,8) Hadamard designs en 2-(36,15,6) Menon designs met automorfismen van oneven priemorde. Hoofdstuk B.6 past de generatiemethode toe op een andere combinatorische structuur: een partiële meetkunde. Tenslotte wordt de enumeratie van de *dubbels* van het projectieve vlak van orde 4, waarin deze methode ook gebruikt wordt voor deelresultaten, beschreven in Hoofdstuk B.7.

Stel dat A de incidentiematrix van een 2-(v,k,λ) design is. Veronderstel dat A een automorfisme φ bevat dat als volgt inwerkt op A's rijen (punten)

$$(1)(2)\cdots(f)(f+1\ \cdots\ f+p)(f+p+1\ \cdots\ f+2p)\cdots(v-p+1\ \cdots\ v)$$

en als volgt op A's kolommen (blokken)

$$(1)(2)\cdots(f')(f'+1\ \cdots\ f'+p)(f'+p+1\ \cdots\ f'+2p)\cdots(b-p+1\ \cdots\ b)$$

[1] Zie http://cs.anu.edu.au/∼bdm/nauty/ voor de *nauty* webpagina.

De eerste f punten en de eerste f' blokken zijn *gefixeerd*. De laatste $h = v - f$ punten en de laatste $g = b - f'$ blokken zijn *ongefixeerd*. We genereren dus alle niet-isomorfe incidentiematrices A die het automorfisme φ bevatten. In de *local approach* methode kan men de volgende 4 deelmatrices onderscheiden in A.

$$A = \begin{pmatrix} F & G \\ H & X \end{pmatrix}$$

Het *gefixeerd deel* wordt gevormd door de $f \times f'$ matrix F, de $f \times g$ matrix G en de $h \times f'$ matrix H. Stel $n = \frac{h}{p}$ en $n' = \frac{g}{p}$. De $h \times g$ matrix X vormt het *ongefixeerd deel*. X bevat nn' circulanten van orde p, dus het is van de vorm $X = (C_{i,j})$ waarin $C_{i,j}$ een circulant van orde p is, $1 \leq i \leq n$, $1 \leq j \leq n'$.

Definitie B.4.1 (Startconfiguratie) *De startconfiguratie A_s is de matrix A waarin de gefixeerde delen bepaald zijn en X ongedefinieerd is. Daarom beschouwen we X als de $h \times g$ nulmatrix:*

$$A_s = \begin{pmatrix} F & G \\ H & 0 \end{pmatrix}$$

Definitie B.4.2 (Orbitmatrix) *De orbit matrix $\hat{X}$ is een $n \times n'$ matrix, waarin de waarde $\hat{x}_{i,j}$ het aantal enen in een rij van de circulant $C_{i,j}$ is, $1 \leq i \leq n$, $1 \leq j \leq n'$.*

Het generatieproces bestaat uit verschillende fases. Elke fase bestaat uit een exhaustieve generatie van matrices die voldoen aan bepaalde voorwaarden. Eerst genereren we alle inequivalente *startconfiguraties*, hetgeen meestal het eenvoudigste deel van de zoektocht vormt. Voor elke startconfiguratie genereren we alle inequivalente orbitmatrices. Een orbitmatrix definieert dus enkel het aantal enen in een rij van elke circulant. Via dubbeltellingen wordt afgeleid aan welke voorwaarden deze matrices moeten voldoen. De laatste fase breidt elke inequivalente orbitmatrix uit, tesamen met zijn startconfiguratie, tot een volledige incidentiematrix door elke waarde $\hat{x}_{i,j}$ van de orbit matrix $\hat{X}$ te vervangen door elke mogelijke circulant met $\hat{x}_{i,j}$ enen per rij. Uiteindelijk moeten uit de verzameling van bekomen designs eventuele isomorfe exemplaren verwijderd worden.

B.5 Hadamard en Menon designs, en verwante Hadamard matrices en codes

Een deel van de resultaten van dit hoofdstuk werden voorgesteld op de *European Conference on Combinatorics 2005* conferentie, gehouden in Berlijn [6]. De resultaten van dit hoofdstuk werden ingezonden naar *Journal of Combinatorial Designs* [7]. Een preprint kan gevonden worden op *http://caagt.ugent.be/preprints*.

Het hoofdresultaat is de classificatie van Hadamard matrices van orde 32 en 36 die voortkomen uit alle Hadamard en Menon designs met een automorfisme van oneven priemorde. We maakten ook een partiële classificatie van Hadamard matrices van orde 32 en 36 die voortkomen uit een partiële classificatie van Hadamard en Menon designs met automorfismen van orde 2. We vonden 21879 Hadamard matrices van orde 32 en 24920 Hadamard matrices van orde 36. Alle geconstrueerde Hadamard matrices van orde 36 zijn Hadamard equivalent met een reguliere Hadamard matrix.

B.5.1 Inleiding

Een *Hadamard matrix* H van orde n is een $n \times n$ ± 1 matrix waarvoor $HH^t = nI$. Twee Hadamard matrices H_1 en H_2 zijn *Hadamard equivalent* als H_2 kan bekomen worden uit H_1 door het toepassen van een aantal rijpermutaties, kolompermutaties, rijnegaties en kolomnegaties. Een automorfisme van een Hadamard matrix is een equivalentie met zichzelf. Een *genormaliseerde* rij/kolom bestaat uitsluitend uit enen. Een *genormaliseerde* Hadamard matrix is een Hadamard matrix waarvan één rij en kolom genormaliseerd zijn. Een *reguliere* Hadamard matrix heeft een constante rijsom en kolomsom.

Hadamard matrices zijn volledig geclassificeerd tot orde 28. Voor hogere ordes zijn er slechts partiële classificaties gekend. Lin, Wallis en Zhu [30] vonden 66104 inequivalente Hadamard matrices van orde 32. Uitgebreide resultaten over orde 32 verschenen in [31] en [32]. Bij de aanvang van dit onderzoek waren er minstens 235 inequivalente Hadamard matrices van orde 36 gekend [20] [22] [41]. Astronomische grenzen werden bekomen voor het aantal Hadamard matrices van orde 32 en 36 in [38], waarin de auteur ook onze matrices gebruikte om deze grenzen te bekomen. Een volledige classificatie van alle Hadamard matrices van orde 32 en 36 is utopisch. Een motivatie voor dit onderzoek is het bepalen van het aantal Hadamard matrices van orde 32 en 36 met symmetrie.

