SYMMETRY IN INTEGER PROGRAMMING

by

James Ostrowski

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Industrial and Systems Engineering

Lehigh University

December 15 2008

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Dr. Ted Ralphs
Chairman

Accepted Date

Committee:

Dr. Ted Ralphs, Co-Advisor

Dr. Jeff Linderoth, Co-Advisor

Dr. Garth Isaak

Dr. François Margot

Acknowledgments

I would like thank my advisor professor Jeff Linderoth for his guidance and assistance. His efforts have contributed a great deal to this thesis. I would also like to thank professor Fabrizio Rossi and professor Stefano Smriglio for their help in the writing of this thesis. I would also like to thank my committee members, professor Garth Isaak, professor Françios Margot, and professor Ted Ralphs for their efforts and advice. Most of my stay at Lehigh University was supported by an IGERT fellowship from the National Science Foundation and I would like to thank Dean Wu for his efforts in obtaining this funding.

I would especially like to thank my wife, Jessie, for her support and encouragement.

Contents

List of Tables

List of Figures

Abstract

This thesis focuses on solving integer programs whose feasible regions are highly symmetric. Symmetry has long been considered a curse for solving integer programs, and auxiliary (often extended) formulations are often sought to reduce the amount of symmetry in an integer linear programming (ILP) formulation. The approach taken in this work is different in that it seeks to *exploit* the symmetry, not avoid it by reformulation.

A standard method for solving integer programs is *branch-and-bound*. In branch-and-bound, the set of feasible solutions is partitioned, forming more easily-solved subproblems. The presence of symmetry means that many of these subproblems are equivalent. Only one member of each collection of equivalent subproblems needs to be solved. Failure to recognize that many subproblems are symmetric results in a waste of computational effort that can render an instance unsolvable by branch-and-bound.

In an effort to reduce the deleterious effects of symmetry, we first introduce *orbital branching*, a branching method effective for binary integer programs exhibiting symmetry. This method is based on computing sets of variables that may be equivalent with respect to the symmetry remaining in the problem after branching, including symmetry that is not present at the root node. These sets of equivalent variables, called *orbits*, are used to create a valid partitioning of the feasible region that significantly reduces the effects of symmetry. We also show how to use the symmetries present in the problem to fix variables throughout the branch-and-bound tree. Orbital branching is an effective symmetry-breaking algorithm that can be easily incorporated into standard integer programming software.

The importance of orbital branching is that it considers the effects of symmetry during the branching process. Fixing one variable through branching can often lead to the fixing of other variables as a result of symmetry. The additional variables that can be fixed by symmetry can have a significant affect on the LP relaxation solution and should be taken into account in the branching process. Through an empirical study on a test suite of symmetric integer programs, the question as to the most effective orbit on which to base the branching decision is investigated. The resulting method was shown to be quite competitive with a similar method known as *isomorphism pruning* and significantly better than a state-of-the-art commercial solver on symmetric integer programs. Another important contribution of this work is that it offers a way to identify and exploit the symmetry that arises in the problem as a result of branching decisions.

Orbital branching does not, by itself, fully exploit the symmetry present in the problem. Specifically, some redundant subproblems may still be explored. While orbital branching can be a very effective method for finding an optimal solutions, because there is no guarantee that all solutions found are non-isomorphic, it is not recommended to generate all non-isomorphic solution. Determining if the set of solutions contains only non-isomorphic solutions requires a comparison of each pair of solutions generated. The time needed to perform these tests could outweigh most of the benefits of using orbital branching.

The second major contribution of this work is the development of a modification to isomorphism pruning, the current state-of-the-art symmetry-breaking technique for ILP. Isomorphism pruning provides a way to prune nodes and set variables in a way that guarantees no two symmetric subproblems are solved. However, the current implementation of isomorphism pruning in ILP requires a very rigid branching rule. The proposed method removes any restrictions on the choice of branching without the need for significantly more computational effort than the standard isomorphism pruning algorithm. The modification to isomorphism pruning allows us to use orbital branching to incorporate symmetry information into branching, strengthening the branching disjunctions.

Unlike orbital branching, isomorphism pruning can use information provided by the symmetry group to prune nodes in the branch-and-bound tree. This pruning ensures that all solutions found are non-isomorphic. With a guarantee of complete symmetry removal, isomorphism pruning is an ideal choice when generating all optimal solutions.

For many integer programs, the partitioning of the feasible region for branch-and-bound search is best done via general disjunctions, rather than simple variable disjunctions. The third major contribution is to extend the concept of orbital branching to this more general case. Using the symmetry of the problem, we can generate collections of symmetric inequalities that can be used to partition the feasible region. A major difficulty with branching on general disjunctions is determining how to generate them. Many highly symmetric problems contain subproblems with the same structure as the original problem, and the subproblems can be used to define effective branching disjunctions. In addition, if there are relatively few near-optimal (non-isomorphic) solutions to the embedded subproblems, the feasible region can be partitioned based on these solutions. The subproblems resulting from this partition can be easily solved in parallel. The power of the methods presented in chapter 5 are demonstrated by proving the optimality, for the first time, of well-known instances of Steiner Triple Systems of incidence-widths of 135 and 243.

Chapter 1

Introduction

1.1 Motivation

Even though finding an optimal solution to a pure integer linear program is a NP-hard problem [69], many large instances can be solved in a reasonable amount of time. Advanced techniques such as cutting planes, preprocessing, and heuristics have contributed to this great success and turned integer linear programming into a practical success [8]. However, significant challenges remain. Symmetry is one of those challenges. This thesis focuses on techniques for improving the solvability of difficult symmetric instances of integer linear programs.

In general, it is possible to solve a typical pure integer linear program (ILP) instance with as many as tens of thousands of variables. For instance, MIPLIB 2003 [58], a collection of challenging ILPs, contains problems with as many as 87,000 variables that can be solved within 2 hours. However, problems with a large degree of symmetry containing merely 100's of variables remain unsolved. Most notably, an instance of the football pool problem that contains only 743 variables remains unsolved despite enormous computational effort [46]. This thesis aims to close that gap.

Standard formulations of many different classes of important problems exhibit symmetry. One of these is graph coloring problems, a class of great importance both for its theoretical results and its application to many real world problems. In fact, the popular puzzle game, Sudoku, can be thought of as a graph coloring problem. Typical ILP formulations of graph coloring problems contain symmetry. In addition to graph coloring, symmetry appears in job scheduling problems when there are identical machines, covering design problems that have applications in statistics, and code construction.

1.2 Integer Programming

A (pure) integer linear program is the problem of finding values of decision variables that maximize a linear function, subject to a set of linear constraints with the additional restriction that values of all variables should be integral. An ILP can be written as

$$Z_{ILP} = \max_{x \in \mathbb{Z}_+^n} \{c^T x \mid Ax \leq b\} \tag{ILP}$$

where $\mathbb{Z}_+^n$ denotes the set of n-dimensional non-negative integer vectors, $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, and $c \in \mathbb{Q}^n$ where $\mathbb{Q}$ denotes the set of rational numbers. The set $\{x \in \mathbb{Z}_+^n | \ Ax \leq b\}$ is called the *feasible region* of the ILP, which we denote as $\mathcal{F}$. A point is *feasible* if it is in the feasible region. The function that is maximized, $c^T x$, is known as the *objective function*. The problem (ILP) is known to be NP-hard, and there is a wide body of research devoted to solving ILPs. One approach uses the linear program (LP) relaxation. The relaxation

$$Z_{LP} = \max_{x \in \mathbb{R}_+^n} \{c^T x \mid Ax \leq b\} \tag{LP}$$

is formed by relaxing the integrality constraints of (ILP). The space $\{x \in \mathbb{R}_+^n \ | Ax \leq b\}$ is called the *feasible region* of the LP relaxation and will sometimes be denoted as $\mathcal{F}$(LP). The values Z_{ILP} and Z_{LP} are not guaranteed to exist. The values Z_{ILP} and Z_{LP} do not exist if $\mathcal{F}$(LP) is empty (in which case neither exist), or if $\mathcal{F}$(LP) is non-empty but contains no integer points (Z_{LP} exists, but Z_{ILP} does not). In this case we call the ILP problem *infeasible*. If $\mathcal{F}$(LP) contains a sequence of points $\{x^i\}_{i=1}^{\infty}$ such that $\lim_{i \to \infty} c^T x^i = \infty$ then we say (LP) is *unbounded* and Z_{LP} does not exist. If there is such a sequence that consists only of integral points, then (ILP) is *unbounded* and Z_{ILP} does not exists. (LP) can be solved in polynomial time, while no polynomial time algorithm for a general (ILP) is known. Since $\mathcal{F} \subset \mathcal{F}$(LP), solving (LP) yields an upper bound on (ILP), i.e., $Z_{LP} \geq Z_{ILP}$, as some variables in an optimal solution to the relaxation may not have integer values. Information on how to solve (LP) can be found in Bertsimas and Tsitsiklis [7].

Branch-and-bound is a common method used to solve (ILP). Branch-and-bound begins by first solving the LP relaxation to obtain an optimal solution x_{LP}^*. If $x_{LP}^* \in \mathbb{Z}_+^n$, then x_{LP}^* is both a feasible solution and an optimal solution to (ILP). If $x_{LP}^* \notin \mathbb{Z}^n$, then x_{LP}^* must be removed from the region that is being searched. In branch-and-bound, removing a fractional solution is done by *branching*. For a vector $\phi \in \mathbb{Z}^n$, no feasible solution $x \in \mathcal{F}$ satisfies $\lfloor \phi^T x_{LP}^* \rfloor < \phi^T x < \lceil \phi^T x_{LP}^* \rceil$. Thus, (ILP) can be solved by dividing (ILP) into two smaller problems, one with the additional constraint $\phi^T x \leq \lfloor \phi^T x_{LP}^* \rfloor$, and the other with the additional constraint $\phi^T x \geq \lceil \phi^T x_{LP}^* \rceil$. Because x_{LP}^* does not satisfy either the constraint $\phi^T x \leq \lfloor \phi^T x_{LP}^* \rfloor$ or the constraint $\phi^T x \geq \lceil \phi^T x_{LP}^* \rceil$, x_{LP}^* is not feasible in the LP relaxation of either of the smaller problems. Traditionally, the disjunction is obtained by choosing $\phi = e_j$ for a variable x_j with $\lfloor x_{j_{LP}}^* \rfloor < x_{j_{LP}}^* < \lceil x_{j_{LP}}^* \rceil$. This is called *branching on a variable*.

This thesis focuses on *binary integer linear programs*, ILPs where x is restricted to $\{0,1\}^n$, not $\mathbb{Z}^n$. Constraints added to subproblems through branching either fix a fractional variable x_i to 0 or to 1. A subproblem $a = (F_1^a, F_0^a)$ can then be identified by the set of variables fixed to 1, F_1^a, and the set of variables fixed to 0, F_0^a.

Adding inequalities as a result of branching disjunctions create *subproblems*. These subproblems are known as *children*. The problem instance from which the child nodes are created is called the *parent*. The original problem is the *root*. Each of the children are solved in a similar way to the parent. If at any point in the process we find a solution $\hat{x}$ that is both an optimal solution to the LP relaxation and integral, the value $c^T\hat{x}$ can be used as a lower bound for (ILP). If there is any subproblem whose LP relaxation is less than the lower bound, that subproblem cannot contain an optimal solution. As a result, the subproblem can be discarded. Removing a subproblem whose LP bound is less than the lower bound is referred to as *pruning* by bound. Subproblems can also be pruned when the constraints added make the subproblem infeasible. Branch-and-bound can be expressed by a tree where each node represents a subproblem created by branching. Two nodes are adjacent in this tree if and only if they represent a parent-child pair. The tree associated with the search is the *branch-and-bound tree*.

At any node in the branch-and-bound tree, there may be multiple variables that have fractional values in the LP solution. Any of the fractional variables may be chosen for branching, however, variables should be chosen with care. It is well known that the choice of variable on which to branch can significantly affect the solution time [48]. The basis for solving (ILP) is to increase the lower bound (by finding better feasible solutions) while decreasing the upper bound (by adding constraints by branching) until the bounds meet. These two competing goals can lead to conflicting branching strategies. However, branching rules are generally aimed at reducing the upper bound [47] [1]. It is generally assumed throughout this thesis that the lower bounds on (ILP) are found using heuristics and not the responsibility of the branch-and-bound process. By not attempting to find solutions, branch-and-bound can focus its efforts on decreasing the upper bound as quickly as possible. Decreasing the bound is a result of the change in the LP relaxation solution that comes from branching. Variables that decrease the LP relaxation significantly when branched upon are ideal candidates for branching. Therefore, determining the variables that effect the LP solution is very important.

1.2.1 Symmetry in Integer Programming

The focus in this thesis is on cases where an (ILP) is highly symmetric, a concept that will be formalized later in this section. Some intuition can be illustrated with the following example from Bertsimas and Tsitsiklis [7]:

$$\min_{x\in\{0,1\}^{n+1}} \{x_{n+1} \mid 2x_1 + 2x_2 + \ldots + 2x_n + x_{n+1} = 2k+1\} \text{ where } k \in \mathbb{Z}_+,\ k \leq n. \tag{1.1}$$

The integer program in (1.1) can be easily solved without the use of a computer for any choice of n and k. The sum

$2x_1 + 2x_2 + \ldots + 2x_n + x_{n+1}$ must always be odd, forcing x_{n+1}, and hence the objective function, to take the value 1. Despite the problem's obvious solution, traditional branch-and-bound methods like those described in Section 1.2 cannot solve even modest-sized instances. Table 1.1 gives computational results obtained from solving small problem instances of (1.1) using the commercial solver CPLEX with its advanced features turned off.

n	**k**	**Time** (seconds)	**Nodes**
20	5	3.24	54,262
20	6	6.97	116,278
20	7	12.24	203,400
20	8	17.83	293,928
20	9	21.68	352,714
20	10	21.74	352,714
25	5	13.86	23,228
25	6	38.96	657,798
25	7	92.71	1,562,273
30	5	42.7	736,284
30	6	160.15	2,629,573

Table 1.1: Computational Effort Required to Solve ILP

For any $k \in \mathbb{Z}$, $0 \leq k < n$, there will be a fractional solution feasible to the LP relaxation of (1.1) with $x_{n+1} = 0$. Thus, nodes in the branch-and-bound-tree will not be pruned until either k variables are fixed to 1 or $n - k$ variables are fixed to zero. As a result, the enumeration tree will contain at least $\min(2^k, 2^{n-k})$ nodes. Inspection of the enumeration tree, however, reveals that most of the work performed by traditional branch and bound is unnecessary. Many nodes in the tree represent identical subproblems that only need to be solved once.

To illustrate this phenomenon, consider the specific instance

$$\min_{x \in \{0,1\}^5} \{x_5 \mid 2x_1 + 2x_2 + 2x_3 + 2x_4 + x_5 = 3\}. \tag{1.2}$$

The tree in Figure 1.1 is generated using traditional branch and bound methods, by branching on the fractional variable with the smallest index. This branching method is, in general, a poor branching strategy. However, in this instance, the branching strategy is guaranteed to produce the smallest tree. Only nodes for which the LP relaxation is feasible are included in the figure. The LP relaxations of all non-leaf nodes in the tree have optimal values of 0, while the optimal solutions to each of the leaf nodes is also an optimal solution to the ILP (and hence has an objective value of 1).

Consider node D, the subproblem formed by fixing x_1 to zero and x_2 to one. We can rewrite the subproblem as

$$\min_{x \in \{0,1\}^5} \{x_5 \mid 2x_3 + 2x_4 + x_5 = 1\}. \tag{D}$$

Now consider node E, the subproblem formed by fixing x_1 to zero and x_2 to one. This subproblem can also be

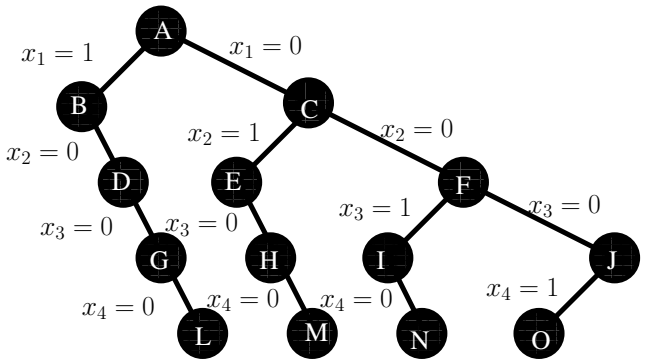


Figure 1.1: Enumeration Tree for ILP Instance 1.2

written as

$$\min_{x \in \{0,1\}^5} \{x_5 \mid 2x_3 + 2x_4 + x_5 = 1\}. \tag{E}$$

The subproblem at node D is identical to the subproblem at node E. It is easy to see why traditional methods have difficulty with this type of problem as both nodes D and E are solved. It is not necessary to consider *both* nodes D and E, but traditional methods provide no way of recognizing that two nodes are identical. Further inspection shows that there are many other nodes that represent identical subproblems (for instance, nodes G, H, and I represent the same subproblem, as well as nodes L, M, N, and O).

Not only are identical subproblems solved multiple times; the subproblems themselves can be difficult to solve. Note that an optimal solution is feasible in each node in Figure 1.1. Nodes whose feasible region contains an optimal solution are more difficult to prune than nodes that do not because nodes with optimal solutions need more of the integrality gap to be closed.

A good branch-and-bound algorithm should avoid solving identical subproblems. Ideally, sets of identical subproblems should be recognized during the branching process and all but one member of each set should be pruned. For example, node E can be pruned because it is identical to node D; nodes I and O can be pruned because they are identical to nodes G and L (respectively). Pruning to avoid this repetition will significantly reduce the number of nodes in the enumeration tree, making even very large symmetric problems solvable by integer programming methods.

1.3 Mathematical Preliminaries

1.3.1 Group Theory

The many identical subproblems in the example arise because of the symmetry in (1.2). To formalize the definition of symmetry, we need to briefly introduce some basic concepts from group theory. A thorough review of group theory

can be found in L.C. and C.T. [45], J.J. [37], and P.J. [71].

A nonempty, finite, set of elements $\mathcal{G}$ together with a binary operation denoted by " $\circ$" is called a *group* if the following properties hold:

i $a,\ b \in \mathcal{G}$ implies $a \circ b \in \mathcal{G}$. (The set is closed under $\circ$).

ii $a,\ b,\ c\ \in \mathcal{G}$ implies $a \circ (b \circ c) = (a \circ b) \circ c$. (The operator $\circ$ is associative).

iii There exists $e \in \mathcal{G}$ with $a \circ e = e \circ a = a$ for all $a \in \mathcal{G}$. (The group contains an identity element).

iv For every $a \in \mathcal{G}$, there exists $a^{-1} \in \mathcal{G}$ with $a \circ a^{-1} = e$. (For every element in the group the group also contains that element's inverse).

