Branch-and-cut for symmetrical ILPs and combinatorial designs

Sebastian Raaphorst

A thesis submitted in conformity with the requirements

for the degree of Master's of Science (Computer Science).

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario, Canada

October, 2004

Abstract

Many combinatorial problems can be formulated as a 0-1 integer linear program (0-1 ILP), which consists of an objective function subject to linear constraints over 0-1 variables. A method called branch-and-cut has been successfully used to attack ILPs, but ILPs are still difficult to solve in practice.

The problems we investigate demonstrate large numbers of equivalent (isomorphic) solutions. We are only interested in generating one solution from each equivalence class. To accomplish this, we study and extend a method by Margot [22] that avoids considering unnecessary partial solutions by generating an isomorph-free search tree. We implement a branch-and-cut library, and solve combinatorial design problems involving t-designs, packings, coverings and intersecting set systems.

We experimentally analyze the strengths and limitations of our algorithms and determine when it is efficient to use isomorph-free branch-and-cut. Our framework generates new results and reproduces, in competitive time, existing ones.

Acknowledgments

First and foremost, I would like to extend my deepest gratitude to my supervisor, Lucia Moura, who not only helped me immeasurably on this thesis, but provided me with opportunities while I was an undergraduate student that kindled my passion for teaching and research.

I would also like to thank the other members of my defense committee, Sylvia Boyd and Brett Stevens, for their feedback and suggestions on this thesis.

Additionally, I cannot thank Daniel Panario, Lucia, and Sylvia enough for happily going out of their way to write me numerous letters and recommendations for funding and applications. I'd also like to acknowledge Dr. François Margot for taking the time to speak with me and help me with this work on many occasions, and, of course, for his papers and algorithms.

Karen Meagher (the Meagherinator), my sassy office mate, kept things in our cell lively and never failed to distract us from our work. A huge thank you to her for helping me stave off insanity for yet another day. Blair Laugher was both a source of motivation and emotional support; without him, this would not have been possible. Norma Hannant, my best friend, encouraged me to no end and endured the roller coaster ride of emotions that was this thesis.

Of course, I would like to thank my parents, Ginette and Peter Raaphorst, for always standing behind me and offering me constant support, and for taking an interest in my work. My cat, Fritz, and his insistence on us taking afternoon naps together in sunbeams for much needed breaks cannot go without thanks as well.

Lastly, I would like to acknowledge my gratitude to my husband, Jeff, for all his encouragement and companionship. His optimism helped me to persevere when I

thought that I couldn't possibly manage to type another line of code. Here's to all the adventures we've shared together thus far, and for many more to come.

Contents

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and thesis overview

In this thesis, we are interested in investigating the efficiency of using isomorph-free and isomorph-reduced branch-and-cut trees to solve 0-1 integer linear programs that, structurally, have a large number of symmetries.

In the study of combinatorics, there are many classes of problems that can be formulated as an objective function that we wish to either maximize or minimize to give an optimal solution subject to certain linear constraints over a set of variables. Such a formulation is called an *integer linear program* (or ILP). If we restrict the values of the variables to be either 0 or 1, we have a *0-1 integer linear program*. Solving general ILPs is a difficult problem and has been shown to be NP complete. Several techniques have been established in order to solve ILPs: these include branch-and-bound [21, 40], cutting-plane methods [13, 36, 40], and the branch-and-cut algorithm [1, 15, 16, 36, 40]. Thorough treatments of theory of ILPs can be found in [38] and [46].

Many combinatorial objects demonstrate a large number of *isomorphs*, which are other objects that have the same structure. For a given problem, we can hence divide the solutions into *equivalence classes*, or sets of objects with identical structure. In

the majority of cases, we are only interested in generating one object from each of these sets, which we will call a *canonical representative* of the class. The others are usually not of importance, and if they are, they may be trivially constructed from the canonical representative.

These isomorphisms reflect themselves in the ILP formulation for the problem as permutations over the variables or *symmetries*. Since the combinatorial questions that we will be investigating have a large number of symmetries, it is our desire to discover and exploit these symmetries with the goal of reducing the number of partial solutions and solutions that we consider; this will allow us to limit our consideration of the search space dramatically. One possible technique is to reject certain partial solutions if we can ascertain that they will lead to noncanonical solutions. This is a partial method that has been used successfully [34, 35]. Another option is to build a search space that is entirely isomorph-free, i.e. we never consider a partial solution unless we are guaranteed that it will lead to a canonical solution. Methods for exploring nonisomorphic branch-and-cut trees have been proposed [22, 23].

It is the goal of this thesis to study and extend the isomorph-free branch-and-cut framework presented by Margot [22]. We do so through implementing Margot's algorithms, and then extend them by adding the option to generate isomorph-reduced trees and by expanding their applicability to four different types of problems: search for an optimal solution, generation of all canonical optimal solutions, generation of all canonical maximal solutions, and generation of all canonical feasible solutions. We then apply this framework to several areas of interest with regards to combinatorial designs to seek to gain insight into the capabilities and limitations of these algorithms, and to attempt to obtain some new results. As a secondary goal, we wish to release an efficient and flexible isomorph-free branch-and-cut programming library to the public domain in order to facilitate solving search and generation problems.

The accomplishments of this thesis include a deeper understanding of the techniques

used by Margot and extended by us. We were able to highlight strengths and weaknesses of this approach, as well as determine what types of ILPs are better suited to these algorithms. We were able to reproduce, with competitive times, some well-known results from combinatorial design theory, and generate several new ones.

In the remainder of this chapter, we present some definitions regarding integer linear programs and group theory that are necessary to the understanding of the framework and the concept of a symmetry group. We also show examples demonstrating the great reduction obtained on the search-tree sizes by using isomorph-free trees. Chapter 2 explores the combinatorial design problems that we are interested in addressing, and demonstrates possible integer linear programs and their respective symmetry groups for each problem. In Chapter 3, we provide a more detailed description of integer linear programs and discuss branch-and-bound, cutting plane methods, and branch-and-cut. In Chapter 4, we delve into the topic of symmetry groups, their representations as data structures, the algorithms that exploit these representations in order to generate isomorph-free or reduced branch-and-cut trees, and techniques for dynamically discovering these groups. Our implementation of the cutting plane, along with the three cuts that we used (two types of clique cuts, and Margot's isomorphism cuts [22]) are presented in Chapter 5. In Chapter 6, we describe algorithms for the four types of problems that our framework is able to address: search for an optimal solution, generation of all canonical optimal solutions, generation of all canonical maximal solutions, and generation of all canonical solutions. In Chapter 7, we explain our specific implementation. We proceed to use our framework to solve several different families of problems, and provide our experimental results in Chapter 8. We present a summarized analysis of these results, and discuss potential questions for future research in Chapter 9.

1.2 Linear programs and integer linear programs

There is a large class of optimization problems that can be formulated as a set of variables and a linear function that we wish to maximize or minimize over the variables subject to a certain set of linear constraints, or inequalities. While the origins of solving optimization problems with equalities dates back to Lagrange, linear programming is largely recognized as having developed in the 1940s to solve planning problems for war operations. After the second world war had ended, industry became aware of the power of linear programs, and it rapidly gained widespread acceptance. Both George B. Dantzig (who introduced the simplex method) and John von Neumann (who derived the theory of duality) are considered the founders of linear programming. The actual name "linear programming" was suggested by T. Koopmans in 1948. [8, 10]

Definition 1.2.1. A *linear programming problem*, or LP, is a mathematical model of the form:

$$\begin{aligned} \text{Maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

where x is an n-vector of variables, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. We can similarly define LPs that minimize the objective function.

Definition 1.2.2. Given a linear programming problem and an n-vector x^*, x^* is a *feasible solution* to the problem if $Ax^* \leq b$. We say that x^* is an optimal solution if, for any other feasible solution y^*, $c^T x^* \geq c^T y^*$.

Definition 1.2.3. An *integer linear programming problem*, or ILP, consists of an LP with the added requirement that the variables take integer values. A *0-1 integer linear programming problem* is an ILP such that the values of all the variables are restricted to either 0 or 1.

There are several techniques for solving linear programs: some of the most common are the simplex method [8], the ellipsoid method [20], and the interior point method

[19].

Solving integer linear programming problems is a harder problem. We investigate algorithms used to do so in Chapter 3. While the algorithms in that chapter can be used to solve any ILP, in this thesis, our focus will be specifically on 0-1 ILPs.

1.3 Group theory, permutation groups, and symmetry groups

As we rely on permutation groups in our algorithms, we begin by presenting some history and definitions from group theory. For an introduction to group theory, see [39].

Group theory is one of the oldest and best studied fields in algebra. The development of modern day group theory was largely driven by the work of Evariste Galois in the early 19th century in his investigations of permutations of the roots of polynomials. Even though prior to the introduction of the concepts of groups in their abstract form it was common for mathematicians to work with permutation groups, the actual definition of a group was not proposed until 1854 by Arthur Cayley [39].

Definition 1.3.1. A *group* is an ordered pair $(G, \circ)$, where G is a set of elements and $\circ$ is a binary operation $\circ : G \times G \to G$ such that the following properties hold:

1. *Associativity of* $\circ$*:* for all $a, b, c \in G$, $a \circ (b \circ c) = (a \circ b) \circ c$
2. *Existence of an identity:* there exists $i \in G$ such that for all $a \in G$, $a \circ i = i \circ a = a$
3. *Existence of inverses:* for all $a \in G$, there exists $b \in G$ such that $a \circ b = b \circ a = i$.

A group is *finite* if the set G contains a finite number of elements.

Definition 1.3.2. A *subgroup* G' of a group G is a subset of G that is a group under the operation of G. We say that $G' \leq G$.

When it is unambiguous to do so, we write $a \circ b$ simply as ab. For $a \in G$, we refer to its inverse as a^{-1}. If the operator on the group is implied, we omit the ordered-pair notation and refer to a group simply by its base set G. We denote the identity element of G as 1 or 1_G to be precise if we are dealing with multiple groups.

Definition 1.3.3. A finite group G is said to be *generated* by a set $X = \{g_0, g_1, \ldots\} \subseteq G$ if for any $g \in G$,

$$g = g_{i_0} g_{i_1} \ldots \text{ with } g_{i_j} \in X \text{ for } i = 0, 1, \ldots$$

We then say that X generates G, denoted $G = \langle X \rangle$. If, by removing any element $x \in X$, we have $G \neq \langle X \setminus \{x\} \rangle$, then we call X a *minimal set of generators*.

Representing a group by a set of generators is required to store large groups and may be useful in algorithms involving them, as the ones proposed in Chapter 4.

A group G is said to act on a set X if, for all $g \in G$, we can define a bijection $g : X \to X$. An example of such groups are the permutation groups.

Definition 1.3.4. Let X be a set with a group G acting on it. For $A \subseteq X$, we define the *orbit of A in G* as follows:

$$\text{orb}(A, G) = \{g(A) \mid g \in G\}.$$

We define the *stabilizer of A in G* as follows:

$$\text{stab}(A, G) = \{g \in G \mid g(A) = A\}.$$

If $A = \{x\}$, we omit set notation and simply write $\text{orb}(x, G)$ and $\text{stab}(x, G)$.

Definition 1.3.5. For a group G and a family of subgroups $G_0, G_1, \ldots, G_{k-1}$, G is said to be the *direct product* of $G_0, G_1, \ldots, G_{k-1}$ (written $G = G_0 \odot G_1 \odot \ldots \odot G_{k-1}$) if:

1. $G_i \cap G_j = \{1_G\}$ for all $\{i, j\} \subseteq \mathbb{Z}_k$,

2. for all $g \in G$ there exists $g_0 \in G_0, g_1 \in G_1, \ldots, g_{k-1} \in G_{k-1}$ such that:

$$g = g_0 g_1 \ldots g_{k-1}.$$

When we are discussing permutation groups, we use the term *direct composition*. If we can find such a family of subgroups and minimal sets of generators $X_0, X_1, \ldots, X_{k-1}$ for each subgroup, then $X_0 \cup X_1 \cup \ldots \cup X_{k-1}$ is a minimal set of generators for G.

Definition 1.3.6. Given two groups (not necessarily distinct), $(G_1, \circ)$ and $(G_2, \cdot)$, a *homomorphism* ϕ is a mapping $\phi : G_1 \to G_2$ such that the following properties hold:

1. $\phi(1_{G_1}) = 1_{G_2}$
2. for all $a, b \in G_1$, $\phi(a \circ b) = \phi(a) \cdot \phi(b)$

An *isomorphism* is a bijective homomorphism. Two groups are said to be *isomorphic* if we can find an isomorphism from one to the other; we then consider them to be structurally the same group. If G_1 and G_2 are isomorphic, we write this $G_1 \cong G_2$.

The groups that we shall be most interested in will be the groups of permutations over a base set (for example, in the case of symmetry groups, the set of variables of our ILP). If our base set is B with $|B| = n$, we denote the group of all permutations over the elements of B as $S(B)$. Let $\mathbb{Z}_n$ denote the integers modulo n. As a matter of simplicity, we denote the permutation group of $\mathbb{Z}_n$ simply as S_n (the symmetric group of order n). By choosing a bijection from B to $\mathbb{Z}_n$, we can use this bijection to create an isomorphism from $S(B)$ to S_n, and hence, we conclude that $S(B) \cong S_n$. We will often refer to permutations over arbitrary sets of size n as elements of S_n.

In this thesis, we will represent permutations interchangeably in two different ways: the vector and the cycle notation. In the vector notation, if $p \in S_n$, we can identify p with $[a_0, a_1, \ldots, a_{n-1}]$, where $p(x) = a_x$ for $x \in \mathbb{Z}_n$. In the cycle notation, a permutation is written as a composition of disjoint cycle: a cycle c is a permutation represented by

a k-tuple $(a_0\ a_1\ \ldots\ a_{k-1})$ where:

$$c(x) = \begin{cases} x & \text{if } x \notin \{a_0, \ldots, a_{k-1}\}, \\ a_{i+1 \bmod k} & \text{if } x = a_i. \end{cases}$$

Any permutation p can be written as a product of disjoint tuples, which represent a composition of cycles.

Example: Consider the permutation $p \in S_6$ that maps $0 \to 3$, $1 \to 1$, $2 \to 5$, $3 \to 4$, $4 \to 0$, and $5 \to 2$. In vector notation, we represent p as $[3, 1, 5, 4, 0, 2]$. In cycle notation, this is written $(0\ 3\ 4)(2\ 5)$.

Theorem 1.3.1. *The set* $\{(0\ 1), (0\ 2), \ldots, (0\ n-1)\}$ *is a set of minimal generators of* S_n.

If we have a permutation group G over a set S, we can define the action of a permutation $g \in G$ on a subset $X \subseteq S$ as follows:

$$g(X) = \{g(x) \mid x \in X\}.$$

For a permutation group G over the set $\mathbb{Z}_n$ and any set S, we may extend the action of G to vectors in S^n. Let $g \in G$ and $s = (s_0, s_1, \ldots, s_{n-1}) \in S^n$. Then:

$$g(s) = (s_{g^{-1}(0)}, s_{g^{-1}(1)}, \ldots, s_{g^{-1}(n-1)})$$

Example: If we have the permutation $g = (01)(245) \in S_6$ and a vector $v = (0, 2, 3, 1, 2, 1) \in \mathbb{Z}_4^6$, then $g(v) = (2, 0, 1, 1, 3, 2)$.

We will be interested in one type of permutation group called the *symmetry group of the ILP*, which will be the basis for many of the algorithms that we will be examining.

Definition 1.3.7. Consider an ILP:

$$\begin{aligned} \text{Maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \in \{0, 1\}^n \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, and $b \in \mathbb{R}^m$. Let $p \in S_n$ and $q \in S_m$, and let $A(p,q)$ be the matrix A with its columns permuted according to p and its rows permuted according to q (via the above defined action of permutations on vectors). Define the group:

$$G = \{p \in S_n \mid p(c) = c \text{ and there exists } q \in S_m \text{ such that } q(b) = b \text{ and } A(p,q) = A\}.$$

Clearly, G is a subgroup of S_n, and acts on the variables of our ILP. We call G the *symmetry group* of the ILP.

1.4 Two examples of the effectiveness of isomorph-free branch-and-cut

Example: The ILP:

$$\begin{aligned}
\textbf{ILP1: Maximize} \quad & x_1 + 2x_2 + x_3 + 2x_4 \\
\text{subject to} \quad & x_1 + x_2 = 2, \\
& x_1 + x_3 = 1, \\
& x_2 + x_4 = 1, \\
& x_3 + x_4 = 2, \\
& x \in \{0,1\}^4,
\end{aligned}$$

has symmetry group $G = \{1_G, (1\ 3), (2\ 4), (1\ 3)(2\ 4)\}$. A standard branch-and-cut tree with depth-first branching is given in Figure 1.1 and contains 23 nodes. An isomorph-free branch-and-cut tree is shown in Figure 1.2 and contains only five nodes.

This ILP is small and not indicative of combinatorial problems that arise in the real world. We present branch-and-cut trees for the generation of all canonical 2-$(5,3,1)$ designs to illustrate how isomorph-free branch-and-cut trees fare in practical problems: Figures 1.3 and 1.4 show the tree for a standard branch-and-cut, and Figure 1.5 shows the isomorph-free tree. There is only one unique solution; the full branch-and-cut tree

presents all 15 isomorphs in 175 nodes, whereas the isomorph-free tree gives one unique solution in only seven nodes.

Even this problem is extremely small, having only 10 variables and 10 constraints, and a symmetry group of size 120. For a larger problem, the 2-$(8, 3, 1)$ packing generation, we have 56 variables and 28 constraints. Our standard branch-and-cut tree contains 80063 nodes, whereas the isomorph-free tree contains only 33.

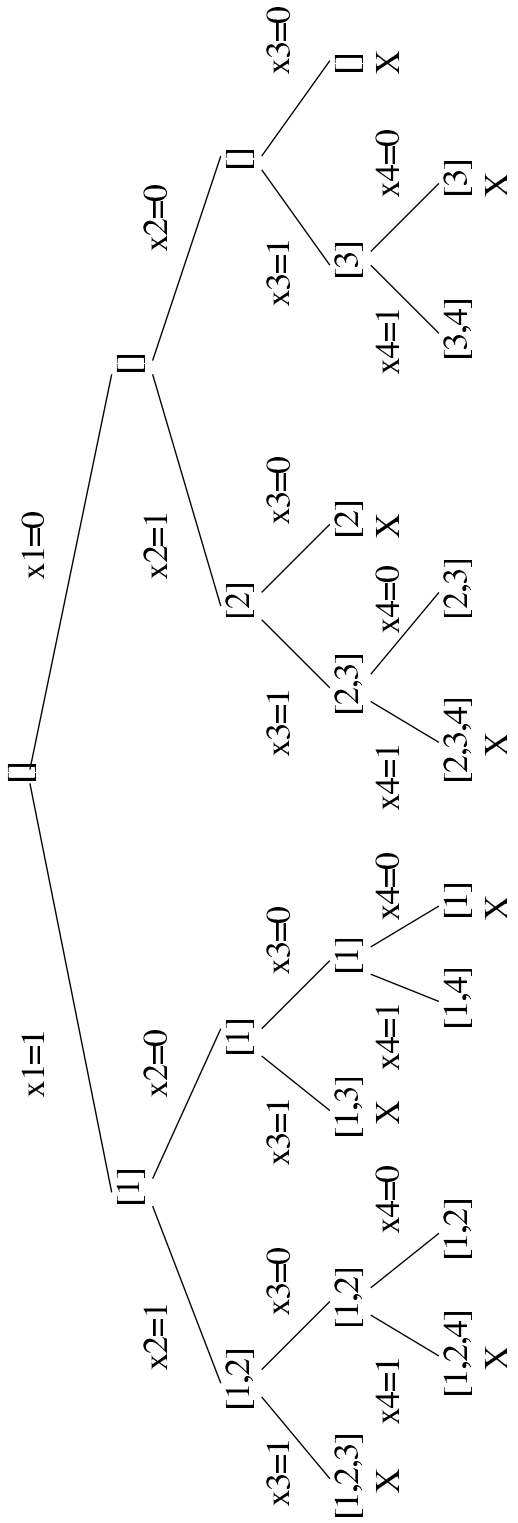

Figure 1.1: The standard branch-and-cut tree for the generation of all solutions to ILP1.

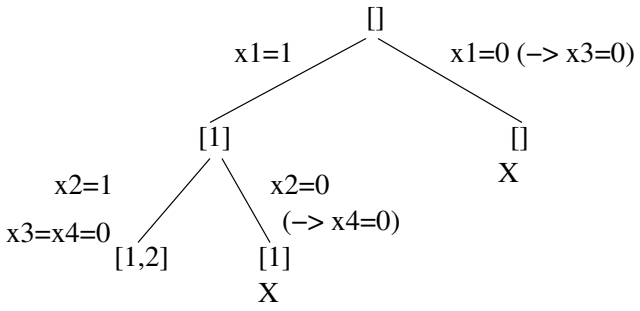

Figure 1.2: The isomorph-free branch-and-cut tree for the generation of all solutions to ILP1.

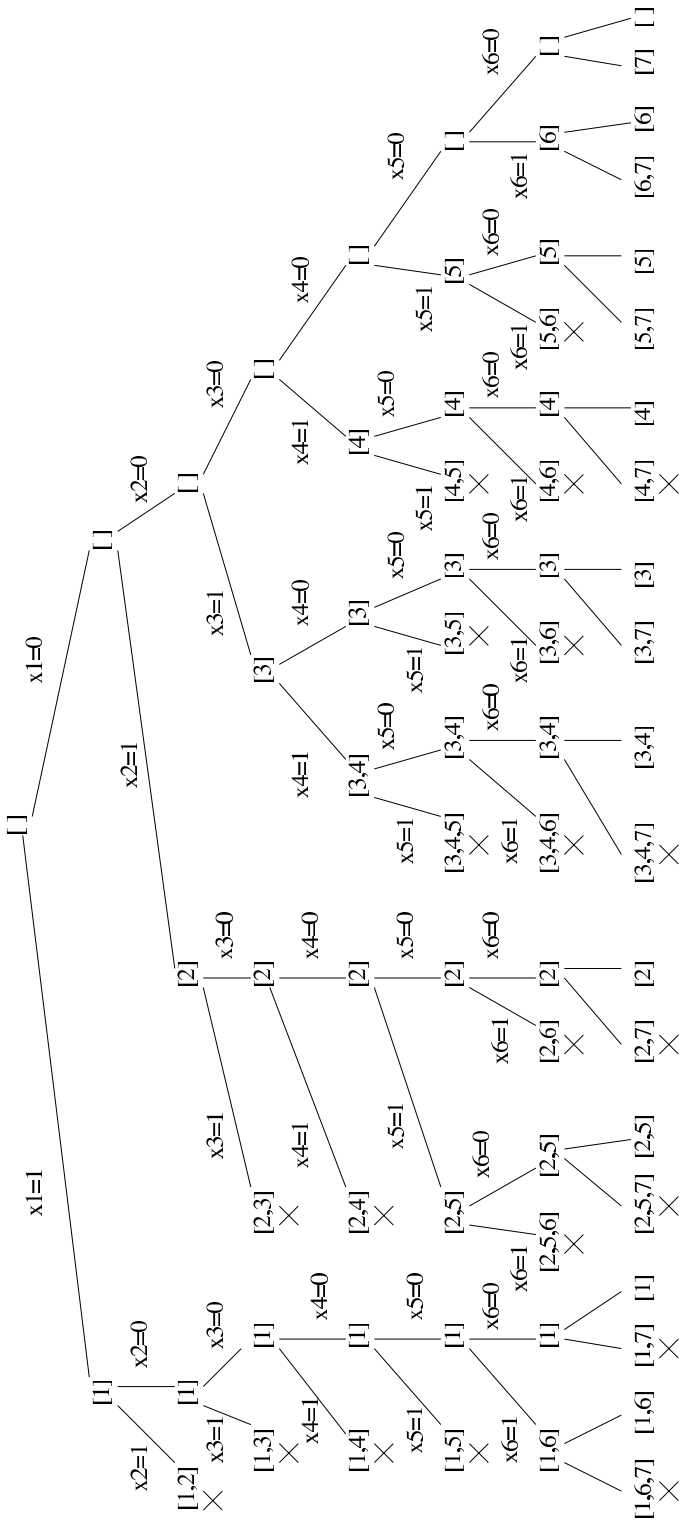

Figure 1.3: Part 1 of the standard branch-and-cut tree for the generation of all solutions to the 2-$(5, 3, 1)$ design problem.

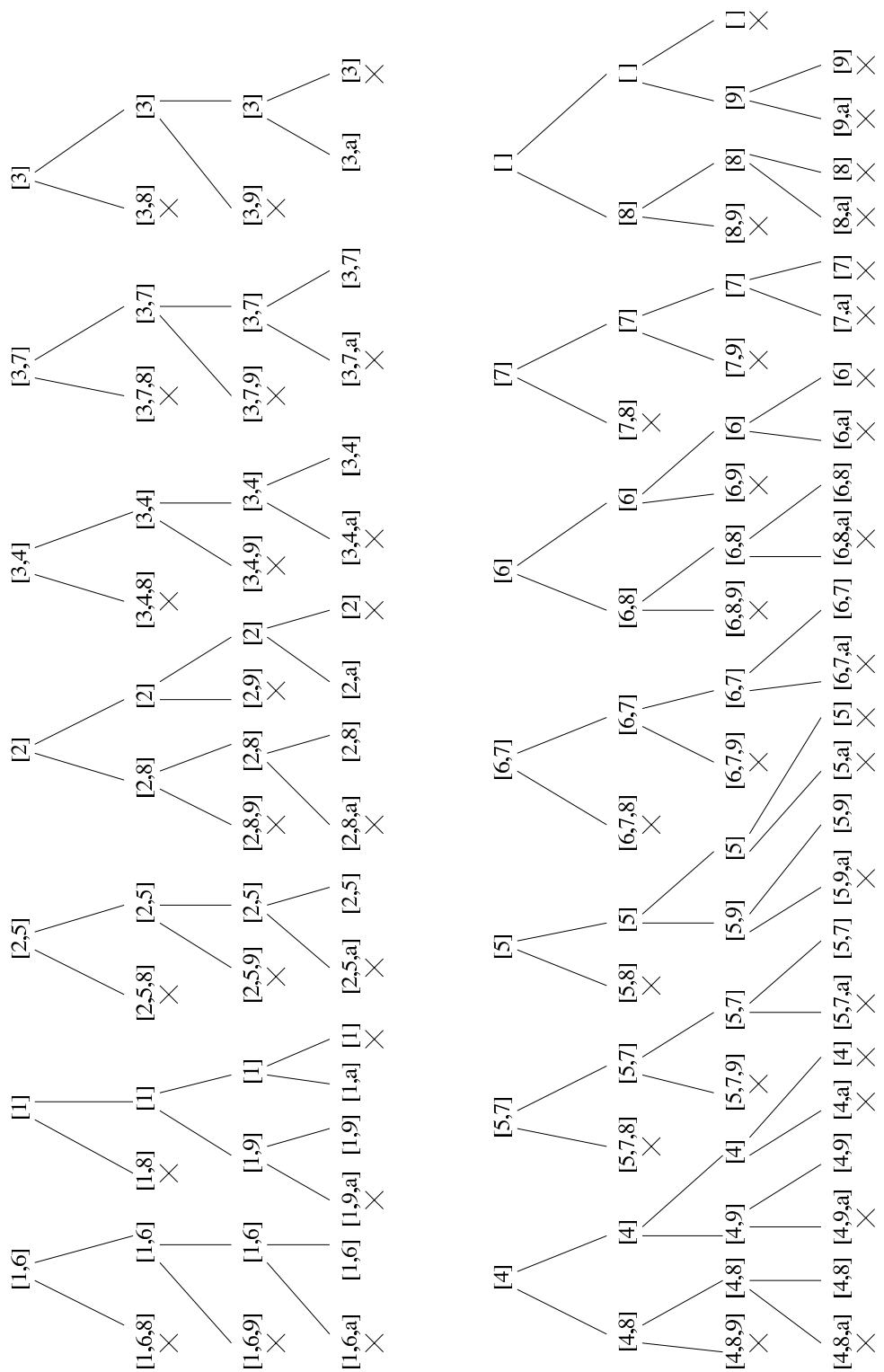

Figure 1.4: Part 2 of the standard branch-and-cut tree for the generation of all solutions to the 2-$(5, 3, 1)$ design problem.

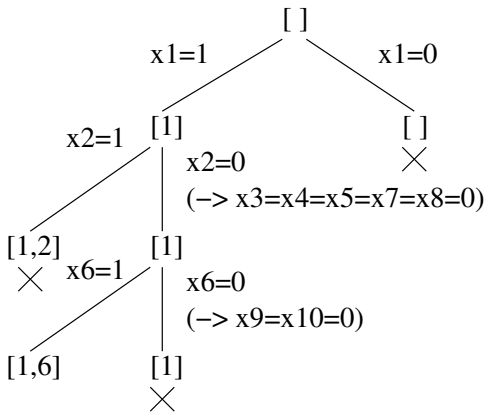

Figure 1.5: The isomorph-free branch-and-cut tree for the generation of all solutions to the 2-$(5, 3, 1)$ design problem.

Chapter 2

Combinatorial design problems

We now define the specific combinatorial problems in which we are interested, their formulation as integer linear programs, their respective symmetry groups, and their isomorphisms. More information regarding these structures may be found in [28, 33, 43, 44] and their integer programming formulations in [32, 35, 45].

2.1 t-(v, k, λ) designs, packings, and coverings

We are interested in studying the existence, generation, and enumeration of t-(v, k, λ) designs, packings, and coverings.

Let $\binom{V}{j}$, where V is a v-set, denote $\{Q \subseteq V \mid |Q| = j\}$.

Definition 2.1.1. A t-(v, k, λ) *design* is a pair $(V, \mathcal{B})$ in which V is a v-set and $\mathcal{B}$ is a multiset with entries from $\binom{V}{k}$ such that for all $T \in \binom{V}{t}$, there exists a maximal multiset $\{B_0, \ldots, B_{\lambda-1}\} \subseteq \mathcal{B}$ such that $T \subseteq B_i$ for all $i \in \mathbb{Z}_\lambda$. We refer to V as the set of points of the design, $\mathcal{B}$ as the set of blocks of the design, and λ as the index of the design.

Typically, we let $V = \mathbb{Z}_v$, so we can simply describe a t-(v, k, λ) design by its list of k-sets $\mathcal{B}$.

Example: Here we demonstrate a 2-$(7, 3, 1)$ design by listing its 3-sets, or triples. Note that, as per the above definition, every 2-set, or pair, appears in exactly one of the triples.

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

Theorem 2.1.1. *A t-(v, k, λ) design contains $\lambda\frac{\binom{v}{t}}{\binom{k}{t}}$ blocks.*

Proof. There are $\binom{v}{t}$ t-sets that must appear, each occurring in exactly λ blocks. Each block contains $\binom{k}{t}$ distinct t-sets. Hence to cover all $\lambda\binom{v}{t}$ t-sets, we require $\lambda\frac{\binom{v}{t}}{\binom{k}{t}}$ blocks. □

Definition 2.1.2. A t-(v, k, λ) *packing (resp. covering)* is a pair $(V, \mathcal{B})$ in which V is a v-set and $\mathcal{B}$ is a multiset with entries from $\binom{V}{k}$ such that for all $T \in \binom{V}{t}$, there exists a maximal (resp. minimal) multiset $\{B_0, \ldots, B_{j-1}\} \subseteq \mathcal{B}$ with $j \leq \lambda$ (resp. $j \geq \lambda$) such that $T \subseteq B_i$ for all $i \in \mathbb{Z}_j$. For fixed t, v, k, λ, we are often interested in packings (resp. coverings) with a maximum (resp. minimum) number of blocks: when we refer to packings and coverings henceforth, unless otherwise explicitly stated, it is assumed that this is what we mean. We call the quantity $|\mathcal{B}|$ the *packing number* (resp. *covering number*).

Example: The following is a 2-$(6, 3, 1)$ packing. Some 2-sets appear once (e.g. $\{1, 5\}$) and some do not appear at all (e.g. $\{0, 5\}$). It is easy to see that this packing is maximal with respect to set inclusion.

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

Example: The following is a 3-$(6, 4, 2)$ covering. Note that certain 3-sets appear more than twice (e.g. $\{0, 1, 2\}$), but every 3-set appears in at least two 4-sets.

$$\begin{array}{llll}\{\{0,1,2,3\}, & \{0,1,2,4\}, & \{0,1,2,5\}, & \{0,1,3,4\},\\ \{0,1,3,5\}, & \{0,2,3,4\}, & \{0,2,4,5\}, & \{0,3,4,5\},\\ \{1,2,3,5\}, & \{1,2,4,5\}, & \{1,3,4,5\}, & \{2,3,4,5\}\}\end{array}$$

There are two bounds, presented in Theorems 2.1.2 and 2.1.3, that we will use in our branch-and-cut.

Theorem 2.1.2 (First Johnson bound). *(Theorem 33.5 of [44].) For a t-(v,k,λ) packing, the packing number is bounded above by:*

$$\left\lfloor \frac{v}{k} \left\lfloor \frac{v-1}{k-1} \cdots \left\lfloor \frac{\lambda(v-t+1)}{k-t+1} \right\rfloor \right\rfloor \right\rfloor .$$

Theorem 2.1.3 (Schönheim bound). *(Theorem 8.6 of [43].) For a t-(v,k,λ) covering, the covering number is bounded below by:*

$$\left\lceil \frac{v}{k} \left\lceil \frac{v-1}{k-1} \cdots \left\lceil \frac{\lambda(v-t+1)}{k-t+1} \right\rceil \right\rceil \right\rceil .$$

2.1.1 Block incidence ILP model for designs

For any t-(v,k,λ) design, packing, or covering $(\mathbb{Z}_v, \mathcal{B})$, $\mathcal{B}$ is a multiset with elements from $\binom{V}{k}$. We represent $D = (\mathbb{Z}_v, \mathcal{B})$ by a $\binom{v}{k}\lambda$ vector X indexed by a pair (B,i) in $\binom{V}{k} \times \mathbb{Z}_\lambda$ such that:

$$x_{(B,c)} = \begin{cases} 1 & \text{if the } c\text{-th copy of } B \text{ appears in } \mathcal{B}, \\ 0 & \text{otherwise.} \end{cases}$$

In this way, we have λ 0-1 variables corresponding to each of the possible blocks. This allows a single block to occur up to λ times in a design.