Hadamard matrices zijn verwant met zelfduale codes [47] [43]. Het bestaan van een extreme zelfduale $[72, 36, 16]$ code vormt een belangrijk open probleem in codeertheorie [42]. Zoals aangetoond in [15] kan een code met dergelijke parameters bekomen worden vanuit Hadamard matrices van orde 36 met een triviaal automorfisme of met een automorfisme van orde 2, 3, 5 of 7. Dit is een andere motivatie voor dit onderzoek.

Om de incidentiematrix van een symmetrische 2-$(4m-1, 2m-1, m-1)$ *Hadamard design* te bekomen, verwijderen we de genormaliseerde rij en kolom van een genormaliseerde Hadamard matrix van orde $4m$ en vervangen we -1 door 0. Niet-isomorfe Hadamard designs kunnen bekomen worden vanuit één Hadamard matrix, afhankelijk van de keuze van de genormaliseerde rij/kolom. Maar slechts één Hadamard matrix kan bekomen worden vanuit een Hadamard design.

Een *Menon design* [14] is een 2-$(4u^2, 2u^2 \pm u, u^2 \pm u)$ design. Een Menon 2-(36,15,6) design $(u = 3)$ bestaat als en slechts als er een reguliere Hadamard matrix van orde 36 bestaat. Ze worden op eenvoudige wijze uit elkaar bekomen door 0 en -1 te verwisselen. Bovendien kunnen we ook 2-(35,17,8) designs afleiden uit de 2-(36,15,6) designs. Dit gebruiken we om onze resultaten te controleren.

B.5.2 Classificatie van 2-$(31, 15, 7)$, 2-$(35, 17, 8)$ en 2-$(36, 15, 6)$ designs

Eerst bepalen we alle mogelijke priemordes p en alle mogelijke waarden voor f voor 2-$(31, 15, 7)$, 2-$(35, 17, 8)$ en 2-$(36, 15, 6)$ designs. Indien een 2-(v,k,λ) design een automorfisme van priemorde p bezit, dan geldt $p \leq k$ of $p \mid v$ (anders zouden alle punten/blokken gefixeerd zijn). Zo ook geldt $(v - f) \mod p = 0$ en $f \leq v/2$ [11]. Tonchev (Lemma 1.8.1 [46]) bewees dat een automorfisme van orde 3 van een 2-(v,k,λ) design hoogstens $b - 3(r - \lambda)$ blokken fixeert. Hieruit volgt dat voor het geval waarin $p = 3$: $f \leq 7$ voor 2-$(31, 15, 7)$, $f \leq 8$ voor 2-$(35, 17, 8)$ en $f \leq 9$ voor 2-$(36, 15, 6)$.

De mogelijke priemdelers p voor 2-$(31, 15, 7)$ en 2-$(36, 15, 6)$ zijn 2, 3, 5, 7, 11 en 13. De mogelijke priemdelers p voor 2-$(35, 17, 8)$ zijn 2, 3, 5, 7, 11, 13 en 17. Het bestaan van

de meeste gevallen werd verworpen door een eenvoudig telargument. Er bestaan bijvoorbeeld geen dergelijke designs voor $p = 11, 13$. Voor de andere gevallen gebruikten we ons programma uit Hoofdstuk B.4. Voor 2-$(31, 15, 7)$, 2-$(35, 17, 8)$ en 2-$(36, 15, 6)$ genereerden we alle designs met automorfismen van orde 3, 5 en 7. Voor $p = 2$ genereerden we enkel alle 2-$(31, 15, 7)$ designs met 1 of 3 gefixeerde punten, alle 2-$(35, 17, 8)$ designs met 1, 3 of 5 gefixeerde punten en alle 2-$(36, 15, 6)$ designs met 2 of 4 gefixeerde punten. Voor grotere waarden van f slaagden we er niet om alle designs met een automorfisme van orde 2 met f gefixeerde punten te genereren. Deze isomorfieklassen blijken te groot om te genereren. Voor orde 17 voor 2-$(35, 17, 8)$ werden de 11 designs en corresponderende 11 Hadamard matrices van orde 36 (beschikbaar op [40]) reeds geconstrueerd [44]. Als extra controle op deze resultaten, maakte I. Bouyukliev een onafhankelijke implementatie van delen van deze classificatie.

B.5.3 Resultaten voor Hadamard matrices

Elk automorfisme van een Hadamard design leidt tot een automorfisme van de gerelateerde Hadamard matrix dat de toegevoegde genormaliseerde rij/kolom fixeert. Elk automorfisme van een Menon design is vanzelfsprekend een automorfisme van de gerelateerde Hadamard matrix. Als er geen 2-$(35, 17, 8)$ designs met een automorfisme van orde p met f gefixeerde punten/blokken bestaan, dan bestaan er ook geen 2-$(36, 15, 6)$ designs met een automorfisme van orde p met $f + 1$ gefixeerde punten/blokken. Dit kan ingezien worden als volgt. Beschouw een 2-$(36, 15, 6)$ met een automorfisme van orde p en $f + 1$ gefixeerde punten/blokken. Converteer de gerelateerde reguliere Hadamard matrix naar een 2-$(35, 17, 8)$ design door één gefixeerd punt en blok te normaliseren en te verwijderen. De bekomen 2-$(35, 17, 8)$ design heeft een automorfisme van orde p met f gefixeerde punten/blokken. De verzameling van reguliere Hadamard matrices bekomen vanuit de classificatie van 2-$(36, 15, 6)$ Menon designs met een automorfisme van orde p met $f + 1$ gefixeerde punten/blokken, moet een deelverzameling zijn van de verzameling die bekomen wordt uit 2-$(35, 17, 8)$ met een automorfisme van orde p met f gefixeerde punten/blokken.