Let I^n be the ground set $\{1, \ldots,\ n\}$. Let S^n be the collection of permutations of the ground set, i.e., the collection of all functions π such that $\pi : I^n \rightarrow I^n$ is a one-to-one and onto (bijective) mapping. The only binary operation acting on permutations throughout this thesis is the *composition* operation, i.e., $\pi_i \circ \pi_j = \pi_i(\pi_j)$. Thus, for notational convenience, a group containing permutations will be referred to only by the set of elements $\mathcal{G}$. S^n is the *symmetric group* of I^n. Any subset of S^n that satisfies the properties of a group is called a *permutation group*. To provide emphasis and to distinguish S^n from a general permutation group, S^n will often be referred to as *the complete symmetric group*.

Given a vector $\lambda \in \mathbb{R}^n$ and a permutation group $\mathcal{G}$, $\pi \in \mathcal{G}$ acts on λ by permuting its coordinates: $\pi(\lambda) = (\lambda_{\pi_1}, \lambda_{\pi_2}, \ldots \lambda_{\pi_n})$. A permutation group can also act on a collection of vectors. For $X \subset \mathbb{R}^n$, $X^{\mathcal{G}}$ is given by $\{\pi(x) \mid x \in X \text{ and } \pi \in \mathcal{G}\}$.

Throughout this thesis, *cyclic notation* is used to describe permutations. For any $i \in \{1, \ldots,\ n\}$ and permutation π, π's action on i can be represented by the sequence $(i,\ \pi(i),\ \pi^2(i),\ \ldots)$, where $\pi^2(i) = \pi(\pi(i))$. Because $\mathcal{G}$ is a finite group, no element of this sequence is distinct. Let p be the first power of π such that $\pi^p(i) = i$. The cycle $(i,\ \pi(i),\ \pi^2(i),\ \ldots,\ \pi^{p-1}(i))$ can be used to describe how the permutation acts on a subset of the elements. The permutation maps i to $\pi(i)$. It also sends the element $\pi(i)$ to $\pi^2(i)$. The set I^n can be partitioned using subsets formed by cycles of π. The collection of cycles formed by the partition of I^n defines the permutation. For example, suppose π is a permutation with $\pi(1) = 1$, $\pi(2) = 3$, $\pi(3) = 2$, $\pi(4) = 5$, $\pi(5) = 4$. The permutation π can be written as $(1)(2,\ 3)(4,\ 5)$. Since π does not permute element 1, the cycle (1) need not be included. The permutation π can be written more succinctly as $(2,\ 3)(4,\ 5)$. For any element not found in a cycle, it can be assumed that the element is not changed by the permutation.

Let $\mathcal{G}$ be an arbitrary permutation group acting on $\{1,\ 2, \ldots,\ n\}$. The group $\mathcal{G}$ can be extended to act on sets of elements. For $S \subseteq \{1,\ 2, \ldots,\ n\}$, the set $\pi(S) = \{\pi(i) \mid i \in S\}$. For a point $z \in \mathbb{R}^n$, the *orbit* of z under the action

of the group $\mathcal{G}$ is the set of all elements of $\mathbb{R}^n$ to which z can be mapped by permutations in $\mathcal{G}$, i.e.,

$$\mathrm{orb}(\mathcal{G}, z) \stackrel{\mathrm{def}}{=} \{z' \in \mathbb{R}^n \mid \exists \pi \in \mathcal{G} \text{ such that } z' = \pi(z)\} = \{\pi(z) \mid \pi \in \mathcal{G}\} = z^{\mathcal{G}}.$$

By definition, if $j \in \mathrm{orb}(\{k\}, \mathcal{G})$, then $k \in \mathrm{orb}(\{j\}, \mathcal{G})$, i.e., the elements j and k share the same orbit. Therefore, the union of the orbits

$$\mathcal{O}(\mathcal{G}) \stackrel{\mathrm{def}}{=} \bigcup_{j=1}^{n} \mathrm{orb}(\{j\}, \mathcal{G})$$

forms a partition of $I^n = \{1, 2, \ldots, n\}$, that is referred to as the orbital partition of $\mathcal{G}$, or simply the *orbits* of $\mathcal{G}$. Any two elements of I^n that share an orbit under the group $\mathcal{G}$ are *equivalent* or *symmetric* with respect to $\mathcal{G}$.

The *stabilizer* of a set $S \subseteq I^n$ in $\mathcal{G}$ is the set of permutations in $\mathcal{G}$ that send S to itself:

$$\mathrm{stab}(S, \mathcal{G}) = \{\pi \in \mathcal{G} \mid \pi(S) = S\}.$$

The stabilizer of S is a subgroup of $\mathcal{G}$. The set of permutations that stabilize each set in the collection $\{S_1,\ S_2, \ldots,\ S_k\}$ is written as

$$\mathrm{stab}(S_i,\ S_2,\ \ldots,\ S_k, \mathcal{G}) = \bigcap_{j=1}^{k} \mathrm{stab}(\{S_j\}, \mathcal{G}).$$

For any collection of permutations, $\mathcal{A}$, $\langle \mathcal{A} \rangle$ is defined to be the smallest group containing all permutations in $\mathcal{A}$. The group $\langle \mathcal{A} \rangle$ is the group generated by $\mathcal{A}$, and similarly the set $\mathcal{A}$ is a *generating set*, or *generator*, of $\langle \mathcal{A} \rangle$. Thus, any group can be compactly represented by a generating set.

Example Let $\mathcal{A} \subset S^4$ consist of the permutations $\pi^1 = (2, 3)$ and $\pi^2 = (1, 4)$. $\langle \mathcal{A} \rangle = \{(), (2, 3),\ (1, 4),\ (1, 4)(2, 3)\}$ is the smallest group containing $\mathcal{A}$. The permutations π_1 and π_2 are generators of $\langle \mathcal{A} \rangle$. Because $\pi_1(1) = 1$ and $\pi_1(4) = 4$, $\pi_1 \in \mathrm{Stab}(\{1\},\ \langle \mathcal{A} \rangle)$, $\pi_1 \in \mathrm{Stab}(\{4\},\ \langle \mathcal{A} \rangle)$, and $\pi_1 \in \mathrm{Stab}(\{1\},\ \{4\},\ \langle \mathcal{A} \rangle)$. Also, $\pi_1 \in \mathrm{Stab}(\{2,\ 3\},\ \langle \mathcal{A} \rangle)$ since elements 2 and 3 are not mapped outside the set $\{2,\ 3\}$. Since π_1 maps element 2 to 3, we have $\mathrm{Orb}(\{2\},\ \langle \mathcal{A} \rangle) = \{2,\ 3\}$. The orbital partition, $\mathcal{O}(\langle \mathcal{A} \rangle)$, is $(\{1,\ 4\}, \{2,\ 3\}$. We can also consider orbits of sets of elements. For example, $\mathrm{orb}(\{1,\ 2\},\ \langle \mathcal{A} \rangle) = (\{1,\ 2\}, \{1,\ 3\}, \{2,\ 4\}, \{3,\ 4\})$.

For any subgroup $\mathcal{H}$ of $\mathcal{G}$ and element $\pi \in \mathcal{G}$, the set $\mathcal{H} \circ \pi = \{\sigma \circ \pi \mid \sigma \in \mathcal{H}\}$ is a *right coset* of $\mathcal{G}$.

1.3.2 Order Theory

To effectively deal with symmetry, nodes in the branch-and-bound tree that represent equivalent subproblems must be recognized and all but one node pruned. To ensure that at least one subproblem is not pruned, pruning techniques require inducing an order on sets of equivalent subproblems or sets of equivalent solutions. Nodes are pruned or

kept based on this order. This section provides a brief introduction to concepts in order theory. More information on ordering can be found in Davey and Priestley [11].

A *quasi order* $\lesssim$ is an order on a set $\mathcal{S}$ that is reflexive and transitive, i.e., for all a, b in $\mathcal{S}$, we have that:

Reflexive: $a \lesssim a$

Transitive: $a \lesssim b$ and $b \lesssim c$ implies $a \lesssim c$

The set $\mathcal{S}$ along with the quasi order $\lesssim$ is a *quasi ordered set*.

A *total order* is a relation, $\preceq$, on a set $\mathcal{S}$ that is antisymmetric, transitive, and total, i.e. for every a, b in $\mathcal{S}$:

Antisymmetric: $a \preceq b$ and $b \preceq a$ implies $a = b$

Transitive: $a \preceq b$ and $b \preceq c$ implies $a \preceq c$

Totality: $a \preceq b$ or $b \preceq a$

Note that totality implies reflexivity. The set $\mathcal{S}$ along with a total order $\preceq$ is a *totally ordered set*. For every total order $\preceq$, there is a *strict order* $\prec$ with $a \prec b$ implies $a \preceq b$ and $a \neq b$.

It is important to note that because $\lesssim$ is a quasi order, it is not antisymmetric. The relations $a \lesssim b$ and $b \lesssim a$ does not imply that $a = b$.

1.3.3 Graph Theory

Many symmetric ILPs have their origin in graph theory. A *graph* $G = (V, E)$ consists of a finite set of *vertices*, V, and a set E of unordered pairs of vertices, called *edges*. Each edge $e = \{i, j\} \in E$ is *incident* to vertices i and j. Two vertices are called adjacent if there is an edge incident to both vertices.

For two disjoint collections of vertices S and M, S *covers* M if each vertex in M is adjacent to at least one vertex in S. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E$. A *clique* is a graph $G = (V, E)$ such that for every $i, j \in V$, the edge (i, j) is in E.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there exists a bijection $\phi : V \to V'$ where $(a, b) \in E$ if and only if $(\phi(a), \phi(b)) \in E'$. The function ϕ is then an isomorphism of G and G'. An isomorphism sending G to itself is called an automorphism. The set of automorphisms of a graph G is referred to as $\mathrm{Aut(G)}$. $\mathrm{Aut(G)}$ is a group which will sometimes be referred to as the permutation group of the graph G.

A graph $G = (V, E)$ is *colored* if the graph is associated with functions $c_v : V \to \mathbb{R}$ and $c_E : E \to \mathbb{R}$. For any $v \in V$, vertex v is assigned color $c_v(v)$. Similarly, for any $e \in E$, edge e is colored $v_E(e)$. The automorphism group

of G colored by c_V and c_E is a subset of $\mathrm{Aut}(G(V,E))$ containing permutations that map vertices to vertices of like color and edges to edges of like color.

$$\mathrm{Aut}(G(V,E),c_v,c_E) = \{\pi \in \Pi^{|V|} \mid \pi \in \mathrm{Aut}(G(V,E)),\ c_V(\pi(v)) = c_V(v) \text{ and } c_E(\pi(e)) = c_E(e)\ \forall v \in V,\ e \in E\}.$$

A *bipartite graph* $G(V,V',E)$ is a graph with vertices V and V' where no edge in E is adjacent to either two vertices in V or two vertices in V'. A *dominating set* for a graph $G = (V,E)$ is a subset D of V such that every vertex in V is adjacent to at least one vertex in D. More information on graph theory can be found in West [82].

1.4 Symmetry Groups of Integer Linear Programs

The *symmetry group* $\mathcal{G}(ILP)$ of an integer program is the collection of permutations in S^n that map every feasible solution of (ILP) of value t to another feasible solution of (ILP) also of value t [57]. Formally,

$$\mathcal{G}(ILP) \stackrel{\text{def}}{=} \{\pi \subseteq S^n \mid \forall x \in \mathcal{F}(ILP),\ \pi(x) \in \mathcal{F} \text{ and } c^T\pi(x) = c^T x\}.$$

Computing the symmetry group of a general ILP is NP-hard, and typically more difficult than solving the ILP itself. As a result, practical methods aimed at exploiting symmetries are forced to use a subset of the symmetry group that is found by examining the problem formulation.

Given a permutation $\pi \in S^n$ and a permutation $\sigma \in S^m$, let $A(\pi,\sigma)$ be the matrix obtained by permuting the columns of A by π and the rows of A by σ, i.e., $A(\pi,\sigma) = P_\sigma A P_\pi$, where P_σ and P_π are appropriate permutation matrices. The *formulation group* $\mathcal{G}(A,b,c)$ of (ILP) is the set of permutations

$$\mathcal{G}(A,b,c) \stackrel{\text{def}}{=} \{\pi \in \Pi^n \mid \pi(c) = c,\ \exists\, \sigma \in S^m \text{ with } \sigma(b) = b \text{ such that } A(\pi,\sigma) = A\}.$$

For any $\pi \in \mathcal{G}(A,b,c)$, if $\hat{x}$ is feasible for (ILP) (or the LP relaxations of (ILP)), then $\pi(\hat{x})$ is also feasible. Moreover, the solutions $\hat{x}$ and $\pi(\hat{x})$ have the same objective value. Thus, $\mathcal{G}(A,b,c) \subseteq \mathcal{G}(ILP)$. As an example, we consider the following ILP:

Example

$$\min \sum_{i=0}^{8} x_i$$

$$\begin{aligned}
\text{subject to} \quad & x_0 + x_1 + x_2 + x_3 + x_6 \geq 1 \\
& x_0 + x_1 + x_2 + x_4 + x_7 \geq 1 \\
& x_0 + x_1 + x_2 + x_5 + x_8 \geq 1 \\
& x_0 + x_3 + x_4 + x_5 + x_6 \geq 1 \\
& x_1 + x_3 + x_4 + x_5 + x_7 \geq 1 \\
& x_2 + x_3 + x_4 + x_5 + x_8 \geq 1 \\
& x_0 + x_3 + x_6 + x_7 + x_8 \geq 1 \\
& x_1 + x_4 + x_6 + x_7 + x_8 \geq 1 \\
& x_2 + x_5 + x_6 + x_7 + x_8 \geq 1 \\
& x \in \{0,1\}^9
\end{aligned}$$

This problem amounts to finding the smallest dominating set of the graph G in Figure 1.2.

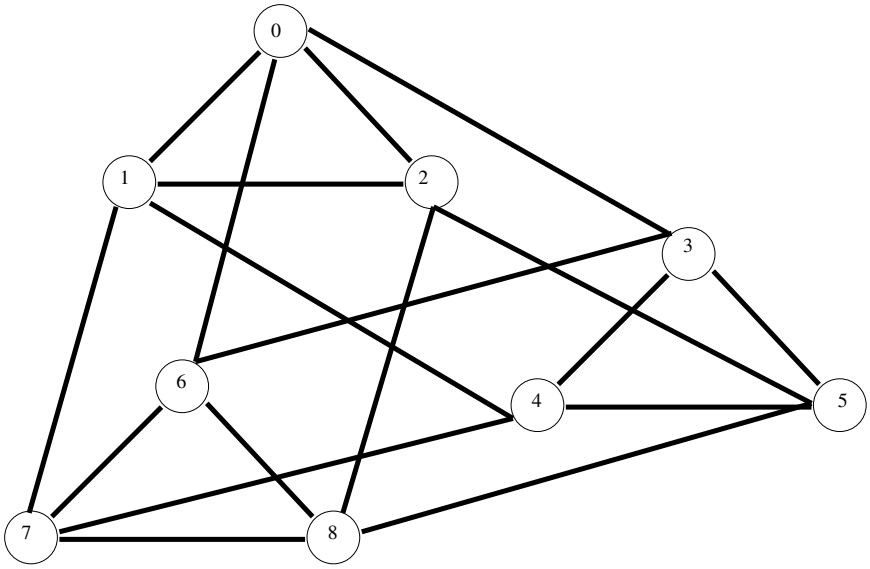


Figure 1.2: Graph Resulting in a Symmetric ILP.

The formulation group $\mathcal{G}(A, b, c)$ in this example contains 72 permutations. However, it can be generated using just 4 permutations. The generators are listed in table 1.2 along with the corresponding σ such that $A(\pi, \sigma) = A$.

π	σ
(3, 6)(4, 7)(5, 8)	(3, 6)(4, 7)(5, 8)
(1, 2)(4, 5)(7, 8)	(1, 2)(4, 5)(7, 8)
(1, 3)(2, 6)(5, 7)	(1, 3)(2, 6)(5, 7)
(0, 1)(3, 4)(6, 7)	(0, 1)(3, 4)(6, 7)

Table 1.2: Generators for $\mathcal{G}$(A,b,c).

Because A is a symmetric matrix, each permutation π and its corresponding σ are identical. For general ILP problems, it is unlikely that any such pair of permutations will be identical. Indeed, if A is not a square matrix, π and σ will not act on the same set of elements. For this example, it can be shown that $\mathcal{G}(A, b, c) = \mathrm{Aut(G)}$.

The orbital partition $\mathcal{O}(\mathcal{G}(A, b, c))$ consists of just one orbit, $\{0,\ 1,\ \ldots,\ 8\}$. The stabilizer of 0 (i.e. $\mathrm{stab}(\{0\}, \mathcal{G}(A, b, c))$) contains only 8 permutations, shown in Table 1.3.

π
(3, 6)(4, 7)(5, 8)
(1, 2)(4, 5)(7, 8)
(1, 3)(2, 6)(5, 7)

Table 1.3: Generators for Stab$(\{0\}, \mathcal{G}(A, b, c))$.

The orbits of the stabilizer $\mathrm{stab}(\{0\}, \mathcal{G}(A, b, c))$ are $\{0\},\ \{1,\ 2,\ 3,\ 6\}$, and $\{4,\ 5,\ 7,\ 8\}$.

An optimal dominating set for the graph is the set $\{0,\ 3,\ 6\}$. We can find other equivalent solutions by permuting this set with any of the permutations in $\mathcal{G}(A, b, c)$. For example, consider permuting the set $\{0,\ 3,\ 6\}$ with the last generator, $(0,\ 1)(3,\ 4)(6,\ 7)$. In this case, $\{0,\ 3,\ 6\}$ is mapped to the set $\{1,\ 4,\ 7\}$. The cover consisting of $\{1,\ 4,\ 7\}$ is then an equivalent optimal solution.