In the case of packings (coverings), we want to maximize (minimize) the number of blocks in our design, so our objective function becomes:

$$\text{Maximize (Minimize)} \sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda} x_{(B,c)}.$$

If we are considering designs, the parameters uniquely determine the number of blocks in our design (namely $\lambda\frac{\binom{v}{t}}{\binom{k}{t}}$) and hence it is a problem of feasibility and the objective function is irrelevant.

The number of occurrences of a t-set T in blocks of the design can be counted by the expression:

$$\sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda : T\subseteq B} x_{(B,c)}.$$

This gives us the following formulation for t-(v,k,λ) designs:

Find a feasible solution for

$$\begin{aligned}
\sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda : T\subseteq B} x_{(B,c)} &= \lambda, && T \in \tbinom{V}{t},\\
x_{(B,c)} &\geq x_{(B,d)}, && B \in \tbinom{V}{k}, c < d, c, d \in \mathbb{Z}_\lambda,\\
x_{(B,c)} &\in \{0,1\}, && (B,c) \in \tbinom{V}{k} \times \mathbb{Z}_\lambda.
\end{aligned}$$

We obtain this formulation for t-(v,k,λ) packings:

$$\begin{aligned}
\text{Maximize} \quad & \sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda} x_{(B,c)} &&\\
\text{subject to} \quad & \sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda : T\subseteq B} x_{(B,c)} \leq \lambda, && T \in \tbinom{V}{t},\\
& x_{(B,c)} \geq x_{(B,d)}, && B \in \tbinom{V}{k}, c < d, c, d \in \mathbb{Z}_\lambda,\\
& x_{(B,c)} \in \{0,1\}, && (B,c) \in \tbinom{V}{k} \times \mathbb{Z}_\lambda.
\end{aligned}$$

For t-(v,k,λ) coverings, we have:

$$\begin{aligned}
\text{Minimize} \quad & \sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda} x_{(B,c)} &&\\
\text{subject to} \quad & \sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda : T\subseteq B} x_{(B,c)} \geq \lambda, && T \in \tbinom{V}{t},\\
& x_{(B,c)} \geq x_{(B,d)}, && B \in \tbinom{V}{k}, c < d, c, d \in \mathbb{Z}_\lambda,\\
& x_{(B,c)} \in \{0,1\}, && (B,c) \in \tbinom{V}{k} \times \mathbb{Z}_\lambda.
\end{aligned}$$

Example: The ILP for a 2-$(5, 3, 2)$ packing is:

$$
\begin{aligned}
\text{Maximize} \quad & \sum_{(B,c)\in\binom{V}{k}\times\mathbb{Z}_\lambda} x_{(B,c)} \\
\text{subject to} \quad & x_{012,0} + x_{012,1} + x_{013,0} + x_{013,1} + x_{014,0} + x_{014,1} \leq 2 \\
& x_{012,0} + x_{012,1} + x_{023,0} + x_{023,1} + x_{024,0} + x_{024,1} \leq 2 \\
& x_{013,0} + x_{013,1} + x_{023,0} + x_{023,1} + x_{034,0} + x_{034,1} \leq 2 \\
& x_{014,0} + x_{014,1} + x_{024,0} + x_{024,1} + x_{034,0} + x_{034,1} \leq 2 \\
& x_{012,0} + x_{012,1} + x_{123,0} + x_{123,1} + x_{124,0} + x_{124,1} \leq 2 \\
& x_{013,0} + x_{013,1} + x_{123,0} + x_{123,1} + x_{134,0} + x_{134,1} \leq 2 \\
& x_{014,0} + x_{014,1} + x_{124,0} + x_{124,1} + x_{134,0} + x_{134,1} \leq 2 \\
& x_{023,0} + x_{023,1} + x_{123,0} + x_{123,1} + x_{234,0} + x_{234,1} \leq 2 \\
& x_{024,0} + x_{024,1} + x_{124,0} + x_{124,1} + x_{234,0} + x_{234,1} \leq 2 \\
& x_{034,0} + x_{034,1} + x_{134,0} + x_{134,1} + x_{234,0} + x_{234,1} \leq 2 \\
& x_{012,0} \leq x_{012,1} \\
& x_{013,0} \leq x_{013,1} \\
& \vdots \quad \vdots \quad \vdots \\
& x_{234,0} \leq x_{234,1} \\
& x_{(B,c)} \in \{0,1\}, \qquad (B,c) \in \tbinom{V}{k} \times \mathbb{Z}_\lambda \qquad .
\end{aligned}
$$

We now investigate the structure of the symmetry group (say G) of this particular formulation. We observe that symmetries over blocks consist of point relabelings, which are simply all the permutations in S_v. For each $g \in S_v$, we have that, for $(B, c) \in \binom{V}{k} \times \mathbb{Z}_\lambda$, $g(x_{(B,c)}) = x_{(g(B),c)}$. Using this, we can construct an isomorphism such that $S_v \cong G' \leq G$. It turns out that these are the only symmetries of our ILP, so $G \cong S_v$. We thus have, by Theorem 1.3.1, that $G \cong \langle (0\ 1), (0\ 2), \ldots, (0\ (v-1)) \rangle$.

Example: We present the symmetry group for the 2-$(5, 3, 2)$ ILP given above by

examining the effects of the generators of S_5 on the blocks:

$$\begin{aligned}
(0\ 1) &\mapsto (x_{023,i}\ x_{123,i})(x_{024,i}\ x_{124,i})(x_{034,i}\ x_{134,i}), \qquad i \in \mathbb{Z}_\lambda \\
(0\ 2) &\mapsto (x_{013,i}\ x_{123,i})(x_{014,i}\ x_{124,i})(x_{034,i}\ x_{234,i}), \qquad i \in \mathbb{Z}_\lambda \\
(0\ 3) &\mapsto (x_{012,i}\ x_{123,i})(x_{014,i}\ x_{134,i})(x_{024,i}\ x_{234,i}), \qquad i \in \mathbb{Z}_\lambda \\
(0\ 4) &\mapsto (x_{012,i}\ x_{124,i})(x_{013,i}\ x_{134,i})(x_{023,i}\ x_{234,i}), \qquad i \in \mathbb{Z}_\lambda.
\end{aligned}$$

These 4 permutations generate G, which has size $v! = 120$.

2.1.2 Incidence matrix ILP model for designs

Here we investigate an ILP for designs based on incidence matrices. We limit our examination to a specific case, namely 2-(v, k, λ) designs. This ILP formulation is presented in more detail in [45], with additional notes in [35].

Definition 2.1.3. For a 2-(v, k, λ) design, the *incidence matrix* representing the design is a $v \times b$ (where $b = \lambda\frac{v(v-1)}{k(k-1)}$) matrix with entries from $\{0, 1\}$, such that row i represents point $i \in \mathbb{Z}_v$ and column j represents B_j, the jth block in $\mathcal{B}$. If we let $x_{i,j}$ represent the value in row i, column j of our matrix, we have that:

$$x_{i,j} = \begin{cases} 1 & \text{if } i \in B_j \\ 0 & \text{otherwise.} \end{cases}$$

Example: If we have the following 2-$(7, 3, 1)$ design:

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

we obtain the following incidence matrix representing the design, where the rows rep-

resent the point configurations and the columns represent the blocks:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

For a 2-(v, k, λ) design, which is represented by v rows and $b = \lambda\frac{v(v-1)}{k(k-1)}$ columns, we create $v \cdot b$ variables $x_{i,l}$, with $i \in \mathbb{Z}_v$, $l \in \mathbb{Z}_b$. As this is a feasibility problem, we do not require an objective function. Given that each column represents a block and each block is a k-set of $\mathbb{Z}_v$, we obtain the following b equations, one per block:

$$\sum_{i \in \mathbb{Z}_v} x_{i,l} = k, \qquad l \in \mathbb{Z}_b$$

For a block l, pair $\{i, j\}$ occurs in block l if and only if $x_{i,l}$ and $x_{j,l}$ are both 1, i.e. $x_{i,l}x_{j,l} = 1$. Hence, to force each pair $\{i, j\}$ to appear exactly λ times, we have the following equations, one for each pair:

$$\sum_{l \in \mathbb{Z}_b} x_{i,l}x_{j,l} = \lambda, \qquad \{i, j\} \subset \mathbb{Z}_v$$

Since these equations are non-linear, we need to linearize them in order to include them in the ILP. We do so by introducing additional variables: for each block $B_l \in \mathcal{B}$ and each pair $\{i, j\} \subset \mathbb{Z}_v$, we create a variable $y^l_{i,j}$, where:

$$y^l_{i,j} = \begin{cases} 1 & \text{if } \{i, j\} \in B_l, \\ 0 & \text{otherwise.} \end{cases}$$

Again, this is equivalent to saying that $y^l_{i,j} = 1$ if and only if $x_{i,l} = 1$ and $x_{j,l} = 1$, so

we derive three constraints to ensure that this is the case:

$$\begin{aligned} -x_{i,l} + y^l_{i,j} &\leq 0 \\ -x_{j,l} + y^l_{i,j} &\leq 0 \\ x_{i,l} + x_{j,l} - y^l_{i,j} &\leq 1 \end{aligned}$$

We can then remove the non-linear pair constraints and replace them with:

$$\sum_{l \in \mathbb{Z}_b} y^l_{i,j} = \lambda, \qquad \{i,j\} \subset \mathbb{Z}_v$$

Thus, our incidence matrix ILP formulation for 2-(v,k,λ) designs is as follows:

$$\begin{aligned} \text{Maximize} \quad & \textstyle\sum_{i \in \mathbb{Z}_v} \sum_{l \in \mathbb{Z}_b} x_{i,l} \\ \text{subject to} \quad & \textstyle\sum_{i \in \mathbb{Z}_v} x_{i,l} = k, && l \in \mathbb{Z}_b, \\ & \textstyle\sum_{l \in \mathbb{Z}_b} y^l_{i,j} = \lambda, && \{i,j\} \subset \mathbb{Z}_v, \\ & -x_{i,l} + y^l_{i,j} \leq 0, && l \in \mathbb{Z}_b, \{i,j\} \subset \mathbb{Z}_v, \\ & -x_{j,l} + y^l_{i,j} \leq 0, && l \in \mathbb{Z}_b, \{i,j\} \subset \mathbb{Z}_v, \\ & x_{i,l} + x_{j,l} - y^l_{i,j} \leq 1, && l \in \mathbb{Z}_b, \{i,j\} \subset \mathbb{Z}_v, \\ & x_{i,l} \in \{0,1\}, && l \in \mathbb{Z}_b, i \in \mathbb{Z}_v, \\ & y^l_{i,j} \in \{0,1\}, && l \in \mathbb{Z}_b, \{i,j\} \subset \mathbb{Z}_v. \end{aligned}$$

We do not provide an example for this formulation; the smallest design of interest is 2-$(7,3,1)$, which has 196 variables, making it infeasible to present here in its entirety.

We now investigate the symmetry group G of this formulation. We can derive two subgroups of G: G_r, the row permutations of G, and G_c, the column permutations of G. We then claim that $G = G_r \odot G_c$, the direct composition of G_r and G_c. Hence, we determine minimal sets of generators for each of these subgroups.

We first consider G_r. This subgroup consists of all row permutations of the matrix; as the rows represent the points of our design, each row permutation is equivalent to a point permutation, and hence G_r is isomorphic to S_v. Thus, as from Theorem 1.3.1, we know that $S_v = \langle (0\ 1), (0\ 2), \ldots, (0\ v-1) \rangle$, we need simply examine the corresponding

row transpositions. For a generator $(0\ i)$ of S_v, we swap rows 0 and i of our incidence matrix, resulting in the following permutation over the x variables:

$$(x_{0,0}\ x_{i,0})(x_{0,1}\ x_{i,1})\dots(x_{0,v-1}\ x_{i,v-1})$$

As the y variables are simply secondary variables and their values are imposed by the values of the x variables, we do not need to consider the action of our permutations upon them. We will never consider these variables in the branch-and-cut or in solutions, so they are unimportant.

We now turn our attention to G_c, the subgroup of column permutations. These are simply reorderings on the blocks of our design, and hence can be viewed as a subgroup isomorphic to S_b. From Theorem 1.3.1, we know that $S_b = \langle (0\ 1)\dots(0\ b-1)\rangle$, and hence, for each generator $(0\ l)$ of S_b, it suffices to find the corresponding generator of G_c. We first investigate the action of $(0\ l)$ on the x variables of our ILP; this simply involves swapping columns 0 and l, which gives us the following composition of transpositions:

$$(x_{0,0}\ x_{0,l})(x_{1,0}\ x_{1,l})\dots(x_{v-1,0}\ x_{v-1,l})$$

Again, as in the case of G_r, we do not need to consider the action of our permutations upon the y variables and can simply ignore them.

Thus, as $G = G_r \odot G_c$, we have that our group G has as a minimal set of generators the above mentioned permutations and may be represented by $(v-1)+(b-1) = v+b-2$ permutations instead of explicitly listing all $v!b!$ permutations in the group. We note that for 2-$(7,3,1)$ designs, $|G| = 7!7! = 25401600$, whereas our minimal set of generators is of size 12.

It is possible to add constraints to the ILP that force the blocks of the design to appear only in lexicographically increasing order. Instead of translating each block directly into lexicographical number, which would have added a large number of constraints to the problem, we assigned a value to each block. We begin by defining a

function:

$$\begin{array}{rcl} \phi : \binom{\mathbb{Z}_v}{k} & \to & \mathbb{Z} \\ \{v_0, \ldots, v_{k-1}\} & \mapsto & 2^{v-v_0-1} + \ldots + 2^{v-v_{k-1}-1} \end{array}.$$

It is then clear that, for two blocks $\mathcal{B}_1$ and $\mathcal{B}_2$, $\mathcal{B}_1 < \mathcal{B}_2$ if and only if $\phi(\mathcal{B}_1) < \phi(\mathcal{B}_2)$. On 32 bit architectures, this inherently limits us in that $v < 32$, but this poses no constraints for the range of values that we are able to test.

In order to ensure the ordering on the blocks, we can thus add the following family of $v - 1$ constraints to the design:

$$\sum_{i \in \mathbb{Z}_v} 2^{v-i-1} x_{i,l} - \sum_{j \in \mathbb{Z}_v} 2^{v-j-1} x_{j,l-1} \geq 0, \qquad l \in \mathbb{Z}_{\frac{v(v-1)}{k(k-1)}} \setminus \{0\}.$$

This is equivalent to evaluating ϕ on the block in column l and ensuring that it is lexicographically greater than the block in column $l - 1$. This has the advantage of eliminating the column permutations from the symmetry group, thus shrinking its size from $v!b!$ to $v!$; however, experimentally, this was shown to be detrimental to execution time (Section 8.3.2), so we omit these constraints by default.

2.2 (v, k, t) intersecting set systems

Another family of combinatorial problems that is difficult to solve in practice without taking isomorphism into account is the problem of intersecting set systems. These have been investigated in [33].

Definition 2.2.1. A (v, k, t) *intersecting set system* is a pair $(V, \mathcal{B})$ where V is a v-set and $\mathcal{B} \subseteq \binom{V}{k}$ is a set such that for all $B_1, B_2 \in \mathcal{B}$, $|B_1 \cap B_2| \geq t$. We call V the base set of the intersecting set system, and $\mathcal{B}$ the intersecting sets. We are interested in maximal intersecting set systems.

As with designs, packings, and coverings, we may take $V = \mathbb{Z}_v$. In this case, we may describe an intersecting set system simply by listing the members of $\mathcal{B}$.

Example: Two distinct $(6, 4, 3)$ intersecting set systems:

$$\begin{aligned} I_1 &= \{\{0,1,2,3\},\{0,1,2,4\},\{0,1,2,5\}\} \\ I_2 &= \{\{0,1,2,3\},\{0,1,2,4\},\{0,1,3,4\},\{0,2,3,4\},\{1,2,3,4\}\} \end{aligned}$$

We proceed to investigate an ILP formulation for (v, k, t) intersecting set systems. We represent $I = (\mathbb{Z}_v, \mathcal{B})$ by a $\binom{v}{k}$ vector x indexed by $B \in \binom{V}{k}$ such that:

$$x_B = \begin{cases} 1 & \text{if } B \in \mathcal{B}, \\ 0 & \text{otherwise.} \end{cases}$$

As we are interested in finding maximal intersecting set systems, our objective function becomes:

$$\text{Maximize} \sum_{B \in \binom{V}{k}} x_B$$

Let $I = (\mathbb{Z}_v, \mathcal{B})$ be an intersecting set system. By definition, for $B_i, B_j \in \binom{V}{k}$, if we have that $|B_i \cap B_j| < t$, then only one of $\{B_i, B_j\}$ can appear in $\mathcal{B}$, which leads us to the following constraint:

$$x_{B_i} + x_{B_j} \leq 1.$$

Hence, the ILP formulation for a (v, k, t) intersecting set system is:

$$\begin{aligned} \text{Maximize} \quad & \sum_{B \in \binom{\mathbb{Z}_v}{k}} x_B & & \\ \text{subject to} \quad & x_{B_i} + x_{B_j} \leq 1, & & \{B_i, B_j\} \subseteq \tbinom{\mathbb{Z}_v}{k}, |B_i \cap B_j| < t, \\ & x_B \in \{0,1\}, & & B \in \tbinom{\mathbb{Z}_v}{k}. \end{aligned}$$

Example: Here is the ILP for $(5, 3, 2)$ intersecting set systems:

$$\begin{aligned}
\text{Maximize} \quad & \sum_{i \in \mathbb{Z}_{10}} x_i \\
\text{subject to} \quad & x_{012} + x_{034} \leq 1 \\
& x_{012} + x_{134} \leq 1 \\
& x_{012} + x_{234} \leq 1 \\
& x_{013} + x_{124} \leq 1 \\
& x_{013} + x_{234} \leq 1 \\
& x_{014} + x_{023} \leq 1 \\
& x_{014} + x_{123} \leq 1 \\
& x_{014} + x_{234} \leq 1 \\
& x_{023} + x_{124} \leq 1 \\
& x_{023} + x_{134} \leq 1 \\
& x_{024} + x_{123} \leq 1 \\
& x_{024} + x_{134} \leq 1 \\
& x_{034} + x_{123} \leq 1 \\
& x_{034} + x_{124} \leq 1 \\
& x_B \in \{0, 1\}, \qquad B \in \tbinom{\mathbb{Z}_v}{k}.
\end{aligned}$$

The symmetry group for a (v, k, t) intersecting set system is the same as for the block incidence formulation for designs: namely a group isomorphic to S_v.

2.3 Isomorphism, canonical structures, and variable fixings

Packings, coverings, designs, and intersecting set systems clearly have some commonalities. This will allow us to generalize and define the concept of isomorphism over these problems.

Definition 2.3.1. Let V be a v-set, and $\mathcal{B} \subseteq \mathcal{P}(V)$. We call $(V, \mathcal{B})$ a *set system*.

Definition 2.3.2. Two set systems $D_1 = (\mathbb{Z}_v, \mathcal{B}_1)$ and $D_2 = (\mathbb{Z}_v, \mathcal{B}_2)$ are *isomorphic* if there exists a permutation $g \in S_v$ such that $g(\mathcal{B}_1) = \mathcal{B}_2$. This is denoted $D_1 \cong D_2$. An *automorphism* of a set system $D = (\mathbb{Z}_v, \mathcal{B})$ is an isomorphism g from D to itself, that is, $g(\mathcal{B}) = \mathcal{B}$. As S_v is a group, isomorphism is an equivalence relation. We refer to the equivalence classes of an isomorphism as *isomorphism classes*.

Using the standard subset lexicographical ordering on subsets of a base set $\mathbb{Z}_n$, an ordering on set systems is well-defined. Let $D_1 = (\mathbb{Z}_n, \mathcal{B}_1)$ and $D_2 = (\mathbb{Z}_n, \mathcal{B}_2)$ be two distinct structures with $\mathcal{B}_1 = \{B_{1,0}, B_{1,1}, \ldots, B_{1,m}\}$ and $\mathcal{B}_2 = \{B_{2,0}, \ldots, B_{2,l}\}$, indexed by lexicographical ordering on subsets. We say that D_1 is lexicographically smaller than D_2, denoted $D_1 < D_2$, if there exists $i \in \mathbb{Z}_{m+1}$ such that $B_{1,j} = B_{2,j}$ for all $0 \leq i$ and $B_{1,i} < B_{2,i}$, or if $l > m$ and $B_{1,i} = B_{2,i}$ for all $0 \leq i \leq m$. If a set system D_1 is either lexicographically smaller than or equal to another set system D_2, we write $D_1 \leq D_2$.

Definition 2.3.3. Given an isomorphism class I, if $D \in I$ has the property that $D \leq D'$ for all $D' \in I$, we say that D is *canonical* (or the *canonical representative* of I).

It is then our goal to generate only canonical representatives.

We now determine variables of the ILPs that can be fixed without loss of generality due to our choice of canonical structures. First, since in all problems considered, the symmetry groups act transitively on the variables, we can always select the lexicographically smallest k-set, namely $\{0, 1, \ldots, k-1\}$. In the case of t-$(v, k, 1)$ designs, we can select the k-sets passing through $\{0, \ldots, t-2\}$ as the lexicographically smallest partial design with k-sets containing $\{0, \ldots, t-1\}$ and all other elements distinct, namely:

$$\{B_0, B_1, \ldots, B_{\frac{v-(t-1)}{k-(t-1)}}\} \text{ with } B_i = \{0, 1, \ldots, t-2, ik-(i-1)(t-1), \ldots, (i+1)k-it-1\}.$$

Furthermore, we can select the smallest k-set not repeating an already covered t-set, namely $\{0, 1, \ldots, t-3, t-1, k, \ldots\}$.

Example: In the 2-$(7, 3, 1)$ design formulation we know that any optimal canonical solution contains the blocks $\{0, 1, 2\}$, $\{0, 3, 4\}$, $\{0, 5, 6\}$, and $\{1, 3, 5\}$.

Chapter 3

ILPs and branch-and-cut

In this chapter, we examine three different techniques that can be used to solve general ILPs: the branch-and-bound, the cutting plane, and the branch-and-cut algorithms.

3.1 Branch-and-Bound Technique

The idea behind the branch-and-bound technique is that, given an ILP, we will implicitly examine all possible integer solutions and find one that is an optimal solution.

Branch-and-bound employs a search tree to examine the solution space of the ILP. An important aspect of this method involves "bounding", which, for a maximization problem, requires finding an upper bound on a subproblem. Then, we can prune a branch of the search tree whenever the upper bound on this subproblem is small enough to certify that no optimal solution can be found there. A useful upper bound is given by the LP relaxation.

Definition 3.1.1. Given an ILP, we may remove the integrality constraints. The resultant LP is called the *LP relaxation* of the original ILP. For a 0-1 ILP, we substitute the integrality constraints $x_i \in \{0, 1\}$ by $0 \leq x_i \leq 1$.

Theorem 3.1.1. *Given a maximization ILP, the optimal solution of the LP relaxation provides an upper bound on the solution for the ILP.*

Proof. Clearly, as all we have done is remove integrality constraints, if x is a solution to our ILP, then x is a solution to the LP relaxation. Hence, the LP relaxation contains all the solutions of our original ILP, and may contain real-valued solutions with larger objective value. □

A general branch-and-bound algorithm is given in `branch-and-bound` (Algorithm 3.1.1). We begin with a root node which represents the original ILP, and a variable, say `opt`, which stores the value of the best solution found. The term *fathoming* is synonymous with the idea of pruning, or removing nodes from our tree without exploring their associated subtree. This algorithm is presented for maximization problems, but may be easily altered to solve minimization problems.

There are many possible variations of this algorithm; for instance, we may explore the tree using a depth-first, breadth-first, or best-first approach. These all present certain advantages and disadvantages. We will use depth-first search in the algorithms of Chapter 4, since it is required by some of the algorithms that eliminate isomorphs.

An important concept to note about this algorithm is that, given a node n, the ILPs represented by the child nodes of n are simply subproblems of the ILP represented by n. Hence, the solution to the LP relaxation of n serves as an upper bound for the solutions in the search space of the subtree rooted at n. The consequences of this are twofold: first, if the upper bound at n is less than the best solution found so far, we know that we will not find a better solution in the subtree rooted at n, so we may simply fathom this node; and secondly, if the solution of the LP relaxation at n gives us an integer solution, then this integer solution is optimal for the problem represented by n, so for search problems, we may also fathom.

The branch-and-bound technique will always solve an ILP and provide an optimal solution. However, the number of nodes in the search tree can become quite large

Algorithm 3.1.1 branch-and-bound(ILP $\mathcal{I}$)

```
opt = −∞, values = null
create tree T with root node representing I
while T has unexplored nodes do
  get the next unexplored node n of T and its depth i
  try to solve the LP relaxation for the ILP associated with n
  if the LP relaxation is infeasible then
    fathom node n and loop
  end if
  obtain the objective solution sol with variable value vector v
  if sol < opt then
    fathom node n and loop
  end if
  if v is integer then
    if sol > opt then
      opt = sol
      values = v
    end if
    fathom node n and loop
  end if
  branch on x_i to create subnodes n_0 (x_i = 0) and n_1 (x_i = 1)
end while
return (opt, values)
```

even for small problems, thus making them prohibitively difficult to solve using this technique.

3.2 Cutting Plane Technique

Another technique that may be used to solve an ILP is the *cutting plane technique*. We outline the general idea for this method here along with some of the geometry behind its use; more thorough treatments may be found in [13], [38], and [46].

Suppose that we have an ILP over n variables. We can geometrically think of the set of solutions to the ILP as a set of points in the space $\mathbb{R}^n$. As we will specifically be dealing with 0-1 problems, we can think of our solution space as a subset of $\{0,1\}^n$ and the space of our LP relaxation as a subset of $[0,1]^n$, but we examine the cutting plane method from a general viewpoint. We begin by investigating some definitions.

Definition 3.2.1. Let $S \subseteq \mathbb{R}^n$. Then $x \in \mathbb{R}^n$ is said to be a *convex combination* of the points in S if there exists $\{x_1, \ldots, x_t\} \subseteq S$, a finite subset, and $\lambda = (\lambda_1, \ldots, \lambda_t) \in (\mathbb{R}^+)^t$ such that:

$$x = \sum_{i=1}^{t} \lambda_i x_i$$

$$\text{and } \sum_{i=1}^{t} \lambda_i = 1.$$

The set of all convex combinations of the points in S is called the *convex hull of* S and is denoted $\text{conv}(S)$.

We provide, but do not prove the following theorem regarding convex hulls.

Theorem 3.2.1. *(From [38].) Let $S \subseteq \mathbb{R}^n$, $c \in \mathbb{R}^n$. Then:*

$$\max\{c \cdot x \mid x \in S\} = \max\{c \cdot x \mid x \in \text{conv}(S)\}.$$

Definition 3.2.2. A *polyhedron* $P \subseteq \mathbb{R}^n$ is a set of points that satisfy a finite system of linear inequalities:

$$P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$$

where A is an $m \times n$ matrix over $\mathbb{R}$. A polyhedron P is said to be bounded if there exists a $w \in \mathbb{R}^+$ such that:

$$P \subseteq \{x \in \mathbb{R}^n \mid -w \leq x_j \leq w, j = 1, \ldots, n\}.$$

A bounded polyhedron is called a *polytope*. For an example, see Figure 3.1.

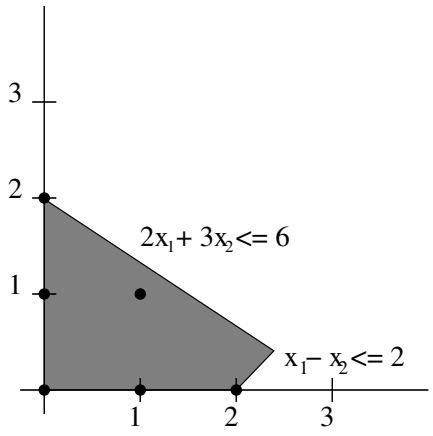

Figure 3.1: Example of a polytope

The above diagram shows the polytope for the LP relaxation of the following ILP:

$$\begin{aligned} 2x_1 + 3x_2 &\leq 6 \\ x_1 - x_2 &\leq 2 \\ x_1, x_2 &\geq 0 \\ x_1, x_2 &\in \mathbb{Z} \end{aligned}$$

The shaded area with the darkened outline represents the polytope P defined by the set of solutions to the LP relaxation. The points represent the family of integer solutions to the problem.

Definition 3.2.3. A set $\{x_1, \ldots, x_t\} \subseteq \mathbb{R}^n$ is said to be *affinely independent* if:

$$\sum_{i=1}^{t} \lambda_i x_i = 0 \text{ and } \sum_{i=1}^{t} \lambda_i = 0 \text{ implies } \lambda_i = 0.$$

The maximum number of affinely independent points for the polytope $P = \{x \in \mathbb{R}^n \mid Ax = b\}$ is $n + 1 - \text{rank}(A)$. We say that P *is of dimension* k if the maximum number of affinely independent points for P is $k + 1$, and we denote this $\dim(P) = k$.

Definition 3.2.4. An inequality $ax \leq b$ is said to be *valid* for a polyhedron P if all points in P satisfy it, i.e. $ax \leq b$ for all $x \in P$. If an inequality $ax \leq b$ is valid, then the set

$$F = \{x \in P \mid ax = b\}$$

is called a face of P. A face of P is called a *facet* if $dim(F) = dim(P) - 1$. An inequality that defines a facet is known as a *facet-inducing inequality.*

For each facet F of a polytope P, one of the inequalities defining F is necessary for the modeling of P. In addition, for full-dimensional polytopes, the facet inequalities are sufficient for describing P.

We now turn our attention specifically to ILPs. If $P = \{x \in (\mathbb{R}^+)^n \mid Ax \leq b\}$ is a polytope, and $S = P \cap \mathbb{Z}^n$, then either $S = \emptyset$ or S contains a finite number of points. By theorem 3.2.1, we can solve the integer program $\max\{c \cdot x \mid x \in S\}$ by focusing our attention on $\max\{c \cdot x \mid x \in \text{conv}(S)\}$. Hence, we want $\text{conv}(S)$, which will give us all feasible and possibly maximal solutions to our ILP described by S. To do this, it suffices to find the facets of $\text{conv}(S)$. This is the motivation behind the cutting plane technique, which iteratively tries to discover these facets by generating "cuts" (i.e. valid inequalities) that describe $\text{conv}(S)$. We do this using integrality and the linear inequalities of P themselves; we begin with a family of linear inequalities of the form $Ax \leq b$, and if they are not sufficiently strong to describe $\text{conv}(S)$, we refine them into stronger ones. There are many different types of cuts, both general and

problem-specific, that may be constructed. General ones include the Chvátal-Gomory cuts [13] and specific ones include cuts for packing problems which will be used in this thesis [22, 35, 36, 37, 38].

The algorithm `cutting-plane` (Algorithm 3.2.1) provides pseudocode detailing a typical cutting-plane method. The step in which we generate one or more cuts such that the nonintegral solution to our LP relaxation, x, is not valid is called the *separation phase*, as we separate x from the family of solutions to our ILP.

Algorithm 3.2.1 cutting-plane(ILP $\mathcal{I}$)

```
loop
  solve the LP relaxation of I to get solution x
  if x is integer then
    return x
  end if
  generate a valid inequality for I that is violated by x
end loop
```

While the cutting plane approach has fewer subproblems to deal with that a pure branch-and-bound technique, it has the disadvantage that after several iterations, it is typical for cuts to show little improvement towards obtaining an integer solution. In essence, we cut off progressively smaller and smaller pieces of the polyhedron of the LP relaxation. This phenomena is called *tailing off*. Depending on the choice of cuts and the severity of the tailing off, it is possible that we may never reach an integer solution using this technique.

3.3 Branch-and-Cut Technique

Due to the inherent limitations of each of the two above techniques, a third method called branch-and-cut (as coined by Padberg and Rinaldi in [40]) was created that

Algorithm 3.3.1 branch-and-cut(ILP $\mathcal{I}$)

```
opt = −∞, values = null
create tree T with root node representing I
while T has unexplored nodes do
  get the next unexplored node n of T and its depth i
  while criterion are satisfied to continue the cutting plane do
    solve the LP relaxation of the ILP of n
    if the LP relaxation is infeasible then
      break and loop on the next node
    end if
    obtain the objective value sol with variable value vector v
    if sol < opt then
      fathom node n and loop to process the next one
    end if
    if v is integer then
      if sol > opt then
        opt = sol
        values = v
      end if
      fathom node n, break, and loop to process the next node
    end if
    generate valid inequalities for the ILP that are violated by v
  end while
  branch on x_i to create subnodes n_0 (x_i = 0) and n_1 (x_i = 1)
end while
return (opt, values)
```

combines the strengths of both branch-and-bound with those of the cutting plane algorithm while guaranteeing that we will reach an integer solution.

The idea is that, given a node, we perform the cutting plane algorithm until some criterion is no longer satisfied: usually, we measure the quality of the cuts to determine if we are tailing off, or ascertain whether or not our algorithm is producing a sufficient number of cuts to make it worthwhile to continue. When the cutting plane algorithm seems to be contributing little towards integrality, we opt to branch as in branch-and-bound. We continue in this fashion until all relevant nodes in our search space have been processed. This method is shown in `branch-and-cut` (Algorithm 3.3.1).

For our implementation, we use a modified branch-and-cut tree where each node is tested for canonicity prior to processing as described in Chapter 4. The details of the cutting plane algorithm we use are given in `cuttingplane` (Algorithm 5.1.1) in Chapter 5.