Stelling B.5.1 (zie 1.5.1 [46]) *Stel H een Hadamard matrix van orde $n \geq 4$ en $p > 2$ een priemdeler van de orde van de volledige automorfismegroep van H. Minstens één van volgende gevallen geldt: (a) p deelt n; (b) p deelt $n - 1$; (c) $p \leq \frac{n}{2} - 1$. Als p n niet deelt, dan is p bovendien de orde van een automorfisme van de gerelateerde Hadamard* 2-$(n - 1, n/2 - 1, n/4 - 1)$ *design.*

Op basis hiervan concluderen we dat we alle Hadamard matrices van orde 32 en 36 kunnen construeren die voortkomen uit alle Hadamard designs met een automorfisme van oneven priemorde, behalve Hadamard matrices van orde 36 met automorfismen van orde 3 zonder gefixeerde punten. Maar we kunnen alle reguliere Hadamard matrices van orde 36 met een automorfisme van orde 3 zonder gefixeerde punten wel bekomen vanuit de Menon designs.

Een opmerkelijk resultaat is dat de verzameling van inequivalente Hadamard matrices die voortkomen uit 2-$(35, 17, 8)$ met een automorfisme van orde p met f gefixeerde punten, gelijk is aan de verzameling van inequivalente Hadamard matrices die voortkomen uit 2-$(36, 15, 6)$ met een automorfisme van orde p met $f + 1$ gefixeerde punten. Alle verkregen

Hadamard matrices zijn dus Hadamard equivalent met een reguliere Hadamard matrix. Met een gretig algoritme vonden we dat dit ook geldt voor de 11 Hadamard matrices met een automorfisme van orde 17. Misschien zijn alle Hadamard matrices van orde 36 Hadamard equivalent met een reguliere Hadamard matrix?

W. P. Orrick gebruikte [2] onze resultaten voor zijn artikel "Switching operations for Hadamard matrices" [38], waarin een methode ontwikkeld werd die vanuit één Hadamard matrix vele anderen construeert. Miljoenen Hadamard matrices van orde 32 en 36 werden op deze wijze geconstrueerd.

B.5.4 Resultaten voor codes

De volledige automorfismegroep van de symmetrische designs 2-$(35, 18, 9)$ (complement van 2-$(35, 17, 8)$) is een deelgroep van de volledige automorfismegroep van de verwante code. We classificeerden 2-$(35, 17, 8)$ designs met automorfismen van orde 7 met 10 cykels en orde 5 met 14 cykels. We controleerden alle verkregen dubbel-even codes, helaas was de minimumafstand hoogstens 12. Vanuit de designs bekomen we 786 dubbel-even $[72, 36, 12]$ codes, welke de best gekende zelfduale codes zijn van deze lengte. Ze hebben 26 verschillende automorfismegroep ordes en 79 verschillende gewicht enumerators. Menon 2-$(36, 15, 6)$ designs met automorfismen van orde 7 met 10 cykels, van orde 5 met 14 cykels, van orde 3 met 24 cykels en van orde 2 met 36 cykels kunnen ook dergelijke codes opleveren. Helaas hebben ook deze dubbel-even codes hoogstens 12 als minimumafstand.

B.6 Een computerzoektocht naar het bestaan van pg(6,6,4)

In dit hoofdstuk passen we de *local approach* methode toe uit Hoofdstuk B.4 voor een zoektocht naar het bestaan van de partiële meetkunde $pg(6, 6, 4)$ met een automorfisme van orde 3 met 7 gefixeerde punten en rechten. Helass bestaat geen dergelijke $pg(6, 6, 4)$. De algemene bestaansvraag van $pg(6, 6, 4)$ blijft onopgelost. Dit is gemeenschappelijk werk met S. Topalova. We maakten elk een onafhankelijke implementatie.

Definitie B.6.1 (Partiële meetkunde) *Stel P en B disjuncte (niet-lege) objectverzamelingen van, respectievelijk, punten en rechten. Stel I een symmetrische punt-rechte incidentierelatie*

$$I \subseteq (P \times B) \cup (B \times P)$$

De partiële meetkunde $S = (P,\ B,\ I)$ met parameters $pg(s, t, \alpha)$ voldoet aan:

1. *Elk punt is incident met $1 + t$ $(t \geq 1)$ rechten en elk puntenpaar is incident met hoogstens één rechte.*
2. *Elke rechte is incident met $1 + s$ $(s \geq 1)$ punten en elk rechtenpaar is incident met hoogstens één punt.*
3. *Als x een punt is dat niet incident is met rechte L, dan bestaan er precies α $(\alpha \geq 1)$ punten $y_1, y_2, \ldots, y_\alpha$ en α rechten $M_1, M_2, \ldots, M_\alpha$ zodat xIM_i, M_iIy_i en y_iIL $(1 \leq i \leq \alpha)$.*

[2] private communicatie

Stel $|P| = v$ en $|B| = b$. Dan worden v en b als volgt afgeleid:

$$v = \frac{(s+1)(st+\alpha)}{\alpha} \tag{B.1}$$

$$b = \frac{(t+1)(st+\alpha)}{\alpha} \tag{B.2}$$

De parameters van $pg(6,6,4)$ zijn

$$s = 6;\ t = 6;\ \alpha = 4;\ v = 70;\ b = 70$$

De incidentiematrix is gedefinieerd op een analoge wijze zoals voor designs, waarbij we rijen nemen voor punten en kolommen voor rechten. De 70×70 incidentiematrix van $pg(6,6,4)$ heeft volgende eigenschappen:

1. Er zijn precies 7 enen per rij/kolom. Het scalair product van twee rijen/kolommen is 0 of 1.

2. Als punt x niet op rechte L ligt, dan bestaan er precies 4 punten y_1, y_2, y_3, y_4 en 4 rechten M_1, M_2, M_3, M_4 zodat volgende incidenties gelden:

	L	M_1	M_2	M_3	M_4
x	0	1	1	1	1
y_1	1	1	0	0	0
y_2	1	0	1	0	0
y_3	1	0	0	1	0
y_4	1	0	0	0	1

Het veronderstelde automorfisme laat toe om diverse bijkomende voorwaarden aan de orbitmatrix op te leggen. Er waren slechts twee startconfiguraties die beiden tot geen enkele orbitmatrix leidden. De eerste symmetrische startconfiguratie leidde tot een zoektocht van 55 seconden. Voor de tweede asymmetrische startconfiguratie volstond een zoektocht van ongeveer 66 uren. Een 1.8 GHz Pentium IV (met het Linux besturingssysteem) werd hiervoor gebruikt.