The set of all optimal dominating sets equivalent to $\{0,\ 3,\ 6\}$, written as $\{0,\ 3,\ 6\}^{\mathcal{G}(A,b,c)}$, is the collection of sets:

$$\{0,\ 3,\ 6\}^{\mathcal{G}(A,b,c)} = \{\{0,\ 3,\ 6\},\ \{1,\ 4,\ 7\},\ \{2,\ 5,\ 8\}\}.$$

Not all optimal dominating sets are equivalent to the set $\{0,\ 3,\ 6\}$. For example, there is no permutation that maps the set $\{0,\ 3,\ 6\}$ to $\{0,\ 4,\ 8\}$. It is easy to see that $\{0,\ 3,\ 6\}$ and $\{0,\ 4,\ 8\}$ are not equivalent by noting the set of vertices $\{0,\ 3,\ 6\}$ forms a clique in Figure 1.2, where the set of vertices $\{0,\ 4,\ 8\}$ forms an independent set. Another point of interest can be seen by examining all dominating sets equivalent to $\{0,\ 4,\ 8\}$.

$$\{0,\ 4,\ 8\}^{\mathcal{G}(A,b,c)} = \{\{0,\ 4,\ 6\},\ \{0,\ 5,\ 7\},\ \{1,\ 3,\ 8\},\ \{1,\ 5,\ 6\},\ \{2,\ 3,\ 7\},\ \{2,\ 5,\ 6\}\}.$$

The number of solutions equivalent to a given solution is solution-dependent, making it difficult to predict how many equivalent optimal solutions exist.

The equivalence of solutions induced by symmetry is a major factor that might confound the branch-and-bound process. For example, suppose x^* is an (integral) optimal solution to a binary ILP. At the root node, the decision is made to branch on variable x_j, creating one subproblem by fixing $x_j = 0$, and another by fixing $x_j = 1$. If $\exists \pi \in \mathcal{G}(ILP)$ such that $[\pi(x^*)]_j = 1 - x^*_j$, then x^* is a feasible solution for one child node and $\pi(x^*)$ is feasible for the other child node. In general, subproblems where feasible regions contain optimal solutions will be more difficult to solve because the proportion of the integrality gap that must be closed is greater than a subproblem that does not contain any optimal solution. If the cardinality of $\mathcal{G}(ILP)$ is large, then there may be many such subproblems, leading to long solution times.

Note that the formulation group is very dependent on the problem formulation. For example, consider adding the constraint $\sum_{i=0}^{8} 2^i x_i \geq 0$ to Problem 1.2. This constraint does not remove any feasible points from the LP relaxation and does not remove symmetry from the ILP, but it does remove all of the symmetry from the formulation.

1.5 Computing Symmetry Groups and Formulation Groups

Computation of the formulation group $\mathcal{G}(A, b, c)$ is done by computing the automorphism group of a related colored graph. An ILP is transformed into a colored complete bipartite graph $G(A, b, c) = (N, M, E)$. For vertex n_i in the set $N = \{n_1, n_2, \ldots, n_n\}$, $c_V(n_i) = c_i$. For vertex m_j in set $M = \{m_1, m_2, \ldots, m_m\}$, $c_V(m_j) = b_j$. For all $(n_i, m_j) \in E$, $c_E((n_i, m_j)) = a_{ij}$ (edges can be omitted from $G(A, b, c)$ if $a_{ij} = 0$).

Theorem 1.1 *The permutation* $\{\pi, \sigma\} \in S^{n+m}$ *with* $\pi \in S^n$ *is in* $\mathrm{stab}(N, \mathrm{Aut}(G(A, b, c)))$ *if and only if* $\pi \in \mathcal{G}(A, b, c)$.

Proof Let $\{\pi, \sigma\}$ be in $\mathrm{stab}(N, \mathrm{Aut}(G(A, b, c)))$. By definition, $c_v(\pi(n_i)) = c_v(n_i)$ for any $n_i \in N$.

By the constriction of $G(A, b, c)$ vertices n_i and n_j in N have the same color only if variables x_i and x_j have the same objective value in the ILP. Hence, $c_{\pi(i)} = c_i$ for all $i \in N$, implying $\pi(c) = c$. Similarly, $\sigma(b) = b$.

For every edge $(i, j) \in G(A, b, c)$, $c_E((\pi,\ \sigma)(i, j)) = c_E((i, j))$. Since $\pi \in S^n$ we can rewrite the edge $(\pi,\ \sigma)(i, j)$ as $(\pi(i), \sigma(j))$, so $c_E((\pi(i), \sigma(j))) = c_E((i, j))$. Edges $(\pi(i), \sigma(j))$ and (i, j) in $G(A, b, c)$ share the same color if and only if $a_{\pi(i),\sigma(j)} = a_{i,j}$. Thus, $A = A(\pi, \sigma)$, and $\pi \in \mathcal{G}$.

Let $\pi \in \mathcal{G}(A, b, c)$. By the definition of $\mathcal{G}(A, b, c)$, there is a corresponding $\sigma \in S^m$ such that $A = A(\pi, \sigma)$ and $\sigma(b) = b$. Because $\pi(c) = c$ and $\sigma(b) = b$, the permutation (π, σ) maps vertices of $G(a, b, c)$ to vertices of the same color. By definition of π, σ, for any i, j, $a_{\pi(i),\sigma(j)} = a_{i,j}$. The permutation (π, σ) maps the edge $(n_i, m_j) \in G(a, b, c)$, colored $a_{i,j}$ to $((\pi, \sigma)(n_i), (\pi, \sigma)(m_j)) = ((\pi)(n_i), (\sigma)(m_j))$, colored $a_{\pi(i),\sigma(j)} = a_{i,j}$. Thus (π, σ) is in $\mathrm{stab}(N, \mathrm{Aut}(G(A, b, c))$. □

1.5. COMPUTING SYMMETRY GROUPS AND FORMULATION GROUPS

There are several software packages that can compute the automorphism groups required to perform orbital branching. The program `nauty` by McKay [59] has been shown to be quite effective [20], and is used throughout the thesis to compute symmetry groups.

The complexity of computing the automorphism group of a graph is not known to be polynomial time. However, `nauty` computes the formulation groups of the problems studied in this thesis very quickly, generally faster than solving the LP at a given node. Times required to compute formulation groups are given in Chapter 3. One explanation for this phenomenon is that the running time of `nauty`'s backtracking algorithm is correlated to the size of the symmetry group being computed. For example, computing the automorphism group of the clique on 2000 nodes takes 85 seconds, while graphs of comparable size with little or no symmetry require fractions of a second.

An excellent resource for computational algebra algorithms is Holt [36]. We present two important algorithms from Holt [36] that are used throughout this thesis. Given a symmetry group $\mathcal{G} \subset S^n$ with $< \pi_1,\ \pi_2, \ldots,\ \pi_k >= \mathcal{G}$ and an n-vector z, the orbit of z with respect to $\mathcal{G}$ can be found using Algorithm 1.1. The complexity of finding $\mathrm{orb}(z, \mathcal{G})$ is $O(k|\,\mathrm{orb}(z, \mathcal{G})|)$.

Algorithm 1.1 Computing Orbits

Input:	Group generators $< \pi_1,\ \pi_2, \ldots,\ \pi_k >$ of $\mathcal{G}$ and n-vector z.
Output:	$\mathrm{orb}(z, \mathcal{G})$.

Step 1.	Initialize $O = \{z\}$, $S = \{z\}$.
Step 2.	While S is non-empty:
Step 2a.	For $s \in S$:
Step 2b.	For $\pi_i \in (\pi_1,\ \pi_2, \ldots,\ \pi_k)$:
Step 2c.	If $\pi_i(s) \notin O$, add $\pi_i(s)$ to O and S.
Step 2d.	Remove s from S.
Step 3.	Return O.

Sometimes it is beneficial to compute not only $\mathrm{orb}(z, \mathcal{G})$, but also to find permutations that map z to each $y \in \mathrm{orb}(z, \mathcal{G})$. Algorithm 1.1 can be updated for such a purpose. Algorithm 1.2, the `orbit-stabilizer` algorithm, returns the set $\{(i, \pi_i) \mid i \in \mathrm{orb}(z, \mathcal{G}),\ \pi(z) = i\}$.

Algorithm 1.2 Orbit Stabilizer

Input:	Group generators $< \pi_1,\ \pi_2, \ldots,\ \pi_k >$ of $\mathcal{G}$ and n-vector z.
Output:	$\triangle = \{(i, \pi_i) \mid i \in \mathrm{orb}(z, \mathcal{G}),\ \pi(z) = i\}$.

Step 1.	Initialize $\triangle = \{(z, e)\}$.
Step 2.	For $(x,\ \pi_x) \in \triangle$
Step 2a.	$\pi_i \in (\pi_1,\ \pi_2, \ldots,\ \pi_k)$:
Step 2b.	if there is not a π with $(\pi_i(x), \pi) \in \triangle$:
Step 2c.	Add $(\pi_i(x), \pi_i \circ \pi_x)$ to $\triangle$.
Step 3.	Return $\triangle$.

Computing the symmetry group $\mathrm{stab}(z, \mathcal{G})$ can be done using Algorithm 1.2 along with the following theorem from Holt [36].

Theorem 1.2 *Let $\sigma_x \in \mathcal{G}$ be any permutation mapping z to x.* $\mathrm{stab}(z, \mathcal{G}) =< \{\sigma^{-1}_{\pi(x)} \circ \pi_i \circ \sigma_x \mid \forall\, i \in \{1, \ldots, k\}, \forall x \in \mathrm{orb}(z, \mathcal{G})\} >$

1.6 Contributions

There are four fundamental contributions of this thesis:

- We develop an effective algorithm, orbital branching, for solving symmetric ILPs. Orbital branching is easily implemented in standard optimization software and can detect and exploit symmetry that is added to the problem as a result of branching decisions.
- We improve the current state-of-the-art symmetry breaking procedure, isomorphism pruning, by allowing for flexible branching. This allows isomorphism pruning to use orbital branching to make branching decisions.
- We suggest and investigate different branching strategies for both orbital branching and isomorphism pruning.
- We develop a way to exploit symmetry when branching on general disjunctions and discuss ways to exploit the structure of symmetric problems.

Each of these contributions has been implemented in software.

1.7 Outline

The current literature dealing with symmetric ILPs can be divided into two separate approaches. In the first approach, symmetry is avoided by reformulating the problem. This can be an effective strategy for dealing with symmetry, but these methods can only be used for problems with specific structures. The second, and more general method, uses the symmetry to prune nodes in the branch and bound tree. In Chapter 2, we will give a comprehensive literature review describing current methods for dealing with symmetry.

In Chapter 3 we introduce *orbital branching*, an effective branching method for integer programs containing a great deal of symmetry. This method is based on using the symmetry information to partition the variables into orbits. The orbits are then used to create a valid partitioning of the feasible region that significantly reduces the effects of symmetry. We also show how to exploit the symmetries present in the problem to fix variables throughout the branch-and-bound tree. Orbital branching can be easily incorporated into standard ILP software using only available software for computing orbits of groups arising at subproblems. Through an empirical study on a test suite of symmetric

integer programs, the question as to the most effective orbit on which to base the branching dichotomy is investigated. The resulting method is shown to be quite competitive with a similar method known as *isomorphism pruning* and significantly better than a state-of-the-art commercial solver on symmetric integer programs.

Unfortunately, orbital branching does not remove all of the negative effects brought about by symmetry. In Chapter 4, the methods of orbital branching and isomorphism pruning are combined to achieve impressive results. Isomorphism pruning fully exploits symmetry found in a problem formulation, but requires a very strict branching rule. In chapter 4 we show that with a slight revision of the proof of validity of isomorphism pruning, we can remove the branching restrictions of isomorphism pruning. Removing the branching restrictions requires a thorough investigation of branching rules. It is important to take fixings done as a result of symmetry into account when choosing variables for branching. We show that using orbital branching to make branching decisions, combined with the symmetry-removal power of isomorphism pruning, can lead to significant improvements in solving symmetric ILPs.

In Chapter 5, the orbital branching methodology is extended so that the branching disjunction can be based on an arbitrary constraint. Many important families of integer programs are structured such that small instances from the family are embedded in larger instances. This structural information can be exploited to define a group of strong constraints on which to base the orbital branching disjunction. The symmetric nature of the problems is further exploited by enumerating non-isomorphic solutions to instances of the small family and using these solutions to create a collection of typically easy-to-solve integer programs. The solution of each integer program in the collection is equivalent to the solution of the original large instance. The effectiveness of this methodology is demonstrated by computing the optimal incidence width of Steiner Triple Systems that were heretofore unsolvable.

Chapter 2

Literature Review

Techniques for dealing with symmetry can be classified into two categories. For certain problem classes, symmetry can be avoided by reformulation. Literature discussing reformulation techniques will be discussed in Section 2.1. A second approach is to remove the symmetry from the problem formulation either by fixing variables or adding additional constraints. This can be done in two ways. Static symmetry-breaking methods detect and exploit symmetry before the solution procedure begins as a preprocessing step while dynamic methods exploit symmetry during the branch-and-bound process. Static methods will be discussed in Section 2.3 and dynamic methods will be discussed in Section 2.4. While many of these methods have been developed by the constraint programming community, they can easily be described in an ILP context.

2.1 Avoiding Symmetry by Reformulation

A popular method for avoiding the negative effects of symmetry in integer programming is reformulating the problem. Reformulation techniques attempt to rewrite the problem in such a way that symmetry is removed. Often, reformulation techniques lift the ILP to a higher dimension where the symmetry does not appear. This section presents a popular reformulation technique that is only applicable to problems with a specific structure. Specifically, this reformulation method can be used if the variables of the ILP can be expressed as an $n \times m$ matrix where $\mathcal{G}(ILP)$ contains all 2^n permutations of the matrix columns. Cutting stock problems (with rolls of identical widths), graph coloring problems, and generalized assignment problems (with identical machines) are examples of problems with this structure. Since problem eligible for reformulation have a similar form, the cutting stock problem will be used as an example.

In the cutting stock problem, M items of varying width are cut from rolls of metal. The goal is to minimize the number of rolls of metal needed to manufacture a predetermined amount of each of the M items. The Kantorovich model [41] for cutting stock problems is the following

$$\begin{aligned} \min \sum_{k=1}^{K} x_0^k \\ \text{s.t. } \sum_{k=1}^{K} x_i^k \geq b_i \ \ \forall i \in \{1, \ \ldots, \ M\} \\ \sum_{i=1}^{m} w_i x_i^k \leq W x_0^k \ \ \forall k \in \{1, \ \ldots, \ K\} \\ x_0^k \in \{0, \ 1\}, \ x_i^k \geq 0, \ x_i^k \in \mathbb{Z} \end{aligned} \tag{2.1}$$

An upper bound on the number of rolls, K, is needed and can be found using heuristics. The variable x_0^k is 1 if roll k is used and x_i^k represents the number of items of type i that are cut on roll k. The width of item i is w_i and the demand is b_i. Symmetry arises in this problem when all rolls have the same width W. For any solution to (2.1), multiple equivalent solutions exist. Specifically, items that are cut on roll 1 can instead be cut on roll 2 (meanwhile, cutting all items that our solution tells us to cut on roll 2 and instead cut them on roll 1). Specifically, for any rolls k and l, the permutation $\pi_{k,l}$ defined by $\pi_{k,l} = (x_0^k, \ x_0^l)(x_1^k, \ x_1^l)(x_2^k, \ x_2^l) \ldots (x_m^k, \ x_m^l)$ is in the symmetry group of the problem. If the $K \times (K+1)$ matrix X were defined by $x_{i,j} = x_i^j$ for all $i \in \{1, \ \ldots, \ M\}$ and $j \in \{0, \ \ldots, \ K\}$, then the permutation $\pi_{k,l}$ corresponds to permuting the kth column with the lth column. Any permutation of the columns of X correspond to a permutation in the symmetry group of the stock cutting problem.

Gilmore and Gomory [30] proposed a reformulation of the Kantorovich model in terms of pattern variables. Each pattern $A^p = (a_1^p, \ \ldots, \ a_m^p)^T$ is an n-vector where a_i^p represents the number of items of type i that are cut in pattern p. Let P be the set of feasible patterns. The reformulation of Gilmore and Gomory is:

$$\begin{aligned} \min \sum_{p \in P} \lambda^p \\ s.t. \sum_{p \in P} a_i^p \lambda^p \geq \ b_i, \text{ for } i = 1 \ldots m \\ \lambda^p \geq 0, \ \lambda^p \in \mathbb{R}^n \end{aligned} \tag{2.2}$$

The reformulation (2.2) does not contain the symmetry that is present in the original formulation (2.1). The solution λ no longer identifies the specific roll from which pattern i is cut. As such, λ does not give the information required to implement the cutting. It is up to the user to assign each pattern to a roll. However, the user is better able to recognize the symmetry that arises from identical rolls and thus, is able to avoid its negative effects.

Similar formulations can be found in the context of graph coloring. Mehrotra and Trick [60] formulate the graph coloring problem by assigning a variable to every maximal independent set in the graph. Then, they then find the minimum number of independent sets that cover the set of vertices. Reformulation has been applied to urban transit

scheduling [12], airline crew scheduling [3] [81] [79], vehicle routing [15], graph coloring [60], as well as binary cutting stock problems [80].

Reformulation may be an effective way of avoiding symmetry. However, this method is only applicable to problems that contain a very specific type of symmetry, i.e., only problems where the variables can be expressed in a matrix form and all permutations of the columns of the variables are contained in the symmetry group. Even in cases where decomposition can be used, implementing branching strategies for the column generation problem may be difficult.

2.2 Removing Symmetry in the Formulation

Removing symmetry by reformulation can be a very effective approach, however, it can be difficult to determine how to reformulate a specific problem. Another class of symmetry-breaking algorithms uses the symmetry group of the problem to reduce the feasible region by removing sets of equivalent solutions. This strategy of reducing the feasible region allows for more general techniques than reformulation.

Recall Example 1.4, a dominating set problem on 9 nodes.

$$
\begin{aligned}
& \min \sum_{i=0}^{8} x_i \\
\text{subject to} \quad & x_0 + x_1 + x_2 + x_3 + x_6 \geq 1 \\
& x_0 + x_1 + x_2 + x_4 + x_7 \geq 1 \\
& x_0 + x_1 + x_2 + x_5 + x_8 \geq 1 \\
& x_0 + x_3 + x_4 + x_5 + x_6 \geq 1 \\
& x_1 + x_3 + x_4 + x_5 + x_7 \geq 1 \\
& x_2 + x_3 + x_4 + x_5 + x_8 \geq 1 \\
& x_0 + x_3 + x_6 + x_7 + x_8 \geq 1 \\
& x_1 + x_4 + x_6 + x_7 + x_8 \geq 1 \\
& x_2 + x_5 + x_6 + x_7 + x_8 \geq 1 \\
& x \in \{0,\ 1\}^9.
\end{aligned}
$$

Let $\mathcal{G}$, be the symmetric group for Problem (2.3). At the root node, all variables in the problem are symmetric. Because it is clear that at least one variable must be equal to 1, an arbitrary variable can be chosen and fixed to 1.