Chapter 4

Group theory algorithms and isomorph-free branching trees

Many of the problems that we will investigate are impractical if we solve them in a standard branch-and-cut framework as the number of nodes is far too high to process; however, these problems often present a large number of "symmetries", and if we are not interested in equivalent solutions, we can simply ignore symmetric subproblems. This has the possibility of dramatically reducing the number of nodes that we need to consider.

A general review of group theory, along with definitions of permutation groups and symmetry groups was presented in Section 1.3. We now proceed to examine ways to store symmetry groups [21, 22], Margot's techniques to exploit these groups to produce isomorph-free branch-and-cut trees [22] and our extension of his algorithms in order to explore isomorph-reduced trees, and three methods we derived to determine the symmetry group of an arbitrary ILP.

4.1 The Schreier-Sims scheme and related algorithms

In this section, we discuss an efficient way to store a permutation group. The two most obvious ways to do this are to store either a list of all elements in the group, or to store a minimal set of generators for the group. Neither one of these techniques is optimal for our needs. In many problems of interest, the symmetry group is isomorphic, for some v, to S_v, which has size $v!$. Because of this, even for small v, a complete list of all permutations is inefficient to store. On the other extreme, Theorem 1.3.1 tells us that S_v can be generated by the $v-1$ permutations of the form $(0\ i)$, where $i \in \mathbb{Z}_v \setminus \{0\}$. Thus, using this technique would require very little memory, but generating all the $v!$ permutations of the group may require a considerable amount of effort and many compositions.

Instead of adopting either of these techniques, we use another intermediate approach specifically designed for permutation groups called the *Schreier-Sims scheme* (see [21, 22]). This scheme stores a non-minimal set of generators such that it becomes easy to construct any element in our group. Let G be a permutation group. We begin by defining the following subgroups of G:

$$\begin{aligned} G_{-1} &= G \\ G_0 &= \{g \in G_{-1} \mid g(0) = 0\} \\ G_1 &= \{g \in G_0 \mid g(1) = 1\} \\ G_2 &= \{g \in G_1 \mid g(2) = 2\} \\ \vdots & \qquad\qquad \vdots \end{aligned}$$

This is clearly a nested family of subgroups of G (in the sense that $G_n \leq G_{n-1} \leq \ldots \leq G_0 \leq G$). In addition, it is important to realize that G_i is the stabilizer of point i in the group G_{i-1}. Thus, G_i stabilizes $\mathbb{Z}_i$.

Definition 4.1.1. Let $G \leq S_n$ be the symmetry group of our problem. Let T be an $n \times n$ table with entries either in G or null (represented by 0) with $T_{i,j}$ representing the entry in the ith row and jth column of our table. For each $k \in \mathbb{Z}_n$, we calculate:

$$\text{orb}(k, G_{k-1}) = \{k_0, k_1, \ldots, k_{r-1}\}$$

which is the orbit, or images of k under the group G_{k-1}. Then, for each $i \in \mathbb{Z}_r$, let $g_{k,i} \in G_{k-1}$ be a permutation such that $g_{k,i}(k) = k_i$. Clearly, this is possible, as $k_i \in \text{orb}(k, G_{k-1})$. Define the entries of the table T as follows:

$$T_{k,j} = \begin{cases} 1 & \text{if } k = j, \\ g_{k,i} & \text{if } j = k_i \in \text{orb}(k, G_{k-1}), \\ 0 & \text{otherwise.} \end{cases}$$

The table T is called the *Schreier-Sims table*, or the *Schreier-Sims scheme* of G.

We note that the table T is not unique for a group G; it is possible that we could have chosen a different permutation from G_{k-1} for each $g_{k,i}$. However, the structure of the table (i.e. the positions of the non-null entries) will always be the same, and the table will always generate the same group, regardless of our choice of permutation.

Example: (From [22].) Let us consider all the symmetries of the 2×2 square shown below:

0	1
2	3

There are eight such symmetries or permutations, namely the identity I, the rotations $R_{90} = (0132)$, $R_{180} = (03)(12)$, and $R_{270} = (0231)$, the reflection in the vertical axis $V = (01)(23)$, the reflection in the horizontal axis $H = (02)(13)$, the reflection in the main diagonal $D_1 = (12)$, and the reflection in the other diagonal $D_2 = (03)$. Then, $G_{-1} = G$, $G_0 = \{I, D_1\}$, and $G_1 = G_2 = G_3 = \{I\}$. We determine that $\text{orb}(0, G_{-1}) = \{0, 1, 2, 3\}$, $\text{orb}(1, G_0) = \{1, 2\}$, $\text{orb}(2, G_1) = \{2\}$, and $\text{orb}(3, G_2) = \{3\}$. A possible Schreier-Sims table for this group is then:

	0	1	2	3
0	I	V	H	R_{180}
1		I	D_1	
2			I	
3				I

Theorem 4.1.1. *(From [21, 22].) Given a permutation group G, a permutation $g \in G$, and a Schreier-Sims table T representing G, we can find elements $g_0, g_1, \ldots, g_{n-1}$ in T (where g_i is an entry in row i of T) such that*

$$g = g_0 g_1 \ldots g_{n-1}.$$

Thus, the permutations in T generate the group G. We call such a set of generators a strong set of generators*, as every $g \in G$ can be expressed as exactly the product of n permutations of T.*

Proof. The general idea is presented in [21, 22], but we expand on the details here. Let $g \in G$ be a permutation. We give a constructive proof which determines the permutations g_i. First, pick g_0 to be the permutation in row 0 of T that maps 0 to $g(0)$, namely $T_{0,g(0)}$. By the property of the nested subgroups, all elements in row 1 of T stabilize 0, so we can seek out a permutation g_1 such that $g(1) = g_0 g_1(1)$, namely $T_{1,g_0^{-1}g(1)}$. We repeat this process, observing that $\mathbb{Z}_i$ is stabilized by all permutations in rows $i \leq j \leq n-1$ of T, so we can select permutation g_i such that $g(i) = g_0 g_1 \ldots g_i(i)$, namely $T_{i,g_{i-1}^{-1}g_{i-2}^{-1}\cdots g_0^{-1}g(i)}$. Then, it is clear that $g_0 g_1 \ldots g_{n-1} = g$. □

We can extend and generalize the definition of the Schreier-Sims scheme by imposing an arbitrary ordering on the elements of the set which we are permuting (i.e. $\mathbb{Z}_n$). We call such an ordering the *base* of the table, and we represent it by an n-vector β with the property that $\beta[0]$ is the first element in our set, $\beta[1]$ the second, etc. Then, given any $g \in G$, we choose permutations $g_0, g_1, \ldots, g_{n-1}$ such that $g_0(\beta[0]) = g(\beta[0])$, $g_0 g_1(\beta[1]) = g(\beta[1])$, etc. as opposed to our original construction.

The advantage to employing a base is that it greatly facilitates arbitrary orbit and stabilizer calculations, as will be seen later in the algorithms `orbit-in-stabilizer` (Algorithm 4.2.2), `first-in-orbit1` (Algorithm 4.2.4), and `first-in-orbit2` (Algorithm 4.2.6). The disadvantage to such an approach is that, when we adjust the base, we are required to rebuild, or at least re-order the table and re-enter certain permutations.

From this point forward, when we discuss Schreier-Sims schemes, we assume that we are discussing schemes that have a base imposed upon them.

The first two algorithms that we examine are used to build the table T itself. We begin by investigating the `test` procedure [21, 22] as shown in Algorithm 4.1.1, which, given a Schreier-Sims table T, a base β, a permutation p, and an index `first`, returns the first i with $\beta[i] \geq$ `first` such that row $\beta[i]$ of the table is modified by the addition of p to T. If such a row does not exist (i.e. the permutation p can already be constructed from T), then the algorithm returns n. We note that when adding permutations to the table T, the value of `first` should always be 0. However, when we modify the base of the table, we will need to remove and re-enter permutations, and we can use the parameter `first` to improve efficiency; if we know that p stabilizes the elements $\beta[0], \ldots, \beta[\texttt{first} - 1]$, then, certainly adding p to T will not modify the rows corresponding to $\beta[0], \ldots, \beta[\texttt{first} - 1]$, so we do not need to check these rows in the algorithm.

Theorem 4.1.2. *The algorithm* `test` *(Algorithm 4.1.1) is correct and returns the index i of the first row of our table that will be modified by the addition of p to our group and a permutation p' such that $p'(j) = j$ for all $j \in \mathbb{Z}_i$ and $G \cup \{p'\} = G \cup \{p\}$.*

Proof. We give our own proof here, while the correctness has been shown in [22]. The general idea behind this algorithm is to examine the Schreier-Sims table T to determine if the permutation p can be constructed by a composition of strong generators as detailed above. By construction, at the end of iteration j, we have found a permutation

Algorithm 4.1.1 test(T, β, p, `first`)

$p' = p$
for $i =$ `first` to $n - 1$ **do**
  $h = T_{\beta[i],p'(\beta[i])}$
  if $h \neq 0$ **then**
    $p' = h^{-1}p'$
  else
    return (i, p')
  end if
end for
return (n, p')

$f_j = h_j^{-1}h_{j-1}^{-1}\dots h_0^{-1}$ such that $f_j p(k) = k$ for all $k \in \mathbb{Z}_{j+1}$. The algorithm terminates when we find an i such that $f_{i-1}p(k) = k$ for all $k \in \mathbb{Z}_i$ but there is no permutation h such that h pointwise stabilizes $\mathbb{Z}_i$ and $h(i) = f_{i-1}p(i)$, and then returns i and $p' = f_{i-1}p$.

Clearly, $h_0h_1\dots h_{i-1}f_{i-1}p = p$, so by inserting $f_j p$ into row i, we can find generators of the following form that construct p:

$$p = h_0h_1\dots h_{i-1}(f_{i-1}p)\underbrace{1\dots 1}_{n-i-1}.$$

If p is already an element represented by our table, no such i exists, and hence, the algorithm returns n and $f_{n-1}p = 1$ to indicate that this is the case. □

The running time of the `test` algorithm is $O(n^2)$. This is because we iterate over the n rows of the table, and for each row, permutation inverse and composition requires n operations. Finding the permutation h can be done in constant time by the structure of the table.

The next algorithm that we examine is used to add a permutation to the strong set of generators representing our group G; it is hence called `enter` [21, 22], and the

general idea is outlined in Algorithm 4.1.2. Again, for a permutation p, if we know that p stabilizes pointwise the elements $\beta[0], \ldots, \beta[\texttt{first} - 1]$, we may pass in a value for the `first` parameter in order to reduce computation.

Algorithm 4.1.2 enter(T, β, p, `first`)

```
(i, p') = test(T, β, p, first)
if i = n then
  return
end if
T_{β[i],p'(β[i])} = p'
for j = first to i do
  for k = 0 to n − 1 do
    h = T_{β[j],k}
    if h ≠ 0 and h ≠ 1 then
      p'' = p'h
      enter(T, β, p'', first)
    end if
  end for
end for
for j = i to n − 1 do
  for k = 0 to n − 1 do
    h = T_{β[j],k}
    if h ≠ 0 and h ≠ 1 then
      p'' = hp'
      enter(T, β, p'', first)
    end if
  end for
end for
```

Theorem 4.1.3. *(From [22].) The algorithm* `enter` *(Algorithm 4.1.2) adds the permutation p to the group represented to the Schreier-Sims table T.*

Proof. We follow the proof in [22] and expand with more details. Let G be the group represented by T before performing `enter`. In order to show that `enter` works correctly, it suffices to show that if T' is the table following a call to `enter`, then G', the structure arising from T', is a group that is generated by G and p. It is obvious that $G \subseteq G'$ and clearly every element in G' is a composition of permutations from $G \cup \{p\}$. Hence, it suffices to show that G' is a group.

Let U'_i represent the i^{th} row of T'. Then, we write $U'_i U'_{i+1} \dots U'_j$ to indicate all the permutations that can be generated by composing elements from row i to row j. Using this notation, it is clear that:

$$G' = U'_0 U'_1 \dots U'_{n-1}$$

We proceed by a proof by induction on $n-i$. Clearly, $U'_{n-1} = \{1\}$ is a group. We will show inductively that if $A_i = U'_i \dots U'_{n-1}$ is a group, then so is $A_{i-1} = U'_{i-1} U'_i \dots U'_{n-1}$.

Let $x_1, x_2 \in A_{i-1}$. We can find elements $h_1, h_2 \in U'_{i-1}$ and $g_1, g_2 \in A_i$ such that:

$$\begin{aligned} x_1 &= h_1 \circ g_1, \\ x_2 &= h_2 \circ g_2. \end{aligned}$$

Obviously, $g_1 \circ h_2(j) = j$ for $j \in \mathbb{Z}_{i-1}$, as the elements of A_i fix all $j < i$, and the elements of U'_{i-1} fix all $j < i - 1$. Because of the compositions in `enter`, there is an $h_3 \in U'_{i-1}$ such that $h_3(i) = g_1 \circ h_2(i)$. Hence, $h_3^{-1} \circ g_1 \circ h_2 = g_3$ for some $g_3 \in A_i$. This gives us:

$$\begin{aligned} x_1 \circ x_2 &= h_1 \circ g_1 \circ h_2 \circ g_2 \\ &= h_1 \circ h_3 \circ h_3^{-1} \circ g_1 \circ h_2 \circ g_2 \\ &= h_1 \circ h_3 \circ g_3 \circ g_2 \end{aligned}$$

where $g_3 \circ g_2 \in A_i$. By again considering the composition operation in `enter`, there must be an $h_4 \in U'_{i-1}$ such that $h_4(i) = h_1 \circ h_3(i)$, and hence, $h_4^{-1} \circ h_1 \circ h_3 = g_4$ for

some $g_4 \in A_i$. This gives us:

$$\begin{aligned} x_1 \circ x_2 &= h_1 \circ h_3 \circ g_3 \circ g_2 \\ &= h_4 \circ h_4^{-1} \circ h_1 \circ h_3 \circ g_3 \circ g_2 \\ &= h_4 \circ g_4 \circ g_3 \circ g_2 \end{aligned}$$

and $h_4 \circ g_4 \circ g_3 \circ g_2 \in U'_{i-1} A_i$. Thus, $x_1 \circ x_2 \in A_{i-1}$, and hence, A_{i-1} is closed under composition. This implies that A_{i-1} is a group. In particular, $A_{-1} = G'$ is a group. □

Remark: We note that by Theorem 4.1.3, it suffices to add a minimal set of generators for G to its Schreier-Sims scheme T in order to have T fully represent G. In fact, not only is this sufficient, but it is desirable: for every permutation that we try to add to our group, we require, in the bare minimum case, a call to the `test` algorithm, so in order to represent the group, we would like to call `enter` only over a minimal set of generators, if possible. This is why, in Chapter 2, we were concerned with finding a minimal generating set for the symmetry group of each of the ILP formulations that we described.

It is simple to make the `enter` algorithm nonrecursive by using a stack of permutations. However, to do so will affect the order in which permutations are entered into the table as the ordering in the stack will not be identical to the order in which recursive calls are made. Thus, it may have an impact on time and memory costs. We offer both implementations in our package.

Proposition 4.1.4. *(From [22].) The algorithm* `enter` *runs in time* $O(n^6)$.

Proof. This proof is taken from [22]. Inserting a permutation into a row is executed a maximum of $O(n^2)$ times. This results in at most n^2 (the number of non-zero entries in the table) recursive calls to `enter`, for a total of $O(n^4)$ calls to `enter`. The compositions can be accomplished in $O(n)$, and the calls to `test` in $O(n^2)$. Hence, `enter` requires at most $O(n^6)$ operations. □

Later, we will see that one of the basic operations that is needed by the branching algorithm involves making small changes to the base. One idea, for dramatic base changes, would simply be to re-enter all the permutations in the table with regards to the new base. However, depending on the structure of our table T, this could require up to $O(n^2)$ calls to enter. Considering that, due to the nature of branch-and-cut, we will be altering our base only in small, predictable ways, rebuilding the entire table would not only be inefficient, but unnecessary. Instead, we use a technique called *downing a point*: given two indices r and s with $s \geq r$, we move the point $\beta[r]$ at position r of our base to position s and update the table T accordingly. This leads to the procedure `down` [21, 22], as detailed in Algorithm 4.1.3.

Algorithm 4.1.3 down(T, β, r, s)

Require: $r \leq s$

P = the non-null entries in row $\beta[r]$ of T
set row $\beta[r]$ of T to be the identity row
$t = \beta[r]$
for $i = r + 1$ to s **do**
  $\beta[i-1] = \beta[i]$
end for
$\beta[s] = t$
for all $p \in P$ **do**
  enter(T, β, p, r)
end for

Theorem 4.1.5. *(From [22].) The algorithm* `down` *(Algorithm 4.1.3) is correct and moves the element $\beta[r]$ from position r to position s of our base for $r \leq s$.*

Proof. We give a proof of the theorem stated in [22]. By setting row i of our table to

be the identity row, we mean that we want the following structure for our row:

$$T_{i,j} = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{otherwise.} \end{cases}$$

Given a position $i \in \{r+1, \ldots, s\}$, it is clear that the permutations in row $\beta[i]$ of T stabilize the points $\beta[0], \beta[1], \ldots, \beta[r], \ldots \beta[i-1]$ by the structure of T. Thus, if we move the element $\beta[i]$ to $\beta[i-1]$ (i.e. moving $\beta[i]$ down one position in our base), the structure of the table is not violated as the necessary points are stabilized by the permutations in this row. Hence, the reordering of the base for $\beta[r+1], \ldots, \beta[s]$ is valid.

It remains to show that the table is maintained for the permutations in row $\beta[r]$. As we are moving $\beta[r]$ forwards in our base to position s, the permutations in row $\beta[r]$ of our table will not stabilize the points $\beta[r+1], \ldots, \beta[s]$ and simply adjusting the base without restructuring the table is insufficient. Hence, we just set the row $\beta[r]$ to the identity row, and simply re-enter all the permutations originally contained in this row. Through this technique, it is clear that the modified table and base will contain all permutations originally in T and thus represents T with regards to the re-ordered base. □

A crude analysis of the algorithm demonstrates that `down` runs, in the worst case, in time $O(n^3 + kn^4)$, where k is the number of non-null entries in the original column $\beta[r]$ of the original table. However, this bound, as well as the bound for `enter`, have been shown to be pessimistic, and both algorithms typically perform much faster than this worst case analysis might indicate.

Incidentally, in certain cases, it may also be desirable to perform a *reverse downing of a point*, i.e. we may wish to move the point $\beta[r]$ from position r to position s, $r > s$. We could simply do so by making $r - s$ calls to down and swap the point with its previous neighbour until it is in the desired position, but we can do so more efficiently. We detail the procedure `reverse-down` in Algorithm 4.1.4.

Algorithm 4.1.4 reverse-down(T, β, r, s)

Require: $r \geq s$

$P = \emptyset$

for $i = s$ to $r - 1$ **do**

$P = P \cup$ non-null entries in row $\beta[i]$ of T

set row $\beta[i]$ of T to be the identity row

end for

$t = \beta[s]$

$\beta[s] = \beta[r]$

$\beta[r] = t$

for all $p \in P$ **do**

enter(T, β, p, s)

end for

This algorithm is identical in function and comparable in complexity to calling `down` $r - s$ times, but we hope that by removing all permutations from rows $\beta[s], \ldots, \beta[r-1]$ at once instead of processing each row individually that we will reduce the complexity of the potentially large number of calls to `enter` that would result.

4.2 Algorithms for building isomorph-free branching trees

We will now examine the methods that we will employ to exploit the symmetry group of an ILP in order to generate a nonisomorphic or partially nonisomorphic branch-and-cut tree. The theory and algorithms here are taken from [22], with the exception of Algorithms 4.2.6 and 4.2.7, which we designed. Note that all the algorithms listed in this section could be applied to any type of branching tree (e.g. branch-and-bound trees), but for the purposes of this thesis, we are predominantly interested in their uses

in branch-and-cut trees.

We begin by explaining how the algorithms use the concept of the table base to impose an ordering on the branching variables of our tree. Given a node a of our branch-and-cut tree over n variables (say x_i for $i \in \mathbb{Z}_n$), we define the following sets:

$$\begin{aligned} F_1^a &= \{j \in \mathbb{Z}_n \mid x_j \text{ is fixed to } 1 \text{ at } a\} \\ F_0^a &= \{j \in \mathbb{Z}_n \mid x_j \text{ is fixed to } 0 \text{ at } a\} \\ F^a &= \mathbb{Z}_n \setminus (F_1^a \cup F_0^a) \end{aligned}$$

Then we think of F_1^a as the indices of the variables fixed to 1 at a, F_0^a as the indices of the variables set to 0 at a, and F^a as the indices of the free variables at a. We will often omit the term "indices" when referring to the elements of F_1^a, F_0^a, and F_a, and simply call their elements the variables fixed to 1 at a, the variables fixed to 0 at a, and the free variables at a, respectively.

We now outline the structure of the base that we will impose on the base set (i.e. the set of variable indices) of our permutation group. At node a, we wish to maintain the following base structure over our variables:

$$\beta = [F_1^a \ F^a \ F_0^a].$$

Furthermore, we want the elements of F_1^a to appear in the order that they were fixed to 1 by our branch-and-cut tree, and we want the elements of F_0^a to appear in the reverse order that they were fixed to 0 by our branch-and-cut tree. The ordering of F^a is unimportant. Here, we state and prove the following theorem:

Theorem 4.2.1. *Let a and b be two nodes of our branch-and-cut tree with a an ancestor of b. If β is the base associated with node b, by the above conditions placed on the bases, β is also a valid base for node a.*

Proof. We observe that β has the following format:

$$\beta = [F_1^b \ F^b \ F_0^b].$$

As a is an ancestor node of b, it is clear that $F_1^a \subseteq F_1^b$ and $F_0^a \subseteq F_0^b$. Since the elements of F_1^a were set to 1 before the elements of $F_1^b \setminus F_1^a$ by our branch-and-cut algorithm, and the elements of F_1^b appear in the order that they were set to 1, then we know that our base can be written as:

$$\beta = [F_1^a \ (F_1^b \setminus F_1^a) \ F^b \ F_0^b].$$

Similarly, since the elements of F_0^a were set to 0 before the elements of $F_0^b \setminus F_0^a$, and the elements of F_0^b appear in the reverse order that they were set to 0, we can write our base as:

$$\beta = [F_1^a \ (F_1^b \setminus F_1^a) \ F^b \ (F_0^b \setminus F_0^a) \ F_0^a].$$

Since $F^a = \mathbb{Z}_n \setminus (F_1^a \cup F_0^a)$, it is obvious that $F^a = (F_1^b \setminus F_1^a) \cup F^b \cup (F_0^b \setminus F_0^a)$. Using this, our base can be rewritten in the form:

$$\beta = [F_1^a \ F^a \ F_0^a].$$

This is just a rewriting of β while maintaining the original structure, and this base satisfies the criteria for being a base for node a. □

We now introduce the concept of canonicity with regards to the nodes of our branch-and-cut tree. Let a and b be two nodes of our tree. If there exists a permutation $g \in G$ such that $g(F_1^a) = F_1^b$ and $g(F_0^a) = F_0^b$, then we say that the nodes a and b are *isomorphic*. As groups are closed and contain inverses, isomorphism is clearly an equivalence relation; because of this, node-isomorphism yields a partition of the nodes of our tree. We will refer to sets in our node-isomorphism partition as *isomorphism classes*. We can impose a lexicographical ordering on the elements in an isomorphism class in the following way: if $A, B \subseteq \mathbb{Z}_n$ (with $A \neq B$) are sorted sets with $A[i]$ and $B[j]$ $(i \in \mathbb{Z}_{|A|}, j \in \mathbb{Z}_{|B|})$ indicating the ith element of A and the jth element of B respectively, if we can find $k \in \mathbb{Z}_{|A|}$ such that $A[i] = B[i]$ for all $i \in \mathbb{Z}_k$ and $A[k] < B[k]$ (by imposing the natural integer ordering of $\mathbb{Z}_n$), then we say that A is lexicographically

smaller than B, which we will denote $A < B$. If no such k exists but $A \subset B$, then we also say that $A < B$.

As it is our goal to have an isomorph-free branch-and-cut tree, we only wish to investigate one node from each isomorphism class. To do so, given an isomorphism class C, we will only consider the lexicographically smallest node in C (say a). We designate a the *canonical representative* of C, or simply say that a is *canonical*. We note that this concept of canonicity respects the concept of canonicity as outlined in Section 2.3. Given a node b, to test it for canonicity, we simply calculate $\text{orb}(F_1^b, G)$. If F_1^b is the lexicographically smallest in this set, then we conclude that b is canonical, and if not, we say that b is *non-canonical*.

During our branch-and-cut, for any node a, we only process a if it is canonical; if not, we can immediately prune it and not consider it. We call this method *isomorphism pruning*. It is not immediately obvious that we will not prune valid solutions from our tree using this technique, so we briefly prove this here. As an additional and necessary condition on our tree, we impose that at any node a, when we branch, we must branch on variable x_f where $f = \min(F^a)$. This strategy is called *minimum index branching*.

Lemma 4.2.2. *(From [22].) Let $S \subseteq \mathbb{Z}_n$ be the lexicographically smallest set in* $\text{orb}(S, G)$ *under a permutation group G. Then $S' = S \setminus \max(S)$ is also the lexicographically smallest set in* $\text{orb}(S', G)$.

Proof. If S' is not the smallest in $\text{orb}(S', G)$, then we can find a permutation $g \in G$ such that $g(S') < S'$. Hence, $g(S) < S$, which contradicts the lexicographically smallest property of S. □

Theorem 4.2.3. *(From [22].) Let S be the set of all canonical nodes in our branch-and-cut tree. Then:*

1. *Given any node a in S, the parent node of a is also in S.*
2. *S contains all optimal canonical solutions of our ILP.*

Proof. Our proof is based upon the one provided in [22], and we extend with extra details.

1. Let a be a node in S which is not the root. Let b be the parent node of a in a full branch-and-cut tree without isomorphism pruning. As F_1^a is the smallest in $\text{orb}(F_1^a, G)$ by property of the canonicity of a, and as $F_1^b \subseteq F_1^a$ which contains the smallest $|F_1^b|$ elements of F_1^a (by minimum index branching), by repeated application of Lemma 4.2.2, F_1^b is the lexicographically smallest in $\text{orb}(F_1^b, G)$. Hence, b is canonical, so $b \in S$.

2. Let a be a node in our tree with F_1^a an optimal solution to our ILP. Let H be the lexicographically smallest set in $\text{orb}(F_1^a, G)$. Hence, there exists a node $b \in S$ with $F_1^b = H$. By repeated application of (1), every ancestor of b may be found in S, and hence, the subtree containing node b will be explored by our branch-and-cut algorithm with isomorphism pruning.

□

We proceed to introduce a type of fixing that we can perform in order to significantly reduce the number of nodes that we must consider; this technique is called *0-fixing*, and was taken from [22]. While this is actually an algorithm for the branch-and-cut as opposed to a group algorithm, it requires the concept of orbits and symmetry groups, so we detail it here. Let a be a node in our branch-and-cut tree that was reached by branching on variable x_j (let $j = -1$ if a is the root node). We then have the procedure outlined in Algorithm 4.2.1.

It is not obvious that a branch-and-cut tree using the above techniques will contain all nonisomorphic optimal solutions. We now prove this.

Lemma 4.2.4. *(From [22].) Using 0-fixing in a branch-and-cut tree with minimum index branching and isomorphism pruning will not destroy any canonical solutions.*

Algorithm 4.2.1 0-fixing(a, G)

if $j \geq 0$ and $x_j = 0$ **then**

set all variables in $\mathrm{orb}(j, \mathrm{stab}(F_1^a, G)) \cap F^a$ to 0

end if

$m = 0$

while $m \neq -1$ **do**

$m = -1$

if $|F^a| > 0$ **then**

$m = \min(F^a)$

if $F_1^a \cup \{m\}$ is not canonical **then**

set all variables in $\mathrm{orb}(m, \mathrm{stab}(F_1^a, G)) \cap F^a$ to 0

else

return m

end if

end if

end while

return n

Proof. Here we follow the proof in [22]. Let a be a canonical node in our branch-and-cut tree with F_1^a an optimal solution to our ILP. We assume that there does not exist a canonical node b with $F_1^b = F_1^a$. Then, by Lemma 4.2.2, we can find a canonical parent node c such that F_1^c contains the smallest $|F_1^c|$ elements of F_1^a, and some variable in $F_1^a \setminus F_1^c$ was set to 0 by 0-fixing. As this may have happened multiple times in our branch-and-cut tree, let us choose c to be as close as possible to the root node. Then, we can find $f \in F_0^c$ and $j \in F_1^a \setminus F_1^c$ with $j \in \text{orb}(f, \text{stab}(F_1^c, G))$ such that:

$$\max(F_1^c) < f < m = \min(F_1^a \setminus F_1^c) \leq j.$$

We note that the inequality $\max(F_1^c) < f$ holds because at node c, due to minimum index branching, all indices in F^c are greater than those in F_1^c. Thus, any variable fixed to 0 during 0-fixing at c trivially has index greater than $\max(F_1^c)$. The second inequality follows from the fact that $F_1^c \cup \{m\}$ is clearly the smallest in $\text{orb}(F_1^c \cup \{m\}, G)$: if m is set to 0 during 0-fixing, it must be done from an index $f < m$, and hence, all f considered by the 0-fixing algorithm are strictly smaller than m.

By construction, we can find a permutation $g \in \text{stab}(F_1^c, G)$ such that $g(j) = f$. Clearly, $g(F_1^c \cup \{j\}) = F_1^c \cup \{f\}$. This set is lexicographically smaller than $F_1^c \cup \{m\}$, and hence, we have a contradiction as $F_1^c \cup \{j\} \subseteq F_1^a$ by design. □

Thus, the use of 0-fixing preserves optimal canonical solutions. All that remains to complete the description of our algorithm is to devise a technique to calculate the orbit of a point in the stabilizer of a set. To do so, we examine a procedure `orbit-in-stabilizer` (described in Algorithms 4.2.2 and 4.2.3 [22, 23]), which exploits the structure of the Schreier-Sims table and the base.

This algorithm relies on backtracking in order to find the images of $\beta[k]$ in the stabilizer of the set $\{\beta[0], \ldots, \beta[k-1]\}$ under the action of our symmetry group, G. We begin by calculating J_k, which we call the *basic orbit* of $\beta[k]$ in G. This set is

Algorithm 4.2.2 orbit-in-stabilizer(T, β, k)

$J_k = \{i \in \mathbb{Z}_n \mid T_{\beta[k],i} \neq 0\}$

$p = 1 \in S_n$

$R = \{\beta[0], \ldots, \beta[k-1]\}$

$O = J_k$

orbit-in-stabilizer-aux(T, β, k, J_k, p, R, O, 0)

Algorithm 4.2.3 orbit-in-stabilizer-aux(T, β, k, J_k, p, R, O, i)

for all $j \in R$ **do**

  $h = T_{\beta[i],j}$

  if $h \neq 0$ **then**

    $R' = \{h^{-1}(r) \mid r \in R \setminus \{j\}\}$

    $g' = ph$

    if $i < k - 1$ **then**

      orbit-in-stabilizer-aux(T, k, J_k, g', R', O, $i + 1$)

    else

      $O = O \cup \{g'(s) \mid s \in J_k\}$

    end if

  end if

end for

simply the image of $\beta[k]$ under the action of permutations g of the form:

$$g = \underbrace{1 \dots 1}_{k-1 \text{ times}} g_k g_{k+1} \cdots g_{n-1}$$

where 1 represents the identity permutation, and g_i is a permutation from column $\beta[i]$ of our table T, i.e. a permutation fixing elements $\beta[0], \dots, \beta[i-1]$. Because the permutations $g_{k+1}, \dots, g_{n-1}$, by property of T, will not permute k and we only concern ourselves with the images of k, we need not consider them in our construction of J_k.

Now that we have established J_k, in order to calculate the complete orbit of k in the stabilizer, all we need to do is find all permutations g' of the form:

$$g' = g_0 \dots g_{k-1}$$

where we enforce that $g'(\{\beta[0], \dots, \beta[k-1]\}) = \{\beta[0], \dots, \beta[k-1]\}$. Then, by finding all such g' and all such g as detailed above, the set of $g'g$ will comprise the entire stabilizer that we are seeking. Hence, our orbit becomes elements of the form $g'g(j)$ for $j \in J_k$. As we have already computed J_k, it is sufficient to simply calculate all permutations g'.

This is where the backtracking comes in. We think of the set R as the set of remaining untargeted elements of $\{\beta[0], \dots, \beta[k-1]\}$ (i.e. the variables that have not been mapped to by the g' that we are constructing). The parameter i in the call to `orbit-in-stabilizer-aux` dictates which $\beta[i]$ we are currently considering. What we do is to find all permutations that map $\beta[i]$ to an element left in R. This gives us some g_i (indicated by h in our algorithm) in the construction of g as detailed above. We then simply extend the permutation g' (as occurs in the step $g' = ph$). As it is likely that g_i has permuted other elements of R apart from $\beta[i]$, we must take this into account; hence, we create the set R', which no longer contains $j = g_i(i)$ as something has been mapped to j. As we keep inverse-permuting the elements of R in this way, at any stage i when we have determined permutations $g_0, \dots, g_{i-1}$ with $g' = g_0 \dots g_{i-1}$, we have that R contains elements of the form $g_{i-1}^{-1} \dots g_0^{-1}(r)$, where r consists of elements

of $\beta[0], \ldots, \beta[k-1]$ that have not been the image of permutations yet. Hence, if we consider $g'(s)$ for $s \in R$, the inverses cancel out with the permutations to give us original elements of $\beta[0], \ldots, \beta[k-1]$ which is what we want, because we want these g' permutations to map $\{\beta[0], \ldots, \beta[k-1]\}$ to itself.

Thus, we proceed in this way, making recursive calls to `orbit-in-stabilizer-aux` until either we cannot find a permutation that maps i to a remaining element of R (in which case, the `for` loop finds no permutation and makes no recursive calls, so we backtrack) or until we have completed g' (in which case, $i = k-1$). When we have a complete permutation g', we extend our orbit O in the fashion explained above.