B.7 Enumeratie van de dubbels van het projectieve vlak van orde 4

De resultaten van dit hoofdstuk zijn verschenen in [18]. Er zijn precies 1 746 461 307 niet-isomorfe dubbels van het projectieve vlak van orde 4. Aangezien de designs met automorfismen van oneven priemorde reeds gekend waren, moesten we enkel nog designs met automorfismen van orde 2 construeren. Omdat we konden aantonen dat een 2-$(21,5,2)$ dubbel design uniek reduceerbaar is, was het mogelijk om het aantal designs te bepalen die enkel het triviaal automorfisme bezitten. De meeste computerresultaten werden verkregen door twee verschillende implementaties, een door mezelf en een door S. Topalova.

B.7.1 Inleiding

Elke 2-(v, k, λ) design impliceert het bestaan van een 2-$(v, k, m\lambda)$ design voor elk geheel getal $m > 1$. Door m incidentiematrices van een 2-(v, k, λ) design "aan elkaar te plakken", bekomen we de incidentiematrix van een 2-$(v, k, m\lambda)$ design. Een 2-$(v, k, m\lambda)$ is *reduceerbaar* in m 2-(v, k, λ) designs indien er een partitie van de blokken in m deelcollecties bestaat die elk een 2-(v, k, λ) design vormen. Dergelijke partitie noemen we een *reductie*. Voor $m = 2$ noemen we 2-$(v, k, m\lambda)$ designs *quasidubbels*, en de reduceerbare quasidubbels *dubbels*.

Met de notatie $(D_1 \cup D_2)$ bedoelen we een dubbel die gereduceerd kan worden in de designs D_1 en D_2. Een *reductie* van een dubbel D met parameters 2-$(v, k, 2\lambda)$ wordt voorgesteld door een verzameling van twee blokkencollecties die elk de helft van de blokken van D bevatten en elk een 2-(v, k, λ) design vormen. Een voor de hand liggende reductie van de dubbel $(D_1 \cup D_2)$ is $\{D_1, D_2\}$. We gebruiken vaak de notatie $D_2 = \mu D_1$, waarin μ een puntpermutatie is die, wanneer toegepast op de punten van D_1, D_2 oplevert. Twee reducties $\{D_1, D_2\}$ en $\{D_3, D_4\}$ van een dubbel zijn *equivalent* als en slechts als er een puntpermutatie μ bestaat zodat $D_3 = \mu D_1$ en $D_4 = \mu D_2$, *of* zodat $D_4 = \mu D_1$ en $D_3 = \mu D_2$. Een dubbel die slechts één inequivalente reductie heeft, wordt *uniek reduceerbaar* genoemd. We toonden aan dat, via een computergeassisteerd bewijs, reduceerbare 2-$(21, 5, 2)$ designs uniek reduceerbaar zijn.

B.7.2 Dubbels van een uniek reduceerbare design

Er bestaat slechts één 2-$(21, 5, 1)$ design (het projectieve vlak van orde 4). In deze sectie beschouwen we de dubbels $(D \cup \varphi D)$ van een design D, waarbij we tevens veronderstellen dat alle dubbels uniek reduceerbaar zijn. G is de volledige automorfismegroep van D. G_φ is de doorsnede van de volledige automorfismegroepen van D en φD.

De verzameling van alle $v!$ dubbels van D van de vorm $(D \cup \varphi D)$ (verkregen door alle $v!$ puntpermutaties φ te beschouwen) kan gepartitioneerd worden in isomorfieklassen $\mathcal{C}_G(\varphi)$ zodat

$$v! = \sum_{\mathcal{C}_G(\varphi)} |\mathcal{C}_G(\varphi)| \tag{B.3}$$

Stel dat N_i (resp. N_i') het aantal isomorfieklassen $\mathcal{C}_G(\varphi)$ voorstelt waarvoor $|G_\varphi| = i$ en $G\varphi G \cap G\varphi^{-1}G = \emptyset$ (resp. $G\varphi G = G\varphi^{-1}G$). Stel N het aantal niet-isomorfe dubbels van D. Volgende vergelijkingen kunnen afgeleid worden.

$$v! = 2|G|^2 N_1 + |G|^2 N_1' + \sum_{i>1} \frac{2|G|^2}{i} N_i + \sum_{i>1} \frac{|G|^2}{i} N_i' \,. \tag{B.4}$$

$$N = N_1 + N_1' + \sum_{i>1} N_i + \sum_{i>1} N_i' \,. \tag{B.5}$$

Het volstaat dus om de dubbels te genereren met niet-triviale automorfismen (getallen N_1', N_i en N_i' bepalen voor alle $i > 1$), om dan met vergelijking (B.4) N_1 te bepalen. Tenslotte gebruiken we vergelijking (B.5) om N te bepalen. Een 2-(v,k,λ) design D wiens dubbels allen uniek reduceerbaar zijn, heeft minstens $v!/(2|G|^2)$ niet-isomorfe dubbels. Het aantal dubbels is dus minstens 1 745 944 200.

B.7.3 Reduceerbare 2-$(21, 5, 2)$ met niet-triviale automorfismen

In [48] werden alle 2-$(21, 5, 2)$ designs met automorfismen van oneven priemorde geconstrueerd. Hiervan waren er 4 170 designs reduceerbaar. Voor de 2-$(21, 5, 2)$ dubbels $(D \cup \varphi D)$ met automorfismen van orde 2 onderscheiden we twee types. Er zijn er die elk 2-$(21, 5, 1)$ design in zichzelf transformeren (D in D, φD in φD) en er zijn er die elk 2-$(21, 5, 1)$ design in elkaar transformeren (D in φD en omgekeerd). We genereerden 40 485 designs van het eerste type en 991 957 van het tweede type. In deze samenvatting bespreken we enkel het geval van de automorfismen waarvoor $|G_\varphi| \neq 1$. Het andere geval ($|G_\varphi| = 1$) wordt behandeld met een gelijkaardige methode.