For example, x_0 can be fixed to 1. The reason for fixing x_0 to 1 is very intuitive. Any optimal solution to (2.3) must contain at least one variable with value 1. Suppose that in a given optimal solution, $\hat{x}$, there is a j with $\hat{x}_j = 1$ for some $j \in \{0, \ldots, 8\}$. Since all variables are equivalent at the root node, there is a permutation $\pi \in \mathcal{G}$ that maps element j to element 0. Then, $\pi(\hat{x})$ is feasible and optimal (by definition of $\mathcal{G}$) with $\pi(\hat{x}_0) = 1$. By setting $x_0 = 1$, solutions are removed from the feasible region, but every solution that is removed is symmetric to at least one solution remaining in the feasible region. Hence, fixing an arbitrary variable to 1 serves to remove some of the problem symmetry as well as to tighten the LP bound.

A similar technique is used in graph coloring problems. Given a graph G, a coloring is valid if no two adjacent vertices are assigned the same color. Given a valid coloring, equivalent valid colorings can be generated by relabeling colors. A typical IP formulation for this problem contains binary variables x_i^j that represent if vertex i is colored j and binary variables y^l that represent if any vertex has color l. The mathematical formulation is

$$\begin{aligned} &\min \sum_{l=1}^{k} y^l \\ &\text{s.t. } x_i^l + x_j^l \leq 1 \ \ \forall\, (i,j) \in E \ \ \forall\, l \in \{1, \ \ldots, \ k\} \\ &x_i^l \leq y^l \ \forall\, i \in V \ \ \forall\, l \in \{1, \ \ldots, \ k\} \\ &\sum_l x_i^l = 1 \ \ \forall\, i \in V \\ &y^l, \ x_i^l \in \{0, 1\}. \end{aligned} \tag{2.3}$$

The symmetry group of formulation (2.3) contains the permutations

$$\pi_{l,m} = (x_1^l, x_1^m)(x_2^l, \ x_2^m) \ldots (x_n^m, \ x_n^k) \ \forall l, \ m \in \{1, \ \ldots, \ k\},$$

simply meaning that colors can be arbitrarily relabeled.

Suppose G contains a clique of size k. No two vertices in the clique can be assigned the same color. Given that no vertex is currently colored, each of the k vertices in the clique can be arbitrarily colored with each of the first k colors. As with fixing a variable to 1 in the Problem (1.4), this fixing of colors removes symmetric solutions from the feasible region and reduces the size of the symmetry group acting on the problem by at least a factor of $k!$. As before, this fixing is valid because for each solution that is removed, there is at least one equivalent solution remaining in the feasible region. These intuitive fixing rules aim to reduce the size of the feasible region (as well as the symmetry) of the problem instance by removing a large set of feasible solutions. A solution can be removed as long as it is guaranteed that for each solution removed a *representative solution* (i.e. an equivalent solution) remains in the feasible region. This is formalized in [21] where Eric Friedman adapts a notion from geometry. For a permutation group Γ and a set $X \subset \mathbb{R}^n$, a *generalized fundamental domain* of X with respect to Γ is a subset F of X such that the X can be

constructed using the points in F along with the permutations in Γ, i.e., $F^\Gamma = X$.

Formally, a set F is a generalized fundamental domain of X with respect to Γ if and only if for every $x \in X$, the set $\mathrm{orb}(x, \Gamma) \cap F$ is nonempty. A fundamental domain is *minimal* if it does not contain a smaller fundamental domain.

Example If $X = \{(0,0),\ (1,0),\ (0,1),\ (1,1)\}$ and $\Gamma = \{e,\ \pi = (1,2)\}$, there are three different fundamental domains. The set X is a trivial a fundamental domain because $X^\Gamma = X$. Also, the set $F_1 = \{(0,0),\ (1,0),\ (1,1)\}$ is a fundamental domain. This can be seen by noting that the only element in X that is not in F_1 is the element $(0,1)$, but $\pi((1,0)) = (0,1)$, so $F_1^\Gamma = X$. For the same reason, the set $F_2 = \{(0,0),\ (0,1),\ (1,1)\}$ is also a fundamental domain. No two elements in F_1 are symmetric, so F_1 is a minimal fundamental domain with respect to Γ. Similarly, F_2 is also minimal.

For a given X and symmetry group Γ, there may be more than one minimal fundamental domain. Example 2.2 shows illustrations of fundamental domains.

Example Consider the polyhedron depicted in Figure 2.1.

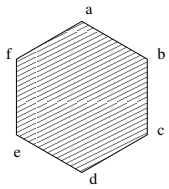


Figure 2.1: Fundamental Domain Example 1

Let Γ contain symmetries that are generated by permutations $\pi_{rotate}(a,b,c,d,e,f)$ and $\pi_{reflect} = (b,f)(c,e)$.

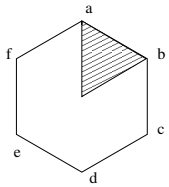


Figure 2.2: Fundamental Domain Example 2

The shaded area in Figure 2.2 represents a fundamental domain of X and Γ. It would be minimal if not for the permutation $\pi_{reflect}$. A minimal fundamental domain is given in Figure 2.3.
There may be many different minimal domains. Figure 2.4 is another example of a minimal domain that is not convex.

For a polyhedron $X \subseteq \mathbb{R}^n$, a permutation group Γ, and an n-vector c consider the set

$$F_c(X, \Gamma) = \{x \in X | c^T x \geq c^T \pi(x)\ \ \forall \pi \in \Gamma\}.$$

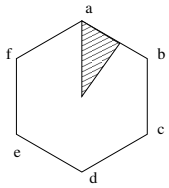


Figure 2.3: Fundamental Domain Example 3

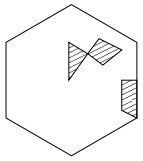


Figure 2.4: Fundamental Domain Example 4

For any $x \in \mathcal{F}$, at least one element $y \in \mathrm{orb}(x, \Gamma)$ satisfies the constraints $c^T y \geq c^T \pi(x)\ \forall \pi \in \Gamma$ so $F_c(X, \Gamma)$ is a fundamental domain of X with respect to Γ. The set $F_c(X, \Gamma)$ is the fundamental domain generated by c. In cases where X and Γ are clear, the fundamental domain will be referred to as F_c.

Example Consider Problem (1.4). Let X be the feasible region of the LP relaxation and $\mathcal{G}$ be the symmetry group of the problem. Let $c = (3,\ 2,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0)$. Because Γ contains 72 permutations, the fundamental domain F_c is formed by adding 72 inequalities (one for each permutation). A subset of the constraints formed by the generators given in Table 1.2, are listed in Table 2.1

π	Constraint
(3, 6)(4, 7)(5, 8)	$3x_0 + 2x_1 + x_2 \geq 3x_0 + 2x_1 + x_2$
(1, 2)(4, 5)(7, 8)	$3x_0 + 2x_1 + x_2 \geq 3x_0 + x_2 + 2x_3$
(1, 3)(2, 6)(5, 7)	$3x_0 + 2x_1 + x_2 \geq 3x_0 + 2x_3 + x_6$
(0, 1)(3, 4)(6, 7)	$3x_0 + 2x_1 + x_2 \geq 3x_1 + 2x_0 + x_2$

Table 2.1: Constraints Generating a Fundamental Domain

Not all inequalities generated by c will be useful, as some constraints may be redundant, evidenced by the first inequality in Table 2.1. The constraints generated by the vector c can be used to impose an order on the elements of X. Let $\lesssim_c$ be the relation on the set X that has $x \lesssim_c y$ if and only if $c^T x \geq c^T y$. Because $\lesssim_c$ is not necessarily antisymmetric, it is a quasi-order ($x \lesssim_c y$ and $y \lesssim_c$ does not imply that $x = y$). Because Γ is finite, for every $x \in \mathcal{F}$, $\mathrm{orb}(x, \Gamma)$ is finite. Thus, for any quasi-order $\lesssim$ and any $x \in \mathcal{F}$ at least one element in $\mathrm{orb}(x, \Gamma)$ satisfies the constraints

$$x \lesssim \pi(x)\ \ \forall \pi \in \Gamma. \tag{2.4}$$

Hence, constraints (2.4) define a fundamental domain. If a relation $\preceq$ defines a strict total order on the sets $\mathrm{orb}(x, \Gamma)$ for every $x \in X$, then constraints (2.4) generated by $\preceq$ is a minimal fundamental domain (as only one element from $\mathrm{orb}(x, \Gamma)$ satisfies constraints (2.4)). However, determining if a quasi order $\lesssim$ defines a strict total order on each orbit is not trivial. One such total order on $\{0,1\}^n$ comes from enforcing a lexicographic ordering of the variables via $c_{Lex} = (2^{n-1},\ 2^{n-2},\ \ldots,\ 2,\ 1)$. $F_{c_{Lex}}$ is a minimal fundamental domain [21] for all $X \subseteq \{0,1\}^n$ and Γ.

Example For problem (1.4), the constraints of $F_{c_{Lex}}$ that result from the generators of $\mathcal{G}$ are

π	Lexicographical Constraint
(3,6)(4,7)(5,8)	$32x_3 + 16x_4 + 8x_5 + 4x_6 + 2x_7 + x_8 \geq 32x_6 + 16x_7 + 8x_8 + 4x_3 + 2x_4 + x_5$
(1,2)(4,5)(7,8)	$128x_1 + 64x_2 + 16x_4 + 8x_5 + 2x_7 + x_8 \geq 128x_2 + 64x_1 + 16x_5 + 8x_4 + 2x_8 + x_7$
(1,3)(2, 6)(5,7)	$128x_1 + 64x_2 + 31x_3 + 8x_5 + 4x_6 + 2x_7 \geq 128x_3 + 64x_6 + 32x_1 + 8x_7 + 4x_2 + 2x_5$
(0,1)(3,4)(6,7)	$256x_0 + 128x_1 + 32x_3 + 16x_4 + 4x_6 + 2x_7 \geq 256x_1 + 128x_0 + 32x_4 + 16x_3 + 4x_7 + 2x_6$

Table 2.2: Lexicographic Constraints

There is an immediate concern with using lexicographic constraints. Adding constraints that define $F_{c_{Lex}}$ will cause problems for most numerical methods because of the magnitude of the coefficients in c_{lex}. However, these constraints guarantee that the resulting fundamental domain is minimal for any $X \subseteq \{0,1\}^n$ and Γ. Despite the numerical issues, most literature focuses on choosing fundamental domains based on lexicographic ordering (either generating fundamental domains by selecting lexicographic maximal or minimal elements).

In the context of integer programming, X represents the set of feasible solutions, $\mathcal{F}$, to a problem and Γ represents the symmetry group of the problem, $\mathcal{G}$. By definition, for any fundamental domain F of $\mathcal{F}$ with respect to $\mathcal{G}(ILP)$, any optimal solution in $\mathcal{F}$ is symmetric to a solution in F. Hence,

$$\min_{x \in F}\{c^T x\} = \min_{x \in \mathcal{F}}\{c^T x\}.$$

The search of (ILP) can be restricted to a fundamental domain. The issue then becomes how to choose a fundamental domain. Symmetry-breaking tools that use fundamental domains can be split into two different classes. Static symmetry-breaking methods determine the fundamental domain at the start of the algorithm, and dynamic methods determine the fundamental domain during the branching process.

2.3 Static Symmetry-Breaking Methods

Static symmetry-breaking methods choose a fundamental domain prior to starting branch-and-bound. Symmetry breaking can be done in an ad-hoc fashion by generating collections of problem-specific constraints that define a fundamental domain. A more general strategy is to use $\mathcal{G}(ILP)$ to construct a minimal fundamental domain. If it

weren't for the potential numerical issues, the minimal fundamental domain $F_{c_{Lex}}$ would be an ideal fundamental domain to use. To avoid the potential numerical issues, advanced static symmetry-breaking methods find ways to implicitly enforce the constraints generated by c_{Lex}.

Puget [72] gives symmetry-breaking cuts for two specific types of constraint programming problems: the pigeonhole problem and the Ramsey problem. The pigeonhole problem attempts to find a solution (or show no solution exists) to the problem of placing N pigeons in M holes. Let x_i^j be a decision variable that takes the value 1 if pigeon i is in hole j. The ILP formulation of the problem is

$$\begin{aligned} &\min 0^T x \\ \text{s.t} &\sum_{j=1}^{M} x_i^j = 1 \; \forall i \in 1, \ldots, \; N \\ &\sum_{i=1}^{N} x_i^j \leq 1 \; \forall j \in 1, \ldots, \; M \\ &x_i^j \in \{0, 1\}. \end{aligned} \tag{2.5}$$

The LP relaxation of (2.5) is infeasible if $M < N$. However, LP bounds are not used in [72] to prune subproblems. As a result, Puget only prunes nodes in the enumeration tree when there are no feasible branching decisions. Despite the fact that Problem (2.5) is easily solved with common sense, small instances when $M < N$ are very difficult to solve via constraint programming methods not relying on LP.

The major difficulty arises from the fact that the pigeons are not unique. The symmetry group Equation 2.5 contains all $N!$ permutations of pigeons and all $M!$ permutations of holes. Subproblems that can be pruned, i.e. those where all the holes are occupied by a pigeon, are of depth at least M, so the tree is very large (it contains more than 2^M nodes). Many of these subproblems, however, are equivalent.

To combat the symmetry in this problem for any M and N, Puget adds the ordering constraint

$$\sum_{k=1}^{M} k x_i^k < \sum_{k=1}^{M} k x_{i+1}^k \text{ for all } i = 1 \ldots N-1. \tag{2.6}$$

These constraints force an ordering of pigeons and remove the symmetry that was present in the original problem formulation. The resulting feasible region after adding the ordering constraints is a fundamental domain in the case where $M < N$, since the feasible region is empty. Interestingly, even though adding (2.6) removes all the symmetry from the formulation, in the case where (2.5) is feasible, i.e, $M \geq N$, the inequalities in (2.6) only describe a minimal fundamental domain if $M = N$. For instance, suppose $M = 4$ and $N = 3$. The solution with $x_1^1 = 1$, $x_2^2 = 1$, and $x_3^3 = 1$ satisfies constraints in (2.6), but so does the symmetric solution $x_1^1 = 1$, $x_2^2 = 1$, and $x_3^4 = 1$. In the case where $N = M$, the only feasible solution has $x_i^i = 1$ for all i, so the fundamental domain defined by (2.6) is minimal. Nevertheless, inequalities in (2.6) make it very easy to show that no solution exists if $M < N$, as the

ordering constraints make it easy for the set of fixed variables to create a logical inconsistency. Instances with large M and N are easily solved via constraint programming when inequalities (2.6) are added to the formulation.

The second problem Puget provides cuts for is the Ramsey problem. Given a complete graph $K_n = (V, E)$ with N vertices, the Ramsey problem attempts to color the edges of the graph with three colors such that no triangle contains three edges of the same color. There are only feasible colorings when $N \leq 16$, but proving infeasibility for $N > 16$ is difficult. The symmetry group, $\mathcal{G}(Ramsey)$, of this problem is very large. Since the graph is complete, any relabeling of vertices will send feasible solutions to feasible solutions (resulting in $N!$ symmetries). An IP formulation of the Ramsey problem is

$$\begin{gathered} \min 0^T x \\ \text{s.t } x_{ij0} + x_{ij1} + x_{ij2} = 1 \ \ \forall (i,j) \in E \\ x_{hik} + x_{ijk} + x_{hjk} \leq 2 \ \ \forall h,\ i,\ j \in V,\ k = 0,\ 1,\ 2 \\ x_{ijk} \in \{0,1\}, \end{gathered} \tag{2.7}$$

where $x_{ijk} = 1$ if edge (i, j) is colored with color k.

To combat the symmetry Formulation (2.7), Puget adds two different types of constraints. First, the constraints

$$\sum_{i \in V \setminus \{v_0\}} x_{0,i,0} \geq \sum_{i \in V \setminus \{v_j\}} x_{j,i,0} \ \forall j \in V \tag{2.8}$$

enforce that vertex v_0 is adjacent to more edges of color 0 than any other vertex. This is a valid constraint in the context of symmetry as *some* vertex in a solution is adjacent to more edges of color 0 than any other vertex, and since all vertices are symmetric, this property can be arbitrarily assigned to v_0. After adding constraints (2.8) to the original formulation (2.7), the formulation group has changed. Now, vertex v_0 is no longer equivalent to any other vertex in the graph. As a result, permutations that map variables of the type $x_{0,j,k}$ to variables $x_{l,m,p}$ with $l \neq 0$ are removed from the formulation's symmetry group. Interestingly, while constraints (2.8) do remove symmetry from the formulation group, they may not remove symmetries from the symmetry group of the problem. The symmetry group will not change in the (unlikely) event that every feasible solution satisfies (2.8).

All variables representing edges adjacent to vertex v_0 are still symmetric in the formulation including constraints (2.8). To remove additional symmetry, constraints are added to make the colors assigned to edges $(0, i)$ an increasing function of i. This is accomplished by the constraint

$$\sum_{k=0}^{2} k x_{0,j,k} \leq \sum_{k=0}^{2} k x_{0,j+1,k} \ \forall j = 0,\ \ldots,\ n-1. \tag{2.9}$$

The symmetry group of the formulation (2.7) with constraints (2.8) and (2.9) added contains only the identity

permutation, but the feasible region is not a minimal fundamental domain of Equation (2.7) with respect to group $\mathcal{G}(Ramsey)$. Consider the graph in Figure 2.5. This graph represents a feasible solution to (2.7) and also satisfies constraints (2.8) and (2.9). Permuting vertices v_1 and v_2 (corresponding to

$$\pi = (x_{0,1,0}, x_{0,2,0})(x_{0,1,1}, x_{0,2,1})(x_{0,1,2}, x_{0,2,2})(x_{1,3,0}, x_{2,3,0})(x_{1,3,1}, x_{2,3,1})(x_{1,3,2}, x_{2,3,2}) \in \mathcal{G}(Ramsey)$$

) maps the graph in Figure 2.5 to the graph in Figure 2.6.