After we have exhausted all possibilities for mapping elements of R to themselves and hence created all possible g's, the backtracking ends and O contains the appropriate orbit.

A rough examination of `orbit-in-stabilizer` shows us that the worst-case complexity of the algorithm is $O(nk!)$, although, in practice, this algorithm runs significantly faster.

We now proceed to detail the techniques that we use for the canonicity testing itself; as mentioned above, to determine if a node a is canonical, for our purposes, it suffices to check if F_1^a is the lexicographically smallest in $\text{orb}(F_1^a, G)$. Because of the ordering of our base, if $k+1$ is the number of elements fixed to 1, it suffices to check that the set $\{\beta[0], \ldots, \beta[k]\}$ is the lexicographically smallest in its orbit under the action of our symmetry group G.

If we have been performing 0-fixing, we may use this information to perform a reasonably fast canonicity test as proposed in [22]. We call this fast algorithm `first-in-orbit1` and detail it in Algorithms 4.2.4 and 4.2.5.

The explanation for this algorithm comes largely from [22]; we expand upon this. We begin by examining the data structures used in the algorithms. The vector Z is used to hold some required information concerning variables that are fixed to 0. Let

Algorithm 4.2.4 first-in-orbit1(T, β, Z, k)

$p = 1 \in S_n$

$R = \{\beta[0], \ldots, \beta[k]\}$

$f =$ true

first-in-orb1-aux(T, β, Z, k, p, R, 0, f)

return f

Algorithm 4.2.5 first-in-orbit1-aux(T, β, Z, k, p, R, i, f)

if f is false **then**

return

end if

for all $j \in R$ **do**

if $\beta^{-1}[j] \geq Z[i]$ **then**

$f =$ false

return

end if

$h = T_{\beta[i],j}$

if $h \neq 0$ **then**

$R' = \{h^{-1}(r) \mid r \in R \setminus \{j\}\}$

$p' = ph$

if $i < k$ **then**

first-in-orbit1-aux(T, β, Z, k, p', R', $i+1$, f)

end if

end if

end for

$k+1$ be the number of variables at a node a fixed to 1. Then, we know that F_1^a, with the base ordering imposed upon it, is of the form:

$$F_1^a = \{\beta[0], \ldots, \beta[k]\} \text{ with } \beta[0] < \ldots < \beta[k].$$

We then construct the vector Z in the following fashion: $Z[i]$ stores the index, with respect to the base, of the last variable to have been fixed to 0 when $\beta[i]$ was fixed to 1. Then, we can conclude that the variables $\beta[Z[i]], \ldots, \beta[n-1]$, where n is the number of variables in our ILP, were all fixed to 0 when $\beta[i]$ was set to 1.

Example: Consider a node a for an ILP with 10 variables and the following base structure:

$$\beta = [\underbrace{1, 3, 4, 7}_{F_1^a}, \underbrace{9, 8}_{F^a}, \underbrace{6, 5, 0, 2}_{F_0^a}]$$

where the most recent variable fixing was setting x_7 to 1; then we would have that $P[3] = 6$, because when variable x_7 (which corresponds to $\beta[3]$) was fixed to 1, variables $\beta[6], \ldots, \beta[9]$ were fixed to 0.

Theorem 4.2.5. *(From [22].) The algorithm* `first-in-orbit1` *is correct.*

Proof. We follow the proof in [22], and provide more detailed steps.

If, at any point, the condition $\beta^{-1}[j] \geq P[i]$ holds, then, for some $t \leq i$, there is a point $j \in R$ that was fixed to 0 before $\beta[t]$ was fixed to 1 and, provided that $t \geq 2$, after $\beta[t-1]$ was fixed to 1. We define the following set:

$$S = p(\{\beta[0], \ldots, \beta[i-1]\})$$

We will now demonstrate that at any stage, R, the set of remaining target elements (as outlined in the `orbit-in-stabilizer-aux` algorithm) is always a subset of the set $S' = p^{-1}(\{\beta[0], \ldots, \beta[k]\})$ at any given point in the backtracking. At any point i in the algorithm, with $p = g_0 \ldots g_{i-1}$, let R_i denote the set R for this recursive call. We

then have:

$$\begin{aligned}
R_0 &= R \\
R_1 &= \{g_0^{-1}(r) \mid r \in R_0 \setminus \{g_0^{-1}(0)\}\} \\
&= \{g_0^{-1}(r) \mid r \in R \setminus \{g_0^{-1}(0)\}\} \\
R_2 &= \{g_1^{-1}(r) \mid r \in R_1 \setminus \{g_1^{-1}(1)\}\} \\
&= \{g_1^{-1}g_0^{-1}(r) \mid r \in R \setminus \{g_0^{-1}(0), g_1^{-1}g_0^{-1}(1)\} \\
\vdots & \qquad\qquad \vdots \\
R_i &= \{g_{i-1}^{-1}\dots g_0^{-1}(r) \mid r \in R \setminus \{g_0^{-1}(0), g_1^{-1}g_0^{-1}(1), \dots, g_{i-1}^{-1}\dots g_0^{-1}(i-1)\} \\
\vdots & \qquad\qquad \vdots
\end{aligned}$$

As elements of S', at point i in the algorithm, are of the form $g_{i-1}^{-1}\dots g_0^{-1}(s)$ for $s \in \mathbb{Z}_k$, clearly it is the case that R_i is a subset of S' by construction. It is also obviously disjoint from S as defined above (as S consists of all targets of the permutation p up to point i). Because of this, we have that $j = p^{-1}(\beta[s])$ for some $s \in \{i, \dots, k\}$.

As i was fixed to 0 before $\beta[t]$ was set to 1, we can find a $w < \beta[t]$ such that:

$$i \in \mathrm{orb}(w, \mathrm{stab}(\{\beta[0], \dots, \beta[t-1]\}, G)).$$

Because of this, we can find a permutation h in the stabilizer that maps i to w. We define the set $S^o = p(\{\beta[0], \dots, \beta[t-1]\})$. Then, this is clearly a subset of S. We now derive the following statement:

$$\{hp^{-1}(s) \mid s \in S^o\} = \{\beta[0], \dots, \beta[t-1]\}$$

since p^{-1} maps elements of S^o to $\{\beta[0], \dots, \beta[t-1]\}$ and h, by selection, stabilizes these elements. We also have:

$$hp^{-1}(\beta[t]) = w < \beta[t].$$

Thus, we have found a permutation, namely hp^{-1}, that maps the set $S^o \cup \{\beta[t]\}$ to $\{\beta[0], \dots, \beta[t-1], w\}$, which is lexicographically smaller than $\{\beta[0], \dots, \beta[t]\}$. Thus, if the algorithm returns `false`, the set $\{\beta[0], \dots, \beta[k]\}$ is not lexicographically smallest in its orbit.

It remains to show the converse, namely that if $\{\beta[0], \ldots, \beta[k]\}$ is not the lexicographically smallest in its orbit, then the algorithm returns false. Suppose that this is the case. Then we can find a permutation $q \in G$ that maps this set to something lexicographically smaller in its orbit. Furthermore, we can find an index $t \in \mathbb{Z}_{k+1}$ such that q stabilizes $\{\beta[0], \ldots, \beta[t]\}$ and $q(\beta[t]) < \beta[t]$. Pick t to be the smallest such value.

We write $q = h_0 \ldots h_n$ with h_i from the ith column of our Schreier-Sims table T. We note that $w = q(\beta[t])$ was set to 0 before $\beta[t]$ was set to 1. Hence, we have $q^{-1}(\{\beta 0, \ldots, \beta t - 1, w\}) = \{\beta[0], \ldots, \beta[t]\}$. During the algorithm, we note that we will find a permutation p such that $p(\beta[i]) = q^{-1}(\beta[i])$ for $i \in \mathbb{Z}_t$, namely $p = h_1 \ldots h_{t-1}$ as expected. Let $z = q^{-1}(\beta[t])$. Then, we have that $p^{-1}q^{-1}(w) = z$, and that $p^{-1}q^{-1}$ stabilizes $\{\beta[0], \ldots, \beta[t-1]\}$. Thus, it follows that z will be in the orbit of the stabilizer of this set, and that z was fixed to 0 with w or earlier. Thus, R will contain z and $\beta^{-1}[z] \geq P[t]$, so the algorithm correctly returns `false`. □

The worst-case complexity of `first-in-orbit1` is $O(n(k+1)!)$, but as with the algorithm, `orbit-in-stabilizer`, in practice, this method seems to run in much lower time.

As mentioned above, there is, however, one problem with using the above algorithm for testing node canonicity: it is dependent on the fact that we need to have set everything possible to 0 (for the Z array to contain the proper information to be used) via 0-fixing. As we are interested in the effects of "turning off" canonicity testing and 0-fixing at certain points in the tree (as it may be too costly to compute canonicity and orbits in stabilizers for each node when F_1 becomes large, especially given that orbits become smaller as F_1 increases in size) but may still require the ability to turn these back on at later points, we must investigate algorithms that allow this. While it is still possible to use `orbit-in-stabilizer` to determine orbits in stabilizers at any point, for 0-fixing, we need to check whether a set is canonical as well. This leads us to the modified algorithm, `first-in-orbit2`, as described in Algorithms 4.2.6 and 4.2.7.

Algorithm 4.2.6 first-in-orbit2(T, β, k)

return first-in-orbit2-aux(T, β, k, 1, {}, 0)

Algorithm 4.2.7 first-in-orbit2-aux(T, β, k, p, S, i)

if $i = k + 1$ **then**

return true

end if

if $S < \{\beta[0], \ldots, \beta[i-1]\}$ **then**

return false

end if

for all permutation q in row $T_{\beta[i]}$ **do**

$p' = pq$

$S' = S \cup \{p'(\beta[i])\}$

if first-in-orbit2-aux(T, β, k, p', S', $i + 1$) is false **then**

return false

end if

end for

return true

Theorem 4.2.6. *The algorithm* `first-in-orbit2` *is correct.*

Proof. Given the set $B = \{\beta[0], \ldots, \beta[k]\}$, assume that there is some permutation $p \in G$ such that $p(B) < B$. We show that we cover p by our algorithm. Given the Schreier-Sims table T with base β, we can write:

$$p = h_0 h_1 \ldots h_{n-1}$$

where h_i is a strong generator from row $T_{\beta[i]}$. Given that we are only interested in the action of p on the elements of B and do not care about how it affects $\{\beta[k+1], \ldots, \beta[n-1]\}$, we can consider the permutation:

$$p' = h_0 h_1 \ldots h_k \underbrace{1 \ldots 1}_{n-k \text{ times}}$$

If p is in our table, then certainly, so is p', and it is clear that $p'(B) < B$, so it suffices to show that p' will be constructed by our algorithm. However, this is obvious, as we try all combinations of permutations over the rows $\beta[0], \ldots, \beta[k]$ of our table. □

A pessimistic upper bound on this algorithm is $O(n^k)$. If the table T is sparse, it performs significantly better. For problems with large symmetry groups, it may approach this bound. Unfortunately, this algorithm is considerably less efficient than `first-in-orbit1`, which exploits the fact that 0-fixing has been used and hence requires the examination of far fewer permutations.

We note that all three of these algorithms can be made to run non-recursively by using arrays to simulate recursion. We demonstrate them in a recursive fashion for its elegance, but in our implementation, we have opted for the non-recursive implementation to avoid the overhead of repeated method calls.

We may take two different approaches to storing symmetry group information, both with inherent advantages and disadvantages.

1. Store a single Schreier-Sims table for the entire tree and modify it as needed in our depth-first tree search.

2. Store a Schreier-Sims table for each node in the tree.

These two techniques are discussed further in Chapter 6 when we establish specific details on our branch-and-cut algorithm.

4.3 Variable fixings and the base

In some ILPs, it is known that certain variables will be fixed to 1 in a canonical design, and as a consequence of the ILP constraints or symmetry group permutations, other variables will be fixed to 0. Section 2.3 discusses what variables may be fixed in the context of set system problems.

It is not possible to simply fix these variables in the problem formulation since this would violate our use of minimum index branching. Instead, however, it is possible to *execute a change on the initial ordering of the variables* using the base to allow for 1-fixings.

Given an ILP with a set of variables $\mathcal{X}$, if we have subsets $X_1, X_0, X \subseteq \mathcal{X}$ such that we know, for any canonical solution $S = (x_0, x_1, \ldots)$:

$$\begin{aligned} x_i \in X_1 &\Rightarrow S_i = 1 \\ x_i \in X_0 &\Rightarrow S_i = 0, \end{aligned}$$

then we can simply specify an initial base $\beta = [X_1\ X\ X_0]$ for our group before entering the permutations, or, if we wish to 0-fix the variables in X_0 at initialization time, an initial base $\beta = [X_1\ X_0\ X]$ to avoid reverse-downing.

This provides a generalization to minimum index branching; if our problem has n variables, we simply need an ordered set O containing all elements from $\mathbb{Z}_n$ and can use that as our base set for the minimum index branching routine, picking the minimal free element in O with regards to its ordering. In the case of standard minimum index branching, we simply use $\mathbb{Z}_n$ with standard integer ordering, as expected.

4.4 Algorithms for calculating the symmetry group of an ILP

In order to use the above algorithms, it is essential that we can determine the symmetry group of our ILP. For some combinatorial problems, the structure of the group immediately presents itself. In this case, we simply need to determine a set of generators and add them, using `enter`, to the Schreier-Sims table T to create the group. Examples in which we implicitly know the symmetry group are given in Chapter 2.

In other, less structured cases, the symmetry group may not be so immediately obvious. In these cases, it is necessary to use some method in order to determine it. We discuss three such algorithms here. We note that in [22, 23], a simplified variant of the algorithm presented in Section 4.4.3 was used; this original algorithm worked only on matrices with 0-1 coefficients. The algorithms in Section 4.4.1 and 4.4.2, as well as the general algorithm in Section 4.4.3, are proposed by us.

4.4.1 Naïve technique

The first approach that we look at simply iterates over all possible column and row permutations of the coefficient matrix of our ILP. The basic pseudocode is listed in `find-symmetry-group1` (Algorithm 4.4.1), where we denote our n-vector of coefficients as c, our m-vector of bounds as b, and our our $m \times n$ coefficient matrix as A.

Basically, this method iterates over all possible $n!$ column permutations of the matrix A, and for each column permutation, it examines whether or not there exists a row permutation such that the two permutations fix A. Again, to find such a row permutation, every possible permutation is examined, and there are $m!$ such permutations. Thus, the total running time of this algorithm is in the order of $O(m!n!)$.

While this may seem infeasible, it is simple to implement, and the initial idea was that it might run in a reasonable amount of time for small m and n, as is the case

Algorithm 4.4.1 find-symmetry-group1(c, A, b)

```
S_c = S_n
while |S_c| > 0 do
  pick any p ∈ S_1
  S_c = S_c \ {p}
  if p(c) = c then
    S_r = S_m
    while |S_r| > 0 do
      pick any q ∈ S_r
      S_r = S_r \ {q}
      if q(b) = b then
        if A(p, q) = A then
          output p
          exit inner loop
        end if
      end if
    end while
  end if
end while
```

with the 2-$(5, 3, 1)$ packing ($m = 10$, $n = 10$). However, this proved to be an incorrect assumption; on a Sun Fire V880 with eight 900 MHz UltraSparc III Cu processors and 32 GB of RAM (running Solaris 9), it was found that the algorithm took approximately 2.3 seconds (CPU time) to examine each column permutation. Given that there are $10! = 3,628,800$ such permutations, we can easily see that iterating naïvely over all permutations like this would take an approximated 96.6 days (real time).

Clearly, this is not a reasonable amount of time to calculate the symmetry group of such a simple problem, and this algorithm increases factorially in terms of CPU time. Since each column permutation from the construction of the 2-$(5, 3, 1)$ packing considers 10! row permutations, we can (poorly) estimate that it takes $2.3/10! = 6.34 \times 10^{-7}$ seconds to consider each row permutation. Hence, in the case of the symmetry group for the 2-$(7, 3, 1)$ design, which has $\binom{7}{2} = 21$ rows, each column permutation will take at least the following amount of time to consider:

$$21! \text{ perms } \cdot 6.34 \times 10^{-7} \frac{\text{s}}{\text{perm}} \approx 3.24 \times 10^{13} \text{ s} \approx 1.03 \times 10^{6} \text{ years.}$$

As there are $\binom{7}{3} = 35$ columns in our ILP, we consider $35! = 1.03 \times 10^{40}$ column permutations, giving us that it is likely to take at least the following amount of time to calculate the symmetry group for this problem using the naïve technique:

$$1.03 \times 10^{40} \text{ perms } \cdot 1.03 \times 10^{6} \frac{\text{years}}{\text{perm}} = 1.06 \times 10^{46} \text{ years.}$$

Instead, we obviously require a different approach to hope to calculate the symmetry group of even small problems, which motivated our creation of the backtracking / partitioning technique.

4.4.2 Backtracking / partitioning technique

This algorithm is based on ideas from Denny's canonicity testing algorithm in [9]. The general idea behind the second technique we examine is that if a partial column

permutation is unable to fix A, then no extension of the partial permutation is able to fix A. Thus, we build p by backtracking techniques, determining at each step of the backtracking whether or not the possibility of a q exists such that $A(p, q) = A$.

As demonstrated in the naïve technique, it is not feasible to consider all row permutations; thus, we use a technique called *partition refinement* in order to minimize the number of row permutations that we need to try. In essence, when we manage to complete p via backtracking techniques and we have a valid partition refinement, we have found a p and q (or possibly a family of q permutations) that fix A. Thus, we output p and backtrack. The general pseudocode for the algorithm, explained below, is presented in `find-symmetry-group2` (Algorithm 4.4.2).

Note that, at every iteration of the loop, we ensure that our partial permutation p fixes c, and that the family of row permutations represented by our partition refinement is capable of fixing b (if either of these cases fail, we backtrack immediately). Following is a detailed explanation of the various components of the algorithm.

Backtracking on p

This is a fairly simple concept, and one that will not be examined in great depth (it is assumed that the reader has knowledge of backtracking algorithms). The general idea is that we have a partial permutation over the columns, and we extend it via backtracking techniques. Thus, at some stage of the algorithm, if we have $P = [1, 3, 5]$ (where $P[i] = p^{-1}(i)$, for reasons explained in the following section) considered as a partial permutation of S_7, we have the following possible extensions of p:

$$\begin{aligned} P_1 &= [1, 3, 5, 0] \\ P_2 &= [1, 3, 5, 2] \\ P_3 &= [1, 3, 5, 4] \\ P_4 &= [1, 3, 5, 6] \end{aligned}$$

All of these possibilities will be explored by the backtracking tree.

Algorithm 4.4.2 find-symmetry-group2(c, A, b)

calculate the partition scheme of A

$i = 1$

$P = []$

while $i \geq 1$ **do**

try to extend P, the permutation vector, at position i

if this was not possible **then**

$i = i - 1$

loop

end if

refine the partitions of q

if q cannot be refined to a bijection **then**

loop

end if

if P does not represent a complete permutation **then**

$i = i + 1$

loop

end if

output the permutation p represented by P

end while

The advantage to using such an approach is twofold; firstly, if a partial column permutation p does not fix the vector c, then no extension of p can fix c, and secondly, if no row permutation exists that fixes A under the influence of p, then no row permutation will exist that will fix A for any possible extension of p. Therefore, by this technique, provided that the size of the symmetry group of the problem in consideration is a sufficiently small subgroup of S_n, we can prune huge numbers of column permutations from our search space.

Partition schemes and refinement

In this section, we represent permutations inversely; i.e. if p is a permutation, $P[i] = j$ means that $j \to i$ under the action of p. The reason we do this is that we want to build permutations over our matrix columnwise from left to right, and hence it is our goal to know what maps to column i rather than the converse.

Definition 4.4.1. A *partition scheme* is a two-dimensional array of sets, $\mathcal{PS}$, such that $\mathcal{PS}[col, val]$ is the set of positions in which val appears in column col of the matrix A.

Example: For the 2-$(5, 3, 1)$-packing design, we have the following matrix:

$$A = \left[\begin{array}{cccccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}\right]$$

Then, if we index the columns and the rows beginning at 0, $\mathcal{PS}[0,0]$ is simply the set of positions in which 0 appears in the first column (namely $\{2,3,5,6,7,8,9\}$), and $\mathcal{PS}[0,1]$ is the set of positions in which 1 appears in the first column (namely $\{0,1,4\}$). Similarly, $\mathcal{PS}[1,0] = \{1,3,4,6,7,8,9\}$ and $\mathcal{PS}[1,1] = \{0,2,5\}$, etc.

Definition 4.4.2. A *partition refinement* is an array of sets, $\mathcal{PR}$, such that $\mathcal{PR}[row]$ is the set of rows to which it is possible to map row such that A will be preserved under the corresponding partial column permutation. Note that $\mathcal{PR}$ represents a family of row permutations.

Initially, when P is empty ($P = []$), we know that any row can be mapped to any other row, thus, $\mathcal{PR}[i] = \mathbb{Z}_m$ for all $i \in \mathbb{Z}_m$. At any stage of the backtracking, if we extend P by $x \in \mathbb{Z}_n$ at position i, the family of permutations represented by $\mathcal{PR}$ may no longer all fix A under p, so we must refine to remove the invalid permutations.

When P is extended by x at position i (i.e. p maps column x to position i), we perform the following *refinement step*:

$$\mathcal{PR}[j] = \mathcal{PR}[j] \cap \mathcal{PS}[i, A[j, p[i]]] \text{ for all } j \in \mathbb{Z}_m$$

Example: Assume we have the matrix A as defined above for the 2-$(5,3,1)$-packing design. For the purposes of this example, we will begin with p empty and add 2 to it (i.e. mapping column 2 to position 0), so $P = [2]$. Then we perform the following

refinements:

$$\begin{aligned}
\mathcal{PR}[0] &= \mathcal{PR}[0] \cap \mathcal{PS}[0, A[0,2] = 1] \\
&= \mathbb{Z}_m \cap \{0,1,4\} \\
&= \{0,1,4\} \\
\mathcal{PR}[1] &= \mathcal{PR}[1] \cap \mathcal{PS}[0, A[1,2] = 0] \\
&= \mathbb{Z}_m \cap \{2,3,5,6,7,8,9\} \\
&= \{2,3,5,6,7,8,9\} \\
\mathcal{PR}[2] &= \mathcal{PR}[2] \cap \mathcal{PS}[0, A[2,2] = 0] \\
&= \mathbb{Z}_m \cap \{2,3,5,6,7,8,9\} \\
&= \{2,3,5,6,7,8,9\} \\
\mathcal{PR}[3] &= \mathcal{PR}[3] \cap \mathcal{PS}[0, A[3,2] = 1] \\
&= \mathbb{Z}_m \cap \{0,1,4\} \\
&= \{0,1,4\} \\
\vdots & \qquad\qquad \vdots
\end{aligned}$$

Now, say we further extend P with 6, so $P = [2,6]$. Now our "partially permuted" matrix is as follows:

$$A(p,1) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Then our partition refinements become:

$$\begin{aligned}
\mathcal{PR}[0] &= \mathcal{PR}[0] \cap \mathcal{PS}[1, A[0,6] = 0] \\
&= \{0,1,4\} \cap \{1,3,4,6,7,8,9\} \\
&= \{1,4\} \\
\mathcal{PR}[1] &= \mathcal{PR}[1] \cap \mathcal{PS}[1, A[1,6] = 0] \\
&= \{2,3,5,6,7,8,9\} \cap \{1,3,4,6,7,8,9\} \\
&= \{3,6,7,8,9\} \\
\vdots & \qquad\qquad \vdots \\
\mathcal{PR}[4] &= \mathcal{PR}[4] \cap \mathcal{PS}[1, A[4,6] = 1] \\
&= \{2,3,5,6,7,8,9\} \cap \{0,2,5\} \\
&= \{2,5\}
\end{aligned}$$

In our partially permuted matrix, row 0 is [1 0]. If we scan the rows over the first two columns of the unpermuted A, we see that the entries [1 0] occur in rows 1 and 4. Thus, row 0 can only be mapped to rows 1 and 4 of our original matrix to preserve it under our permutations at this point in the construction of p_1. Similarly, row 4 corresponds to [0 1] under the current column permutation, and so we scan our original matrix A to find that rows 2 and 5 begin with [0 1], so they are the only possible images of row 4 under a row permutation designed to fix A. In this case, it is easy to see that our refinement is not extendible to a bijection; by looking at our partially permuted matrix above, we can see that there is no row that corresponds to [1 1]. The first row of A is [1 1], and thus it is not possible to find a surjective mapping between our partially permuted matrix and the first two columns of A. Thus, since permutations are bijections, our refinement represents the empty set of permutations (i.e. no row permutation exists), and we backtrack immediately.

4.4.3 Technique based on coloured graphs

While the partitioning and refinement technique works well in comparison to the naïve technique, it does present one major shortcoming: the algorithm generates every permutation of our symmetry group, which could be quite large. In order to create the Schreier-Sims representation of the group, we need only a minimal set of strong generators. It is this consideration that motivates our third algorithm.

The only necessary condition for this algorithm is that all variables of the ILP have the same range. As we only concern ourselves with 0-1 ILPs, this technique will work fine for our needs. If we have the ILP:

$$\begin{aligned} \text{Maximize} \quad & cx \\ \text{subject to} \quad & b_l \leq Ax \leq b_u \end{aligned}$$

with $c = [c_0, \ldots, c_{n-1}]$, $b_l = [b_{l,0}, \ldots, b_{l,m-1}]$, and $b_u = [b_{u,0}, \ldots, b_{u,m-1}]$, and we denote the entry in row i and column j of A to be $a_{i,j}$, we can derive a graph representing our problem.

Definition 4.4.3. A *coloured graph* is a graph $C = (V, E)$ and a set $K = \{K_0, \ldots, K_{s-1}\}$ of colouring classes such that K partitions V, i.e.:

- $K_0 \cup \ldots \cup K_{s-1} = V$
- $K_i \cap K_j = \emptyset$ for $\{i, j\} \subseteq \mathbb{Z}_s$.

We derive a coloured graph $C = (V, E)$ from our ILP in the following way: let $V = \{x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}\}$ and set $E = \{\{x_j, y_i\} \mid A_{i,j} \neq 0\}$. For each variable x_i, we create a vector $v_i = [c_i \ a_{0,i} \ a_{1,i} \ \ldots \ a_{m-1,i}]$. Then two variables x_i and x_j are in the same colour class if and only if $v_i = v_j$. Additionally, y_i and y_j are in the same colour class if and only if $[b_{l,i} \ b_{u,i}] = [b_{l,j} \ b_{u,j}]$.

Definition 4.4.4. Given a coloured graph $C = (V, E)$ with colouring classes $K = \{K_0, \ldots, K_{s-1}\}$, an *automorphism* $g : V \to V$ is a bijection such that:

- for $v \in V$, if $v \in K_i$, then $g(v) \in K_i$
- $g(E) = E$.

The *automorphism group* of a coloured graph is the group of all automorphisms of the graph. We denote the automorphism group of C as $\mathrm{Aut}(C)$.

Theorem 4.4.1. *Given the above construction, if G is the symmetry group of our ILP, then there exists a surjective homomorphism of groups mapping $Aut(C)$ onto G.*

Proof. Consider G as a subgroup of S_n, and define the group homomorphism $\phi : \mathrm{Aut}(C) \to G$ such that $\phi(g) = p$, where:

$$p(i) = j \text{ for } g(x_i) = x_j, \qquad i \in \mathbb{Z}_n$$

It suffices to show that it is surjective (i.e. every permutation in G is covered). Let $p \in G$. We will show that there is a permutation $g \in \mathrm{Aut}(C)$ such that $\phi(g) = p$. Given that $p \in G$, we know from Definition 1.3.7 that we can find a row permutation $q \in S_m$ such that for the ILP:

- $A(p, q) = A$
- $p(c) = c$
- $q(b) = b$

We need to find a permutation $g \in Aut(C)$ such that $\phi(g) = p$. Define the following permutation:

$$\begin{aligned} g : V &\to V \\ x_i &\mapsto x_{p(i)} \\ y_j &\mapsto y_{q(j)} \end{aligned} .$$

We now claim that g is an automorphism of our graph. By the three above properties, g respects the colour classes of C, so it remains to show that $g(E) = E$. If $e = \{x_j, y_i\} \in$

$E, i \in \mathbb{Z}_m, j \in \mathbb{Z}_n$, $g(e) = \{x_{p(j)}, y_{q(i)}\}$. As $A(p,q) = A$ and $q(b) = b$, $g(e) \in E$. As this map is injective (as p and q are permutations), it is also surjective, and hence, $g(E) = E$. Thus, ϕ is surjective. □

Hence, if we can easily determine $\mathrm{Aut}(C)$ for our coloured graph C, we can then easily obtain the symmetry group G of our ILP. Furthermore, a set of generators of $\mathrm{Aut}(C)$ can be translated to a set of generators for G through the group isomorphism ϕ. The freely distributable package `nauty`, by Brendan McKay [29], allows us to calculate generators for the automorphism group of a coloured graph.

Hence, given an arbitrary ILP with variables all from the same range, we can use `nauty` to compute generators for the coloured graph represented by the problem, translate these to permutations over the variables using ϕ, and then `enter` these permutations into our Schreier-Sims table in order to create our symmetry group.

The pseudocode for this third technique is given in `find-symmetry-group3` (Algorithm 4.4.3).

Algorithm 4.4.3 find-symmetry-group3(c, A, b)

construct C, the coloured graph representing the ILP
call `nauty` on C to get $\mathrm{Aut}(C) = \langle g_0, \ldots, g_{k-1} \rangle$
for $i = 0$ to $k - 1$ **do**
  output $\phi(g_i)$
end for

4.4.4 Comparison of the three techniques

Table 4.1 shows the times required for the three different algorithms to calculate the symmetry groups for several different 2-$(v, 3, 1)$ packings using the block formulation. In the table, we denote the `find-symmetry-group` algorithms by their number. For the first algorithm (the naïve approach), we provide lower-bound estimates on the times

using an approximation of 6.34×10^{-7} seconds to process each row permutation. The times reported are measurements of CPU time in seconds.

v	$\lvert G\rvert$	fsg1 time	fsg2 time	fsg3 time
5	120	8.34×10^{6}	0.05	≤ 0.01
6	720	2.02×10^{24}	2.01	≤ 0.01
7	5040	3.35×10^{53}	65.49	0.01

Table 4.1: CPU times for the three different symmetry group calculation algorithms.

As it is not feasible to continue `find-symmetry-group1` and `find-symmetry-group2` for higher v, in Table 4.2 we give results exclusively for `find-symmetry-group3` for a larger number of values to demonstrate its efficiency and compare it with the time taken to simply enter generators known a priori, as specified in Section 2.1.1. The column S contains the number of strong generators appearing in the Schreier-Sims table, not including identity permutations.

v	$\lvert G\rvert$	S	fsg3 time	Gens time
5	120	15	0.00	0.00
6	720	29	0.00	0.01
7	5040	49	0.01	0.01
8	40320	76	0.05	0.03
9	362880	111	0.15	0.11
10	3628800	155	0.47	0.32
11	39916800	209	1.50	0.83
12	479001600	274	3.72	2.13
13	6227020800	351	8.80	5.00
14	8.72×10^{10}	441	20.82	10.98
15	1.31×10^{12}	545	42.43	22.58
16	2.09×10^{13}	664	82.77	44.44

Table 4.2: CPU times for `find-symmetry-group3` compared with entering generators known a priori.

Chapter 5

Cuts used and our cutting plane implementation

A general cutting plane algorithm was introduced in section 3.2. We now describe our particular implementation of the cutting plane method used for our branch-and-cut, as well as the two cuts that we provide, namely isomorphism cuts [22] and clique inequalities [35, 36].

5.1 Cutting plane implementation

As detailed in section 3.2, the pure cutting plane technique for solving ILPs iterates repeatedly until an integer solution is obtained. In the cutting plane algorithm used by branch-and-cut, we generate cuts until some selected criteria indicates to us that it is no longer likely to be useful to continue to do so, at which point we instead utilize the methods of branch-and-bound and branch on a variable. We here follow the algorithm proposed in [34, 35].

For the purposes of our framework, we selected two measurements to determine the effectiveness of each iteration of the cutting plane. Firstly, we measure the number of

cuts generated. If the algorithm does not generate a sufficient number of cuts during the separation phase, we heuristically decide that our time would be better spent branching as opposed to iteratively producing more cuts. Secondly, as we are only adding cuts to our ILP that are violated by the current fractional solution of the LP relaxation, we may measure the maximum violation amongst all the generated cuts. If the maximum violation does not exceed some minimum value, we suspect that our cutting plane algorithm is either producing cuts that have little benefit towards achieving integrality or that the cutting plane algorithm is tailing off.

In addition, it may not be advantageous to add any violated cut to our ILP. Certain cuts may not result in sufficient violations, and thus will simply complicate our ILP formulation without significantly assisting in obtaining an integral solution. Hence, we only add cuts if they are violated to a specified degree.

We will refer to the minimum number of required cuts parameter as `MNC`. Instead of directly specifying the minimum required violation (`MRV`) to continue the cutting plane algorithm and the worthwhile violation (`WV`) to add a cut to the ILP, we allow a lower bound and an upper bound to be specified on these parameters, say $[v_{\min}, v_{\max}]$ and $[m_{\min}, m_{\max}]$ respectively. Let `numvars` and `numfracvars` be the total number of variables for the ILP and the total number of variables with fractional values in the current LP solution. We then set $\texttt{MRV} = v_{\min} + \frac{\texttt{numfracvars}}{\texttt{numvars}}(v_{\max} - v_{\min})$, and similarly for `WV`. The motivation behind this is to encourage the algorithm to continue and generate cuts if the previous iteration produced a solution that demonstrated a higher degree of integrality. The goal of the cutting plane algorithm is to approach an integer solution; if we are making little progress in attaining this goal as indicated by a large number of fractional variables, we become more restrictive with respect to `MRV` and `WV` and set them closer to their upper bounds $v_{\max}$ and $m_{\max}$; we suspect that continuing is unlikely to help sufficiently, so we instead demonstrate a preference to terminate and resort to branch-and-bound techniques.