Beschouw $(D \cup \varphi D)$ waarvoor de volledige automorfismegroep orde 2^s ($s \geq 1$) heeft en $|G_\varphi| \neq 1$. Dan bevatten D en φD gemeenschappelijke automorfismen van orde 2. Dus we starten vanaf de unieke design D en construeren dan alle dubbels $(D \cup \varphi D)$ door alle puntpermutaties φ te beschouwen die:

(a) gefixeerde punten op gefixeerde punten afbeelden,

(b) een puntenpaar van één orbit op een puntenpaar van één orbit afbeelden.

De unieke 2-$(21, 5, 1)$ design D heeft automorfismen van orde 2 met 5 of 7 gefixeerde punten. We construeerden alle dubbels $(D \cup \varphi D)$ die opgebouwd zijn uit designs D en φD die beiden dergelijke automorfismen van orde 2 bezitten. We construeerden de dubbels op twee verschillende manieren.

De eerste manier bepaalt eerst alle permutaties van volledige orbits van D. Voor alle niet-equivalente gevallen beschouwen we dan alle puntpermutaties binnen elke orbit. Voor alle niet-equivalente gevallen voegen we alle mogelijke permutaties toe van het gefixeerd deel. Zodoende hebben we alle puntpermutaties φ beschouwd.

De tweede manier bepaalt eerst alle automorfismen van D. We genereren alle puntpermutaties φ (die voldoen aan bovengenoemde voorwaarden (a) en (b)) in een bepaalde lexicografische volgorde. Bij het genereren van een bepaalde permutatie φ zoeken we $\alpha, \beta \in G$ zodat $\beta\varphi\alpha$ of $\beta\varphi^{-1}\alpha$ een permutatie is die lexicografisch kleiner is dan φ (en van hetzelfde type). Indien er een dergelijk permutatiepaar $\alpha, \beta \in G$ bestaat, dan is de huidige oplossing equivalent met een reeds beschouwde. Deze techniek laat toe om partiële permutaties te elimineren.

Beide technieken construeren hetzelfde aantal dubbels.

B.7.4 Resultaat

Op basis van de 1 028 899 geconstrueerde dubbels met niet-triviale automorfismen, bepalen vergelijkingen (B.4) en (B.5) het aantal niet-isomorfe dubbels met uitsluitend triviale automorfismen: 1 745 432 408. Het totaal aantal 2-$(21, 5, 2)$ dubbels is 1 746 461 307, hetgeen weinig verschilt van de ondergrens 1 745 944 200.

B.8 Codewoorden van klein gewicht in de codes die voortkomen uit Desarguesiaanse projectieve vlakken van priemorde

In dit hoofdstuk behandelen we een probleem uit de codeertheorie, waarin computerresultaten ons hielpen om tot nieuwe resultaten te komen. Dit werk vormt een aanzienlijk deel van het artikel *Small weight codewords in the codes arising from Desarguesian projective planes*, dat ingezonden werd naar *Designs, Codes and Cryptography* [17]. We verbeteren de resultaten van K. Chouinard [12] over codewoorden van klein gewicht in de codes die voortkomen uit $PG(2,p)$, p priem. We vermelden ook enkele bijkomende resultaten. Mijn voornaamste bijdrage tot deze resultaten was het computerwerk. Voor de weggelaten bewijzen verwijzen we naar het artikel [17].

B.8.1 Inleiding

Het projectieve vlak is een verzameling punten, een verzameling rechten en een verzameling van punt-recht incidenties waarvoor het volgende geldt.

(a) Elk puntenpaar ligt op een unieke rechte.

(b) Elk rechtenpaar snijdt in een uniek punt.

(c) Er bestaan vier punten waarvan er geen drie collineair zijn (dus er zijn vierhoeken).

Het projectieve vlak $PG(2,q)$ van orde $q = p^h$ (p priem, $h \geq 1$) over het veld $\mathbb{F}_q$ heeft q^2+q+1 punten en rechten, en is equivalent met de symmetrische 2-$(q^2+q+1, q+1, 1)$ design.

We definiëren de incidentiematrix A met rijen voor de punten en kolommen voor de rechten. De code C van het projectieve vlak $PG(2,q)$, $q = p^h$, p priem, $h \geq 1$, is de $\mathbb{F}_p$-opspanning van de rijen van A [1] [39].

In [1] wordt aangetoond dat de incidentievectoren van een rechte de codewoorden van minimaal gewicht $q+1$ vormen. Chouinard [12] toonde aan dat er geen codewoorden zijn in het interval $[p+2, 2p-1]$ in de code die voortkomt uit $PG(2,p)$, p priem, en dat de enige codewoorden met gewicht $2p$ de scalaire veelvouden zijn van het verschil van twee verschillende incidentievectoren van een rechte. Aan de hand van een specifieke basis voor deze code, beschreven door Moorhouse [37], karakteriseren we alle codewoorden tot gewicht $2p+\frac{p-1}{2}$ ($p \geq 19$). Bovendien tonen we aan dat de enige codewoorden deze zijn met gewicht $p+1$, $2p$ en $2p+1$, en dat de codewoorden met gewicht $2p+1$ gevormd worden door de lineaire combinaties van twee verschillende incidentievectoren van een rechte, waarin de lineaire combinatie verschillend van nul is in het intersectiepunt.

B.8.2 De Moorhouse basis voor $AG(2,p)$, p priem

De rang van de p-voudige lineaire code van het projectieve (resp. affiene) vlak $PG(2,p)$ (resp. $AG(2,p)$), p priem, is $\binom{p+1}{2}+1$ (resp. $\binom{p+1}{2}$).