Because π was in the formulation group before constraints (2.8) and (2.9) were added to the formulation, the graphs in Figure 2.5 and Figure 2.6 are symmetric. But, the graphs in both Figure 2.5 and Figure 2.6 each satisfy constraints (2.8) and (2.9). It is easy to see that π is not in the symmetry group (when the additional constraints are added) by considering the feasible graph formed by coloring edge (v_0, v_2) in Figure 2.5 with color 1, π maps this graph to a graph that violates (2.9). While the constraints (2.8) and (2.9) remove all symmetry from the problem formulation, they do not define a fundamental domain. However, the resulting fundamental domain is small enough to allow for these problems to be solved in a reasonable amount of time.

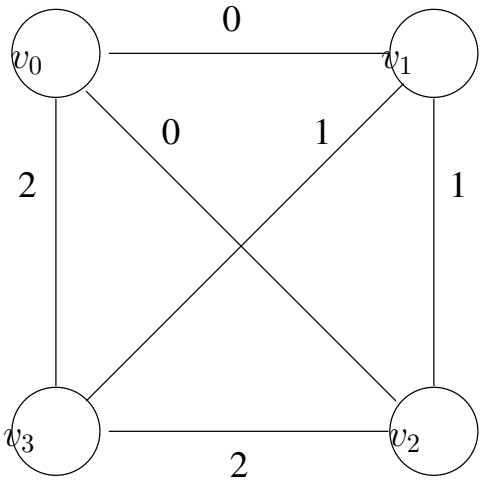


Figure 2.5: Ramsey Graph for $n = 4$

Adding constraints like (2.6), (2.8), or (2.9) are common in symmetry-breaking literature. Meller et al. [61] attack symmetry in an optimal facility layout design using constraints that reduce the size of the fundamental domain. Sherali and Smith [77] present three different applications where symmetry arises: telecommunications network-design problems, noise pollution problems, and machine procurement and operation problems. They also discuss symmetry-reducing constraints for these specific problems. Mendez-Diaz and Zabala present formulations, as well as symmetry-breaking inequalities, for the graph coloring problem [13] [62].

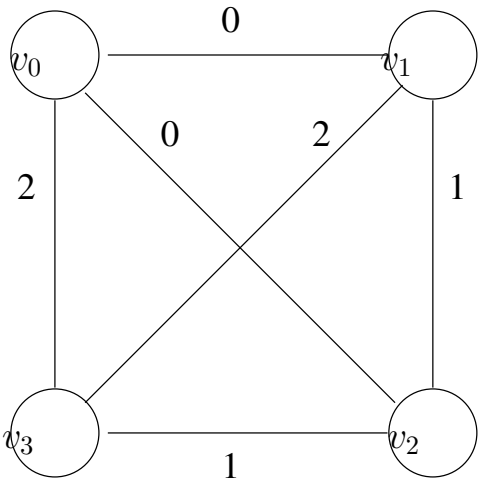


Figure 2.6: An Equivalent Ramsey Graph for $n = 4$

2.3.1 Lexicographic Ordering

The constraints proposed by Puget for the pigeonhole problem and the Ramsey problem can be effective at reducing symmetry, making a previously intractable problem solvable. There is no guarantee, however, that the fundamental domains resulting from the addition of these inequalities will be minimal. Further, the constraints added to reduce the symmetry are very problem-specific. Knowing the actual symmetry group of the problem (or at least a subgroup) allows for a more general method of solving symmetric problems.

Given $\mathcal{G}(ILP)$, a minimal fundamental domain can always be defined by adding lexicographic constraints. Adding these constraints restricts the search to solutions that are lexicographically smallest among equivalent solutions, (sometimes referred to as "the lexicographic leader"). This technique was used to solve planning problems in [38].

Crawford, Ginsberg, Luks, and Roy [9] outline a method for generating lexicographic inequalities. They also discuss partial symmetry-breaking methods; strategies that remove most of the symmetry without adding an exponential number of constraints. Aloul et. al. [2] rewrite these lex-leader constraints in a more efficient way.

Example Consider a simple example of using lexicographic constraints from [29].

$$
\begin{aligned}
& \min \sum_{i=0}^{6} x_i \\
\text{subject to} \quad & x_0 + x_1 + x_2 + x_3 \geq 1 \\
& x_0 + x_1 + x_2 + x_4 \geq 1 \\
& x_0 + x_1 + x_2 + x_5 \geq 1 \\
& x_0 + x_3 + x_4 + x_5 \geq 1 \\
& x_1 + x_3 + x_4 + x_5 \geq 1 \\
& x_2 + x_3 + x_4 + x_5 \geq 1 \\
& x_0 + x_3 \geq 1 \\
& x_1 + x_4 \geq 1 \\
& x_2 + x_5 \geq 1 \\
& x \in \{0,1\}^6
\end{aligned}
$$

Twelve permutations are in the formulation group. These permutations are generated by $(0,3)(1,4)(2,5)$, $(0,1)(3,4)$, and $(0,2)(3,5)$. The following constraints will then enforce the lexicographical ordering $\preceq_{lex}$, and will reduce the

feasible region to $F_{c_{Lex}}$:

$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_0 + 2^4x_2 + 2^3x_1 + 2^2x_3 + 2x_5 + x_4$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_1 + 2^4x_0 + 2^3x_2 + 2^2x_4 + 2x_3 + x_5$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_2 + 2^4x_1 + 2^3x_0 + 2^2x_5 + 2x_4 + x_3$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_1 + 2^4x_2 + 2^3x_0 + 2^2x_4 + 2x_5 + x_G$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_2 + 2^4x_0 + 2^3x_1 + 2^2x_5 + 2x_3 + x_4$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_3 + 2^4x_4 + 2^3x_5 + 2^2x_0 + 2x_1 + x_2$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_3 + 2^4x_5 + 2^3x_4 + 2^2x_0 + 2x_2 + x_1$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_4 + 2^4x_3 + 2^3x_5 + 2^2x_1 + 2x_0 + x_2$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_5 + 2^4x_4 + 2^3x_3 + 2^2x_2 + 2x_1 + x_0$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_4 + 2^4x_5 + 2^3x_3 + 2^2x_1 + 2x_2 + x_0$$
$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_5 + 2^4x_3 + 2^3x_4 + 2^2x_2 + 2x_0 + x_1.$$

Note that there is one constraint for every permutation in the symmetry group. Enforcing these lexicographical constraints by using linear constraints may cause numerical stability issues, as mentioned previously. However, there is hope that some of these numerical issues can be avoided by pre-processing the constraints. The lexicographic inequalities are likely to contain redundancies. Also, inequalities may be simplified in such a way as to remove some of the scaling issues.

Consider the second constraint,

$$2^5x_0 + 2^4x_1 + 2^3x_2 + 2^2x_3 + 2x_4 + x_5 \geq 2^5x_0 + 2^4x_2 + 2^3x_1 + 2^2x_3 + 2x_5 + x_4.$$

This constraint can be written as $2^3x_1 + 2^2x_2 + 2x_4 + x_5 \geq 2^3x_2 + 2^2x_1 + 2x_5 + x_4$ (because $x_0 = x_0$ and $x_3 = x_3$ are always true). If $x_1 > x_2$, then the constraint is satisfied independently of the values of x_4 and x_5. If $x_1 = x_2$, the constraint reduces to $2x_4 + x_5 \geq 2x_5 + x_4$, or just $x_4 \geq x_5$. Thus, the original constraint can be rewritten as

$2x_1 + x_4 \geq 2x_2 + x_5$. Similarly, all constraints can be rewritten as:

$$0 \geq 0$$

$$2x_1 + x_4 \geq 2x_2 + x_5$$

$$2x_0 + x_3 \geq 2x_1 + x_4$$

$$2x_0 + x_3 \geq 2x_2 + x_5$$

$$8x_0 + 4x_1 + 2x_3 + x_4 \geq 8x_1 + 4x_2 + 2x_4 + x_5$$

$$8x_0 + 4x_1 + 2x_3 + x_4 \geq 8x_2 + 4x_0 + 2x_3 + x_5$$

$$4x_0 + 2x_1 + x_2 \geq 4x_3 + 2x_4 + x_5$$

$$4x_0 + 2x_1 + x_2 \geq 4x_3 + 2x_5 + x_4$$

$$4x_0 + 2x_1 + x_2 \geq 4x_4 + 2x_3 + x_5$$

$$4x_0 + 2x_1 + x_2 \geq 4x_5 + 2x_4 + x_3$$

$$16x_0 + 8x_1 + 4x_2 + 2x_3 + x_4 \geq 16x_4 + 8x_5 + 4x_3 + 2x_1 + x_2$$

$$16x_0 + 8x_1 + 4x_2 + 2x_3 + x_4 \geq 16x_5 + 8x_3 + 4x_4 + 2x_2 + x_0.$$

The collection of inequalities can be further strengthened by considering the collection of inequalities as a whole. Note that constraints 2 and 3 imply constraint 4, so constraint 4 is not necessary. The 12 constraints can be reduced to a set of 8 non-redundant constraints:

$$2x_1 + x_4 \geq 2x_2 + x_5$$

$$2x_0 + x_3 \geq 2x_1 + x_4$$

$$4x_0 + 2x_1 + x_2 \geq 4x_3 + 2x_4 + x_5$$

$$4x_0 + 2x_1 + x_2 \geq 4x_3 + 2x_5 + x_4$$

$$4x_0 + 2x_1 + x_2 \geq 4x_4 + 2x_3 + x_5$$

$$4x_0 + 2x_1 + x_2 \geq 4x_5 + 2x_4 + x_3$$

$$8x_0 + 4x_1 + 2x_2 + x_3 \geq 8x_4 + 4x_5 + 2x_3 + x_1$$

$$4x_0 + 2x_1 + x_2 \geq 4x_5 + 2x_3 + x_4.$$

Adding these 8 processed constraints will significantly reduce the stability issues that may occur if the original 12

constraints were added. It is natural to ask the question *by how much should processing be expected to help*?. Since many lexicographic constraints are redundant, are an exponential number of lexicographic constraints necessary to remove all the symmetry? Luks and Roy examined this question in [52]. Unfortunately, even with relatively small symmetry groups, an exponentially large set of lexicographic constraints may be required to enforce the lexicographic ordering of feasible solutions. Even if it were possible to efficiently process a collection of lexicographic constraints, the quantity and the scale of the processed constraints may overwhelm the LP solver.

2.3.2 Static Symmetry Breaking via Orbitopes

Explicitly stating all the lexicographic inequalities (as in Table 2.2) is not practical for many reasonably-sized problems. One avenue of research is to find classes of problems for that there are efficient ways to enforce lexicographic ordering without adding the constraints to the LP formulation. This is the purpose of [39]. In their work, Kaibel and Pfetsch consider the set of all 0/1 matrices of size $p \times q$, $\mathcal{M}_{p,q}$. Given a symmetry group $\mathcal{G}$ acting on the columns of a matrix, they define the *full orbitope* $O_{p,q}(\mathcal{G})$ to be the set of matrices of $\mathcal{M}_{p,q}$ that are lexicographically maximal within their orbits (where the ordering of the variables is row-by-row). For example, the 0/1 matrix:

$$X = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

does not have lexicographically-decreasing columns, so it is not in the orbitope $O_{8,5}(S^5)$. The permutation $(4, 5)$

swaps columns 4 and 5, giving the matrix:

$$X' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

The matrix X' has lexicographically-decreasing columns, so it is in $O_{8,5}(S^5)$.

Kaibel and Pfetsch focus their attention on both *packing* and *partitioning orbitopes*, where either the cyclic group (C^q) or the complete symmetric group (S^q) acts on the columns. The packing orbitope, $O^{\leq}_{p,q}(\mathcal{G}) \subset O_{p,q}(\mathcal{G})$ consists of all lexicographically maximal 0/1 matrices where each row contains at most a single 1. The partitioning orbitope, $O^{=}_{p,q}(\mathcal{G}) \subset O_{p,q}(\mathcal{G})$ consists of all lexicographically maximal matrices where each row contains exactly a single 1.

In [39], Kaibel and Pfetsch provide linear descriptions of all facet-defining inequalities for the convex hull of orbitopes $O^{\leq}_{p,q}(C^q)$, $O^{\leq}_{p,q}(S^q)$, $O^{=}_{p,q}(C^q)$, and $O^{=}_{p,q}(S^q)$. They also provide separation algorithms for all inequalities that run in polynomial time. The facet-defining inequalities have only $\{-1,\ 0,\ 1\}$ coefficients, avoiding the numerical issues that arise when lexicographic constraints are used to eliminate symmetry. However, to completely describe the polytope $O^{\leq}_{p,q}(S^q)$ and $O^{=}_{p,q}(S^q)$, an exponential number of linear inequalities is required.

This technique can be applied to classes of integer programming problems as follows. Elements in a partitioning orbitope represent possible solutions to IP formulations of set-partitioning problems such as graph coloring, while the packing orbitope represents solutions to IP formulations of set-packing problems. For example, imagine a set-partitioning problem in which items are placed in one of q indistinguishable sets. The variable $x_{i,j} = 1$ if item i is placed in set j. A solution to the problem can be represented as a $p \times q$ matrix, where p is the number of items and q is the number of sets. Since the sets are indistinguishable, any permutation of the sets will map feasible partitions to other feasible partitions. Therefore, every permutation of the columns is in the symmetry group. Symmetry in the formulation can be removed by adding the additional constraint that the solution must also be an element of the orbitope $O_{p,q}(S^q)$. Kaibel and Pfetsch [39] show that enforcing membership in either $O^{\leq}_{p,q}(C^q)$ or $O^{=}_{p,q}(C^q)$ is simple. Ensuring that the first column is the lexicographically largest column in the matrix is enough to guarantee that the matrix is in the appropriate orbitope. Note also that C^q contains only $q-1$ permutations, so only $q-1$ constraints are needed to enforce a lexicographic ordering.

Enforcing membership in either $O^{\leq}_{p,q}(S^q)$ or $O^{\leq}_{p,q}(S^q)$ is not as simple as it is for the cyclic group C^q. Kaibel and Pfetsch note that there exists a projection of $O^{=}_{p,q}(S^q)$ that is affinely isomorphic to $O^{\leq}_{p-1,q-1}(S^{q-1})$, so they restrict their attention to the orbitope $O^{=}_{p,q}(S^q)$. The main result of their work is that *shifted column inequalities* (SCI) complete a necessary and sufficient linear description of $O^{\leq}_{p,q}(S^q)$. To describe the shifted column inequalities some notation is necessary. A *bar* is formed by indices of the matrix

$$B = \{(i,j),\ (i,j+1),\ \ldots,\ (i,q)\}$$

for some $2 \leq i \leq p$, $2 \leq j \leq \min\{i,q\}$. A *shifted column* (SC) is a set of indices

$$S = \{(i_1,j_1),\ (i_2,j_2),\ \ldots,\ (i_\eta,j_\eta)\}$$

where $i_1 \leq i_2 \leq i_\eta$, $j_1 \leq j_2 \leq j_\eta \leq j$, and no two elements in S share the same diagonal in the matrix. For any bar B and shifted column S, they call $\sum_{(i,j)\in B} x_{i,j} \leq \sum_{(i,j)\in S}$ a shifted column inequality. The orbitope $O^{=}_{p,q}(S^q)$ can be completely described by non-negativity constraints, the row-sum equations $\sum_j x_{i,j} = 1$ for all i, and the shifted column inequalities formed by all bars and shifted columns.

Example Figures 2.7 through 2.9 are three different shifted columns and bars. A "+" denotes elements in the bar while a "−" denotes elements in the shifted column. The inequality derived from the shifted column and the bar in Figure 2.7 is

$$x_{9,5} + x_{9,6} + x_{9,7} \leq x_{4,4} + x_{5,4} + x_{6,4} + x_{7,4} + x_{8,4}.$$

The inequality derived from the shifted column and the bar in Figure 2.8 is

$$x_{9,5} + x_{9,6} + x_{9,7} \leq x_{2,2} + x_{4,3} + x_{5,3} + x_{7,4} + x_{8,4}.$$

The inequality derived from the shifted column and the bar in Figure 2.9 is

$$x_{9,5} + x_{9,6} + x_{9,7} \leq x_{1,1} + x_{2,1} + x_{4,2} + x_{5,2} + x_{6,2}.$$

Note that in partitioning problems $\sum_{(i,j)\in B} x_{i,j} \leq 1$. If a SCI is violated, it must be because $\sum_{(i,j)\in B} x_{i,j} = 1$ and $\sum_{(i,j)\in S} = 0$. Consider an example of a matrix that violates the first SCI shown above. Clearly the matrix in Figure 2.10 is not lexicographically maximal, as column 4 can be swapped with column 5 to create a lexicographically larger matrix.