The cutting plane procedure used by our package is outlined in `cutting-plane` (Algorithm 5.1.1). The framework allows users to arbitrarily select which cut generation techniques to use, as well as to create new ones. We provide two types of cuts, namely isomorphism cuts and clique inequalities, and examine them in the next sections. This algorithm performs feasibility test and bounds testing on LPs: if an LP is not feasible, or cannot be solved to give an integer solution at least as good as the current best solution, the calling code is informed.

5.2 Isomorphism cuts

Margot [22] has proposed `isomorphism-cuts` (Algorithm 5.2.1). Isomorphism cuts are a family of cuts that are aimed at reducing the number of calls to `canonicity-test` (Algorithm 6.0.4) by allowing the LP solver to prune some non-canonical nodes before they are fully tested for canonicity.

Given a node a of our branch-and-cut tree, define the set $H^a = F_1^a \cup F^a$. We can then think of H^a as the variables that can potentially have value 1 at node a. Suppose we can find $J \subseteq H^a$ with $J \cap F^a \neq \emptyset$ such that J^* is the canonical representative isomorphic to J under $\mathrm{orb}(J, G)$ and $J^* < F_1^a$. Then, if b is any descendant of a with $J \subseteq F_1^b$, this node will be automatically pruned by the canonicity testing. Hence, to improve algorithmic efficiency and avoid generating unnecessary nodes, we can add the following cut to the subtree rooted at node a:

$$\mathrm{IC}(J): \qquad \sum_{j \in J} x_j \leq |J| - 1.$$

We are processing in a depth-first fashion and always exploring the child of a node a where the branching variable was fixed to 1 before exploring the child with branching variable fixed to 0. Thus, we will be enumerating the sets F_1^d, where d is any node in the tree, in a lexicographical fashion from smallest to largest. Because of this,

Algorithm 5.1.1 cutting-plane(Node n, ILP $\mathcal{I}$, `MNC`, $v_{\min}$, $v_{\max}$, $m_{\min}$, $m_{\max}$, `bestObjectiveValue`)

`flag` = true

loop

process the node n

solve the LP relaxation of $\mathcal{I}$ to get solution x^* with objective value b

if the LP is infeasible **then**

return an error code stating that the node is infeasible

end if

if $\lfloor b \rfloor \leq$ `bestObjectiveValue` **then**

return a message saying this node will not lead to an optimal solution

end if

if `flag` is false **then**

return (x^*, b)

end if

if x^* is integer **then**

return (x^*, b)

end if

$\texttt{MRV} = v_{\min} + \frac{\texttt{numfracvars}}{\texttt{numvars}}(v_{\max} - v_{\min})$, $\texttt{WV} = m_{\max} + \frac{\texttt{numfracvars}}{\texttt{numvars}}(m_{\max} - m_{\min})$

generate cuts with violation at least `WV`, recording `numcuts` and `maxviolation`

if `numcuts` $<$ `MNC` **then**

`flag` = false

end if

if `maxviolation` $<$ `MRV` **then**

`flag` = false

end if

end loop

any isomorphism inequality generated at a node a will be valid for any other node investigated later in the branch-and-cut.

The next question that arises is how we pick the subsets J. As H^a can be quite large, it does not make sense to test every possible subset. It is our goal to only generate violated cuts; non-violated cuts are useless to us. Clearly, by the nature of the cut, picking variables x_i who have, by the solution of our LP relaxation, value close to 0 will not likely lead to violated cuts. Hence, we pick some δ such that we only consider variables $x_j > \delta$. Because of this, we can construct the following set:

$$H = \{j \mid j \in H^a \text{ and } x_j > \delta\} \subseteq H^a$$

Then, we need only consider all $J \subseteq H$.

We outline the techniques, taken from [22], that we use to create isomorphism cuts in `isomorphism-cuts` (Algorithm 5.2.1) and `isomorphism-cuts-aux` (Algorithm 5.2.2).

Algorithm 5.2.1 isomorphism-cuts(Node a, Table T, Z, x, δ)

$H = \emptyset$
for $i = 0$ to $n - 1$ **do**
  if $x_i > \delta$ **then**
    $H = H \cup \{i\}$
  end if
end for
$S = \emptyset$
$s[-1] = 0$
`isomorphism-cuts-aux`(a, T, Z, x, H, S, s, 1, 0)

We now explain the algorithm. The parameter represents the Schreier-Sims table defining our symmetry group, and Z, the vector used in `first-in-orbit1` (shown in Algorithm 4.2.4). In a fashion identical to that of the `first-in-orbit1` algorithm,

Algorithm 5.2.2 isomorphism-cuts-aux(Node a, Table T, Z, x, Set R, Set S, s, Perm p, i)

for all $j \in R$ **do**
  $S' = S \cup \{p[j]\}$
  $s[i] = s[i-1] + x[p[j]]$
  if $s[i] \leq i$ **then**
    return
  end if
  if $\beta^{-1}[j] \geq Z[i]$ **then**
    output the cut $\mathrm{IC}(S')$
    loop
  end if
  $h = T_{\beta[i],j}$
  if $h \neq 0$ and $i < |F_1^a|$ **then**
    $R' = \{h^{-1}(m) \mid m \in R \setminus \{j\}\}$
    $p' = ph$
    `isomorphism-cuts-aux`(a, T, Z, x, R', S', s, p', $i+1$)
  end if
end for

we maintain a permutation p to test canonicity. As this is done in the same way as `first-in-orbit1`, it is not explained here. The vector s is used in the backtracking to store sums of the variables selected in our set of remaining elements R (initially H); this information allows us to determine whether or not S represents a violated isomorphism inequality.

Essentially, what we do in this backtracking approach is try to pick a variable (according to the above conditions) from R to add to S. If we cannot find a variable that generates a valid isomorphism cut, we simply backtrack. If we are able to extend our subset S, we check it for canonicity, and if it is canonical, we have found an isomorphism cut and add it to the constraints. Note that we no longer try to extend S at this point; if $S' \supset S$ is an extension of S, then the isomorphism cut generated by S automatically implies the cut generated by S', so we need not consider any extensions once we have constructed a violated cut. If, on the other hand, S is not canonical, we try to extend S in order to get a canonical set that generates violated cuts. To do so, we construct a new set R' which only consists of our next valid choices of R, permuted in the same fashion as in `first-in-orbit1`, and proceed.

A rough estimate for the time required by `isomorphism-cuts` is $O(n|H|!)$, but this algorithm typically performs much faster.

One problem with this algorithm is that it does require canonicity testing: for this to be feasible in a reasonable amount of time, we must use the techniques of algorithm `first-in-orbit1`, which requires that 0-fixing be done up to this node. If we opt to turn 0-fixing or canonicity testing off (which is performed by 0-fixing) at any ancestor of this node, we cannot generate isomorphism cuts using these techniques.

5.3 Clique Inequalities

Another family of inequalities that we can exploit are called *clique inequalities*. Given a 0-1 maximization ILP with 0-1 entries in the $m \times n$ coefficient matrix A and constraints

of the form $Ax \leq 1$, we can define the following set for each column i:

$$\mathcal{C}_i = \{j \in \mathbb{Z}_m \mid A[j][i] = 1\} \text{ for } i \in \mathbb{Z}_n$$

We then say that two columns i and j *intersect* if $\mathcal{C}_i \cap \mathcal{C}_j \neq \emptyset$.

From this definition, using the following technique as outlined in [36], we can create a graph $G = (V, E)$ from our ILP:

$$V = \mathbb{Z}_n$$

$$E = \{\{i, j\} \mid \mathcal{C}_i \cap \mathcal{C}_j \neq \emptyset\}$$

Definition 5.3.1. Given a graph $G = (V, E)$, a *clique* of G is a subset $\mathcal{C} \subseteq V$ such that for all $\{i, j\} \subseteq \mathcal{C}$, $\{i, j\} \in E$. A clique $\mathcal{C}$ is said to be *maximal* if there is no $v \in V \setminus \mathcal{C}$ such that $\mathcal{C} \cup \{v\}$ is a clique.

Let $\mathcal{C}$ be any clique in our graph G. Then the following clique inequality holds for the ILP:

$$\sum_{c \in \mathcal{C}} x_c \leq 1$$

To make these inequalities as strong as possible, it is desirable that $\mathcal{C}$ be a maximal clique. Indeed, if $\mathcal{C}$ is a maximal clique, then the above inequality induces a facet of the packing problem [11].

In the separation phase of the cutting plane algorithm, we attempt to generate clique inequalities that are violated by the current solution of the relaxation. Trivially, vertices corresponding to variables with value 0 in the solution will not help us find violated clique inequalities, so we may omit them in our formulation of the graph G. For packings, $x_i = 1$ implies that x_i is not part of a violated clique, so we may ignore all variables with value 1. Hence, we need only consider the graph with vertices corresponding to variables with fractional values in the current solution [16].

Definition 5.3.2. For a graph $G = (V, E)$, we say that two vertices i and j are *connected* if there is a path $\{i, v_0, \ldots, v_n, j\} \subseteq V$ such that $\{i, v_0\} \in E$, $\{v_i, v_{i+1}\} \in E$

for all $i \in \mathbb{Z}_n$, and $\{v_n, j\} \in E$. We trivially define a vertex to be always connected to itself. Then connectedness defines an equivalence relation on V, and we can find a partition $\mathcal{P} = \{C_0, \ldots, C_k\}$ of V. We call the sets C_i the *connected components* of G.

It has been shown in [16] that decomposing a graph into its connected components resulted in large savings for their clique generation algorithms. The connected components of a graph G can be determined using a simple depth-first search (DFS) of the graph. By the definition of a clique, we can subdivide the problem of finding maximal cliques in G to a family of subproblems of finding maximal cliques in each of the connected components of G. However, even in this case, the clique separation problem is known to be NP-hard, and hence, it may only be feasible to search for violated cliques in small connected components; instead, we can use heuristics to attempt to find violated cliques.

We now investigate two algorithms to generate cliques. The first can be used for generic 0-1 maximization packing ILPs and the second is a specialization that exploits the structure of the block formulation of t-$(v, t+1, 1)$ packings and designs.

5.3.1 Generalized clique detection algorithm

We now discuss a general clique detection technique, as proposed in [35] with ideas taken from [16, 36] and outlined in `general-clique-detection` (Algorithm 5.3.1). For connected components with `enumthreshold` or fewer vertices, we search for all maximal cliques in the graph, and otherwise, we use a heuristic given in [36] to find good cliques.

In order to enumerate all maximal cliques of a component, we use a backtracking technique to attempt all valid possibilities and output the cliques that are discovered. In the case that it is not feasible to enumerate all maximal cliques, we employ a greedy heuristic by Nemhauser and Sigismondi. Given a graph $G = (V, E)$ and a vertex $v \in V$,

Algorithm 5.3.1 general-clique-detection(ILP $\mathcal{I}$, `fracvalue`, `enumthreshold`)

generate a graph $G = (V, E)$ representing $\mathcal{I}$

$C = \emptyset$

find $\{C_0, \ldots, C_k\}$, the connected components of G, using DFS

for $i = 0$ to `k` **do**

  if $|C_i| \leq$ `enumthreshold` **then**

    enumerate all maximal cliques of C_i and store in C

  else

    associate a value $v_c = |x_c -$ `fracvalue`$|$ with $c \in C_i$

    call a greedy heuristic on C_i based on nondecreasing value of v_c to get good cliques and store them in C

  end if

end for

return C

we define:

$$N(v) = \{w \mid \{v, w\} \in E\}.$$

We call this set the *neighbourhood of v in G*. Clearly, if C is a clique containing v, $C \subseteq \{v\} \cup N(v)$. The greedy algorithm considers each vertex v in turn, giving precedence to vertices near `fracvalue`, and finds cliques by focusing on the subgraph $\{v\} \cup N(v)$. Moura [35] experimentally showed that in design problems, choosing a value of 0.5 for `fracvalue` was likely to give good cliques, so we selected that as our default.

After the cliques have been generated, if the inequalities implied by one of these cliques is violated sufficiently, we add the clique inequality to our problem.

5.3.2 t-$(v, t+1, 1)$ packing and design clique inequalities

In the case of t-$(v, k, 1)$ packings and designs, if $k = t+1$, we easily know the structure of the graph detailed above and hence, do not need to construct it; clearly, vertices represented by variables x_i and x_j have an edge between them if and only if the k-sets represented by x_i and x_j have exactly t points in common. From this, the structure of the maximal cliques themselves are also easily determined.

Theorem 5.3.1. *(From [35].) For any $t \geq 1$, $v \geq t+3$, $k = t+1$, and $\lambda = 1$, there exists exactly two distinct (up to isomorphism) clique facets for the polytope associated with the packing / design:*

$$\sum_{K \in \binom{\mathbb{Z}_v}{t+1} : T \subseteq K} x_K \leq 1, \quad \textit{for all } T \in \tbinom{\mathbb{Z}_v}{t},$$

$$\sum_{K \in \binom{L}{t+1}} x_K \leq 1, \quad \textit{for all } L \in \tbinom{\mathbb{Z}_v}{t+2}.$$

Cliques of the first type are included in the LP relaxation for the problems, and so they cannot be violated by its solution. Therefore, the only candidates for violated cliques are the ones of the second type.

Example: For a 2-$(7, 3, 1)$ design, if we consider the 4-set $\{0, 1, 2, 3\}$, we have a clique between the following blocks:

$$x_0 = \{0, 1, 2\}, x_1 = \{0, 1, 3\}, x_5 = \{0, 2, 3\}, x_{15} = \{1, 2, 3\}$$

Clearly, no pair of these blocks can appear in a 2-$(7, 3, 1)$ design, as any two of these blocks share a pair in common. Indeed, in the general case, it is obvious that any two distinct k-sets constructed from a $k+1$ set will share $t = k-1$ points in common, and hence cannot both appear in a packing or in a design.

At a given node, we are only interested in generating clique inequalities that cut off the current non-integer solution. As these clique inequalities are of the form:

$$x_{i_0} + x_{i_1} + \ldots + x_{i_k} \leq 1$$

we can derive that, for the inequality to be violated, at least one of the variables on the left hand side must have value at least $\frac{1}{k} = \frac{1}{t+1}$. In our specialized clique enumeration algorithm, we examine all fractional variables with value at least $\frac{1}{k}$ and then consider all $(k+1)$-sets containing the k-set represented by such a variable. This leads us to the procedure proposed in `specialized-clique-detection` (Algorithm 5.3.2).

Algorithm 5.3.2 specialized-clique-detection(Node n, v, k, Soln x, fracbound)

```
lower = max(fracbound, 1/k)
upper = 1 - fracbound
for all x ∈ F^n do
  if x_i ≥ lower and x_i ≤ upper then
    let K be the block represented by x_i
    for all j ∈ Z_v \ K do
      if we have not considered the (k+1)-set K ∪ {j} then
        determine if the maximal clique over all k-subsets of K ∪ {j} leads to a
        violated inequality, and if so, add it to the node
      end if
    end for
  end if
end for
```

Chapter 6

Isomorph-free branch-and-cut for optimization and exhaustive generation

In this chapter, we develop isomorph-free branch-and-cut algorithms for various types of problems. Margot [22, 23, 24] has proposed an isomorph-free branch-and-cut method for solving an ILP, that is, for searching for a single maximum or generating all maximum solutions. We focus on his basic methods proposed in [22], and we extend his techniques to two other types of problems. We also discuss some specifics of our implementation.

Notation: Given an ILP $\mathcal{I}$ over n variables, let $S_{\mathcal{I}} \subseteq \{0,1\}^n$ denote the set of all solutions satisfying the constraints of $\mathcal{I}$, i.e. the feasible solutions of $\mathcal{I}$. For $x \in S_{\mathcal{I}}$, we denote the value of the objective function evaluated at x as $\mathrm{Obj}_{\mathcal{I}}(x)$.

Definition 6.0.3. Let $\mathcal{I}$ be an ILP. Then a solution $x \in S_{\mathcal{I}}$ is said to be *optimal* if, for all $y \in S_{\mathcal{I}}$, $\mathrm{Obj}_{\mathcal{I}}(y) \leq \mathrm{Obj}_{\mathcal{I}}(x)$.

Definition 6.0.4. Let $\mathcal{I}$ be a 0-1 ILP over n variables. We define the following function:

$$\begin{aligned} Q : \{0,1\}^n &\rightarrow \mathcal{P}(\mathbb{Z}_n) \\ x &\mapsto \{i \mid x_i = 1\} \end{aligned}$$

We then say that $\mathcal{I}$ has *set inclusion property* if for all $x \in S_{\mathcal{I}}$ and $y \in \{0,1\}^n$ such that $Q(y) \subseteq Q(x)$, we have that $y \in S_{\mathcal{I}}$ and $\text{Obj}_{\mathcal{I}}(y) \leq \text{Obj}_{\mathcal{I}}(x)$.

Definition 6.0.5. Given a 0-1 ILP $\mathcal{I}$ with set inclusion property and a solution $x \in S_{\mathcal{I}}$, we say that x is *maximal* if there does not exist a solution $y \in S_{\mathcal{I}}$ such that $\text{Obj}_{\mathcal{I}}(x) \leq \text{Obj}_{\mathcal{I}}(y)$ and $Q(x) \subset Q(y)$.

Example: In the context of 2-$(5,3,1)$ packing designs, $x_1 = \{\{0,1,2\}\}$ is a solution, and $x_2 = \{\{0,1,2\},\{0,3,4\}\}$ is a maximum (and maximal) solution. As $Q(x_1) \subset Q(x_2)$ and $\text{Obj}_{\mathcal{I}}(x_1) < \text{Obj}_{\mathcal{I}}(x_2)$, we conclude that x_1 is not a maximal solution.

Definition 6.0.6. In a branch-and-cut tree associated with a problem with v variables, at a node n, we construct $s' = (s'_1, \ldots, s'_v)$, the *partial solution vector* (or partial solution) for node n, as follows:

$$s'_i = \begin{cases} x_i & \text{if variable } i \text{ is fixed at } n \\ 0 & \text{otherwise.} \end{cases}$$

Given an ILP formulation, there are several problems we may be interested in solving. Our framework is designed to deal with the following problem types:

1. Search for a single optimal solution (standard optimization problem)
2. Generation of all optimal canonical solutions
3. Generation of all maximal canonical solutions (with regards to set inclusion)
4. Generation of all feasible canonical solutions

The four algorithms share much in common, and hence, we provide a general framework for solving all problems in `general-bac` (Algorithm 6.0.5). For a more in-depth general branch-and-cut procedure, see `branch-and-cut` (Algorithm 3.3.1).

The basic idea behind `general-bac` is to explore the relevant part of the search space in the branch-and-cut tree associated with the problem. We process the tree in a depth-first fashion: not only does this allow us to find solutions early on (giving us a lower bound for early pruning), but it allows us to minimize the amount of data that we store at each node when we are using the symmetry group algorithms (Sections 4.1 and 4.2).

To represent the nodes that we still need to process, we utilize a stack data structure S with a backtracking approach. Given a node n which is at depth d in the branch-and-cut tree, let a_i represent the ancestor of n at depth i in the tree. The structure of our stack S is then as follows:

$$S = (a_0, a_1, \ldots, a_{d-1}, n)$$

where the right side of the vector represents the top of the stack. In essence, the stack holds, in order of depth, all of the ancestors of the current node n. At any point in our branch-and-cut, to retrieve the next node to be processed from our stack, we use `get-next-node` (Algorithm 6.0.3).

We begin the search by pushing the root node (marked as unprocessed) on the stack. This node will be immediately retrieved by the first pass through the branch-and-cut loop. Every time we branch on a node n, we will push the child node with branching variable x fixed to 1 onto the stack. Thus, we have already extended the stack to account for the subtree with x fixed to 1, and when we return to n via backtracking, provided that n has already been marked as being processed, we know that we need only consider the subtree of n with x fixed to 0 to explore all children of this node completely.

The reason for this stack-based approach is two-fold. First, it allows us to determine

Algorithm 6.0.3 get-next-node(S)

let F^n denote the indices of the free variables at node n

while S is not empty **do**

  $n = \text{top}(S)$

  if we have not processed n **then**

    mark n processed

    return n

  end if

  if the branching variable to get to n was fixed to 0 **then**

    pop(S) and loop

  end if

  $x = \min(F^n)$, the free variable of minimum index

  create n_0, the node with x fixed to 0

  push(S, n_0)

end while

return 0

the next node to be processed quickly, in worst case $\Theta(d)$, where d is the depth of the previously considered node in the tree. Moreover, in most cases, we are able to retrieve the next unprocessed node in constant time, or time significantly better than $O(d)$; indeed, the amortized time for getting the next node is $\Theta(1)$. The second benefit of the stack-based implementation is that it provides us with an easy partial maximality testing technique, which is needed for our third exploration technique (generation of all maximal solutions). This algorithm is not used in the general case and will be outlined below in Algorithm 6.3.2.

The solution of the LP / feasibility check is performed in conjunction with the generation of cuts; this is done in `cutting-plane` (Algorithm 5.1.1). While our current implementation calls an LP solver to check constraint validity, return a bound on the solution, and give us a fractional solution associated with the bound, this could be done via a different technique. We also note that the variable `bestObjectiveValue` may not be relevant in some of the algorithms; however, it is needed for the `cutting-plane` algorithm, and initially assigning it a value of $-\infty$ ensures correct behaviour.

The preprocessing, processing (as performed in `cutting-plane`), and postprocessing methods of the branch-and-cut algorithm do nothing in the generic framework; they are provided to allow users of the package to easily extend the basic algorithm to include problem-specific optimizations, pruning techniques, etc. For example, we use Margot's symmetry group algorithms as one preprocessing enhancement: we use them to both optimize (via 0-fixing) and possibly reject (via canonicity testing). We present `canonicity-test` (Algorithm 6.0.4) which shows how the algorithms of Chapter 4 may be used in order to produce an isomorph-free branch-and-cut. The preprocessing returns `true` if the node is found to be valid, and `false` otherwise.

Given any node a and its base $B = [F_1^a \ F^a \ F_0^a]$, if we maintain an ordering on the sets F_1^a and F_0^a so that the elements in F_1^a appear in order of when they were fixed to 1, and elements in F_0^a appear in reverse order of when they were fixed to 0, then for

Algorithm 6.0.4 canonicity-test(Node a, Group G)

let k be the branching variable index or -1 if this is the root node

let (T, β) be the tree and base associated with G

if we wish to test canonicity at node a **then**

  if 0-fixing has been done at all parent nodes **then**

    `0-fixing`(a, G)

    return `true`

  else

    return `first-in-orbit2`(T, β, k)

  end if

end if

return `true`

Algorithm 6.0.5 general-bac(ILP $\mathcal{I}$)

create a stack $S = \{a_0\}$ with a_0 the root node representing $\mathcal{I}$

`bestObjectiveValue` $= -\infty$

while ($n =$ `get-next-node`(S)) $\neq 0$ **do**

  preprocess n

  let (x^*, b) be the results of a call to `cutting-plane`

  if the LP at n is infeasible or the objective function bounds are not met **then**

    prune n and loop

  end if

  postprocess n

  if there are free variables at n **then**

    determine x, the lowest indexed free variable at n

    push(S, n_1), with n_1 the child node of n where $x = 1$

  end if

end while

any ancestor node b of a, B is also a valid base for b. This is the case because $F_1^b \subseteq F_1^a$, $F_0^b \subseteq F_0^a$, and furthermore, because of the ordering imposed on F_1^a and F_0^a, we can think of our base B as having the form:

$$B = [F_1^b \ (F_1^a \setminus F_1^b) \ F^a \ (F_0^a \setminus F_0^b) \ F_0^b].$$

Since $F^b = F^a \cup (F_1^a \setminus F_1^b) \cup (F_0^a \setminus F_0^b)$, and the ordering on the free variables is not important to the structure of the base, this is obvious. This is rigorously proven in Theorem 4.2.1.

We can exploit this fact in order to use the same group and base data structure for the entire tree. We begin by noting that if the tree were to be processed in another fashion (e.g. breadth-first), it would be necessary to maintain a group and base data structure for each currently unprocessed node: there is no guarantee of any structural relation between bases of nodes located on different branches of the tree. However, due to our depth-first exploration, when we finish processing a subtree and return to some parent node where we branched by fixing variable x_i to 1, we can simply examine the other child tree of this parent node by fixing x_i to 0, shifting x_i in the base by a call to `reverse-down` (Algorithm 4.1.4), and proceeding. We note that we need not consider the other case, namely returning to a parent where we branched by fixing x_i to 0. When we return to such a parent, as by the algorithm we always consider the subtree with x_i fixed to 1 before the subtree with x_i fixed to 0, we can conclude that the entire subtree rooted at this parent has been fully explored, and we simply backtrack further up the branch-and-cut tree.

Alternatively, we can store a group and base data structure per node and simply make a copy of it for child nodes of a node. Since, for any given node n, F^n will always be maintained in ordered form, the lowest indexed variable will always appear in the first position of F^n, removing the need to ever call `reverse-down`.

There are several advantages and disadvantages to both techniques. In storing a single Schreier-Sims table for the entire tree, we dramatically reduce the amount of

memory that might be spent in storing many copies of the symmetry group, but we may require many calls to `reverse-down`, which can be an extremely costly operation due to the number of calls to `enter` that it may require. On the other hand, storing a Schreier-Sims table per node will raise memory costs and there is overhead involved in duplicating the table structure and copying all the permutations, but this guarantees that the free variables are always sorted and removes the need to ever call `reverse-down`. We investigate the efficiency of both of these techniques in Section 8.2.

We now turn our attention to a detailed examination of the techniques used to derive each of the four aforementioned problem types.

6.1 Search for a single optimal solution

In this types of problems, we seek to find only one out of the possibly many optimal solutions of the ILP.

The general algorithm, `search-for-optimal`, is given in Algorithm 6.1.1. This algorithm relies on the solver, which, given a node n, will solve a relaxation of the problem associated with n. The key to this is that the relaxed solution must be no worse than any integer solution that can be found in the subtree rooted at n. If an LP solver like CPLEX is used, for instance, the integrality constraint of the ILP is relaxed and an upper bound b on the objective function for this node is returned.

If s, the solution vector associated with the bound b, is integer, then we have found a vertex of the convex hull associated with the general problem. The subtree rooted at node n may contain other integer solutions whose objective values are also equal to b, but this subtree cannot contain any nodes with integer solutions that exceed b. Hence, at node n, we have found one optimal solution for this subtree, namely s, and as we are only concerned with generating a single optimal solution for the ILP, we need not explore this subtree.

At any node n, if the bound returned by the solver is less than or equal to

`bestObjectiveValue`, the best value found for the objective function by an integer solution thus far, then we know that any solution in the subtree rooted at n will give us a solution no better than the best solution we have found so far. Because of this, we can prune node n immediately.

When all the valid nodes have been processed, an optimal solution is returned if one was found; if the value of `bestObjectiveValue` is still -1, then we have proven that no solution exists for this problem.

Algorithm 6.1.1 search-for-optimal(ILP $\mathcal{I}$)

---

create a stack $S = \{a_0\}$ with a_0 the root node representing $\mathcal{I}$

`bestObjectiveValue` $= -\infty$, `bestSolutionVector` $= 0$

while ($n =$ `getNextNode(`S`)`) $\neq 0$ **do**

  preprocess n

  let (x^*, b) be the results of a call to `cutting-plane`

  if the LP at n is infeasible or $\lfloor b \rfloor \leq$ `bestObjectiveValue` **then**

    prune n and loop

  end if

  if x^* is integer **then**

    `bestObjectiveValue` $= b$, `bestSolutionVector` $= x^*$

    pop(S) and loop

  end if

  postprocess n

  if there are free variables at n **then**

    determine x, the lowest indexed free variable at n

    push(S, n_1), with n_1 the child node of n where $x = 1$

  end if

end while

return (`bestObjectiveValue`, `bestSolutionVector`)

6.2 Generation of all optimal solutions

Whereas in the previous technique, we concerned ourselves solely with finding an optimal solution to the supplied ILP, in this case, we wish to find all optimal solutions to the problem.

We outline the algorithm `generate-optimal` in Algorithm 6.2.1. This algorithm is similar to the algorithm for the generation of a single optimal solution with several modifications. Again, one of the key ideas used in this algorithm involves the solver that is employed to process the relaxation of the problem. As in the previous case, we require that the solver return a bound b such that no solution in the subtree rooted at n gives us an objective value better than b.

As enumerating all solutions for a given ILP involves exploring considerably more of the search space than, say, a generation algorithm might require, our goal is to attempt to prune subtrees that will not yield optimal solutions as quickly as possible. We do this by trying to find an integer solution that will give us a bound for the optimal solution, thus allowing us to discard a large number of subtrees that cannot give optimal solutions. Intuitively, due to the nature of the majority of the 0-1 maximization problems that we will be considering, the objective function will entail some linear sum of the variables in our problem with positive coefficients. Hence, it is reasonable to hypothesize that solution vectors with a large number of variables set to 1 will maximize the value of the objective function. This is largely our motivation in the order of the creation of the subnodes of n: by first exploring the branch n_1 with x (the branching variable) fixed to 1, and then n_0, the node with x fixed to 0, as n_1 has more variables fixed to 1 than does n_0, we suspect that the subtree rooted at n_1 will be more likely to result in a canonical node with high objective value. This branching order is the same in all algorithms, but is of particular importance in this case.

Unlike the search algorithm, we cannot stop processing the subtree rooted at a node n if the partial solution vector s at n is integer; there may be nodes in this subtree

with solution vectors that differ from s, but that are still considered optimal. Thus, we may only use the information generated by the solver to obtain an early bound: if s is integer and gives us a bound that is better than our current bound, then the subtree rooted at n contains at least one optimal solution better than any solution we have seen so far (namely s), and so we clear the set of all optimal solutions as any past stored solutions are not optimal, and we record the new bound b immediately (as opposed to when a solution is recorded at a leaf in the tree) in hopes that we may use it to prune branches of the subtree rooted at n.

We also note that, unlike the search case, if the bound b obtained for a node n of the search space is the same as the best objective value seen so far, we must still consider the node n and its associated subtree as there may be other canonical optimal solutions rooted at n.

When all the relevant nodes of the search space have been explored, the best objective value found and all solutions that provide that objective value are returned. If `bestObjectiveValue` is -1, then we have shown that no valid solution to the ILP exists.

6.3 Generation of all maximal solutions

Given Definition 6.0.5 of maximality, it is our goal to generate all maximal solutions (with respect to set inclusion) of our ILP.

We begin by noting that if x_1 and x_2 are maximal solutions of some ILP $\mathcal{I}$, $|Q(x_1)|$ may not be equal to $|Q(x_2)|$, and likewise there may be no relation between $\text{Obj}_{\mathcal{I}}(x_1)$ and $\text{Obj}_{\mathcal{I}}(x_2)$, so unlike the previous two algorithms, we cannot exploit bounds while exploring the tree.

The question that arises is how can a solution be tested efficiently for maximality? Many problems might demonstrate large sets of solutions, and pairwise testing to see if a new solution contains or is contained in an already discovered solution may not be

Algorithm 6.2.1 generate-optimal(ILP $\mathcal{I}$)

create a stack $S = \{a_0\}$ with a_0 the root node representing $\mathcal{I}$

`bestObjectiveValue` $= -\infty$, `bestSolutionVectorSet` $= \{\}$

while ($n =$ `getNextNode(`S`)`) $\neq 0$ **do**

preprocess n

let (x^*, b) be the results of a call to `cutting-plane`

if the LP at n is infeasible or $\lfloor b \rfloor <$ `bestObjectiveValue` **then**

prune n and loop

end if

if x^* is integer and $b >$ `bestObjectiveValue` **then**

`bestObjectiveValue` $= b$, `bestSolutionVectorSet` $= \{\}$

end if

if there are no free variables at n **then**

`bestSolutionVectorSet` $=$ `bestSolutionVectorSet` $\cup\{x^*\}$

pop(S) and loop

end if

postprocess n

if there are free variables at n **then**

determine x, the lowest indexed free variable at n

push(S, n_1), with n_1 the child node of n where $x = 1$

end if

end while

return (`bestObjectiveValue`, `bestSolutionVectorSet`)

a feasible way to approach this task.

Instead, we implement maximality testing using two different techniques. The first is called *partial maximality testing*, where we use the structure of the node stack itself to reject some non-maximal solutions. For each node n, we have a flag, `possibly-maximal`(n). The value of `possibly-maximal`(root) is set to `true`. Let n' be the parent node of n. The value of this flag for node n is set as follows:

$$\texttt{possibly-maximal}(n) = \begin{cases} \texttt{true} & \text{if we branched with value 1 at } n', \\ \texttt{possibly-maximal}(n') & \text{otherwise.} \end{cases}$$

When we reach a leaf node n in our tree with valid solution s, we immediately traverse backwards through the stack, setting every node's `possibly-maximal` flag to `false`. By the structure of the stack, every node in the stack is an ancestor of n. Given any ancestor m of n, the trivial solution extension of m (i.e. setting all variables in F^m to 0) is clearly contained in the solution s. It is only when we fix a variable in F^m to 1 that we can possibly have a maximal solution, which explains the condition on `possibly-maximal` as detailed above. If we branch at a node, fixing a variable to 0, then the child can only be possibly maximal if the parent is also possibly maximal.

The partial maximality test does not reject all non-maximal solutions: if we consider the set of solutions, listed in order of their discovery by our algorithm,

$$S = \{(1,0,1),(1,0,0),(0,0,1)\} \subset \mathbb{Z}_2^3,$$

then our algorithm will recognize that $(1,0,0)$ is contained in $(1,0,1)$, but it will not recognize that $(0,0,1)$ is contained in $(1,0,1)$). However, it allows us to recognize a large class of non-maximal solutions and reject them without more complete maximality testing.