In [37] wordt de volgende constructie voor een basis van de incidentiematrix A van $PG(2,p)$ gegegeven, waarbij $r_0, r_1, \ldots, r_p$ de punten zijn van een willekeurige rechte M:

- Voor $i \in \mathbb{N},\ 0 \leq i \leq p-1$: neem $p-i$ willekeurige rechten door r_i. Deze $\binom{p+1}{2}$ rechten vormen een basis voor het affiene vlak.
- Rechte M vervolledigt deze basis tot een basis voor het projectieve vlak.

Door met de computer de coördinaten te bestuderen van de rechten t.o.v. dergelijke basis, kwamen we tot volgende lichte variatie van de Moorhouse basis, hetgeen bewezen wordt in ons artikel [17]. Deze variatie wordt gebruikt in de karakterisatie van de codewoorden.

Stelling B.8.1 *De ruimte voortgebracht door de affiene rechten van de Moorhouse basis door r_0, r_1 en r_2, kan ook voortgebracht worden door $p-1$ affiene rechten door elk van de punten r_0, r_1 en r_2 te kiezen, waarbij de drie niet geselecteerde rechten niet concurrent zijn.*

B.8.3 Computerresultaten

We onderzochten het effect van het verwijderen van alle rechten van de hermitische kromme in $PG(2,q)$, q een kwadraat. De computerresultaten toonden dat de rang verlaagde door deze rechten te verwijderen. Vervolgens werd het volgende algemene resultaat bewezen [17].

Stelling B.8.2 *Beschouw het verwijderen van een verzameling kolommen uit A. De rang van A verlaagt als en slechts als er een codewoord in C bestaat wiens niet-nul posities bevat zijn in de verzameling verwijderde kolommen uit A.*

Omwille van dit resultaat was het interessant om te onderzoeken welke verzameling verwijderde rechten er een rangverlaging veroorzaakt. We onderzochten exhaustief alle mogelijke verwijderingen van m kolommen voor de projectieve vlakken van kleinste orde. We zien dat bij het verwijderen van minder dan $2p$ rechten, de rang slechts verlaagt indien we alle rechten door een punt verwijderen. Bij het verwijderen van $2p$ rechten, verlaagt de rang wanneer we alle rechten door twee punten, behalve hun verbindingsrechte, verwijderen. Wanneer we ook de verbindingsrechte verwijderen, verlaagt de rang met 2.

Wanneer we alle rechten door drie punten verwijderen, verlaagt de rang met 4 wanneer we alle rechten door drie collineaire punten verwijderen, en met 3 wanneer we alle rechten door drie niet-collineaire punten verwijderen.

Wanneer we $(p-1)$ rechten verwijderen door elk van drie collineaire punten a, b en c, maar niet hun verbindingsrechte ab, dan verlaagt de rang als en slechts als de drie niet-verwijderde rechten ($\neq ab$) concurrent zijn. We construeerden tevens een algoritme om het gerelateerde codewoord te construeren dat overeenstemt met dergelijke verzameling van $3(p-1)$ rechten.

We beschouwen ook het verwijderen van $(p-2)$ rechten door vier collineaire punten, behalve hun verbindingsrechte. In dit geval verlaagt de rang indien de 8 niet-verwijderde rechten kunnen gepartitioneerd worden in 2 disjuncte verzamelingen van concurrente rechten. We kunnen ook de codewoorden construeren voor $PG(2,p)$, p priem, $p \leq 23$.

B.8.4 Verbeterde resultaten voor $PG(2,p)$, p priem

We beschouwen alle gevallen waarin we hoogstens $2p + \frac{p-1}{2}$ rechten verwijderen die overeenstemmen met een verzameling niet-nul posities van een codewoord. Door inductie op het gewicht tonen we het volgende aan.

Stelling B.8.3 *De enige codewoorden c, met $0 < wt(c) \leq 2p + \frac{p-1}{2}$, in de p-voudige lineaire code C die voortkomt uit $PG(2, p)$, p priem, $p \geq 19$, zijn:*

- *Codewoorden met gewicht $p + 1$: scalaire veelvouden van incidentievectoren van een rechte.*
- *Codewoorden met gewicht $2p$: $\alpha(c_1 - c_2)$, met c_1 en c_2 de incidentievectoren van twee rechten.*
- *Codewoorden met gewicht $2p + 1$: $\alpha c_1 + \beta c_2$, $\beta \neq -\alpha$, met c_1 en c_2 de incidentievectoren van twee rechten.*

B.9 Farming package

De meeste exhaustieve backtrackalgoritmen kunnen op eenvoudige wijze opgesplitst worden in onafhankelijke processen die elk een verschillend deel van de zoekboom genereren. Beschouw bijvoorbeeld een zoekruimte die n_d toppen bevat op diepte d van de zoekboom. De taak om alle toppen op grotere diepte te genereren, is opdeelbaar in m onafhankelijke deeltaken die elk $\frac{n_d}{m}$ deelzoekbomen uitwerken, met als wortel een van de toppen op diepte d.

Om dergelijke parallelle programma's te schrijven, ontwikkelden we een *farming* pakket, geschreven in *Java* met *Java RMI* [3] [19]. De Engelstalige appendix is geschreven in de vorm van een tutorial om het pakket te gebruiken. We beschrijven hier slechts de conceptuele samenhang van het pakket.

Een farming applicatie bestaat uit één meesterproces en verscheidene slaafprocessen. Deze processen draaien bij voorkeur op afzonderlijke processoren. We schrijven kortweg *meester* en *slaaf* voor, respectievelijk, meesterproces en slaafproces. De meester beheert de taken die uitgevoerd moeten worden door de slaven. Slaven worden geïdentificeerd door een unieke naam. Meestal mag om het even welke slaaf een bepaalde taak uitvoeren. De communicatie tussen meester en slaaf gebeurt als volgt. Eerst wordt de meester gestart, wachtend op slaven die zich zullen registreren. Vervolgens worden de slaven opgestart. De slaven registreren zich vervolgens bij de meester met een bepaalde slaafnaam. Wanneer een slaaf niet wordt toegelaten (bijvoorbeeld omdat de slaafnaam reeds gebruikt is), eindigt de slaaf. In het normale geval vraagt de slaaf een taak aan de meester, waarop de meester de gevraagde taak teruggeeft. Wanneer er geen taken meer zijn, krijgt de slaaf een *null* taak terug, waarop de slaaf zichzelf beëindigt. In het normale geval voert de slaaf zijn taak uit en geeft het resultaat terug aan de meester. Daarna vraagt de slaaf een nieuwe taak, enzovoort. De meester beëindigt zichzelf wanneer alle resultaten behandeld zijn. Taken beschikken ook over de mogelijkheid om bepaalde (tussentijdse) resultaten te schrijven naar de meester m.b.v. standaard Java uitvoerstromen. Het ontwikkelde *farming* pakket is slechts bruikbaar voor problemen die in relatief grote deelproblemen kunnen opgesplitst worden. Een taak moet toch enkele seconden duren opdat de RMI overhead verwaarloosbaar zou zijn. Merk op dat de taakresultaten niet in dezelfde volgorde de meester hoeven te bereiken zoals