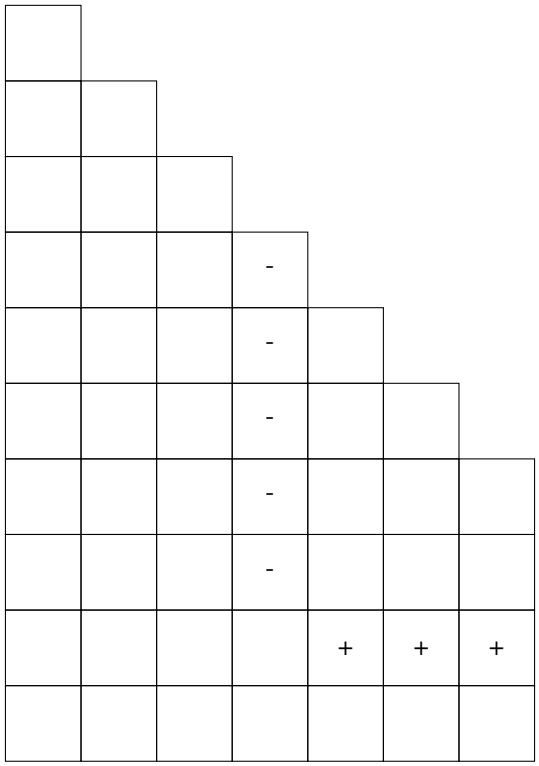


Figure 2.7: Shifted Column 1

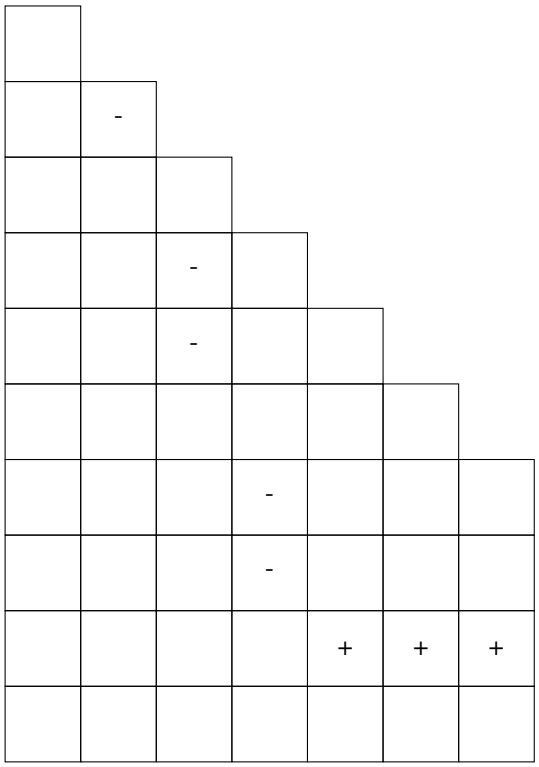


Figure 2.8: Shifted Column 2

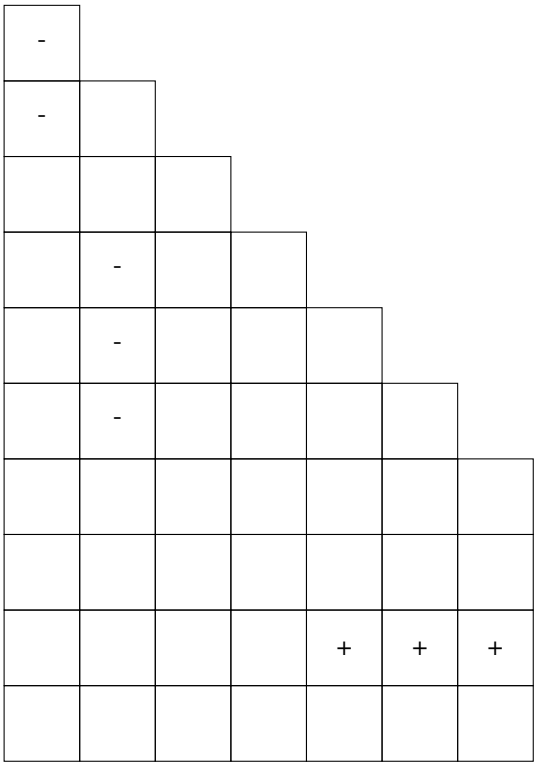


Figure 2.9: Shifted Column 3

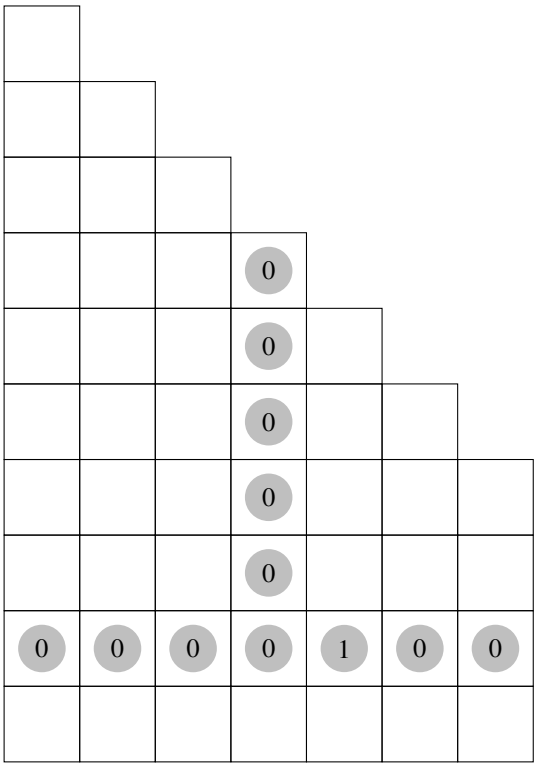


Figure 2.10: Matrix that does not satisfy SCI

In [40], Kaibel, Peinhardt, and Pfetsch use the linear time separation algorithm for these SCIs to fix variables throughout the enumeration tree. They call their fixing *orbitopal fixing*. They also test orbitopal fixing on a class of graph partitioning problems and compare it to a more general symmetry breaking method, isomorphism pruning, which will be detailed in Section 2.4.1. Unlike isomorphism pruning, using orbitopal fixing does not restrict branching behavior, and the results suggest that the gain in performance as a result of using orbitopal fixing over isomorphism pruning is a result of the flexible branching.

2.3.3 Double Lex

Orbitopal fixing considers symmetry that acts on the columns of the matrix. It is possible for symmetry to act on both the columns and the rows. For instance, the pigeonhole problem could be formulated using variable x_{ij} to indicate that pigeon i is in hole j. Since both the pigeons and the holes are indistinguishable, permutations in the symmetry group act on both the columns and the rows.

If solution matrix X is a $m \times n$ matrix, the symmetry group of the problem contains $m!n!$ symmetries. A subset of the symmetries, $m!$ of the symmetries, can be broken by requiring the rows of X to be lexicographically ordered (where row i is lexicographically larger than row j if $i < j$), or $n!$ symmetries can be broken by requiring the columns to be lexicographically ordered. Flener et al. [18] shows that, in fact, requiring both the rows and the columns to be lexicographically ordered in the same direction results in a fundamental domain. However, enforcing lexicographically-increasing columns and decreasing rows does not. Enforcing this lexicographic ordering requires at most $n! + m!$ many constraints, or can be accomplished using fixing algorithms such as orbitopal fixing (where lexicographic ordering is enforced on the columns of X and the columns of X^T). However, adding constraints that enforce lexicographically-increasing rows and columns does not remove all symmetry. Example (2.10) shows a case where two equivalent solutions both have lexicographically-increasing rows and columns.

Example Suppose the following feasible solution is found for a problem whose symmetry group contains all permutations of the rows and all permutation of the columns:

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

For any values of A and B, this solution will have both lexicographically-increasing rows and columns. Let $\pi_{i,j}$ be a column permutation that swaps columns i and j and let $\rho_{m,n}$ be a row permutation that swaps rows m and n. These

permutations act on th current solution as follows:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & A \\ 1 & 0 & B \end{bmatrix} \rightarrow_{\rho_{2,3}} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & B \\ 0 & 1 & A \end{bmatrix} \rightarrow_{\pi_{1,2}} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & B \\ 1 & 0 & A \end{bmatrix}. \tag{2.10}$$

This sequence of permutations gives a new matrix that also has lexicographically-increasing rows and columns that are isomorphic to the original.

2.4 Dynamic Symmetry Breaking

Static symmetry-breaking methods can be very effective at solving highly symmetric problems. Methods such as adding symmetry breaking constraints only require an alteration of the problem formulation and do not require special software. Also, knowing the domain explicitly at every node in the tree makes it possible to fix variables based on the fundamental domain constraints (implicit or explicit).

Static symmetry-breaking methods have some limitations. The first, as previously mentioned, is the potentially large number of inequalities needed to remove all of the symmetry from the problem. This problem can sometimes be avoided with clever algorithms. Secondly, restricting the search to an arbitrarily chosen fundamental domain could distort the branching process by changing the relative importance of the variables. For example, restricting the feasible region to the set of lexicographically minimal solutions would significantly increase the importance of variables with small indices. Algorithms that, in theory, have no branching restrictions may require branching in a rigid way to achieve the best results.

Static methods can make finding optimal solutions more difficult, as the solution found must also be in the chosen fundamental domain. Adding symmetry-breaking constraints to the root node may in fact remove optimal solutions that, along with a specified branching rule, would have been easier to find than the equivalent optimal solution remaining in the fundamental domain. This has been studied in the context of constraint programming in [25], where the authors show that branching strategies can drastically alter the time required to find solutions. The problem caused by pruning these easily-found optimal solutions can be overcome with more intelligent branching. For instance, if the set of representative solutions is chosen based on a lexicographic order, then branching in that order would avoid this problem. However, rules like this are not ideal, as branching is very important even in integer programs that do not contain symmetry. Therefore, choosing a branching rule a priori is not desirable.

Dynamic symmetry-breaking strategies differ from static methods because they construct the fundamental domain during the solution process, not a priori. Because the fundamental domain is not predefined, the branching decisions can influence the fundamental domain, and not vice versa.

Because the fundamental domain is defined during the branching process, dynamic symmetry breaking methods cannot use off-the-shelf software. Also, because the domain is not fully defined at nodes in the tree, fewer variables can be fixed than with static methods.

2.4.1 Isomorphism Pruning

The leading approach for solving

$$\min_{x \in \{0,1\}^n} \{c^T x \mid Ax \leq b\}, \qquad \text{BIP}$$

where (BIP) is highly symmetric ($|\mathcal{G}(BIP)|$ is large), is isomorphism pruning, developed for integer programming by Margot [54, 55]. Let the set F_1^a be the set of variables that have been fixed to 1 in subproblem a of a branch-and-bound tree and F_0^a be the set of variables fixed to zero. The fundamental idea behind isomorphism pruning is that for each node a in the branch-and-bound tree, the orbits $\mathrm{orb}(F_1^a, \mathcal{G}(BIP))$ of the equivalent sets of variables to F_1^a are computed. If there is a node $b = (F_1^b, F_0^b)$ elsewhere in the enumeration tree such that $F_1^b \in \mathrm{orb}(F_1^a, \mathcal{G}(BIP))$, then the node a need not be evaluated and is pruned by isomorphism. A very distinct and powerful advantage of this method is that *no* nodes whose sets of fixed variables are isomorphic will be evaluated. One disadvantage of this method is that computing $\mathrm{orb}(F_1^a, \mathcal{G}(IP))$ can require computational effort on the order of $O(n|F_1^a|!)$ in the worst case. A more significant disadvantage of isomorphism pruning is that $\mathrm{orb}(F_1^a, \mathcal{G}(BIP))$ may contain many subsets equivalent to F_1^a, and the entire enumeration tree must be compared against this list to ensure that a is not isomorphic to any other node b. In a series of papers, Margot offers a way around this second disadvantage [54, 55]. The solution is to declare one *unique representative* among the members of $\mathrm{orb}(F_1^a, \mathcal{G}(BIP))$. Then, if F_1^a is not the unique representative, the node a may safely be pruned. The advantage of this extension is that it is trivial to check whether or not node a may be pruned once the orbits $\mathrm{orb}(F_1^a, \mathcal{G}(IP))$ are computed.

Margot uses the concept of lexicographical ordering to determine a minimal fundamental domain. Rather than adding lexicographic inequalities, he determines rules for pruning nodes in the branch-and-bound. In his first paper [54] on isomorphism pruning, he restricts the search to the minimal fundamental domain $F_{c_{Lex}}$. A key theorem of [54] is that if F_1^a is not the lexicographically-smallest element of $\mathrm{orb}(F_1^a, \mathcal{G}(IP))$, then node a can be pruned by isomorphism. This theorem holds only if a specific branching rule is used. This branching rule requires branching on variable x_i at every node of depth i in the branch-and-bound tree. The isomorphism pruning described in [54] is a static symmetry-breaking method. This is changed in [55].

The dynamic isomorphism pruning method described in [55] is based on the realization that variables are assigned indices arbitrarily. Branching based on the index of a variable should not be expected to produce favorable results.

However, variables can be re-indexed so as to reflect a desired branching decision. This re-indexing is done by referring to variables not by their index but by their rank. The rank is determined by the branching behavior. A variable is given rank i the first time a node at depth i is reached in the branch-and-bound tree. The variable chosen for branching at that node, using any branching strategy, is assigned rank i. Once that branching decision is made, however, that variable is essentially given index i, and must be branched on at all other nodes of the same depth. As a result, the fundamental domain formed by Margot [55] is based on the lexicographic order of the ranks of the variables.

Isomorphism pruning allows for the use of some traditional integer programming methodologies, such as cut-generation based on implications derived from preprocessing and variable fixing based on reduced costs. In isomorphism pruning, for a variable fixing to be valid, *all* optimal solutions must be in agreement with the fixing. A powerful idea, called *orbit setting*, is to combine the variable fixing with symmetry considerations in order to fix many additional variables. For instance, if a variable can be set to zero as a result of any logical implication, then all variables sharing the same orbit can also be fixed to zero.

2.4.2 Symmetry Breaking via Dominance Detection

Symmetry Breaking via Dominance Detection (SBDD) [16] [19] [28] [35] [70] [74] is a state-of-the-art method for dealing with symmetry in constraint programming. Like isomorphism pruning, SBDD checks at every node whether the current node can be pruned due to symmetry considerations. Unlike isomorphism pruning, SBDD does not look for other nodes in the tree that are isomorphic to the current node. Instead, it searches for nodes that *dominate* the current node. Node a dominates node b if $\exists \pi \in \mathcal{G}(ILP)$ with $\pi(\mathcal{F}^b) \subseteq \mathcal{F}^a$. Note that π does not need to map $\mathcal{F}^b$ onto $\mathcal{F}^a$. In other words, b is dominated by a if the feasible region of b is equivalent to a subset of the feasible region of a.

Example Recall the set covering problem in Example 1.4

$$\min \sum_{i=0}^{8} x_i$$

subject to

$$\begin{aligned}
x_0 + x_1 + x_2 + x_3 \qquad + x_6 \qquad &\geq 1 \\
x_0 + x_1 + x_2 \qquad + x_4 \qquad + x_7 \qquad &\geq 1 \\
x_0 + x_1 + x_2 \qquad + x_5 \qquad + x_8 &\geq 1 \\
x_0 \qquad + x_3 + x_4 + x_5 + x_6 \qquad &\geq 1 \\
x_1 \qquad + x_3 + x_4 + x_5 \qquad + x_7 \qquad &\geq 1 \\
x_2 + x_3 + x_4 + x_5 \qquad + x_8 &\geq 1 \\
x_0 \qquad + x_3 \qquad + x_6 + x_7 + x_8 &\geq 1 \\
x_1 \qquad + x_4 \qquad + x_6 + x_7 + x_8 &\geq 1 \\
x_2 \qquad + x_5 + x_6 + x_7 + x_8 &\geq 1 \\
x \in \mathbb{B}^9
\end{aligned}$$

Suppose the node a in the enumeration tree represents the subproblem formed by fixing x_0 to 1. Let b be a node formed by setting $x_0 = 0$ and $x_1 = 1$.

Clearly $\mathcal{F}^a$ and $\mathcal{F}^b$ are not equivalent since $\mathcal{F}^a$ has a dimension of 8 while $\mathcal{F}^b$ has a dimension of 7. Consider the permutation $\pi = (0,\ 1)(3,\ 4)(6,\ 7)$ (recall this is a generator for the symmetry group of the problem). For any $x \in \mathcal{F}^b$, $\pi(x)_0 = x_1 = 1$, so $\pi(x) \in \mathcal{F}^a$. π maps every element of $\mathcal{F}^b$ to an element in $\mathcal{F}^a$, node a dominates node b.

The goal of the SBDD algorithm is to prune dominated nodes from the enumeration tree. To ensure that one representative of each solution is kept, a node is pruned only if it is dominated by a node to its left in the tree (for an arbitrary orientation of the tree). Testing for dominance may be computationally burdensome, as there could be an exponentially large set of nodes to the left of the current node. However, the number of domination tests required is significantly reduced by noticing that all parent nodes dominate their children. Instead of testing every node to the left of the current node, it is only necessary to test nodes whose parents are ancestors of the current node. If these nodes do not dominate the current node, then their children will not dominate the current node either.

SBDD guarantees that no equivalent nodes are processed. Unfortunately, this can still be a very computationally expensive approach in deep trees. If node a has depth i, it is possible to perform this dominance check up to i times. Testing for dominance is equivalent to solving subgraph isomorphism problems, and is known to be NP-complete. The

deeper the enumeration tree gets, the more costly this algorithm will be. Of course, dominance detection does not have to be performed at every node. Fahle et al. [16] have achieved better results by testing for dominance intermittently throughout the tree.

One potential problem with SBDD is that it may lead to the pruning of nodes in the tree much deeper is necessary (although this problem is remedied by additional variable fixing algorithms). While a node may not be dominated by any single node found to the left in the tree, it may still be dominated by some collection of such nodes.

Example Let Figure 2.11 represent a branch-and-bound tree for an ILP with 4 variables. Let $\mathcal{G} = \{e,$ $pi = (1,4)(2,3)\}$ be the symmetry group of the ILP.

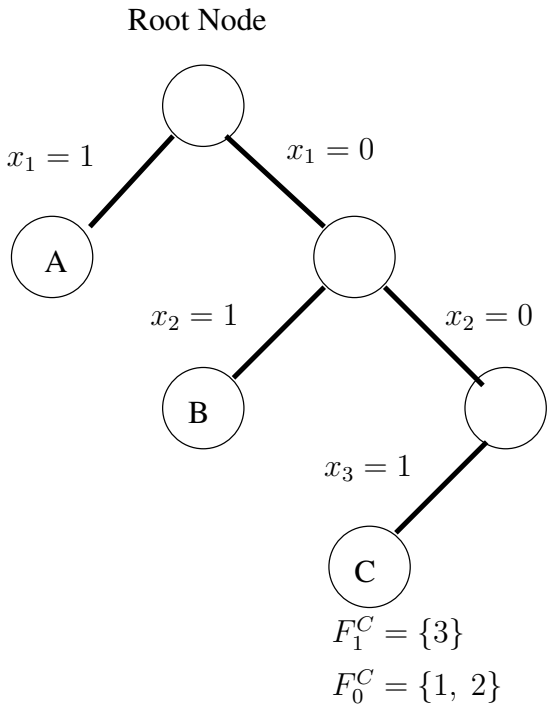


Figure 2.11: SBDD Example

At node C in the tree, SBDD tests if node C is dominated by either nodes A or B. Node A cannot dominate node C because the solution $[0,\ 0,\ 1,\ 0]$ is feasible at C and $\pi([0,\ 0,\ 1,\ 0]) = [0,\ 1,\ 0,\ 0]$ is not feasible at node A. Node B does not dominate C because the solution $[0,\ 0,\ 1,\ 1]$ is feasible at C and $\pi([0,\ 0,\ 1,\ 1]) = [1,\ 1,\ 0,\ 0]$ is not feasible at node B. As a result, node C is not pruned by SBDD. However, every feasible solution in C is equivalent to some feasible solution in either A or B. Note that node C would have been pruned by isomorphism pruning because $\pi(F_1^C) = \{2\}$ is lexicographically smaller than $F_1^C = \{3\}$. In this particular case, it would be easy to set x_4 to 0 at the parent node of B or x_3 to 0 at the parent node of C by using algorithms like orbit setting, avoiding the processing of unnecessary nodes. Using variable fixing algorithms like orbit setting will avoid this problem.