To supplement this approach, we perform a *complete maximality test* on each solu-

tion. If we are working with n variables, we first define the following function:

$$\begin{aligned} Q^{-1} : \mathcal{P}(\mathbb{Z}_n) &\rightarrow \{0,1\}^n \\ K &\mapsto [x_0, x_1, \ldots, x_{n-1}] \end{aligned}$$

$$\text{where } x_i = \begin{cases} 0 & \text{if } i \notin K, \\ 1 & \text{otherwise.} \end{cases}$$

Our test is dependent on the following condition: Let S be our solution set, and let $s, t \in S$ be two solutions associated respectively with nodes a and b such that $s \neq t$ and s is contained in t. Then, if we consider the set $D = Q(t) \setminus Q(s)$, the set of indices of all variables fixed to 1 in t but not in s, and we pick $d \in D$, then $Q^{-1}(Q(s) \cup \{d\})$ is a solution to our problem. Furthermore, $Q(s) \subset Q(s) \cup \{d\} \subseteq Q(t)$, so s is contained in $Q^{-1}(Q(s) \cup \{d\})$. Because of this assumption, we need only try "flipping" variables in F_0^a to 1 and see if the constraints are still valid. The algorithm `maximality-test` (Algorithm 6.3.1) is derived from this idea.

Algorithm 6.3.1 maximality-test(ILP $\mathcal{I}$, Solution $s \in S \subseteq \{0,1\}^n$)

for all $i \in \mathbb{Z}_n \setminus Q(s)$ **do**
  let $s' = Q^{-1}(Q(s) \cup \{i\})$
  if $\text{Obj}_{\mathcal{I}}(s') < \text{Obj}_{\mathcal{I}}(s)$ **then**
    loop
  end if
  if s' is a feasible solution to $\mathcal{I}$ **then**
    return false
  end if
end for
return true

This algorithm requires us to check for feasibility of s'. This is simple, since we only need to check whether s' satisfied all constraints of the ILP.

Thus, at a node a with solution s, we consider every $x = x_i \in F_0^a$. If flipping x to 1 does not invalidate any of our constraints and positively benefits the objective value associated with the solution, we have found a solution $Q^{-1}(Q(s) \cup \{i\})$ that contains s, so s is not maximal. If t is a maximal solution such that $Q(s) \subseteq Q(t)$, this technique depends fully on the assumption that intermediate subsets via Q between s and t are also valid solutions. If this assumption does not hold, maximality testing becomes extremely time consuming, as with a naive flipping approach, we would need to test $2^{|F_0^a|}$ flips and check a possibly large number of constraints for each flip.

We note that this approach works only for problems such as packings. In the case where we want to minimize the objective function, we would take the opposite approach: for any two solutions s, t, $Q(s) \subseteq Q(t)$ would indicate that t is contained in s. However, it is often simple to formulate a minimization problem of this nature as a maximization problem through a simple change of variables, so we do not provide the framework for this testing. This leads us to `generate-maximal`, as outlined in Algorithm 6.3.2.

Note that in this algorithm, if no lower bound is given, we do not need to solve an LP; we check the constraints directly for feasibility instead. If we have a lower bound, we solve the LP in order to determine if nodes can be pruned. Additionally, the cutting plane method serves no function unless a fractional optimal solution to the LP relaxation is calculated.

6.4 Generation of all feasible solutions

This technique is fairly simple; we want all solutions that satisfy our constraints, regardless of their objective value (i.e. for an ILP $\mathcal{I}$, we want $S_{\mathcal{I}}$). The procedure is more-or-less a simplification of Algorithm 6.2.1 with the bounds checking removed. Pseudocode for this approach is provided in `generate-all` (Algorithm 6.4.1).

Again, as with the enumeration of all maximal solutions algorithm, we simply need

Algorithm 6.3.2 generate-maximal(ILP $\mathcal{I}$)

create a stack $S = \{a_0\}$ with a_0 the root node representing $\mathcal{I}$

`bestObjectiveValue` $= -\infty$, `maximalSolutionVectorSet` $= \{\}$

while ($n =$ `getNextNode(`S`)`) $\neq 0$ **do**

preprocess n

call `cutting-plane` if appropriate (e.g. lower bound specified on problem)

if the LP at n is infeasible **then**

prune n and loop

end if

if there are no free variables at n **then**

if `possibly-maximal`(n) is set to false **then**

pop(S) and loop

end if

set the `possiblyMaximal` flag of all nodes in S to false

let x^* be the solution at node n

if `maximality-test(`$\mathcal{I}$`,` x^*`)` **then**

`maximalSolutionVectorSet` $=$ `maximalSolutionVectorSet` $\cup\{x^*\}$

pop(S) and loop

end if

end if

postprocess n

if there are free variables at n **then**

determine x, the lowest indexed free variable at n

push(S, n_1), with n_1 the subnode of n where $x = 1$

end if

end while

return `maximalSolutionVectorSet`

Algorithm 6.4.1 generate-all(ILP $\mathcal{I}$)

create a stack $S = \{a_0\}$ with a_0 the root node representing $\mathcal{I}$

`bestObjectiveValue` $= -\infty$, `solutionVectorSet` $= \{\}$

while ($n =$ `getNextNode(`S`)`) $\neq 0$ **do**

preprocess n

let (x^*, b) be the results of a call to `cutting-plane`

if the LP at n is infeasible **then**

prune n and loop

end if

if there are no free variables at n **then**

`solutionVectorSet` $=$ `solutionVectorSet` $\cup\{x^*\}$

pop(S) and loop

end if

postprocess n

if there are free variables at n **then**

determine x, the lowest indexed free variable at n

push(S, n_1), with n_1 the child node of n where $x = 1$

end if

end while

return `solutionVectorSet`

to check whether or not the constraints are violated by any of our fixings, and if so, we prune the current node.

Chapter 7

The NIBAC package: overview and user options

Our branch-and-cut package, called NIBAC (Non-Isomorphic Branch-And-Cut), provides an easily programmable C++ framework for formulating combinatorial problems as ILPs and then solving them either using pure branch-and-cut techniques or an isomorph-reduced or isomorph-free branch-and-cut tree which exploits the symmetry group.

NIBAC has been written in a style that may be processed by the doxygen documentation system, and references, including class descriptions and diagrams may be found in the `doc` subdirectory of the project in HTML and LaTeX formats. The source code may be found in `src` with a `Makefile` which is set up for Sun Solaris and must be edited for your architecture.

A brief summary of the classes and packages is presented below. UML class diagrams are given in Figures 7.1, 7.2, 7.3, and 7.4. Note that, because the class diagram is large and has been spread over several pages, some of the "has-a" relationships are not explicitly presented; these should, however, easily be inferred.

- `BAC`

  The necessary framework for a branch-and-cut. This is the implementation and related data structures of the algorithms in Chapter 6.

- `BACOptions`

  A set of options for the branch-and-cut. This class allows users to specify to what depth NIBAC should process until hands the problem to the ILP solver, along with holding values for various parameters like which cut producers will be used, bounds on the objective function, etc. It provides an `initialization` method that, given the command line arguments passed to `main`, will parse command line options automatically and return a modified command line argument list and count to the calling code.

- `bitstring`

  This class is not intended for users of the framework and is provided for use in `find-symmetry-group2` (Algorithm 4.4.2).

- `Block`

  A convenience class to represent blocks from final solutions.

- `BlockGroup`

  A `SchreierSimsGroup` that acts as the symmetric group over a set of points on variables representing blocks of the problem.

- `CliqueCutProducer`

  A `CutProducer` that finds violated clique inequalities as specified in Section 5.3.

- `Column`

  This class is not intended for users of the framework. It is a component of the `Formulation` class and represents a column of our ILP.

- `ConstraintC`

  A constraint for our problem. As problems manage their own constraints, users are expected to initialize a variable of type `Constraint` (a pointer to a `ConstraintC`) using the provided static methods, and then add the `Constraint` to either the `Formulation` (for a globally valid constraint) or a `Node` (for a constraint valid for a subtree rooted at the node).

- `CPLEXSolver`

  An implementation of `LPSolver` using CPLEX 7.x or 8.x.

- `CutProducer`

  Abstract interface for classes that generate cuts. `BACOptions` holds a `vector` of `CutProducers` that will be used by `BAC` in the cutting-plane algorithm.

- `DefaultSolutionManager`

  A simple, memory-based `SolutionManager` that takes care of all solutions reported and only maintains those that fit the problem type and criteria (with regards to things like maximality, optimality, etc.).

- `Formulation`

  The ILP formulation of the problem. Users are expected to create their ILP using this class by supplying it with an objective function and a set of instances of `Constraint`.

- `GeneratedGroup`

  An abstract subclass of `Group` which encompasses methods used for a group that is represented by a set of generators. The `find-symmetry-groupN` family of algorithms (Algorithms 4.4.1, 4.4.2, and 4.4.3) are presented here.

- `Graph`

  Not intended for users of the framework. This is an internal class used by `CliqueCutProducer` to hold the graph associated with the ILP.

- `Group`

  Abstract superclass for a symmetry group.

- `IsomorphismCut`

  Not intended for users of the framework. This class represents a linked list of isomorphism cuts, written in order to efficiently maintain a list of minimal and unique isomorphism cuts during isomorphism cut production. Used by `IsomorphismCutProducer`.

- `IsomorphismCutProducer`

  Producer of isomorphism cuts.

- `LPSolver`

  Abstract interface to which an LP solver must adhere to be used by NIBAC. Currently, a concrete implementation that uses CPLEX (`CPLEXSolver`) is provided, but other LP solving packages could easily be used as an alternative.

- `MargotBAC`

  A subclass of `BAC` that defines the `preprocess` method by employing Margot's algorithms as described in Section 4.2. For isomorph-free branch-and-cut, or isomorph-reduced branch-and-cut, this is the class that should be used.

- `MargotBACOptions`

  This class extends the functionality of the `BACOptions` class by allowing users to specify depths and criteria for canonicity testing.

- `matrix`

  A package of C++ routines for allocating and deallocating two or three dimensional arrays of arbitrary types.

- `MatrixGroup`

  A `SchreierSimsGroup` that represents all the row and / or column permutations over an $m \times n$ matrix of variables, as in an incidence matrix style formulation.

- `Node`

  This class is not intended for use by programmers of the framework. It encapsulates the data needed by the branch-and-cut and related algorithms for a single node of the search tree.

- `NodeStack`

  This class is not intended for use by programmers of the framework. It stores a stack of `Node`s and implements the `get-next-node` algorithm (Algorithm 6.0.3).

- `PermutationPool`

  This class manages memory for permutations (arrays of `int`) by offering either a static pool or dynamic allocation.

- `Problem`

  The main class of interest for users of the framework. It instantiates and initializes objects needed by BAC and presents a set of methods that must be overridden that allow users to define the ILP, determine fixings, set up the cut producers, etc.

- `SchreierSimsGroup`

  A concrete implementation of `GeneratedGroup` that uses the algorithms of Section 4.1 to represent a group. If users intend to create their own groups without

using `BlockGroup` or `MatrixGroup`, they should instantiate an instance of this class and enter permutations into it; a minimal set of generators will suffice.

- `SolutionManager`

  An abstract interface for a class used to process solutions. A concrete implementation is provided with `DefaultSolutionManager`.

- `Statistics`

  A set of statistics maintained about the branch-and-cut process. These are held in, and accessible through the `BACOptions` class. They need not be instantiated by the user, and come with convenience printing routines in order to output statistics to an output stream.

- `Timer`

  A millisecond-precise stopwatch for measuring time elapsed.

The basic technique to formulate a problem is to extend the `Problem` class. The methods of this class, as detailed in the doxygen documentation, can be overridden to construct the formulation, determine the symmetry group, set up the cut producers, etc. Examples of implementations of the abstract `Problem` class can be found for all of the problems in Chapter 2, as well as for the generation and enumeration of cyclic basis representations of subspaces of dimension two over the $\mathbb{F}_2$-vector space $\mathbb{F}_2^n$.

A full user manual will be released with the package in PDF format in the `doc` subdirectory along contact information for the author regarding questions and bugs.

For the remainder of this chapter, we examine the various parameters that may be altered in order to affect the performance of the branch-and-cut and how these relate to the algorithms previously discussed.

Figure 7.1: Class diagram (part one of four) for the NIBAC package.

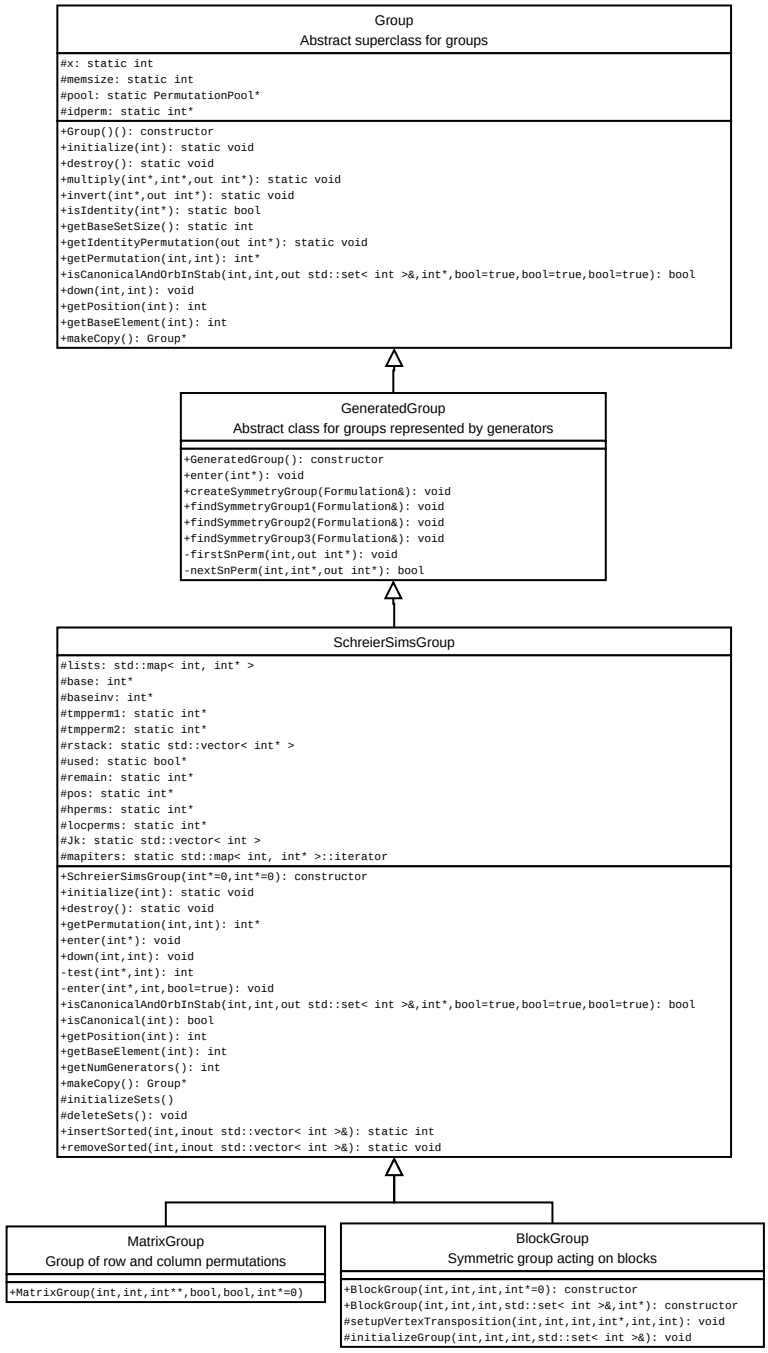

Figure 7.2: Class diagram (part two of four) for the NIBAC package.

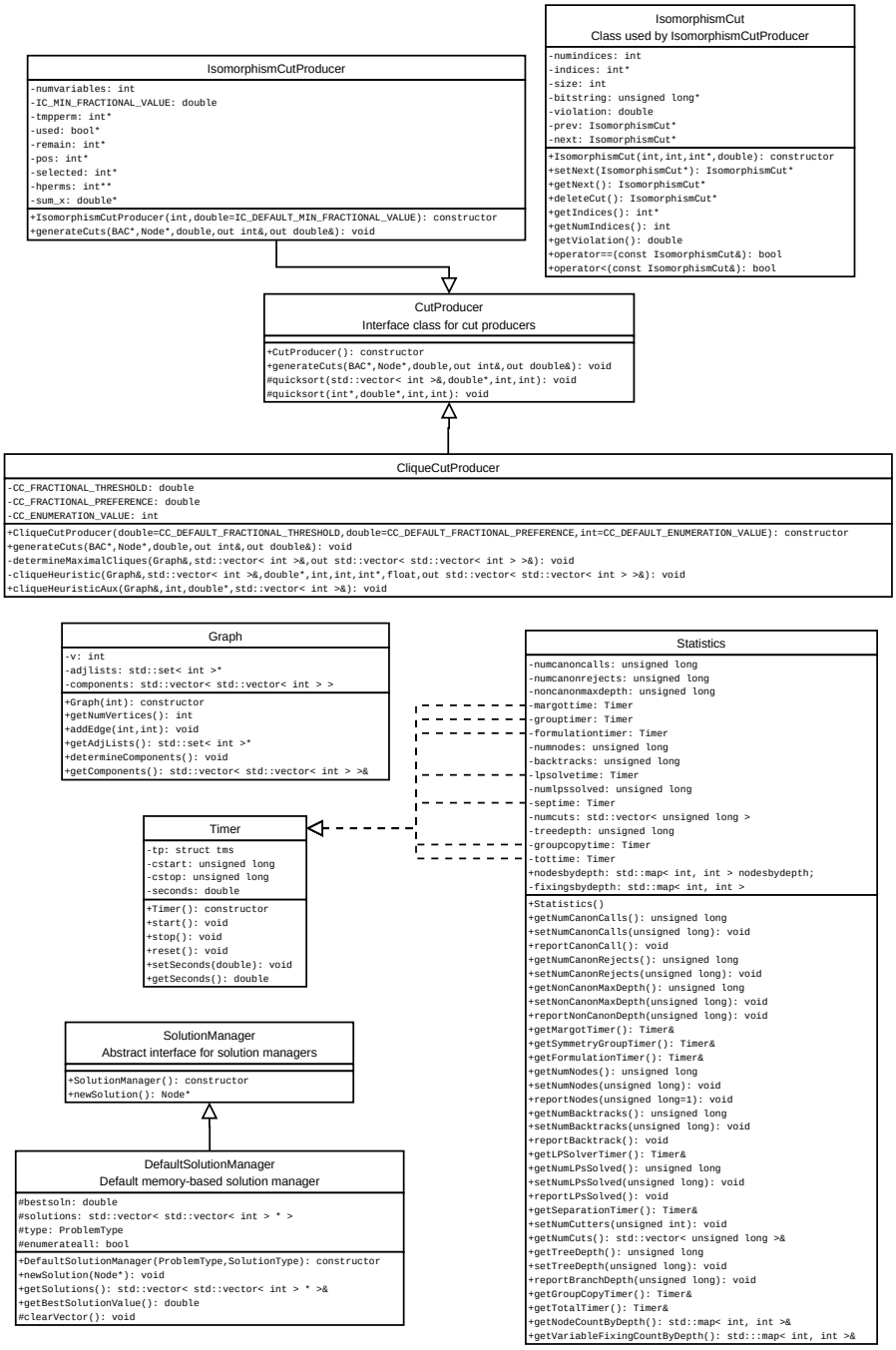

Figure 7.3: Class diagram (part three of four) for the NIBAC package.

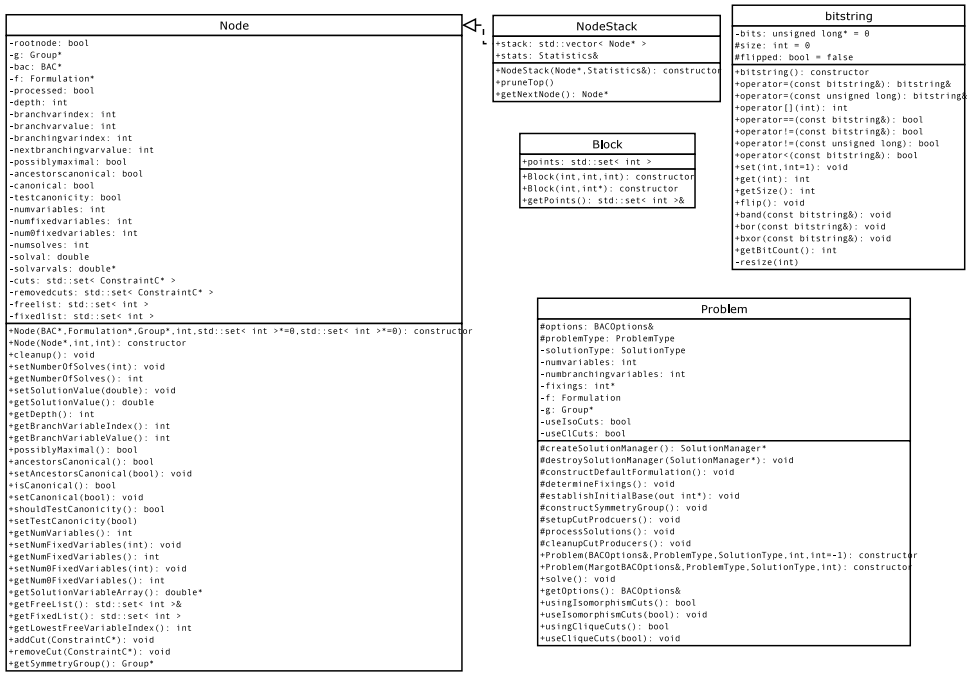

Figure 7.4: Class diagram (part four of four) for the NIBAC package.

7.1 Constants and variable parameters

In order to optimize the branch-and-cut framework, a number of user-adjustable parameters have been provided. These are outlined in the following sections.

7.1.1 Constants

Constants are defined in the file `common.h` and must be redefined at compile time if they are to be changed.

`EPSILON`

Default: 10^{-3}

A real number close to 0 used in floating point calculation. A floating point number x is considered to be equal to y if $|x - y| \leq$ `EPSILON`. This constant is also used in determining integrality.

7.1.2 Variable parameters

Variable parameters are either specified in a constructor of a class or as a setting in the `BACOptions` class (or a subclass thereof) when it makes sense to do so. We proceed to detail the various parameters here and their default values.

Branch-and-bound parameters

- `BB_LBOUND`

  Set via: `BACOptions`

  Default: `INT_MIN`

  This parameter allows the user to specify a lower bound on the optimal value of the solution if one is known. This allows early pruning of branches from the tree in a maximization problem, and in a minimization search problem, we know that a solution is optimal and terminate if we ever achieve this bound.

- BB_UBOUND

  Set via: BACOptions

  Default: INT_MAX

  This parameter allows the user to specify an upper bound on the optimal value of the solution if one is known. This allows early pruning of branches from the tree in a minimization problem, and in a maximization search problem, we know that a solution is optimal and terminate if we ever achieve this bound.

- BB_DEPTH

  Set via: BACOptions

  Default: BAC::BB_NEVER

  This parameter specifies the depth of our branch-and-cut tree in which we stop the branch-and-cut algorithm and hand the ILP to the LPSolver class. The LPSolver class must solve it fully at that level if such behaviour is supported. In the case of the default implementation, this consists of asking CPLEX to solve an ILP instead of the LP relaxation. For the default value, BB_NEVER, branch-and-cut continues at arbitrarily high depths.

Cutting plane parameters

- CP_MIN_NUMBER_OF_CUTS

  Set via: BACOptions

  Default: BAC::CP_DEFAULT_MIN_NUMBER_OF_CUTS = 5

  In order to continue running the cutting plane algorithm, we need to produce at least this many cuts meeting our criteria in a single iteration.

- CP_VIOLATION_TOLERANCEL

  Set via: BACOptions

  Default: BAC::CP_DEFAULT_VIOLATION_TOLERANCEL = 0.3

This is the parameter $v_{\min}$, as outlined in Algorithm 5.1.1.

- `CP_VIOLATION_TOLERANCEU`

  Set via: `BACOptions`

  Default: `BAC::CP_DEFAULT_VIOLATION_TOLERANCEU` = 0.6

  This is the parameter $v_{\max}$, as outlined in Algorithm 5.1.1.

- `CP_MIN_VIOLATIONL`

  Set via: `BACOptions`

  Default: `BAC::CP_DEFAULT_MIN_VIOLATIONL` = 0.3

  This is the parameter $m_{\min}$, as outlined in Algorithm 5.1.1.

- `CP_MIN_VIOLATIONU`

  Set via: `BACOptions`

  Default: `BAC::CP_DEFAULT_MIN_VIOLATIONU` = 0.6

  This is the parameter $m_{\max}$, as outlined in Algorithm 5.1.1.

- `CP_ACTIVITY_TOLERANCE` Set via: `BACOptions`

  Default: `BAC::CP_DEFAULT_ACTIVITY_TOLERANCE` = 0.1

  A cut $ax \leq b$ is considered active if $b - ax \leq$ `CP_ACTIVITY_TOLERANCE`. Inactive cuts are not used in a node or in subnodes: they are added when we return to the parent of the node in which they were determined to be inactive if the `CP_KEEP_CUTS` variable is set to `TRUE` (default) in the `BACOptions` class. The ability to permanently discard cuts is offered to reduce memory consumption.

Canonicity testing and 0-fixing parameters

- Canonicity flag string

  Set via: `MargotBACOptions`

  Default: "-", i.e. always

A user-supplied string that determines at what depths canonicity testing should be performed in the branch-and-cut. The string consists of ranges of the form "*a*-*b*", where all depths in range between a and b inclusive are tested for canonicity. If a is omitted, 0 is assumed. If b is omitted, `INT_MAX` (i.e. every depth greater than a) is assumed. Multiple ranges may be specified by separating the ranges with commas. Note that in the case of any enumeration, canonicity testing will always be performed on complete solutions to ensure that an isomorph-free catalogue is output.

- Orbit flag string

  Set via: `MargotBACOptions`

  Default: "-", i.e. always

  A user-supplied string that determines at what depths 0-fixing should be performed in the branch-and-cut. The string consists of ranges of the form "*a*-*b*", where all depths in range between a and b inclusive are tested for canonicity. If a is omitted, 0 is assumed. If b is omitted, `INT_MAX` (i.e. every depth greater than a) is assumed. Multiple ranges may be specified by separating the ranges with commas.

Isomorphism cut producer parameters

- `IC_MIN_FRACTIONAL_VALUE`

  Set via: `IsomorphismCutProducer` constructor

  Default: `IsomorphismCutProducer::IC_DEFAULT_MIN_FRACTIONAL_VALUE` $= 0.5$

  At a node a, we have the set $H^a = F_1^a \cup F^a$. In the interests of efficiency and likelihoods of generating isomorphism cuts, we restrict our consideration to the set $H' = \{i \in H^a \mid x_i \geq \texttt{IC\_MIN\_FRACTIONAL\_VALUE}\}$. The default value for this parameter was proposed in [22].

Clique cut producer parameters

- CC_FRACTIONAL_THRESHOLD

  Set via: CliqueCutProducer constructor

  Default: CliqueCutProducer::CC_DEFAULT_FRACTIONAL_THRESHOLD = 0.15

  For clique cuts, in the interests of efficiency and producing valid cliques, we only consider variables x such that CC_FRACTIONAL_THRESHOLD $\leq x \leq 1-$CC_FRACTIONAL_THRESHOLD.

- CC_ENUMERATION_VALUE

  Set via: CliqueCutProducer constructor

  Default: CliqueCutProducer::CC_DEFAULT_ENUMERATION_VALUE = 20

  When generating cliques, if a connected component has CC_ENUMERATION_VALUE or fewer vertices, we enumerate all maximal cliques: otherwise, we rely on the heuristic. The default value for this parameter was proposed in [16].

- CC_FRACTIONAL_PREFERENCE

  Set via: CliqueCutProducer constructor

  Default: CliqueCutProducer::CC_DEFAULT_FRACTIONAL_PREFERENCE = 0.5

  When generating cliques using the heuristic, we give priority to vertices that are near to CC_FRACTIONAL_PREFERENCE.

Chapter 8

Experimental results and case studies

We now proceed to investigate the performance of the NIBAC framework and use it to solve some combinatorial problems. Using his framework, Margot was able to address a number of different problems: showing nonexistence of a 4-$(10, 5, 1)$ covering of size 50 [22, 24]; generating all canonical 4-$(10, 5, 1)$ coverings of size 51 [24]; searching for a single optimal solution for a number of different covering problems [23]; finding an optimal error correcting code $A(8, 3)$ [22], $A(9, 4)$, and $A(10, 5)$ [23]; and solving several set covering problems derived from Steiner triple systems [23].

In this chapter, we test the problems detailed in Chapter 2 using the four techniques outlined in Chapter 6, thus examining and expanding the applicability of isomorph-free branch-and-cut to a larger variety of combinatorial problems. Additionally, we investigate how specific variations of the algorithm affect performance: for example, we study the differences in efficiency using one group representation for the entire tree versus one group representation per node, as proposed in Chapter 6.

We begin by establishing estimates on good values for several of the parameters for NIBAC algorithms. After these have been determined, we investigate the effects

of different types and combinations of cuts. Once we have determined which NIBAC configuration we will use by default, we perform several case studies on the problems of interest defined in Chapter 2.

We use the bounds given in Theorems 2.1.1, 2.1.2, and 2.1.3. For all t-(v, k, λ) design problems, we set the upper and lower bounds:

$$\texttt{BB\_LBOUND} = \texttt{BB\_UBOUND} = \frac{\lambda\binom{v}{k}}{\binom{v}{t}}.$$

If $\lambda = 1$, we also use the standard fixing of $\binom{v-1}{k-1} + 1$ variables as described in Section 2.3; otherwise, we fix only the first block. For all t-(v, k, λ) packings, we set the upper bound given by the First Johnson bound in Theorem 2.1.2:

$$\texttt{BB\_UBOUND} = \left\lfloor \frac{v}{k} \left\lfloor \frac{v-1}{k-1} \cdots \left\lfloor \frac{\lambda(v-t+1)}{k-t+1} \right\rfloor \right\rfloor \right\rfloor.$$

For t-(v, k, λ) coverings, we use the lower bound provided by the Schönheim bound:

$$\texttt{BB\_LBOUND} = \left\lceil \frac{v}{k} \left\lceil \frac{v-1}{k-1} \cdots \left\lceil \frac{\lambda(v-t+1)}{k-t+1} \right\rceil \right\rceil \right\rceil.$$

In both packing and covering problems, we also use the standard fixing of 1 variable as described in Section 2.3.

The tests were run on a Sun Fire V880 with eight UltraSPARC III Cu 900 MHz processors and 32 GB of RAM running Solaris 9. The programs were compiled with the Sun Forte Developer 7 C++ compiler, and for the LP solver (and ILP solver, in some instances as detailed below), CPLEX 7.0 was used.

A legend of the headings used in the results tables is provided in Table 8.1. We note that results that are new are indicated in the tables in bold face.

B	The number of backtracks in the node stack.
C	The number of calls to the fast canonicity algorithms.
CC	The number of clique cuts produced by the cutting plane.
CPs	Types of cut producers used: i = isomorphism, s = specialized cliques, g = general cliques.
CR	The number of canonical rejections.
CT	Time spent in fast canonicity testing / 0-fixing (in seconds).
CV	Variation of the clique cut producer used: SC for specialized clique producer, GC for general clique producer.
#D	The number of forward (regular) `down` operations performed.
D	The highest depth amongst all processed nodes.
DT	Time spent performing forward `down` operations.
#GC	The number of group copies performed by the branch-and-cut.
GCC	The number of general clique cuts produced by the cutting plane.
GCT	The time spend copying groups.
GT	Time spent in the Schreier-Sims group algorithms.
N	The number of nodes explored by the tree.
IC	The number of isomorphism cuts produced by the cutting plane.
LP	The number of LPs solved during execution.
LPT	Time spent solving linear programs (in seconds).
#RD	The number of `reverse-down` operations performed.
RDT	Time spend performing `reverse-down` operations.
SC	The number of calls to the slow canonicity algorithm.
SCC	The number of special clique cuts produced by the cutting plane.
SCT	Time spent in slow canonicity testing.
ST	Time spent in the separation (in seconds).
T	Total time spent in branch-and-cut (in seconds).

Table 8.1: Description of the columns in the tables of results.

8.1 Setting up the default NIBAC configuration

We begin by determining a good configuration for the NIBAC parameters through experimentation on several problems. After this has been established, we will use this configuration when dealing with the case studies in the next section.

8.1.1 The effects of varying $[v_{\min}, v_{\max}]$ and $[m_{\min}, m_{\max}]$

In Section 5.1, we describe the use of the parameters $[v_{\min}, v_{\max}]$ and $[m_{\min}, m_{\max}]$ in the cutting plane algorithm. It was shown in [35] that the values $[0.3, 0.6]$, $[0.3, 0.6]$ performed best for the cuts and problems that were used in that particular branch-and-cut implementation. We will examine how these parameters impact our branch-and-cut package. Because during search for the 2-$(v, 3, 1)$ designs that we attempted, no isomorphism cuts were generated for the parameter ranges attempted in [35], we included a smaller range in the experimental values in hopes that we might find such a cut. We test various combinations of parameters as detailed in Table 8.2 for several different problems: block formulation designs and packings both in the search for one structure (Table 8.3) and generation of all optimal structures (Table 8.4). The results are similar for other designs and packings, even if the parameter λ is varied.

p#	$v_{\min}$	$v_{\max}$	$m_{\min}$	$m_{\max}$
1	0.01	0.01	0.01	0.01
2	0.1	0.1	0.1	0.3
3	0.1	0.1	0.3	0.3
4	0.3	0.3	0.3	0.3
5	0.3	0.6	0.3	0.6

Table 8.2: Parameter combinations for the cutting plane algorithm.

Table 8.5 demonstrates the effects of varying these parameters in the generation of a single 2-$(13, 3, 1)$ design using the incidence matrix formulation. The lowest execution time is reported when the cutting plane algorithm is turned off completely. Other 2-(v, k, λ) designs are either generated or enumerated too quickly to provide information or are infeasible for testing purposes using this technique.