[3] http://java.sun.com/products/jdk/rmi/. Java Remote Method Invocation (Java RMI) maakt het mogelijk om gedistribueerde Java-gebaseerde applicaties te schrijven, waarbij methoden van Java objecten kunnen opgeroepen worden vanuit een andere Java virtuele machine, die zich op een andere machine kunnen bevinden.

ze uitgedeeld werden. De meester kan het falen van een bepaalde slaaf opvangen omdat elke slaaf op regelmatige tijdstippen een signaal naar de meester stuurt dat aanduidt dat hij nog steeds leeft. Wanneer dit slaafsignaal te laat komt, zal de meester de betreffende taak uitdelen aan een andere slaaf. Dit pakket wordt gebruikt in de practica van het vak "Parallelle Algoritmen".

B.10 Software tools

We verwijzen naar de webpagina *http://users.ugent.be/∼jpwinne/phd* voor de software en zijn documentatie.

C Software tools

We give a brief overview of the developed packages in Section C.1. For the software and its full documentation, we refer to the website *http://users.ugent.be/∼jpwinne/phd*

In Section C.2 we list other used packages which were written by other members of our research group CAAGT.

We also used the *trove* library, available from *http://trove4j.sourceforge.net*. This is an implementation of high performance collections for Java. In particular, we used its collections for primitive types.

C.1 Developed packages

The following packages, which contain many subpackages, were developed.

- **be.ugent.caagt.backtrack**: Provides a framework for the exhaustive generation of combinatorial objects that satisfy certain user defined constraints.
- **be.ugent.caagt.codes**: Package for the classification of the related graphs of the article about the projective two-weight codes with small parameters and their corresponding graphs.
- **be.ugent.caagt.design**: General package for the exhaustive generation of designs.
- **be.ugent.caagt.design.doubles**: Package related to the enumeration of the doubles of the projective plane of order 4.
- **be.ugent.caagt.design.example**: Package containing the example of Section 2.4.
- **be.ugent.caagt.design.orbit**: Package for the exhaustive generation of designs with an assumed automorphism.
- **be.ugent.caagt.farming**: Package to farm a problem on a cluster.
- **be.ugent.caagt.gui.gentool**: Package which contains the visualizer application of the backtrack framework.

- **be.ugent.caagt.nauty**: Package to call nauty from Java.
- **be.ugent.caagt.nonsymmetricbacktrack**: General package for the exhaustive generation of combinatorial objects which are represented by a rectangular integer matrix.
- **be.ugent.caagt.pg**: Package related to the article about small weight codewords in the codes arising from projective planes of prime order.
- **be.ugent.caagt.pg664**: Package for the search of the partial geometry $pg(6, 6, 4)$.
- **be.ugent.caagt.rim**: Package containing integer matrix implementations for rectangular integer matrices.

C.2 Other packages of the CAAGT library

The following packages were used.

- **be.ugent.caagt.algebra**: Package containing finite field implementations.
- **be.ugent.caagt.perm**: Package about permutation groups.
- **be.ugent.caagt.util**: Package containing general utility classes.
- **be.ugent.caagt.im**: Package containing interfaces and standard implementations for integer matrices and graphs.

List of Figures

List of Tables

List of Algorithms

List of Java Source

Bibliography

[1] E. F. Jr. Assmus and J. D. Key. Cambridge University Press, Cambridge, 1992.

[2] S. Ball. Multiple blocking sets and arcs in finite planes. *J. London Math. Soc.*, 54(3):581–593, 1996.

[3] Th. Beth, D. Jungnickel, and H. Lenz. *Design Theory, 2nd edition.* Cambridge University Press, 1999.

[4] J. Bloch. *Effective Java Programming Language Guide.* Addison Wesley Professional, 2001.

[5] I. Bouyukliev, V. Fack, W. Willems, and J. Winne. Projective two-weight codes with small parameters and their corresponding graphs. *Des. Codes Cryptogr.*, 41:59–78, 2006.

[6] I. Bouyukliev, V. Fack, and J. Winne. Hadamard matrices of order 36 and doubly-even self-dual [72,36,12] codes. *Discrete Mathematics and Theoretical Computer Science Proceedings*, AE:93–98, 2005.

[7] I. Bouyukliev, V. Fack, and J. Winne. 2-(31,15,7), 2-(35,17,8) and 2-(36,15,6) designs with automorphisms of odd prime order, and their related hadamard matrices and codes. *J. Combin. Designs*, 2006. Submitted.

[8] S. Bouyuklieva. On the automorphisms of order 2 with fixed points for the extremal self-dual codes of length $24m$. *Des. Codes Cryptogr.*, 25:5–13, 2002.

[9] S. Bouyuklieva. On the automorphism group of a doubly-even (72,36,16) code. *IEEE Trans. Info. Theory*, 50:544–547, 2004.

[10] Van Lint J.H. Cameron P.J. *Designs, Graphs, Codes and their Links.* Cambridge University Press, 1991.

[11] A.R. Camina. A survey of the automorphism groups of block designs. *J. Combin. Designs*, 2(2):79–100, 1994.

[12] K. L. Chouinard. *Weight distributions of codes from planes.* PhD thesis, 2000.

[13] C.J. Colbourn and J.H. (Eds.) Dinitz. *The CRC Handbook of Combinatorial Designs.* CRC Press, Boca Raton, FL., 1996.