2.4.3 SBDS

Symmetry Breaking During Search (SBDS) [6] [27] [26][28] is another approach to dynamic symmetry breaking. At every node of the search tree, SBDS adds constraints that ensure that no two isomorphic solutions are allowed. While SBDS was developed for constraint programming, it will be discussed in terms of integer programs where the decision variables are binary.

In SBDS, at the node $a = (F_1^a, F_0^a)$, a variable x_i, $i \in N^a$, is chosen for branching. The disjunction imposed, however, is not the standard $x_i = 1 \vee x_i = 0$. Instead, the branching disjunction is

$$\sum_{j \in F_1^a} x_j + x_i = |F_1^a| + 1 \vee \sum_{j \in F_1^a} \pi(x_j) + \pi(x_i) \leq |F_1^a| \; \forall \pi \in \mathcal{G}(ILP).$$

Note that the left branch is equivalent to fixing x_i to 1. The disjunction mentioned above is trivial if $\mathcal{G}$ contains only the identity permutation. In this case, it reduces to $x_i = 1 \vee x_i = 0$. The case where symmetry is present in the problem is best described in the context of set covering. The left child is created by adding i to the cover. Constraints added to the right child ensure that no permutation variables in $F_1^a \cup j$ appear in a cover found to the right of a in the tree. If such a cover was found, it would be equivalent to a cover found in the left child of a. Consider the following example:

Example To demonstrate SBDS, the set covering problem from Example 1.4 is used.

$$\min \sum_{i=0}^{8} x_i$$
subject to

$$\begin{aligned}
x_0 + x_1 + x_2 + x_3 \qquad\qquad + x_6 \qquad\qquad &\geq 1 \\
x_0 + x_1 + x_2 \qquad + x_4 \qquad\qquad + x_7 \qquad &\geq 1 \\
x_0 + x_1 + x_2 \qquad\qquad + x_5 \qquad\qquad + x_8 &\geq 1 \\
x_0 \qquad\qquad + x_3 + x_4 + x_5 + x_6 \qquad &\geq 1 \\
x_1 \qquad + x_3 + x_4 + x_5 \qquad + x_7 \qquad &\geq 1 \\
x_2 + x_3 + x_4 + x_5 \qquad\qquad + x_8 &\geq 1 \\
x_0 \qquad\qquad + x_3 \qquad\qquad + x_6 + x_7 + x_8 &\geq 1 \\
x_1 \qquad\qquad + x_4 \qquad + x_6 + x_7 + x_8 &\geq 1 \\
x_2 \qquad\qquad + x_5 + x_6 + x_7 + x_8 &\geq 1 \\
x \in \mathbb{B}^9 &
\end{aligned}$$

Recall that the symmetry group of this ILP is generated by permutations found in Table 2.3.

π
(3, 6)(4, 7)(5, 8)
(1, 2)(4, 5)(7, 8)
(1, 3)(2, 6)(5, 7)
(0, 1)(3, 4)(6, 7)

Table 2.3: Generators for SBDS Example

At the root node, x_0 is chosen for branching. The left node then has $x_0 = 1$, while the right node is formed by adding the constraints $\pi(x_i) \leq 0 \; \forall \pi \in \mathcal{G}(ILP)$. Because all variables are equivalent at the root node, the constraints added to the right child fix all variables to zero. This makes the right problem infeasible.

The second branch is not as easy. Suppose x_1 was branched on. Again, the left child is formed by fixing x_1 to 1. The set of constraints $\pi(x_0) + \pi(x_1) \leq 1$ for every $\pi \in \mathcal{G}$ are added to the right child. The constraints are:

i $x_0 + x_1 \leq 1$

ii $x_0 + x_2 \leq 1$

iii $x_0 + x_3 \leq 1$

iv $x_0 + x_6 \leq 1$

v $x_1 + x_2 \leq 1$

vi $x_1 + x_4 \leq 1$

vii $x_1 + x_7 \leq 1$

viii $x_2 + x_5 \leq 1$

ix $x_2 + x_8 \leq 1$

x $x_3 + x_4 \leq 1$

xi $x_3 + x_5 \leq 1$

xii $x_3 + x_6 \leq 1$

xiii $x_4 + x_5 \leq 1$

xiv $x_4 + x_8 \leq 1$

xv $x_5 + x_9 \leq 1$

xvi $x_6 + x_7 \leq 1$

xvii $x_6 + x_8 \leq 1$

xviii $x_7 + x_9 \leq 1$

In this example, only 18 constraints are listed, even though the symmetry group $\mathcal{G}$ contains 72 elements. This is because constraints generated by each permutation may be redundant. For instance, the permutation $(3,6)(4,7)(5,8)$ is in the $\mathcal{G}$, however, it leaves the inequality $x_0 + x_1 \leq 1$ unchanged. Also note that these constraints are not ill-conditioned like the lexicographic constraints, but an exponential number may exist.

SBDS guarantees that all optimal solutions found are non-isomorphic. Also, each canonical solution is the leftmost solution (amongst other isomorphic solutions) in the tree. The implication of this structure is that unlike the static symmetry-breaking methods, for any branching strategy, the first optimal solution that would have been found without using symmetry breaking will still be found with the SBDS. The major disadvantage of this method, however, is the large number of constraints added to the subproblems. To reduce the number of inequalities added, Puget, in [73] only adds constraints for the permutations that leave the sets F_1^a and F_0^a alone (i.e. permutations that are in both $\mathrm{stab}(F_1^a, \mathcal{G})$ and $\mathrm{stab}(F_0^a, \mathcal{G})$. He calls this method the STAB method.

Example Using the STAB method, only the following constraints are added to the second subproblem:

i $x_0 + x_1 \leq 1$

ii $x_0 + x_2 \leq 1$

iii $x_0 + x_3 \leq 1$

iv $x_0 + x_6 \leq 1$

While the STAB method can significantly reduce the number of constraints needed, it does not guarantee that all generated solutions will be non-isomorphic.

2.4.4 Using Local Symmetry

In [63], Meseguer and Torras are not concerned with solving a given ILP, but instead focus on attempting to quickly find optimal or near optimal solutions. They examine how different branching strategies can affect the efficiency of finding a desired solution in cases where the ILP has a large amount of symmetry. Because their motivation is not to prove optimality, their strategy in branching is different than the above methods. They wish to maximize the number of solutions feasible to each subproblem in the tree. This is accomplished a general ILP by using the *minimum domain heuristic*. This heuristic chooses a branching variable with the smallest domain. The intuition behind this heuristic is to minimize the number of children early in the tree. This heuristic is discussed in [34] [78]. A result is that the chosen variable maximizes the number of final states considered in the subproblem. This is favorable because the goal of the CSP is to find a solution, and thus considering a wider set of final states would increase the likelihood of finding a solution. This branching strategy is not applicable to binary integer programs, since the domain of any free variable is $\{0, 1\}$, but the intuition of this branching rule can be adapted for problems with symmetry.

Generally, it is not sufficient to consider only subproblems with a large number of solutions. In fact, it is desirable for the subproblem to consider a large number of *non-isomorphic* solutions. The heuristic presented in [63], the *variety-maximization heuristic* aims to do just that. When all variables are binary, the variety-maximization heuristic always chooses to fix a variable from the largest orbit. This will break as many symmetries as possible, resulting in a subproblem with a higher density of non-isomorphic solutions. Of special interest is that the approach in [63] uses the local symmetry group of the subproblem to generate orbits, i.e., it includes symmetries that come about as a result of the branching process.

This paper is noteworthy in the context of this thesis for two reasons. First, it introduces the concept of branching on orbits and even develops a branching rule for orbits. This idea is discussed in great detail in Chapter 3. Also, this paper introduces an attempt to exploit symmetry that is introduced into a problem as a result of fixing variables. Using the concept of a local symmetry group is also discussed in Chapter 3.

2.5 Summary

There have been two major efforts to deal with symmetry in integer programming. The first, symmetry removal by reformulation, is a side-effect of methods developed to improve lower bounds. It is very effective at removing symmetry, but is only applicable to a specific type of problem. Also, the reformulation requires adding a potentially large number of variables to the problem. A more general method of mitigating the effects of symmetry is to use the symmetry group of the problem formulation to reduce the size of the feasible region. This method is very useful, as it is the only way to solve even small sized problems that cannot be reformulated.

Many methods that use symmetry to reduce the feasible region for a general problem require the use of sophisticated algebra packages. For example, isomorphic pruning, SBDS, and SBDD all require a significant amount of algebraic computations. This can make implementing these methods very complicated. In Chapter 3 we discuss an easily-implemented branching method that uses ideas from Meseguer and Torras [63] to solve ILP instances, by choosing a small, but not necessarily minimal, fundamental domain.

Most symmetry breaking methods, for instance, isomorphism pruning or orbitopal fixing, either explicitly restrict branching decisions or distort the value of branching on variables. Chapter 4 discusses a revision to isomorphism pruning that not only removes the branching restrictions, but doesn't distort branching information in the process.

Chapter 3

Orbital Branching

In this chapter, we will discuss a branching method called *orbital branching*. It is an easily implemented way to exploit symmetry in integer programming. Also, orbital branching provides a way to take advantage of symmetry that is introduced into the enumeration tree through the branch-and-bound process. Fixing variables can lead to changes in the structure of the problem that can have a considerable effect on the symmetry group. These fixings can create subproblems whose symmetry groups contain new symmetries and can allow for additional variables to be set at any nodes throughout the branch-and-bound tree.

In problems containing a great deal of symmetry, branching on a variable may lead to the fixing of the values of additional variables. It is important to consider these effects when choosing a variable on which to branch. For instance, fixing a single variable to either 1 or 0 may not affect the corresponding LP solution. However, variables that can be set as a result of the fixing may have a significant affect on the value of the resulting LP solution. In orbital branching, we attempt to develop branching rules that takes into account information provided by the symmetry group to better choose branching variables. The outline of the chapter is as follows: Section 3.1 develops the basic algorithm for solving an integer program using only the symmetry found in the original problem. Section 3.2 presents a way to further fix variables using symmetry. Section 3.4 discusses branching rules and gives computational results. Orbital branching does not fully exploit the symmetry. A discussion of how symmetry can remain and how branching rules can effect the use of symmetry can be found in Section 3.5. A comparison of orbital branching with other symmetry breaking methods is found in Section 3.6.1.

3.1 Orbital Branching

3.1.1 Method

Orbital branching is an intuitive way of exploiting the symmetry group $\mathcal{G}(ILP)$ during branching decisions. The classic 0-1 variable branching dichotomy does not take advantage of the problem information encoded in the symmetry group. The presence of symmetry provides ways of strengthening both branching disjunctions and variable fixing algorithms. Symmetry can be exploited ether by fixing variables by symmetry considerations either at the node or during branching. Algorithms discussed in Chapter 2 such as orbital setting and zero-fixing use symmetry to fix variables only at the node. Orbital branching fixes variables during branching. By using symmetry considerations to strengthen branching decisions, the affects of branching on a given variable are better known at the time the branching decision is made. Thus results in better branching decisions.

To take advantage of information provided by symmetry, orbital branching uses orbits of variables to create the branching dichotomy, not individual variables. Informally, suppose that at the current subproblem there is an orbit of cardinality k. In orbital branching, the current subproblem is split into $k+1$ subproblems: the first k subproblems are obtained by fixing each variable in the orbit to one while the $(k+1)^{\text{st}}$ subproblem is obtained by fixing all variables in the orbit to zero. For any pair of variables x_i and x_j with i and j sharing an orbit, the subproblem created when x_i is fixed to one is essentially equivalent to the subproblem created when x_j is fixed to one. Therefore, we can keep in the subproblem list only *one* representative subproblem, pruning the $(k-1)$ equivalent subproblems. This is formalized below.

3.1.2 Description of Methods

Let $\mathcal{G}^a$ be the symmetry group of the subproblem represented by node a in the branch-and-bound tree. Let $O = \{i_1, i_2, \ldots, i_{|O|}\} \subseteq N^a$ be an orbit of the symmetry group $\mathcal{G}^a$. Given a subproblem a, the disjunction

$$x_{i_1} = 1 \vee x_{i_2} = 1 \vee \ldots x_{i_O} = 1 \vee \sum_{i \in O} x_i = 0 \tag{3.1}$$

splits the feasible region. In what follows, it will be shown that for any two variables x_j and x_k with $i, j \in O$, the two children $a(j)$ and $a(k)$ of a, obtained by fixing respectively x_j and x_k to 1, have the same optimal solution value. As a consequence, disjunction (3.1) can be replaced by the binary disjunction

$$x_h = 1 \vee \sum_{i \in O} x_i = 0, \tag{3.2}$$

where h is an arbitrary element in O. Note that the additional variable fixings in 3.2 can be done by the orbit-setting algorithm in isomorphism pruning and equivalent algorithms in constraint programming literature. Formally, we have

Theorem 3.1.

Theorem 3.1 *Let O be an orbit in the orbital partitioning $\mathcal{O}(\mathcal{G}^a)$, and let j, k be two variable indices in O. If $a(j) = (F_1^a \cup \{j\}, F_0^a)$ and $a(k) = (F_1^a \cup \{k\}, F_0^a)$ are the child nodes created when branching on variables x_j and x_k, then $z^*(a(j)) = z^*(a(k))$.*

Proof. Let x^* be an optimal solution of $a(j)$ with value $z^*(a(j))$. Obviously x^* is also feasible for a. Since j and k are in the same orbit O, there exists a permutation $\pi \in \mathcal{G}^a$ such that $\pi(j) = k$. By definition, $\pi(x^*)$ is a feasible solution of a with value $z^*(a(j))$ such that $x_k = 1$. Therefore, $\pi(x^*)$ is feasible for $a(k)$, and $z^*(a(k)) = z^*(a(j))$□

The basic *orbital branching* method is formalized in Algorithm 3.1.

Algorithm 3.1 Orbital Branching

Input:	Subproblem $a = (F_1^a, F_0^a)$, non-integral solution $\hat{x}$.
Output:	Two child subproblems b and c.
Step 1.	Compute orbital partition $\mathcal{O}(\mathcal{G}^a) = \{O_1, O_2, \ldots, O_p\}$.
Step 2.	Select orbit O_{j^*}, $j^* \in \{1, 2, \ldots, p\}$.
Step 3.	Choose arbitrary $k \in O_{j^*}$. Return subproblems $b = (F_1^a \cup \{k\}, F_0^a)$ and $c = (F_1^a, F_0^a \cup O_{j^*})$.

A general binary ILP is then solved by using LP-based branch-and-bound, where children of processed nodes are created by algorithm 3.1. The consequence of Theorem 3.1 is that the search space is limited, but orbital branching also has the relevant effect of reducing the likelihood of encountering symmetric solutions. Namely, no solutions in the left and right child nodes of the current node will be symmetric with respect to the local symmetry. This is formalized in Theorem 3.2. Methods for selecting an orbit on which to branch (Step 2 of algorithm 3.1) are discussed in Section 3.3.2.

Theorem 3.2 *Let b and c be any two subproblems in the enumeration tree. Let a be the first common ancestor of b and c. If $a \neq \{b, c\}$ then there $\nexists x \in \mathcal{F}^b$ such that $\exists \pi \in \mathcal{G}^a$ with $\pi(x) \in \mathcal{F}^c$.*

Proof. We will show this by contradiction. Suppose that $\exists x \in \mathcal{F}^b$ and a permutation $\pi \in \mathcal{G}^a$ such that $\pi(x) \in \mathcal{F}^c$. Let $O_i \in \mathcal{O}(\mathcal{G}^a)$ be the orbit chosen to branch on at subproblem a. W.l.o.g. we can assume $x_k = 1$ for some $k \in O_i$, that is, b is in the left branch of a. We have that $x_k = [\pi(x)]_{\pi(k)} = 1$, but $\pi(k) \in O_i$. By the orbital branching dichotomy, $\pi(k) \in F_0^c$, so $\pi(x) \notin \mathcal{F}^c$. □

Note that by using the group $\mathcal{G}^a$, orbital branching attempts to use symmetry found at all nodes in the enumeration tree, not just the symmetry found at the root node. This makes it possible to prune nodes whose corresponding solutions are not symmetric in the original ILP. As a result, the area searched by orbital branching need not be a fundamental domain of $\mathcal{F}$ with respect to the symmetry group found at the root node.

3.1.3 Illustrative Example

Example In order to demonstrate the effects of orbital branching, consider the following example. Let $G = (V, E)$ be the graph in Figure 3.1 and the associated PILP:

$$\begin{aligned} \max & \sum_{i \in V} x_i \\ x_i + x_j & \leq 1 \qquad \forall \{i, j\} \in E, \\ x_i & \in \{0, 1\} \qquad \forall i \in V \end{aligned}$$

which corresponds to computing the stability number of G.

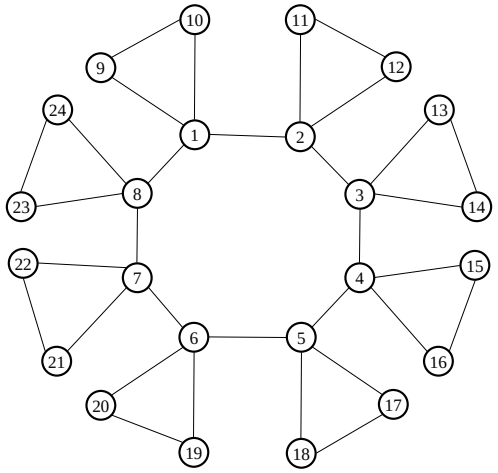
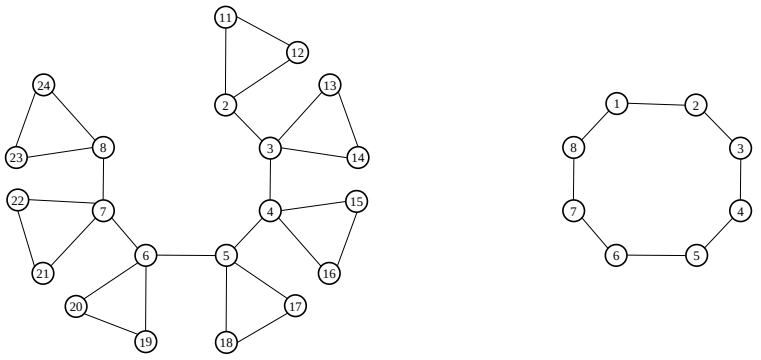


Figure 3.1: Example

The LP relaxation of the root subproblem, r, gives an upper bound of 12. Applying Step 1 of Algorithm 3.1 at the root node results in a group $\mathcal{G}^r$ containing 4096 permutations and an orbital partition $\mathcal{O}(\mathcal{G}^r)$ containing two orbits, namely, $O_1 = \{1,\ldots,8\}$ and $O_2 = \{9, \ldots, 24\}$. Thanks to the structure of the problem, in which each constraint corresponds to an edge of G, the orbits of $\mathcal{G}^r$ can be intuitively visualized on the graph.