In some cases, the times for the separation algorithm may seem very high: reasons for this are discussed in the following section. In the meantime, we observe that for all attempted parameter intervals higher than $[0.3, 0.6]$, no cuts are obtained in all examples: the value $[0.6, 0.6]$ generated no cuts in all cases, thus making the cutting plane algorithm useless.

From the examples, it is not easy to determine the best values for the cutting plane parameters; clearly, there is no advantage to $[0.01, 0.01]$ as we still do not generate any isomorphism cuts for the majority of generation problems. Intuitively, this makes sense, as there is no backtracking in the branch-and-cut tree, indicating that we are only creating a canonical set representing the solution. In the case of incidence matrices, the cost of the separation does not justify its benefits.

In light of these experiments, we will, from this point forward for the general case, proceed to use the values $[0.3, 0.6]$ as per [35] for both ranges, as there are certain instances where each interval performs well and this interval demonstrates reasonably low times in most cases.

v	p#	N	LPT	LP	ST	IC	CV	CC	T
15	-	15	0.12	15	-	-	-	-	17.99
	1	17	0.08	24	25.60	0	SC	24	31.67
		9	0.18	24	8.18	0	GC	52	16.40
	2	17	0.16	23	22.39	0	SC	15	44.63
		15	0.16	28	19.54	0	GC	42	37.67
	3	17	0.14	23	22.56	0	SC	15	44.77
		14	0.18	26	14.99	0	GC	38	30.84
	4	13	0.10	16	9.74	0	SC	3	23.63
		13	0.13	18	10.16	0	GC	7	23.97
	5	13	0.08	15	15.69	0	SC	2	29.33
		10	0.11	15	6.59	0	GC	9	15.72
19	1	29	0.87	49	3935.44	0	SC	86	11789.40
		28	1.14	70	4864.78	0	GC	191	12271.30
	2	30	0.77	46	3739.90	0	SC	57	12349.40
		29	1.34	71	5339.79	0	GC	180	13195.10
	3	28	0.70	41	3078.49	0	SC	48	10490.70
		29	0.96	65	4271.28	0	GC	148	12121.10
	4	29	0.74	38	3213.54	0	SC	11	11066.80
		28	0.85	48	3170.27	0	GC	49	10587.20
	5	29	0.73	36	3170.85	0	SC	8	11035.20
		30	0.90	44	3434.46	0	GC	28	12055.00

Table 8.3: Search for several 2-$(v, 3, 1)$ designs using the block incidence formulation for determining good cutting plane algorithm parameters. There are never any backtracks in the tree.

v	p#	N	B	LPT	LP	ST	IC	CV	CC	T
11	-	319	159	0.68	319	-	-	-	-	9.82
	1	197	98	0.69	235	0.94	415	SC	31	6.31
		183	91	0.73	254	0.86	237	GC	190	5.99
	2	259	129	0.67	295	0.87	157	SC	28	9.17
		193	96	0.74	260	0.78	130	GC	167	5.93
	3	259	129	0.84	295	0.94	157	SC	28	9.23
		193	96	0.57	260	0.82	130	GC	167	5.98
	4	263	131	0.64	285	0.83	113	SC	12	9.43
		191	95	0.56	234	0.75	297	GC	86	5.74
	5	261	130	0.60	288	0.83	260	SC	8	9.54
		247	123	0.66	292	0.73	126	GC	69	8.76
12	-	6701	3350	19.23	6701	-	-	-	-	695.43
	1	5523	2761	17.21	6151	48.00	3997	SC	59	660.28
		3835	1917	13.57	4631	25.20	2903	GC	1751	484.44
	2	1793	896	8.83	2313	29.50	3816	SC	49	148.64
		3871	1935	14.18	4651	24.73	3131	GC	1734	489.86
	3	1793	896	8.37	2311	29.91	3834	SC	59	148.29
		3909	1954	24.56	4690	24.56	2924	GC	1723	491.47
	4	1829	914	8.36	2288	28.73	3780	SC	31	148.09
		4691	2345	15.65	5428	27.16	2967	GC	1146	579.80
	5	1847	923	8.39	2283	27.86	3424	SC	18	146.65
		4677	2338	14.98	5394	26.37	3179	GC	1058	576.44

Table 8.4: Generation of all optimal 2-$(v, 3, 1)$ packings for determining good cutting plane algorithm parameters.

p#	N	B	LPT	LP	ST	IC	T
-	754	344	189.85	754	-	-	30949.0
1	739	338	179.45	807	12488.3	348	42496.6
2	736	338	179.83	801	12207.9	305	42006.2
3	736	338	179.80	801	12205.9	305	42001.7
4	739	338	181.70	785	11816.8	136	41816.3
5	742	338	174.81	792	12128.5	198	42471.8

Table 8.5: Search for a 2-$(13, 3, 1)$ design using the incidence matrix formulation for determining good cutting plane algorithm parameters.

8.1.2 The effects of different types of cuts

Tables 8.3, 8.4, and 8.5 demonstrate some large times for the cutting plane algorithm. We investigate this phenomenon here to determine its causes.

We begin by examining the statistics by choosing different cut producers for the search problem for a 2-$(19, 3, 1)$ design (Table 8.6) and the optimal generation problem for 2-$(12, 3, 1)$ packings (Table 8.7). We report on combinations of isomorphism cuts (i), specialized clique generation (s), and general clique generation (g).

CPs	N	LPT	LP	ST	IC	GCC	SCC	T
-	30	0.59	30	-	-	-	-	8645.10
g	30	0.92	44	0.02	-	28	-	8629.89
s	29	0.73	36	0.02	-	-	8	7862.46
gs	29	0.86	52	0.09	-	31	7	7852.21
i	30	0.62	30	3368.60	0	-	-	11984.60
ig	30	0.96	44	3431.93	0	28	-	12053.80
is	29	0.68	36	3168.15	0	-	8	11031.60
igs	29	1.00	52	3244.26	0	31	7	11098.30

Table 8.6: The effects of using different cut producers on the search for an optimal 2-$(19, 3, 1)$ design.

It is clear that in certain cases, as with the 2-$(19, 3, 1)$ design search, the high amount of time spent generating isomorphism cuts is not worthwhile: indeed, in this case, no isomorphism cuts are generated, so there is no effect on canonicity. This is due to the nature of the isomorphism cut generation algorithm. The backtracking approach that we use to create lexicographically smaller sets is quite intensive when the number of variables and the size of the symmetry group are large.

In the case of the 2-$(12, 3, 1)$ packing generation, the isomorphism cuts are obviously

CPs	N	LPT	LP	ST	IC	GCC	SCC	T
-	6973	19.28	6973	-	-	-	-	717.98
g	5283	17.20	6010	0.35	-	1241	-	591.91
c	2977	10.52	2991	0.24	-	-	20	189.60
gc	1849	9.06	2444	0.45	-	1054	16	144.44
i	5547	16.96	6052	44.74	3586	-	-	650.93
ig	4677	15.03	5394	26.41	3179	1058	-	574.46
ic	1847	8.44	2283	27.53	3424	-	18	146.09
igc	1395	7.52	1959	18.77	3127	853	17	112.92

Table 8.7: The effects of using different cut producers on the generation of all optimal 2-$(12, 3, 1)$ packings.

beneficial: despite the large amount of time spent in the separation, they decrease the number of nodes by 10% to 37% depending on the other types of cuts selected, and always reduce overall run time, sometimes significantly (by up to 22%).

In both examples, it is always beneficial to use both general clique cuts and the specialized clique cuts when possible. In the enumeration of all maximal $(9, 5, 3)$ intersecting set systems, cuts do not assist the problem. While the cutting plane algorithm takes no more than 0.05 seconds in all cases, the execution time with cuts enabled rises from 1401.77 s to 1472.96 s (or by approximately 5%). As this increase is negligible, we are not concerned by enabling all cuts in the general case.

From the above tables, it is clear that generating isomorphism cuts can increase the separation time for a problem. We proceed to investigate whether or not they are advantageous, and if so, to what types of problems.

As we have seen in Table 8.3, isomorphism cuts do not appear to be generated for small v in the search problems for 2-$(v, 3, 1)$ designs, so we no longer investigate those here and conclude that isomorphism cuts are detrimental in these cases.

We examine several different design generations in Table 8.8 and packing enumerations in Table 8.9. Each of these formulations was run for generation as well, and none of them reported an isomorphism cut in this case; however, the separation never took more than a fraction of a second for each problem.

v	k	λ	IsoCuts	N	ST	IC	CT	C	CR	T
7	3	2	off	147	-	-	0.10	323	108	0.43
			on	133	0.05	10	0.09	290	96	0.43
7	3	3	off	637	-	-	0.64	1454	511	2.55
			on	559	0.17	42	0.72	1256	430	2.45
7	3	4	off	3423	-	-	5.21	7357	2263	16.42
			on	3011	1.51	118	4.93	6434	1958	16.32
7	3	5	off	17649	-	-	50.73	34607	8284	112.65
			on	15849	14.16	382	46.84	30819	7196	119.07
8	4	3	off	163	-	-	1.02	547	307	2.71
			on	159	0.20	4	0.96	522	288	2.89
9	3	2	off	3179	-	-	18.93	8677	3960	33.67
			on	2349	4.13	615	12.11	5928	2453	28.13
9	4	3	off	1717	-	-	9.10	6996	4439	35.87
			on	1419	1.14	107	7.19	5438	3325	30.66
10	4	2	off	131	-	-	1.62	638	445	7.52
			on	131	0.37	13	1.67	638	445	8.08

Table 8.8: The effects of isomorphism cuts on several 2-(v, k, λ) design generations. No cuts were generated for the 2-$(11, 5, 2)$, and 2-$(13, 4, 1)$ design generations, nor did the separation take more than 0.01 s, so these results are omitted.

It seems that it is difficult to predict whether or not isomorphism cuts will assist the problem. In the examples studied, in the worst case, execution time increased by

v	IsoCuts	N	ST	IC	CT	C	CR	T
10	off	163	-	-	1.04	733	512	2.10
	on	151	0.23	106	0.91	634	430	2.24
11	off	319	-	-	6.71	1272	839	9.77
	on	275	0.81	244	5.93	1023	654	9.82
12	off	6973	-	-	641.33	43680	33904	715.17
	on	5591	44.16	3893	545.76	33942	26210	654.36

Table 8.9: The effects of isomorphism cuts on several 2-$(v, 3, 1)$ packing generations.

7.74%. The greatest reduction in execution time was 16.99%. In exactly half of the test cases where the execution time was affected by isomorphism cuts, the time dropped. In the instances where isomorphism cuts hindered the problem runtime, despite the fact that they almost always reduced the number of nodes in the tree and the number of canonicity tests performed, the time spent in the separation algorithm outweighed the benefits.

In problems like the search for a 2-$(13, 3, 1)$ design using the incidence matrix formulation (Table 8.5), isomorphism cuts are shown to be detrimental to the execution time; however, this is understandable as the isomorphism cuts are generated using an algorithm similar to that of canonicity testing, and canonicity testing dominates the run time when the cutting plane algorithm is turned off (taking 83.53% of the CPU time). This brings into question whether this type of ILP is suited for the NIBAC algorithms.

Because of their possible usefulness and in order to follow Margot's proposed framework more closely, from this point forth we will leave isomorphism cuts enabled unless otherwise indicated.

8.2 Using node groups versus tree groups

Chapter 6 discusses two possible approaches to storing the group for a branching tree: either we can maintain one group representation for the entire tree, which has the advantage that it requires far less memory and requires no overhead in copying permutations, or we can hold one representation per node, which allows us to avoid making calls to the potentially costly `reverse-down` (Algorithm 4.1.4). In the results in the previous sections, we employed the one group per node philosophy; we now proceed to compare the two possibilities.

It seems that in the case where the symmetry group is isomorphic to a symmetric group or the direct composition of a small number of symmetric groups that there is very little difference between the two representations; in fact, it seems that using one group for the entire tree reduces the execution time because there is no need to perform a group copy operation for each visited node. We demonstrate results for this on design problems (Table 8.10) and packing problems (Table 8.11), focusing on generation rather than search; for the majority of search problems where a structure exists, the algorithms usually require no backtracking, and thus, there is no need to ever call `reverse-down`.

Clearly, in the above examples, there is little variation between the two different group representations. However, this is not always the case: if the structure of the symmetry group is instead isomorphic to the direct composition of a large number of small symmetric groups, then the time spent downing increases significantly, and the time spent in reverse-downing becomes exorbitant. To illustrate this, we consider an ILP for t-(v, k, λ) designs using the block formulation and $\lambda > 1$, in which we remove the inequalities of the form:

$$x_{(B,c)} \geq x_{(B,d)}, \qquad B \in \binom{V}{k}, \quad c < d, \quad c, d \in \mathbb{Z}_\lambda.$$

The canonical solutions will be the same, but we introduce a large number of isomorphs.

t	v	k	λ	Tech	#GC	GCT	#D	DT	#RD	RDT	T
2	8	4	3	TG	-	-	774	0.08	392	0.10	2.94
				NG	159	0.07	783	0.08	-	-	2.93
2	9	3	2	TG	-	-	5416	0.16	4384	0.12	26.46
				NG	2351	0.87	5449	0.14	-	-	28.19
2	9	4	3	TG	-	-	5105	0.55	4277	0.42	27.99
				NG	1441	1.27	5210	0.50	-	-	31.00
2	10	4	2	TG	-	-	1386	1.05	522	0.67	8.12
				NG	131	0.27	1396	1.18	-	-	8.04
2	11	5	2	TG	-	-	1687	19.78	98	11.03	46.33
				NG	35	0.22	2705	23.51	-	-	39.35
2	13	3	1	TG	-	-	1573	≤ 0.01	1123	≤ 0.01	122.90
				NG	229	0.35	1598	0.01	-	-	123.86

Table 8.10: The effects of using a node group (NG) versus a tree group (TG) for several t-(v, k, λ) design generation problems.

This causes the symmetry group G to grow from size $v!$ to $v!\lambda!^{\binom{v}{k}}$, where:

$$G = S_v \odot \underbrace{S_\lambda \odot \ldots \odot S_\lambda}_{\binom{v}{k} \text{ times}}.$$

It is natural to expect that larger symmetry groups require more time spent in group algorithms; however, by the examples in Tables 8.10 and 8.11, this time seems to increase evenly between the two approaches. Using the formulation that yielded the larger symmetry group, for the 2-$(9, 4, 3)$ design generation problem, the execution time increased from 159.47 seconds to 2601.84 when a tree group was used instead of a node group; 2329.91 seconds of this time was spent in reverse-downing.

Upon investigation, it became apparent that it was not so much the size of the symmetry group that imposed this large increase, but the structure of the symmetry

t	v	k	λ	Tech	#GC	GCT	#D	DT	#RD	RDT	T
2	10	3	1	TG	-	-	803	0.12	397	0.06	1.72
				NG	123	0.06	844	0.12	-	-	1.76
2	11	3	1	TG	-	-	1128	0.34	467	0.22	5.43
				NG	189	0.13	1128	0.37	-	-	5.48
2	12	3	1	TG	-	-	7660	1.00	6757	0.89	110.23
				NG	1391	1.52	7673	1.10	-	-	110.76

Table 8.11: The effects of using a node group (NG) versus a tree group (TG) for several t-(v, k, λ) packing generation problems.

group itself. When the symmetry group is either isomorphic to a symmetric group or a small number of direct compositions of symmetric groups, the majority of the permutations in the Schreier-Sims representation become concentrated in the early rows of the table (vis-a-vis the base). Since it is extremely unlikely that we will be downing the variables appearing early in the base as they were likely fixed to 1 by preprocessing or early in the branch-and-cut, the permutations in these rows will not have to be re-entered through calls to `down` or `reverse-down`. Indeed, the majority of calls to `down` and `reverse-down` will not require us to re-enter any permutations. When, however, the symmetry group is instead isomorphic to the direct composition of a large number of small symmetric groups, the permutations become spread throughout the Schreier-Sims table. Thus, every call to `down` is likely to require entering at least one permutation, and calls to `reverse-down` may require entering many more.

Because the node group representation is likely to work better in the general case, the time spent copying groups is not a significant part of total execution time, and the memory requirements for doing so are not prohibitive, we opt to store a group per node for our purposes.

8.3 Case studies: solving various problems with NIBAC

We now turn our attention to the effectiveness of using NIBAC to solve a variety of different problems. We focus our efforts on the following tactics:

1. using an isomorph-free branch-and-cut tree with 0-fixing,
2. using an isomorph-free branch-and-cut tree up to a depth d and then proceeding with normal branch-and-cut via NIBAC,
3. using an isomorph-free branch-and-cut tree up to a depth d and then proceeding with normal branch-and-cut via CPLEX, and
4. turning on and off canonicity testing at various points in the branch-and-cut tree.

8.3.1 Block incidence formulation for designs, packings, and coverings

The block incidence ILP formulation for designs, packings, and coverings was given in Section 2.1.1. We examine the effectiveness of NIBAC at solving these types of problems.

2-$(v, 3, \lambda)$ designs, packings, and coverings

Tables 8.12 and 8.13 detail the results of several packing, covering, and design searches with regards to three different techniques: allowing CPLEX to completely solve the problem (CPLEX), allowing NIBAC to solve the problem without canonicity testing (NIBAC iso-off), and allowing NIBAC to solve the problem using canonicity testing (NIBAC iso-on). In all tests in these tables, while canonicity testing significantly

decreased the number of nodes explored by our branch-and-cut, in terms of execution time, it was usually worse to use canonicity testing.

			CPLEX		NIBAC iso-off			NIBAC iso-on			
v	k	λ	N	T	N	D	T	N	D	CT	T
11	5	2	1	0.59	674	338	15.85	18	10	0.05	20.57
14	3	1	1	0.13	289	151	1.70	13	12	1.02	4.56
15	3	1	1	0.29	224	118	1.82	10	9	8.93	15.71
19	3	1	1	0.60	626	325	15.73	30	27	8611.74	12324.70

Table 8.12: Search for several 2-(v, k, λ) packings and designs using CPLEX, NIBAC with isomorphism testing turned off, and regular NIBAC.

	CPLEX		NIBAC iso-off			NIBAC iso-on			
v	N	T	N	D	T	N	D	CT	T
5	1	≤ 0.01	1	0	0.01	1	0	≤ 0.01	≤ 0.01
6	1	≤ 0.01	44	16	0.04	22	10	0.01	0.03
8	1	0.01	14322	52	16.30	488	30	5.55	9.04

Table 8.13: Search for several 2-$(v, 3, 1)$ coverings using CPLEX, NIBAC with isomorphism testing turned off, and regular NIBAC.

In Table 8.14, we give experimental results for a search for a 2-$(14, 3, 1)$ packing where we turn canonicity testing off at a specified depth and allow either CPLEX or NIBAC (iso-off) to complete the problem. In all tests in this table, it was advantageous or comparable to allow CPLEX to perform branch-and-cut. This is typical of many search problems where an object is easily generated (i.e. no, or very few backtracks occur in the tree). Canonicity testing affects the branch-and-cut tree in unpredictable ways (as shown by the fluctuating number of nodes and execution times), and since

the object is easily produced by traditional branch-and-cut, canonicity testing is not beneficial.

	NIBAC iso-on + CPLEX		NIBAC iso-on + NIBAC iso-off						
depth	N	T	N	D	LPT	LP	ST	CT	T
never	1	0.13	289	151	0.60	300	0.17	≤ 0.01	1.65
0	1	2.63	216	113	0.45	255	0.12	≤ 0.01	3.83
1	35	3.34	73	39	0.16	80	0.09	≤ 0.01	3.24
2	16	3.23	307	161	0.55	316	0.18	≤ 0.01	4.55
3	4	2.91	147	79	0.32	159	0.12	0.01	3.77
4	74	3.50	135	73	0.32	145	0.13	0.02	3.71
5	10	3.30	242	129	0.53	249	0.22	0.09	4.56
6	62	3.45	85	48	0.28	96	0.17	0.17	3.62
7	12	3.34	111	62	0.34	125	0.22	0.23	3.95
8	9	3.39	202	110	0.57	210	0.34	0.33	4.78
9	10	3.56	14	12	0.16	22	0.39	0.4	3.71
10	11	3.82	27	19	0.21	36	0.50	0.50	3.97
11	12	4.12	13	12	0.14	21	0.63	0.72	4.24
always	-	-	13	12	0.15	21	0.63	1.03	4.54

Table 8.14: The effects of canonicity testing in the search for a 2-$(14, 3, 1)$ packing. We use NIBAC with canonicity testing turned on up to depth, and compare completion by CPLEX with completion by NIBAC with canonicity testing turned off.

We now turn our attention to the possibility of turning off canonicity testing at a certain depth, and then resuming it later on in the tree for search problems. As an example that is typical for this technique, we examine the search for a 2-$(10, 3, 1)$ packing where canonicity is turned off at depth 3 in Table 8.15 and turned back on

at a specified depth. In all cases, the execution time was higher than always using canonicity testing (0.16 seconds, 10 nodes, depth 8) and never using canonicity testing (0.09 seconds, 36 nodes, depth 19). We could not find larger problems where we could turn off canonicity testing at a certain depth and then turn it back on, even for a single row of our tree, and obtain results in a reasonable amount of time. For example, in the case of search for a 2-$(14, 3, 1)$ packing, shutting off canonicity at depth 7 (as per Table 8.14) resulted in a total runtime of 3.95 seconds, but resuming it even for only depth 10 caused the problem to run for over 24 hours without completing. There may be many nodes to test for canonicity at depth 10, and while a large number of these might be found to be isomorphic, there could be a significant portion of nodes (those leading to maximum solutions, and those leading to sub-optimal solutions or infeasible solutions that are still canonical) that need to be tested.

CDepth	N	D	CT	T
-	41	23	0.01	0.16
4	37	21	13.97	14.11
5	37	21	14.05	14.20
6	42	24	36.56	36.71
7	42	24	36.61	36.79
8	41	23	25.19	25.38
9	41	23	25.22	25.41
10	41	23	22.90	23.06
11	19	11	0.03	0.17

Table 8.15: The effects of turning off canonicity at depth 3 in the search for a 2-$(10, 3, 1)$ packing and then resuming at depth CDepth.

In all examined search cases for existence of a design or packing, if the upper bound provided was tight, it appeared that canonicity testing only increased the execution

time dramatically: this increase was from 1.65 to 4.54 for finding a single 2-$(14, 3, 1)$ packing (see Table 8.14). However, if the bounds are looser, canonicity testing is advantageous. We demonstrate this in Table 8.16 in the search for a 2-$(11, 3, 1)$ packing, where the Johnson bound provides us with an upper bound of 18 but the packing number is actually 17. We provide details for continuing the branch-and-cut with either CPLEX or NIBAC with canonicity turned off and note that the former performed better than the latter. We can see that canonicity testing is extremely beneficial since execution time increases from 1.03 seconds to 1840.64 seconds when it is not used. Turning canonicity testing off could be done without much change in running time from depth 7 or greater.

We now turn our attention to a generation problem to view the effects of canonicity testing in such an example: we will create an isomorph-free list of all 2-$(10, 3, 1)$ maximum packings (there are two). Table 8.17 details the times spent testing canonicity up to a prescribed depth and then turning the testing off. Since this is an exhaustive generation problem and we only want isomorph-free list of solutions, any solution encountered after this depth must be tested for canonicity using the slow canonicity test `first-in-orbit2` (Algorithm 4.2.6) in order to determine whether or not it belongs in our list. It is interesting to note that if we cut off canonicity testing at depth 1, we are able to perform 27359 calls to `first-in-orbit2` in 103.50 seconds; however, if we cut off canonicity testing at depth 13, we require 66.19 seconds in order to run this algorithm three times, two of which are used to determine that the tested solution is one of the canonical packings that will appear in our final list. This demonstrates the strengths and weaknesses of `first-in-orbit2`: in most cases, it is able to reject isomorphic solutions quickly, but it requires a large amount of time (as it must try every permutation over the set F_1) to determine that a solution is canonical.

In Table 8.18, we show the results of using NIBAC to generate all 2-$(v, 3, 1)$ coverings for several small values of v. We were unable to generate all 2-$(9, 3, 1)$ coverings with

the block incidence formulation.

	NIBAC iso-on + CPLEX		NIBAC iso-on + NIBAC iso-off						
depth	N	T	N	D	LPT	LP	ST	CT	T
never	132012	262.24	513619	82	735.64	542051	202.31	-	1840.64
0	132012	263.38	473887	82	748.35	508842	204.13	≤ 0.01	1911.53
1	17992	31.46	59137	65	95.93	64072	28.09	≤ 0.01	244.37
2	4995	6.12	11107	57	18.70	12133	5.37	≤ 0.01	46.7
3	1083	1.79	3535	52	6.28	3879	1.72	≤ 0.01	15.59
4	910	1.89	3623	46	6.08	3940	1.27	0.01	15.14
5	48	0.81	437	31	0.92	484	0.14	≤ 0.01	2.47
6	116	0.80	383	29	0.70	418	0.11	0.02	2.19
7	23	0.82	91	25	0.23	109	0.06	0.04	1.05
8	19	0.82	79	22	0.19	95	0.07	0.04	1.09
9	42	0.94	41	18	0.15	51	0.11	0.05	1.01
10	25	0.97	31	14	0.13	38	0.14	0.09	1.04
always	-	-	25	11	0.15	32	0.09	0.10	1.03

Table 8.16: The effects of canonicity testing in the search for a 2-$(11, 3, 1)$ packing, where the Johnson bound is higher than the packing number.

In Table 8.19, we detail the results of some more difficult exhaustive generation problems, namely the generation of all 2-$(14, 3, 1)$ packings and 2-$(15, 3, 1)$ designs. The column #ni lists the number of nonisomorphic structures. Using a full canonicity test, NIBAC was able to generate all 80 2-$(15, 3, 1)$ designs in 2 days, 134 seconds: this required visiting 20233 nodes with a tree depth of 48, where the majority of execution time (170831 seconds) was spent in canonicity testing. Due to the large number of designs, it is not feasible to turn off canonicity testing for this problem: the algorithm

`first-in-orbit2` would take far too long to completely test 80 designs for canonicity. Similarly, the 787 2-$(14, 3, 1)$ packings were generated in 6.42 days, where 4.69 days were spent testing canonicity.

depth	N	D	LPT	LP	ST	CC	CT	SC	SCT	T
1	2558309	88	2075.33	2598647	31.60	6	≤ 0.01	27359	103.50	5987.80
2	493145	82	398.20	500343	4.84	11	≤ 0.01	5328	103.76	1221.87
3	130277	72	110.09	133432	1.71	25	≤ 0.01	1884	195.49	495.92
4	23043	65	18.09	23533	0.24	36	≤ 0.01	348	100.41	152.58
5	14687	58	11.57	15019	0.21	54	0.03	248	170.35	203.70
6	4309	45	3.55	4383	0.06	76	0.02	78	167.54	177.57
7	3453	45	2.72	1785	0.06	95	0.02	64	171.43	179.60
8	1493	42	1.39	1520	0.12	128	0.02	30	164.94	168.86
9	957	41	1.00	987	0.09	152	0.10	19	156.29	159.13
10	697	39	0.77	721	0.11	220	0.16	14	155.89	158.29
11	493	39	0.60	521	0.14	302	0.23	10	150.80	152.86
12	251	36	0.48	278	0.11	381	0.42	6	126.17	127.92
13	183	32	0.28	206	0.07	470	0.44	3	66.19	67.91
14	127	18	0.28	152	0.09	505	0.59	1	31.14	32.87
always	123	15	0.22	148	0.11	532	0.64	-	-	1.78

Table 8.17: The effects of canonicity testing in the generation of all 2-$(10, 3, 1)$ packings.

v	N	D	LP	LPT	ST	CT	T	#ni
4	7	3	7	≤ 0.01	≤ 0.01	≤ 0.01	≤ 0.01	1
5	11	4	11	≤ 0.01	≤ 0.01	≤ 0.01	0.01	1
6	161	16	162	0.06	≤ 0.01	0.02	0.15	1
8	11915	46	11942	6.26	20.68	88.20	124.72	5

Table 8.18: Exhaustive generation of minimum 2-$(v, 3, 1)$ coverings. We omit $v \in \{7, 9\}$ since these are designs.

v	N	D	LP	LPT	ST	CT	T	#ni
14	739879	55	1068325	7788.74	101246.00	432755	554887	787
15	20233	48	21141	121.20	1567.49	160711	172934	80

Table 8.19: Exhaustive generation of all 2-$(14, 3, 1)$ packings and 2-$(15, 3, 1)$ designs.

λ-families of simple and non-simple triple systems

For given t, v, and k, we now investigate exhaustive generation of families of designs as λ varies. Using NIBAC with canonicity testing turned on, we were able to enumerate some results for the first time.

Definition 8.3.1. A design $(V, \mathcal{B})$ is said to be *simple* if $\mathcal{B}$ is a set (i.e. no blocks are repeated).

In some cases, due to the large number of distinct structures, we are not able to generate all t-(v, k, λ) designs, in which case we focus on finding the simple ones only.

Mathon [25] was able to determine the number of 2-$(6, 3, 2n)$ designs.

Theorem 8.3.1. *(From [25].) The number of nonisomorphic* 2-$(6, 3, 2n)$ *designs is:*

$$\sum_{i=0}^{5} a_{i,n \bmod 12} \left\lfloor \frac{n}{12} \right\rfloor^i,$$

where $A = (a_{i,j})$ *is a* 6×12 *matrix defined as follows:*

$$A = \frac{1}{10} \begin{bmatrix} 10 & 10 & 40 & 60 & 130 & 190 & 340 & 480 & 760 & 1050 & 1530 & 2060 \\ 111 & 136 & 321 & 476 & 821 & 1176 & 1771 & 2436 & 3411 & 4536 & 6061 & 7836 \\ 295 & 620 & 1005 & 1480 & 2075 & 2820 & 3745 & 4880 & 6255 & 7900 & 9845 & 12120 \\ 1220 & 1400 & 1700 & 2120 & 2660 & 3320 & 4100 & 5000 & 6020 & 7160 & 8420 & 9800 \\ 360 & 720 & 1080 & 1440 & 1800 & 2160 & 2520 & 2880 & 3240 & 3600 & 3960 & 4320 \\ 864 & 864 & 864 & 864 & 864 & 864 & 864 & 864 & 864 & 864 & 864 & 864 \end{bmatrix}.$$

Using NIBAC with canonicity testing always turned on, we were able to confirm this formula for $n \in [1, 15]$ and determine all simple 2-$(6, 3, \lambda)$ designs: there are only two such simple designs, namely one for $n = 1$ and one for $n = 2$. The results are given in Table 8.20. The column #ni gives the number of nonisomorphic designs. Both Ivanov [17] and Gronau and Prestin [14] were able to determine the number of designs for $n = 2$ and $n = 3$. Ivanov [17, 18] was additionally able to solve for $n \in [4, 6]$, and

n	b	N	D	LPT	LP	ST	CT	T	#ni
1	10	33	16	0.03	33	0.01	0.01	0.06	1
2	20	225	39	0.35	228	0.24	0.60	1.45	4
3	30	791	87	1.09	793	0.28	0.95	3.54	6
4	40	2441	123	3.76	2452	2.41	7.09	18.36	13
5	50	6427	157	12.05	6470	3.93	11.52	42.06	19
6	60	15297	218	35.14	15361	15.04	47.15	137.61	34
7	70	32991	275	86.41	32372	29.63	76.52	287.19	48
8	80	63691	316	186.83	63866	81.99	215.91	697.28	76
9	90	117515	359	383.72	117788	166.81	371.33	1356.12	105
10	100	204841	399	742.95	205303	370.12	816.78	2756.46	**153**
11	110	341043	439	1358.19	341704	698.16	1390.22	4955.94	**206**
12	120	547563	479	2400.10	548484	1308.27	2722.75	9071.63	**286**
13	130	847233	519	4042.61	848546	2302.34	4444.47	15225.80	**375**
14	140	1274737	559	6751.18	1276651	4052.31	7809.90	25856.90	**501**
15	150	1870649	599	10691.40	1873101	6695.58	12309.20	41047.30	**644**

Table 8.20: Results for the generation of 2-$(6, 3, 2n)$ designs.

the generation for $n \in [7, 9]$ was completed by Denny [9]. The values for $n \in [10, 15]$ seem to have been generated here for the first time.

Additionally, NIBAC is able to generate all 2-$(7, 3, \lambda)$ designs for $\lambda \in [1, 8]$. For higher values of λ, the program runs out of memory. Table 8.21 details the results. The columns #ni and #sp give the number of nonisomorphic general and simple designs respectively. While the details in the table reflect the statistics for producing all designs, we provide the column SiT, which lists only the time required to produce exclusively the simple designs. These confirm previous results by Ivanov [18], Gronau and Prestin [14], and Pietsch [41]. Mathon and Pietsch [27] were able to solve this problem for

$n \in [10, 12]$, and Denny [9] addressed the cases where $n \in [13, 16]$.

λ	b	N	D	LPT	LP	ST	CT	T	SiT	#ni	#sp
1	7	7	3	≤ 0.01	7	≤ 0.01	0.01	0.02	0.02	1	1
2	14	133	26	0.10	137	0.04	0.09	0.43	0.09	4	1
3	21	559	51	0.55	575	0.14	0.62	2.49	0.15	10	1
4	28	3011	94	3.98	3056	1.69	4.63	16.25	0.47	35	1
5	35	15849	140	24.75	16000	14.11	47.42	119.77	3.05	109	1
6	42	86459	194	150.65	86850	53.80	151.28	559.62	-	418	0
7	49	409349	234	806.65	410481	296.70	797.06	2970.75	-	1508	0
8	56	1731055	270	3811.60	1733790	1560.72	4054.98	14466.90	-	5413	0

Table 8.21: Results for the generation of 2-$(7, 3, \lambda)$ designs.