[14] C.J. Colbourn and J.H. (Eds.) Dinitz. *The CRC Handbook of Combinatorial Designs.* CRC Press, Boca Raton, FL., 1996.

[15] R. Dontcheva, A.J. van Zanten, and S.M. Dodunekov. Binary self-dual codes with automorphisms of composite order. *IEEE Trans. Info. Theory*, 50(2):311–318, 2004.

[16] V. Fack, I. Bouyukliev, W. Willems, and J. Winne. Projective two-weight codes with small parameters and their corresponding graphs. In *Proceedings of OC'2005, Fourth International Workshop on Optimal Codes and Related Topics (Pamporovo, Bulgaria)*, pages 139–145, 2005.

[17] V. Fack, L. Storme, G. Van de Voorde, and J. Winne. Small weight codewords in the codes arising from desarguesian projective planes. *Des. Codes Cryptogr.*, 2007. Submitted.

[18] V. Fack, S. Topalova, J. Winne, and R. Zlatarski. Enumeration of the doubles of the projective plane of order 4. *Discrete Math.*, 306:2141–2151, 2006.

[19] J. Farley. *Java distributed computing.* O'Reilly, 1998.

[20] S. Georgiou, C. Koukouvinos, and J. Seberry. Hadamard matrices, orthogonal designs and construction algorithms. In W. D. Wallis, editor, *Designs 2002: Further Combinatorial and Constructive Design Theory*, pages 133–205. Kluwer, Academic Publishers, Norwell, Massachusetts, 2002.

[21] B. Huppert. *Endliche Gruppen I.* Springer-Verlag, 1967.

[22] Z. Janko. The existence of a Bush-type Hadamard matrix of order 36 and two new infinite classes of symmetric designs. *J. Combin. Theory Ser. A*, 95:360–364, 2001.

[23] D. Jungnickel. Quasimultiples of projective and affine planes. *J. Geom.*, 26:172–181, 1986.

[24] D. Jungnickel and K. Vedder. Simple quasidoubles of projective planes. *Aequationes Math.*, 34:96–100, 1987.

[25] S.N. Kapralov, I.N. Landgev, and V.D. Tonchev. 2-(25,10,6) designs invariant under the dihedral group of order 10. *Ann. Discrete Math.*, 34:301–306, 1987.

[26] P. Kaski and P. Östergård. Miscellaneous classification results for 2-designs. *Discrete Math.*, 280:65–75, 2004.

[27] P. Kaski, P. Östergård, S. Topalova, and R. Zlatarski. Steiner triple systems of order 19 and 21 with subsystems of order 7. *Discrete Math.*

[28] C. Lam, S. Lam, and V.D. Tonchev. Bounds on the number of affine, symmetric and Hadamard designs and matrices,. *J. Combin. Theory Ser. A*, 92:186–196, 2000.

[29] C. Lam, S. Lam, and V.D. Tonchev. Bounds on the number of Hadamard designs of even order,. *J. Combin. Designs*, 9:363–378, 2001.

[30] C. Lin, D. Wallis, and Zhu Lie. Generalized 4-profiles of Hadamard matrices. *J. Comb. Inf. Syst. Sci.*, 18:397–400, 1993.

[31] C. Lin, D. Wallis, and Zhu Lie. Hadamard matrices of order 32 II. Technical Report 93-05, Department of Mathematical Science, University of Nevada, Las Vegas, Nevada, 1993. Preprint.

[32] D.K.J. Lin and N.R. Draper. Screening properties of certain two-level designs. *Metrika*, 42:99–118, 1995.

[33] R. Mathon. Symmetric (31,10,3) design with non-trivial automorphism group. *Ars Combin.*, 25:171–183, 1988.

[34] Rosa A. Mathon R. Some results on the existence and enumeration of bibds. Technical Report 125, Math. Report, Dept.of Math.&Stat, McMaster Univ., 33, 1985.

[35] B.D. McKay. Hadamard equivalence via graph isomorphism. *Discrete Math.*, 27:213–214, 1979.

[36] B.D. McKay. Nauty users' guide (version 2.2). Technical Report, Computer Science Department, Australian National University, 2004.

[37] G. E. Moorhouse. Bruck nets, codes, and characters of loops. *Des. Codes Cryptogr*, 1(1):7–29, 1991.

[38] W.P. Orrick. Switching operations for Hadamard matrices. 2005. http://arxiv.org/abs/math/0507515.

[39] H. Sachar. *Error-correcting codes associated with finite planes.* PhD thesis, 1973.

[40] J. Seberry. Library of Hadamard matrices. http://www.uow.edu.au/∼jennie/hadamard.html.

[41] J. Seberry and M. Yamada. *Hadamard matrices, sequences and block designs*, pages 431–560. J. Wiley, New York, 1992.

[42] N.J.A. Sloane. Is there a (72,36), $d = 16$ self-dual code? *IEEE Trans. Info. Theory*, 19:251, 1973.

[43] E. Spence and V.D. Tonchev. Extremal self-dual codes from symmetric designs. *Discrete Math.*, 110:265–268, 1992.

[44] V.D. Tonchev. Hadamard matrices of order 36 with automorphisms of order 17. *Nagoya Math. J.*, 104:163–174, 1986.

[45] V.D. Tonchev. *Combinatorial Configurations.* Longman Scientific and Technical, New York, 1988.

[46] V.D. Tonchev. *Combinatorial Structures and Codes*. Kliment ohridski University Press, 1988.

[47] V.D. Tonchev. Symmetric designs without ovals and extremal self-dual codes. *Ann. Discrete Math.*, 37:451–458, 1988.

[48] S. Topalova. Enumeration of 2-(21,5,2) designs with automorphisms of an odd prime order. *Diskretnii Analiz i Issledovanie Operatsii*, 5(1):64–81, 1998. In Russian.

[49] S. Topalova and J. Winne. Construction techniques for incidence structures. In *Combinatorics 2004 conference (Sicilie)*.

[50] R.M. Wilson. Nonisomorphic steiner triple systems. *Math. Z.*, 135:303–313, 1974.