For 2-$(9, 3, \lambda)$ designs, NIBAC was only able to generate all 36 designs with $\lambda = 2$ (Table 8.22). Mathon and Lomas [26] showed that there are 22521 2-$(9, 3, 3)$ designs, but this was too difficult for NIBAC to reproduce. However, we were able to compute all simple 2-$(9, 3, \lambda)$ designs. These results are shown in Table 8.23. As we are unable to perform a full generation for these designs, the data given in the table reflects the statistics of the generation for simple designs only.

λ	b	N	D	LPT	LP	ST	CT	T	#ni	#sp
2	24	2349	69	3.74	2514	3.94	10.15	28.58	36	13

Table 8.22: Results for the generation of 2-$(9, 3, \lambda)$ designs.

We were unable to generate all 2-$(10, 3, 2n)$ designs for $n > 1$, even in the simple case, but it took NIBAC 2136.57 seconds (98687 nodes, tree depth 134, 101626 LPs solved in 195.06 seconds, 300.04 seconds in the separation, and 1230.63 seconds in canonicity testing) to find all 960 2-$(10, 3, 2)$ designs, 394 of which were simple. This is

λ	b	N	D	LPT	LP	ST	CT	T	#sp
2	24	1547	53	1.19	1566	0.44	17.56	21.16	13
3	36	42157	66	35.23	42406	26.37	2088.52	2218.29	332
5	60	4065	71	9.21	4109	97.49	1220.10	1691.47	13
6	72	191	72	2.32	194	30.23	116.45	342.57	1
7	84	159	79	0.22	159	≤ 0.01	2584.74	2585.16	1

Table 8.23: Results for the generation of simple 2-$(9, 3, \lambda)$ designs.

detailed in Table 8.24. These results concur with previous findings by Ganter, Gülzow, Mathon, and Rosa [12]; Colbourn, Colbourn, Harms, and Rosa [2]; and Ivanov [17].

n	b	N	D	LPT	LP	ST	CT	T	#ni	#sp
1	30	98687	134	195.06	101626	300.04	1036.28	2136.57	960	394

Table 8.24: Results for the generation of 2-$(10, 3, 2n)$ designs.

Maximal partial triple systems

Definition 8.3.2. A *maximal partial triple system* MPT(v) is a 2-$(v, 3, 1)$ (not necessarily maximum) packing $(\mathbb{Z}_v, \mathcal{B})$ such that $\mathcal{B}$ is a maximal set of triples.

We are interested in generating all unique MPT(v). We begin by examining the *spectrum* of numbers of triples in MPT(v)s, denoted $S^{(3)}(v)$.

Theorem 8.3.2. *(From [3].) The largest number of triples that an MPT(v) can have is:*

$$\mu(v) = \begin{cases} \left\lfloor \frac{v}{3} \left\lfloor \frac{\lambda(v-1)}{2} \right\rfloor \right\rfloor - 1 & \text{if } v \equiv 5 \pmod 6, \\ \left\lfloor \frac{v}{3} \left\lfloor \frac{\lambda(v-1)}{2} \right\rfloor \right\rfloor & \text{otherwise.} \end{cases}$$

Theorem 8.3.3. *(From [3].) The smallest number of triples that an MPT(v) can have is* $\frac{v^2+d(v)}{12}$*, where:*

$$d(v) = \begin{cases} -2v + 36 & \text{if } v \equiv 0, 8 \pmod{12}, \\ -2v & \text{if } v \equiv 2, 6 \pmod{12}, \\ -2v + 4 & \text{if } v \equiv 4 \pmod{12}, \\ -2v + 16 & \text{if } v \equiv 10 \pmod{12}, \\ -1 & \text{if } v \equiv 1, 5 \pmod{12}, \\ 3 & \text{if } v \equiv 3 \pmod{12}, \\ 11 & \text{if } v \equiv 7, 11 \pmod{12}, \\ 15 & \text{if } v \equiv 9 \pmod{12}. \end{cases}$$

The *spectrum*, or values between the maximum and minimum values that can be realized, were established by Colbourn, Rosa, and Zńam [4] and are given in [3]. NIBAC was able to generate all nonisomorphic MPT(v) for $v \in [5, 12]$, thus enumerating designs for each block count. The results we present are new, and are detailed in Tables 8.25 and 8.26.

v	N	D	CT	T	#ni
5	3	1	≤ 0.01	≤ 0.01	**1**
6	9	3	≤ 0.01	≤ 0.01	**2**
7	25	6	≤ 0.01	0.04	**2**
8	61	8	0.06	0.15	**4**
9	323	14	0.42	1.21	**10**
10	3359	27	11.75	23.42	**47**
11	112919	44	1161.85	1724.81	**472**
12	10004627	46	314922.00	449517.00	**14771**

Table 8.25: Generation of all MPT(v) for $v \in [5, 12]$.

$b \setminus v$	5	6	7	8	9	10	11	12
2	1	1						
3								
4		1						
5			1					
6								
7			1	**3**				
8				1	**4**	**1**		
9					**3**			
10					**2**	**10**		
11						**15**	**3**	
12					1	**19**	**9**	
13						2	**132**	**12**
14							**209**	**25**
15							**110**	**586**
16							**7**	**3638**
17							2	**6855**
18								**3196**
19								**454**
20								5
total	1	2	2	**4**	**10**	**47**	**470**	**14771**

Table 8.26: Spectral information in the generation of MPT(v) for $v \in [5, 12]$.

Triple systems with holes

In this section, we provide some exhaustive generation results for triple systems with holes. To the best of our knowledge, the values have not been computed before; however, the results for $w = 3$ could be produced directly from the complete lists of 2-$(v, 3, 1)$ designs, known up to $v = 19$, instead of generated as we did.

Definition 8.3.3. (From [3].) An *incomplete triple system* ITS(v, w; λ) is a 2-$(v, 3, \lambda)$ packing $(\mathbb{Z}_v, \mathcal{B})$ such that there exists a $W \subseteq \mathbb{Z}_v$ with the property that for each $\{v_0, v_1\} \in V$, there is no $B \in \mathcal{B}$ such that $\{v_0, v_1\} \in B$. We then call W the *hole* of the design.

It then becomes easy to make small modifications to our block incidence formulation to look for incomplete triple systems. In the original formulation for each 2-set, we had the following constraint:

$$\sum_{(B,c)\in\binom{V}{3}\times\mathbb{Z}_\lambda : T\subseteq B} x_{(B,c)} = \lambda, \qquad T \in \binom{V}{2}.$$

For the sets T corresponding to 2-sets of W, we need simply change the right hand side of these constraints from λ to 0 in order to ensure that they do not appear in the packing. This has the effect of changing the symmetry group as well: we can no longer permute between elements of W and $V \setminus W$; thus, our symmetry group is now isomorphic to $G' = S_{v-w} \odot S_w$, and is in fact defined by the actions of G' on the blocks representing the variables of our problem.

Theorem 8.3.4. *(From [3].) An ITS(v, w; λ) exists whenever $w = 0$ or $w \equiv 1, 3 \pmod 6$, $v \equiv 1, 3 \pmod 6$, and $v \geq 2w + 1$.*

The number of blocks in an ITS(v, w; λ) is $\lambda\frac{v(v-1)-w(w-1)}{6}$.

We display results for the generation of all ITS(7, 3; λ) for $\lambda \in [1, 18]$ in Table 8.27; there are only three simple designs, namely one ITS(7, 3; 1), one ITS(7, 3; 2), and one

ITS(7, 3; 3). We also detail the generation of some other ITS(v, w; λ) in Table 8.28. To the best of our knowledge, these results have not been computed before.

As with designs without holes, search problems perform significantly worse when canonicity is used (e.g. we require 0.27 seconds to find a single ITS(15, 7; 1) when canonicity is turned off, and 7.95 seconds when it is turned on). Similarly, in the optimal generation scenario, it is not useful to turn canonicity testing off at any level; all experimental results dictate that turning canonicity testing back on increases time significantly, and it is preferable to employ the canonicity test throughout the entire tree.

λ	N	D	LPT	LP	ST	CT	T	#ni
1	19	9	≤ 0.01	19	≤ 0.01	0.01	0.03	**1**
2	97	23	0.08	98	0.01	0.04	0.21	**3**
3	249	41	0.28	251	0.03	0.30	0.96	**5**
4	609	58	0.91	614	0.16	0.55	2.75	**9**
5	1253	73	2.14	1262	0.57	1.39	6.96	**13**
6	2703	88	4.95	2721	1.77	5.92	19.92	**22**
7	4883	103	10.13	4918	3.48	10.70	39.29	**30**
8	8815	119	21.60	8859	8.22	25.11	84.24	**45**
9	14491	135	37.53	14558	17.37	53.96	162.80	**61**
10	23363	150	68.71	23460	33.61	98.53	299.69	**85**
11	35459	165	115.57	35574	58.13	172.83	507.50	**111**
12	53481	180	191.70	53636	107.38	316.16	879.72	**149**
13	76123	195	294.35	76308	173.99	494.31	1358.33	**189**
14	107623	210	458.48	107833	284.50	816.81	2179.60	**244**
15	147611	225	677.98	147926	439.62	1277.38	3308.14	**304**
16	200123	240	989.93	200493	683.40	1929.61	4999.83	**381**
17	264531	255	1393.26	264948	989.12	2809.43	7035.22	**465**
18	347751	270	1950.00	348312	1495.60	4187.10	10231.20	**571**

Table 8.27: Generation of all ITS(7, 3; λ) for $\lambda \in [1, 18]$. The only simple ITS(7, 3; λ) are one ITS(7, 3; 1), one ITS(7, 3; 2), and one ITS(7, 3; 3).

v	w	λ	N	D	LPT	LP	ST	CT	T	#ni	#sp
9	3	1	33	16	0.06	34	≤ 0.01	0.01	0.15	**1**	**1**
		2	2239	73	2.95	2275	1.01	3.68	13.41	**34**	**8**
		3	276057	167	487.31	277371	143.86	482.27	1834.01	**1924**	**26**
13	3	1	1467	142	3.05	1497	1.25	37.06	49.91	**10**	**10**
15	3	1	533889	253	1724.81	535495	887.66	99448.60	104860.00	**1746**	**1746**
15	7	1	395	39	2.06	317	0.99	24.44	41.70	**6**	**6**

Table 8.28: Generation of several different ITS(v, 3; λ).

Steiner Quadruple Systems

Definition 8.3.4. A *Steiner Quadruple System* is a 3-$(v, 4, 1)$ design, and is denoted SQS(v).

Theorem 8.3.5. *(From [5].) A SQS(v) exists if and only if* $v \equiv 2, 4 \pmod 6$.

The number of blocks in an SQS(v) is $\frac{v(v-1)(v-2)}{24}$. This number grows quickly, thus causing the canonicity testing algorithm time to increase dramatically even for small v. Table 8.29 shows our findings for the exhaustive generation of several SQS(v). NIBAC was able to reproduce previously known results by Mendelsohn and Hung [30] for SQS(v) with $v \in \{8, 10, 14\}$.

v	N	D	LPT	LP	ST	CT	T	#ni
8	25	12	0.01	25	≤ 0.01	0.14	0.2	1
10	53	26	0.18	53	≤ 0.01	8.03	8.73	1
14	14939	111	4371.04	20131	15725.10	136924.00	157609.00	4

Table 8.29: Generation of all optimal SQS(v) for $v \in \{8, 10, 14\}$.

As with the other designs studied in this chapter, for SQS(v) search problems, canonicity testing to any degree increased execution time significantly. For example, to produce a 91-block single SQS(14) using canonicity testing took 286.37 seconds, whereas we were able to find one in 27.54 seconds if we used traditional branch-and-cut.

8.3.2 Incidence matrix formulation for 2-(v, k, λ) designs

Our initial impression, from the results in Table 8.5, lead us to believe that the incidence matrix formulation for designs may not perform well in our framework: with default parameters, it took NIBAC 42471.80 seconds to generate a single 2-$(13, 3, 1)$ design using an incidence matrix, with 742 nodes visited (tree depth 57), 174.81 seconds spent solving 792 LPs, 12128.50 seconds taken by the separation, and 30090.00 seconds spent canonicity testing. Given that the only cuts being used in this problem were isomorphism cuts, we note that the backtracking algorithms for canonicity and isomorphism cut generation dominated 99.4% of the total execution time. On the other hand, when using a block incidence formulation, the same problem required only 1.74 seconds to complete. We conclude that it is by far preferable to use the block incidence formulation for these problems.

We conjectured that perhaps the poor execution times were due to the large size of the symmetry group; this is what motivated us to add further constraints to the problem in order to remove the column permutations (as detailed in Section 2.1.2) and hence reduce the size of the group from $v!\left(\frac{v(v-1)}{k(k-1)}\right)!$ to $v!$.

This strategy, however, turned out to be very poor in practice. The problem of generating a single 2-$(9, 3, 1)$ design increased from 1.03 seconds to 13.22 seconds. Enumerating all 2-$(9, 3, 1)$ designs initially required 5.61 seconds to enumerate with the original formulation, and with the new one, took 63.14 seconds.

Further investigation revealed why this turned out to be the case: while the presence of the column permutations increased the time required by canonicity testing algorithms, it dramatically reduced the benefits of 0-fixing. In the enumeration of 2-$(9, 3, 1)$ designs, our original problem required only 53 nodes and was able to fix 7 variables at depth 1 and 21 at depth 2 by 0-fixing. In the new formulation, our tree contained 4161 nodes and could only fix 1 variable at depth 1 and 1 at depth 2.

Because of these limitations, we concluded that the incidence matrix formulation

was poorly suited to NIBAC and demonstrated the weaknesses of some of its algorithms. The canonicity testing algorithms in Chapter 4 depend on both the size of the symmetry group and the number of variables fixed to 1 in a final solution. In the case of an incidence matrix, the number of variables fixed to 1 can grow quite high: namely k times higher than the equivalent block incidence formulation. For a large number of variables, for 0-fixing to be effective, it is beneficial to have a reasonably big symmetry group; however, this symmetry group causes canonicity testing to perform poorly. Thus, for problems with a large number of variables, it is likely that large symmetry groups are needed for NIBAC algorithms to be useful, but it is unlikely that the cost of these algorithms will justify the benefit.

8.3.3 Intersecting set systems

Using NIBAC, we were able to construct all the maximal intersecting set systems presented in [33]. We show our results in Table 8.30, where the number of nonisomorphic maximal structures is given in the column #ni. It was our hopes that we would be to produce a new result, namely the generation of all maximal (16, 4, 1) intersecting set systems, but even after several weeks of CPU time, NIBAC was unable to complete the problem.

v	k	t	N	D	B	CT	T	#ni
5	3	2	9	4	4	≤ 0.01	≤ 0.01	2
6	4	3	11	5	5	≤ 0.01	0.01	2
7	3	1	1205	17	602	1.58	3.91	15
8	4	2	3543	22	1771	11.36	76.58	17
9	5	3	10467	24	5233	86.62	1472.96	17

Table 8.30: The generation of several maximal (v, k, t) intersecting set systems.

We now turn our attention to the issue of turning off canonicity testing for these problems during the branch-and-cut. Because of the large number of solutions to relevant problems, turning off fast canonicity testing early and testing each solution for isomorphism was far too costly: in the case of the $(7, 3, 1)$ intersecting set system, the execution time required for pure branch-and-cut and then checking solutions for isomorphism was 5723.04 seconds (with 2578381 nodes visited) as opposed to always checking canonicity using the fast test, which only required 4.68 seconds (with 1205 nodes visited).

Table 8.31 provides results for the generation of all feasible (v, k, t) intersecting set systems. We note that the number of nonisomorphic systems in column #ni does not include the trivial (i.e. empty) system. These results are new.

v	k	t	N	D	B	CT	T	#ni
5	3	2	9	4	4	≤ 0.01	≤ 0.01	**5**
6	4	3	11	5	5	≤ 0.01	0.01	**6**
7	3	1	1205	17	602	1.52	3.81	**603**
8	4	2	3543	22	1771	11.05	75.19	**1772**
9	5	3	10467	24	5233	85.73	1202.86	**5234**

Table 8.31: The generation of all feasible (v, k, t) intersecting set systems.

Chapter 9

Conclusion and open questions

In this thesis, we have examined an approach for creating isomorph-free or isomorph-reduced branch-and-cut trees to solve 0-1 integer linear programs. This study has led to a deeper understanding as to the types of problems and ILP formulations for which such a method is effective, and has allowed us to solve several new problems and verify many previously known results.

In particular, we examined three different ILP formulations and the effects of isomorph-free branch-and-cut on them: the block incidence formulation for t-(v, k, λ) designs, coverings, and packings (Section 2.1.1) with several variations (maximal packings, triple systems with holes, and Steiner triple and quadruple systems); the incidence matrix formulation for 2-(v, k, λ) designs (Section 2.1.2); and a formulation for (v, k, t) intersecting set systems (Section 2.2).

We created a programmable, redistributable framework, NIBAC (NonIsomorphic Branch-And-Cut) based on the group algorithms [21] and Margot's canonicity testing scheme [22]. We extended this previous work by implementing Margot's algorithms to function with several different problem solving variations: search for a single optimal solution (Section 6.1), generation of all optimal solutions (Section 6.2), generation of all maximal solutions with regards to set inclusion (Section 6.3), and generation of

all feasible solutions (Section 6.4). Additionally, our framework is able to determine the symmetry group for arbitrary ILPs (Section 4.4), accommodate variable fixings through a change of base (Section 4.3), and support additional types of cuts (Section 5.3).

It was our goal to use the NIBAC framework on the above three ILP formulations in order to examine the following four possibilities:

1. using a pure nonisomorphic branch-and-cut tree to solve the problems,

2. using a nonisomorphic branch-and-cut tree up to a specified depth and then resorting to pure branch-and-cut using CPLEX,

3. using a nonisomorphic branch-and-cut tree up to a specified depth and then resorting to pure-branch-and-cut using NIBAC, and

4. turning on and off isomorphism checking at various depths of the branch-and-cut using NIBAC.

Our experimental results indicated to us that an "all-or-nothing" approach was often best. In terms of **search**, unless an object was difficult to find (e.g. no tight bound was known, as in the example of the 2-$(11, 3, 1)$ packing), not enough of the search space was examined in order to merit canonicity testing, and so it was best avoided; and with regards to **exhaustive generation** problems, always performing canonicity testing seemed to be preferable to any of the aforementioned alternatives.

In this chapter, we begin by examining the strengths and limitations of NIBAC with regards to different types of ILPs, and then proceed to detail which combinatorial design problems NIBAC was able to handle well. We conclude with some remaining open questions and possibilities for further research in this field.

9.1 Conclusions on the suitability of NIBAC for ILP problems

First consideration: Nature of the symmetry group

The structure of the symmetry group is perhaps the biggest factor in determining whether or not a problem is suitable for NIBAC techniques. It is important that the symmetry group be large enough with a structure that is beneficial for 0-fixing, meaning that the orbits of variables should be quite large. For example, in the block incidence formulation for designs, the orbit of each block under the action of the symmetry group is the full set of $\binom{v}{k}$ variables (as any block can be permuted to another via a point permutation), whereas in the incidence matrix formulation with column permutations removed (Section 8.3.2), the orbit of each block contains only v variables out of $v\frac{v(v-1)}{k(k-1)}$. Alternatively, large symmetry groups, while perhaps beneficial for 0-fixing, tend to make the algorithms in Section 4.2 perform poorly. In the incidence matrix formulation for 2-$(v, 3, 1)$ designs with column ordering constraints removed, the symmetry group had size $v!\left(\frac{v(v-1)}{k(k-1)}\right)!$, which proved to be too large to work efficiently with our canonicity testing and isomorphism cut generation algorithms (Section 8.3.2). In our case, the block incidence formulation was much better because of both the small size of the symmetry group and the large size of the orbits.

Experimentally, it seemed that symmetry groups isomorphic to a symmetric group performed best. This was largely due to the representation of the group via the Schreier-Sims table: in such cases, permutations tended to be concentrated in a small number of rows near the beginning of the table, and thus calls to `down` during base changes would result in no, or very few permutations needing to be re-entered into the group. If the symmetry group was more complex (e.g. as in the case of the 2-$(11, 5, 2)$ design with block ordering constraints removed, which resulted in the symmetry group involving the direct composition of $\binom{v}{k}$ copies of $\lambda!$ to compensate), it was likely that permutations

would be more evenly distributed over the table and calls to `down` would require one or more permutations to be re-entered, which was a costly operation.

In problems that present multiple ILP formulations, the symmetry groups resulting from each should be taken into consideration.

Second consideration: Nature of the ILP formulation

The algorithms for determining canonicity increase in complexity as both the number of 1s in a final solution and the size of the symmetry group increase. Typically, in most problems, these two factors increase together. Consider the generation of 2-$(v, 3, 1)$ designs, for instance: as the parameter v increases, both the number of variables and the size of the symmetry group increase as well. This was demonstrated by the problem of searching for a single 2-$(19, 3, 1)$ design; the number of variables fixed to 1 in a final solution (57), coupled with the large size of the symmetry group (19!) caused canonicity testing to dominate the overall execution time. This, however, was not the case in the generation of all optimal 2-$(6, 3, 30)$ designs; while the number of variables fixed to 1 was overall considerably higher (150), the small size of the symmetry group (6!) allowed our canonicity testing algorithms to run in a reasonable amount of time.

If the size of final solutions and both the symmetry group are high, the problem may not be suited to NIBAC techniques.

Third consideration: Search problems

If we are working with search problems (problem type 1) that satisfy the above two considerations, it is difficult to determine the exact nature of isomorph-free branch-and-cut. If it is expected that the problem does not have a solution and we are simply seeking to prove this fact, in some cases it will be essential to use isomorph-free branch-and-cut trees to reduce the number of nodes considered to a feasible level (see [22]); this is also likely for problems in which we expect to find a solution but do not know a tight

upper bound. If bounds are known on the problem and are tight, then the problem is likely easily solved by standard branch-and-cut techniques and the added complexity of canonicity testing will increase execution time: this is seen with the search for a single 2-$(19, 3, 1)$ design. On the other hand, when bounds are loose, it is likely that we will have to traverse a much larger portion of the search space in order to ascertain that a solution is optimal: this is demonstrated with the search for a 2-$(11, 3, 1)$ packing (Table 8.16), where the Johnson bound is 18 and the packing number is 17. In this case, using isomorph-free trees is recommended.

Fourth consideration: Generation problems

If we are working with generation problems that fare well with regards to the first two considerations, especially in the case where there are many solutions with large automorphism groups, we will need canonicity testing in the branch-and-cut tree in order to solve these problems in a reasonable amount of time. This is demonstrated by the enumeration of the two maximum 2-$(10, 3, 1)$ packings (Table 8.17) and the enumeration of all maximal $(7, 3, 1)$ intersecting set systems (Table 8.30).

Fifth consideration: Canonicity testing depth

In almost all examined cases, using canonicity testing for the entire branch-and-cut tree was either preferable or not significantly worse than using canonicity testing up to a certain depth and then turning it off. In the case of generation, turning off canonicity proved to be extremely detrimental: final solutions, needing to be checked for isomorphism, could no longer be examined using the quicker `first-in-orbit1` algorithm (Algorithm 4.2.4), and the alternative test, `first-in-orbit2` (Algorithm 4.2.6), required far more time to accept a solution as being canonical. In the case of search, this was largely unpredictable (as demonstrated by the search for a 2-$(14, 3, 1)$ packing in Table 8.14) and likely depends on the nature of the ILP as mentioned in

the second consideration above.

In all investigated cases, it was disadvantageous to turn canonicity testing off at some point in the tree and then later re-enable it. This is due to the fact that, when canonicity testing is resumed, we must use the slower `first-in-orbit2` technique. Depending on where canonicity testing is turned back on, we may be likely to meet many nodes that may be canonical; while `first-in-orbit2` may reject nodes quickly, it may take a significant amount of time to accept a node as being canonical due to the nature of the algorithm. In a Schreier-Sims scheme where the symmetry group is isomorphic to a symmetric group, the majority of the permutations occur in the first few rows of the table; because `first-in-orbit2` backtracks over the rows corresponding to the beginning of the ordered base representing the variables fixed to 1, this number of permutations may be huge.

In conclusion, there was no case found in which it was valuable to stop canonicity testing and recommence at a later depth in the tree. When performing **search**, as solutions do not need to be checked to be nonisomorphic, it may theoretically be useful to turn off canonicity testing at some point, but that point is not easily determined. In **exhaustive generation**, it is recommended that canonicity testing be done at all nodes in the tree.

9.2 Suitability of NIBAC to solve combinatorial design problems

NIBAC fares particularly well in search problems when the upper bound on the problems are not tight, and therefore harder to solve. An example of this can be found in the search for an optimal 2-$(11, 3, 1)$ packing: the Johnson bound for the 2-$(11, 3, 1)$ packing is 18, while the packing number is 17. NIBAC was able to find a 2-$(11, 3, 1)$ packing in only 1.03 seconds, while CPLEX took 262.24 seconds (Table 8.16).

NIBAC more strongly demonstrates its capabilities in exhaustive generation problems and is able to handle problems in which the size of the symmetry group is small. For instance, because of the small size of the symmetry group for the 2-$(6, 3, 2n)$ family of designs, despite the number of blocks and large solution sizes, we were able to generate all such designs for $n \in [1, 15]$. The results for $n \in [10, 15]$ had not been previously verified. Additionally, for small problems, NIBAC was able to perform well and generate new results; this can be seen in the generation of all maximal partial triple systems, where we solved the problem for $v \in [5, 12]$.

9.3 Open questions and future work

While the NIBAC framework runs well in some instances, there are others in which it does not. We would like to extend Margot's algorithms and the NIBAC framework to deal with larger problem sizes and different types of problems. This brings to light a variety of questions regarding isomorphism testing and nonisomorphic branch-and-cut.

- The slow canonicity test, `first-in-orbit2`, which doesn't depend on 0-fixing is extremely restrictive: while it may be able to reject non-canonical nodes fairly quickly, testing canonical nodes requires large amounts of execution time. In order to avoid this limitation and further examine the possibility of turning on and off isomorphism testing in the branch-and-cut tree, we require a more efficient canonicity test. Is it possible to derive one, either through use of the Schreier-Sims group representation, or through some other group representation? Additionally, if canonicity testing is turned off at some point during an exhaustive generation, and only final solutions need to be tested, is there some faster approach (perhaps using McKay's `nauty`) that we can employ in order to determine canonicity?

- Instead of performing full canonicity tests, it may be advantageous to use isomorphism invariants that could quickly reject some noncanonical nodes. This could

be used in conjunction with, or instead of full canonicity tests.

- In our current implementation, our canonicity testing techniques depend on minimum index branching; however, we may be able to derive heuristics to choose better variables upon which to branch. Is there a possible way of generalizing Margot's algorithms so that branching variables can be chosen arbitrarily? Margot has done some research into this problem in [23].

- Our framework currently employs a depth-first tree traversal technique. If one chooses to use a group representation per node (an option available in NIBAC) instead of a group representation for the entire tree, it is possible to select any type of search tree traversal. A best-first technique is often effective for ILPs; what impact would this have on search problems?

- The framework can deal only with 0-1 variables in its current implementation: the algorithms for calculating orbits-in-stabilizers and testing canonicity have been specifically designed to exploit this restriction. However, for problems such as t-(v, k, λ) designs, by allowing for arbitrary integer values, we would not need block duplication for $\lambda > 1$. Is it possible to extend the concept of a base and modify the orbit-in-stabilizer and canonicity testing algorithms so that we can investigate nonisomorphic branch-and-cut trees for general integer problems?

More work in these areas will assist in further extending the power of isomorph-free and isomorph-reduced branch-and-cut trees in order to solve larger classes of symmetric problems.

Bibliography

[1] Alberto Caprara and Matteo Fischetti. Branch-and-cut algorithms. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*, chapter 4, pages 45–63. John Wiley and Sons, Ltd., 1997.

[2] C. J. Colbourn, M. J. Colbourn, J. J. Harms, and A. Rosa. A complete census of $(10, 3, 2)$ block designs and of Mendelsohn triple systems of order ten. III. $(10, 3, 2)$ block designs without repeated blocks. *Congressus Number.*, 37:211–234, 1983.

[3] C. J. Colbourn and A. Rosa. *Triple Systems*. Oxford University Press, 1999.

[4] C. J. Colbourn, A. Rosa, and Š. Znám. The spectrum of maximal partial Steiner triple systems. *Des. Codes Crypt.*, 3:209–219, 1993.

[5] Charles J. Colbourn and Rudolf Mathon. Steiner systems. In Charles J. Colbourn and Jeffrey H. Dinitz, editors, *The CRC Handbook of Combinatorial Designs*, pages 66–75. CRC Press, Boca Raton, 1996.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw Hill, second edition, 2001.

[7] Harlan Crowder, Ellis L. Johnson, and Manfred Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[8] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[9] Paul C. Denny. Search and enumeration techniques for incidence structures. Master's thesis, University of Auckland, 1998.

[10] R. Dorfman. The discovery of linear programming. *Annals of Computing*, 6(3):283–295, July 1984.

[11] D. R. Fulkerson. Blocking and anti-blocking pairs of polyhedra. *Math. Programming*, 1:168–194, 1971.

[12] B. Ganter, A. Gülzow, R. Mathon, and A. Rosa. A complete census of $(10, 3, 2)$ designs and of Mendelsohn triple systems of order ten. IV. $(10, 3, 2)$ designs with repeated blocks. *Kasseler Math Schriften*, 78(5), 1978.

[13] Ralph E. Gomory. *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, 1963.

[14] H.-D. O. F. Gronau and J. Prestin. Some results on designs with repeated blocks. *Rostock Math. Kolloq.*, 21:15–38, 1982.

[15] K. Hoffman and M. Padberg. LP-based combinatorial problem solving. *Annals of Operations Research*, 4(6):145–194, 1985.

[16] Karla L. Hoffman and Manfred Padberg. Solving airline crew-scheduling problems by branch-and-cut. *Management Sci.*, 39:657–682, 1993.

[17] A. V. Ivanov. Konstruktivnoye perečislenie sistem incidentnosti, III. *Rostock. Math. Kolloq.*, 24:4–22, 1983.

[18] A. V. Ivanov. Constructive enumeration of incidence systems. *Ann. Discrete Math*, 26:227–246, 1985.

[19] N. K. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[20] L. G. Khachiyan. Polynomial algorithms in linear programming. *Zhurnal Vychislitel'noi Matematiki i Matematcheskoi Fiziki*, 20:51–68, 1980.

[21] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, Boca Raton, 1999.

[22] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming*, 94:71–90, 2002.

[23] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming Ser. B*, 98:3–21, 2003.

[24] F. Margot. Small covering designs by branch-and-cut. *Mathematical Programming Ser. B*, 94:207–220, 2003.

[25] R. A. Mathon. Computational methods in design theory. In A. D. Keedwell, editor, *Surveys in Combinatorics 1991*, pages 101–117. Cambridge University Press, 1991.

[26] R. A. Mathon and D. Lomas. A census of 2-$(9, 3, 3)$ designs. *Australas. J. Combin.*, 5:145–158, 1992.

[27] R. A. Mathon and Ch. Pietsch. On the family of 2-$(7, 3, \lambda)$ designs, 1999.

[28] Rudolf Mathon and Alexander Rosa. 2-(v, k, λ) designs of small order. In Charles J. Colbourn and Jeffrey H. Dinitz, editors, *The CRC Handbook of Combinatorial Designs*, pages 3–41. CRC Press, Boca Raton, 1996.

[29] Brendan D. McKay. *nauty User's Guide (Version 2.2)*. Australian National University, Australia.

[30] N. S. Mendelsohn and S. H. Y. Hung. On the Steiner systems s(3, 4, 14) and s(4, 5, 15). *Utilitas Math.*, 1:5–95, 1972.

[31] W. H. Mills and R. C. Mullin. Packings and coverings. In Jeffrey H. Dinitz and Douglas R. Stinson, editors, *Contemporary Design Theory: A Collection of Surveys*, chapter 9, pages 371–399. John Wiley and Sons, Ltd., 1992.

[32] Lucia Moura. Computational and constructive design theory. In W. D. Wallis, editor, *Polyhedral methods in design theory*, number 368 in Math. Appl., pages 227–254. Kluwer, 1996.

[33] Lucia Moura. Maximal s-wise t-intersecting families of sets: kernels, generating sets, and enumeration. *Journal of Combinatorial Theory Ser. A*, 87(1):52–73, 1999.

[34] Lucia Moura. A polyhedral algorithm for packings and designs. In *Proceedings of the 7th Annual European Symposium on Algorithms*, pages 462–475. Springer-Verlag, 1999.

[35] Lucia Moura. *Polyhedral Aspects of Combinatorial Designs*. PhD thesis, University of Toronto, 1999.

[36] G. L. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *J. Opl. Res. Soc.*, 43(5):443–457, 1992.

[37] G. L. Nemhauser and L. E. Trotter, Jr. Properties of vertex packing and independence of system polyhedra. *Mathematical Programming*, 6:48–61, 1974.

[38] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, Ltd., New York, 1988.

[39] W. Keith Nicholson. *Introduction to Abstract Algebra*. Wiley-Interscience, New York, second edition, 1999.

[40] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, March 1991.

[41] Ch. Pietsch. On the enumeration of 2-$(7, 3, \lambda)$ block designs. *J. Comb. Math. Comb. Comput.*, 16:103–114, 1994.

[42] E. A. Severn. *Maximal Partial Steiner Triple Systems*. PhD thesis, University of Toronto, 1984.

[43] Douglas R. Stinson. Coverings. In Charles J. Colbourn and Jeffrey H. Dinitz, editors, *The CRC Handbook of Combinatorial Designs*, pages 260–265. CRC Press, Boca Raton, 1996.

[44] Douglas R. Stinson. Packings. In Charles J. Colbourn and Jeffrey H. Dinitz, editors, *The CRC Handbook of Combinatorial Designs*, pages 409–413. CRC Press, Boca Raton, 1996.

[45] Dana Wengrzik. Schnittebenenverfahren für blockdesign-probleme. Master's thesis, Technische Universität Berlin, 1995.

[46] Laurence A. Wolsey. *Integer Programming*. John Wiley and Sons, Ltd., New York, 1998.