Step 2 of Algorithm 3.1 selects an orbit on which to base the branching dichotomy. Suppose the largest orbit O_2 is chosen, and the branching index $k = 9 \in O_2$ is used. Then, two subproblems b and c are generated as follows: $F_1^b = \{9\}$ and $F_0^b = \emptyset$; $F_1^c = \emptyset$ and $F_0^c = \{9, \ldots, 24\}$. The structure of subproblems b and c, where fixed variables have been removed, is drawn in Figure 3.2. The LP relaxation of subproblem c is 4. Because a solution of size 8 is assumed to be known, node c can be pruned by bound. However, node b has an LP relaxation value of 11.5 and cannot be pruned.



subproblem b subproblem c

Figure 3.2: Child subproblems

The advantage of orbital branching over classical branching on a variable is highlighted by completely executing two branch-and-bound algorithms on the PILP of Example 3.1.3. It is assumed that a feasible solution of (optimal) value 8 is found at the root node. In the first algorithm, the branching decision is carried out by branching on the largest orbit. In the second algorithm, ordinary branching is performed on the variable corresponding to the vertex of G with maximum degree in the remaining graph, a typically effective strategy for stable set problems [76]. In Figures 3.3 and 3.4, the complete enumeration trees obtained by orbital branching and branching on variables, respectively, are drawn. At each node a, the variables fixed (F_1^a, F_0^a) and the value of the LP relaxation z_{LP} are reported. Orbital branching results in fewer evaluated subproblems: 21 vs. 49 for the variable-branching dichotomy.

An insightful explanation of orbital branching's improved performance is obtained by examining the structure of subproblems. For instance, Figure 3.5 shows the graphs remaining at subproblems 9 and 19 of the variable-branching enumeration tree. The graphs are isomorphic, but both subproblems are evaluated when branching on variables. In contrast, orbital branching breaks such a symmetry at the root subproblem. The complete catalog of graphs and orbital partitions for each subproblem in the orbital branching branch-and-bound tree is reported in the Appendix. Looking at the catalog of subproblems, one can observe that no isomorphic subproblems are evaluated when orbital branching is used on this example. This is not, however, true in general (see Section 3.5).

3.2 Enhancements to Orbital Branching

This section demonstrates how additional variables may be fixed during branch-and-bound by considering the implications of symmetry. This section also discusses how to perform orbital branching by considering a subgroup of the original symmetry group. Orbital branching is compared to a related technique for exploiting symmetry in integer programs, isomorphism pruning. The section concludes with a brief discussion on how to most effectively employ

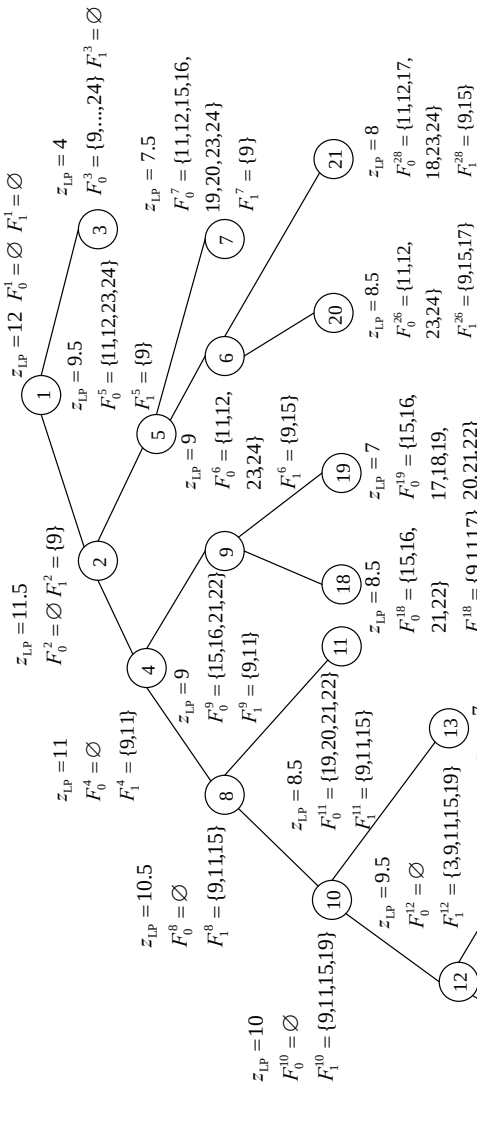


Figure 3.3: Enumeration tree with orbital branching

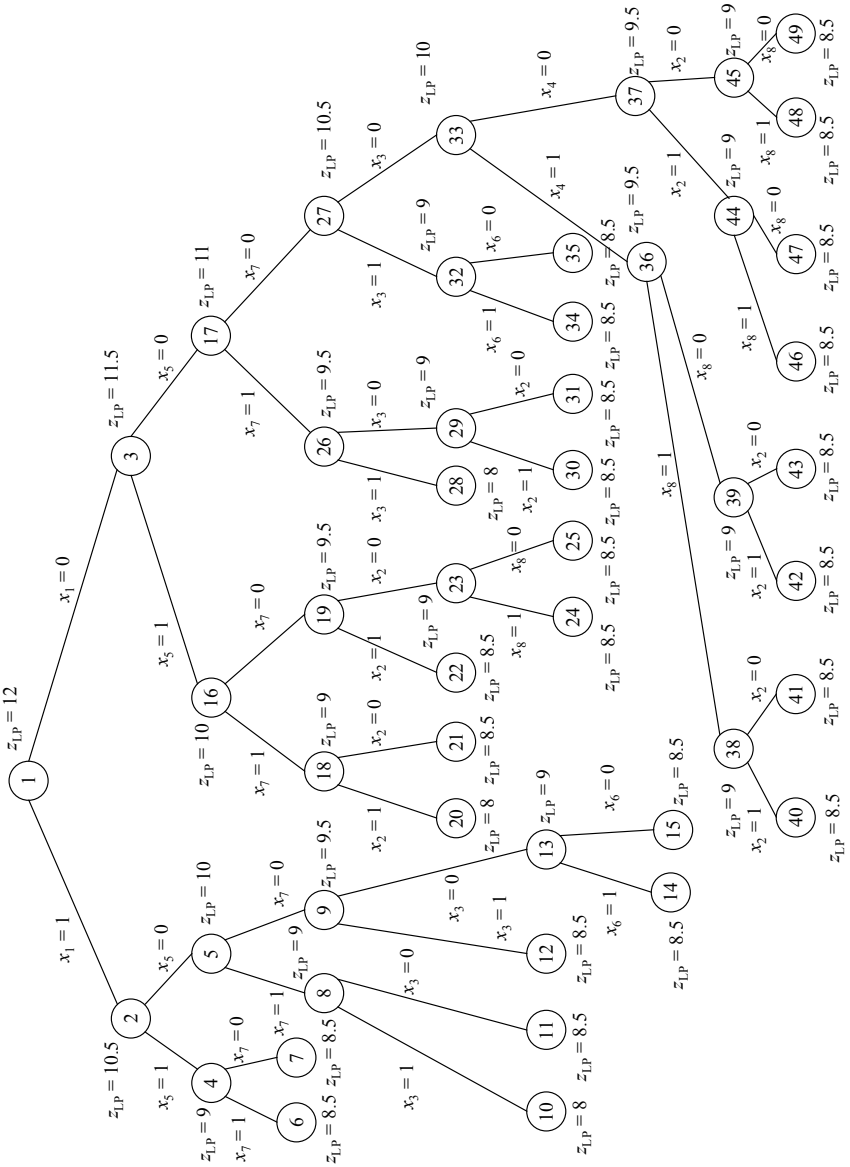
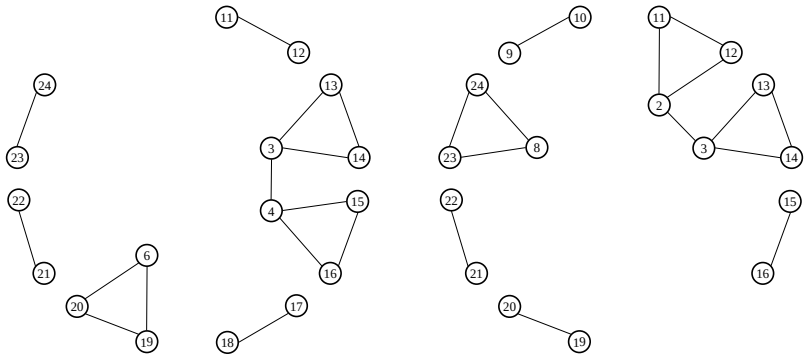


Figure 3.4: Enumeration tree with branching on variable



subproblem 9 subproblem 19

Figure 3.5: Isomorphic Subproblems when Branching on Variables

orbital branching on integer programs whose optimal solution has a large support.

3.2.1 Orbital Fixing

As Theorem 3.2 demonstrates, using orbital branching ensures that no two nodes that are equivalent with respect to the symmetry found at their first common ancestor can be created. It is possible, however, for two child subproblems to be equivalent with respect to a symmetry group found elsewhere in the tree. In order to combat this type of symmetry, *orbital fixing* is performed.

Permutations in the symmetry group $\mathcal{G}^a$ tend to stabilize the sets of elements representing fixed variables. By considering a larger symmetry group, one that does not stabilize the set F_0^a, variables can be found that can be fixed to zero. For $a = (F_1^a, F_0^a)$, let $\hat{a}$ represent the subproblem $\hat{a} = (F_1^a, \emptyset)$, where all variables in F_0^a are free variables. If there exists an orbit O in the orbital partition $\mathcal{O}(\mathcal{G}^{\hat{a}})$ that contains variables such that $O \cap F_0^a \neq \emptyset$ and $O \cap N^a \neq \emptyset$, then all variables in O can be fixed to zero. In the following theorem, it is shown that orbital fixing excludes feasible solutions only if there exists a feasible solution of the same objective value to the left of the current node in the branch-and-bound tree. (It is assumed that the enumeration tree is oriented so that the branch with an additional variable fixed to one is the left branch). Similar methods of fixing variables are discussed in isomorphism pruning (orbit setting) and in constraint programming literature (zero-fixing).

Orbital fixing is able to fix variables by ensuring that if any optimal solution is removed by the fixing, then there is an optimal solution found somewhere to the left of the current node. To aid in the development, the concept of a *focus node* is introduced. For $x \in \mathcal{F}(a)$, we call node $b(a, x)$ a focus node of a with respect to x if $\exists y \in \mathcal{F}(b)$ such that $e^T x = e^T y$ and b is found to the left of a in the tree.

Theorem 3.3 *Let $\{O_1, O_2, \dots O_q\}$ be an orbital partitioning of $\mathcal{G}^{\hat{a}}$ at node a, and let the set*

$$S \stackrel{\text{def}}{=} \{j \in N^a \mid \exists k \in F_0^a \text{ and } j, k \in O_\ell \text{ for some } \ell \in \{1, 2, \dots q\}\}$$

be the set of free variables that share an orbit with a variable fixed to zero at a. If $x \in \mathcal{F}(a)$ with $x_i = 1$ for some $i \in S$, then either there exists a focus node for a with respect to x or x is not an optimal solution.

Proof:

If S is nonempty, then there exists $j \in F_0^a$, $i \in S$, and $\pi \in \mathcal{G}^{\hat{a}}$, with $\pi(i) = j$. W.l.o.g., suppose that j is any of the first of such variables fixed to zero on the path from the root node to a and let c be the first subproblem in which j is fixed. Let $\rho(c)$ be the parent node of c. For any x feasible at a with $x_i = 1$, $\pi(x)$ is not feasible at a or at c (since $\pi(i) = j$ and $j \in F_0^c \subseteq f_0^a$. However, by our choice of j as the first fixed variable, x is feasible in $\rho(c)$ and has the same objective value of x.

The variable x_j could have been fixed either (i) as a result of a branching decision, or (ii) because it was deduced that no optimal solution exists with $x_j = 1$ at node $\rho(c)$ (and the fixing applied to the child nodes), or (iii) by orbital fixing (at $\rho(c)$).

i If j was fixed by orbital branching, then the left child of $\rho(c)$ has $x_h = 1$ for some $h \in \text{orb}(j, \mathcal{G}^{\rho(c)})$. Let $\pi' \in \mathcal{G}^{\rho(c)}$ have $\pi'(j) = h$. Then $\pi'(\pi(x))$ is feasible in the left node with the same objective value of x. The left child node of $\rho(c)$ is then the focus node of a with respect to x.

ii If it was deduced that no optimal solution feasible at $\rho(c)$ exists with $x_j = 1$, then since $\pi(x)$ is feasible in $\rho(c)$ with $x_j = 1$, and π preserves the objective value, x cannot be an optimal solution.

iii Lastly, j could have been fixed by orbital fixing. This implies that the set S is nonempty in $\rho(c)$ and the argument can be repeated until the first ancestor d of a is reached such that F_0^d does not contain variables fixed by orbital fixing. Therefore, a sequence of permutations $\pi^1, \dots, \pi^r$ have been found such that $\pi^r \circ \pi^{r-1} \circ \dots \circ \pi^1 \circ \pi(x)$ is feasible in d and has the same value of x.

Then, either argument (i) or (ii) can be applied. That is, either there is a focus node f of d with respect to $\pi^r \circ \pi^{r-1} \circ \dots \circ \pi^1 \circ \pi(x)$ (which would also be a focus node for a with respect to x), or j was fixed by an optimality condition (which implies $\pi^r \circ \pi^{r-1} \circ \dots \circ \pi^1 \circ \pi(x)$ and thus x is not optimal).

There may be elements in S that do not share an orbit with j. One can show that these elements can also be fixed by adding the fixed variables to F_0, updating S, and repeating the argument. As long as S is nonempty, each iteration will fix at least one variable. □

Algorithm 3.2 Orbital Branching with Orbital Fixing

Input:	Subproblem $a = (F_1^a, F_0^a)$ (with free variables $N^a = I^n \setminus F_1^a \setminus F_0^a$), fractional solution $\hat{x}$.
Output:	Two child nodes b and c.
Step 1.	Compute orbital partition $\mathcal{O}(\mathcal{G}^{\hat{a}}) = \{\hat{O}_1, \hat{O}_2, \ldots, \hat{O}_q\}$. Let $S \stackrel{\text{def}}{=} \{j \in N^a \mid \exists k \in F_0^a \text{ and } (j \cap k) \in \hat{O}_\ell \text{ for some } \ell \in \{1, 2, \ldots q\}\}$.
Step 2.	Compute orbital partition $\mathcal{O}(\mathcal{G}^a) = \{O_1, O_2, \ldots, O_p\}$.
Step 3.	Select orbit O_{j^*}, $j^* \in \{1, 2, \ldots, p\}$.
Step 4.	Choose arbitrary $k \in O_{j^*}$. Return child subproblems $b = (F_1^a \cup \{k\}, F_0^a \cup S)$ and $c = (F_1^a, F_0^a \cup O_{j^*} \cup S)$.

An immediate consequence of Theorem 3.3 is that for all $i \in F_0^a$ and for all $j \in \text{orb}(i, \mathcal{G}^{\hat{a}})$, one can set $x_j = 0$. Orbital branching is updated to include orbital fixing in Algorithm 3.2. With orbital fixing, the set S of additional variables set to zero depends on F_0^a. Variables may appear in F_0^a due to a branching decision or due to traditional methods for variable fixing in integer programming, e.g., reduced cost fixing or implication-based fixing. Orbital fixing, then, *enhances* traditional variable-fixing methods by including the symmetry present at a node of the branch-and-bound tree.

Example (continued) When orbital branching with orbital fixing is applied to the PILP of Example 3.1.3, it generates the enumeration tree shown in Figure 3.6. Orbital fixing is performed at subproblem 6, a node that has $F_0^6 = \{11, 12, 23, 24\}$ and $F_1^6 = \{9, 15\}$. The group $\mathcal{G}^{\hat{6}}$ yields the orbits: $\{2, 3\}$ $\{5, 8\}$ $\{6, 7\}$ $\{11, 12, 13, 14\}$ $\{17, 18, 23, 24\}$ $\{19, 20, 21, 22\}$. The orbit $\{11, 12, 13, 14\}$ contains variables that have already been set to zero: $\{11, 12, 13, 14\} \cap F_0^a = \{13, 14\}$. Therefore, the variables x_{13} and x_{14} are fixed to 0 by orbital fixing. In the same way, considering the orbit $\{17, 18, 23, 24\}$, orbital fixing sets variables x_{17} and x_{18} to 0. All the variables fixed to 0 by orbital fixing are underlined in Figure 3.6.

The effect of orbital fixing is clear at subproblem 6, where the optimal value of the LP relaxation is reduced from 9 to 7, as compared to the algorithm without orbital fixing, avoiding further branching (see the tree of Figure 3.3).

The example also helps illustrate the existence of a focus node if orbital fixing is performed (Theorem 3.3). Define a as the subproblem found at node 6. The set of variables fixed by orbital fixing is $S = \{13, 14, 17, 18\}$. Consider the solution $x \in \mathcal{F}(a)$: $x_2 = x_5 = x_8 = x_9 = x_{13} = x_{15} = x_{19} = x_{21} = 1$, and all other variables set to 0. The solution x is removed from $\mathcal{F}^a$ by orbital fixing because there is a solution to the left of node a with the same cardinality. Following the proof of Theorem 3.3, $i = 13$ and $j \in \text{orb}(\{i\}, \mathcal{G}^{\hat{6}})$ with $x_j \in F_0^6$, i.e., $j = 12$. A permutation $\pi \in \mathcal{G}^{\hat{6}}$ such that $\pi(12) = 13$ is: $[(2, 3), (12, 13), (11, 14)]$. Then, $\bar{x} = \pi(x)$, that is, $\bar{x}_3 = \bar{x}_5 = \bar{x}_8 = \bar{x}_9 = \bar{x}_{12} = \bar{x}_{15} = \bar{x}_{19} = \bar{x}_{21} = 1$, and all other variables set to 0. Notice that $\bar{x} \notin \mathcal{F}(a)$, since $x_{12} = 1$, but $\bar{x}$ is feasible in $\mathcal{F}^2$. By definition, subproblem 5 is the subproblem c in the proof of Theorem 3.3, and subproblem 2 is the subproblem $\rho(c)$. The solution $\bar{x}$ is not feasible at node 4, but $\bar{x}$ is equivalent to a feasible solution

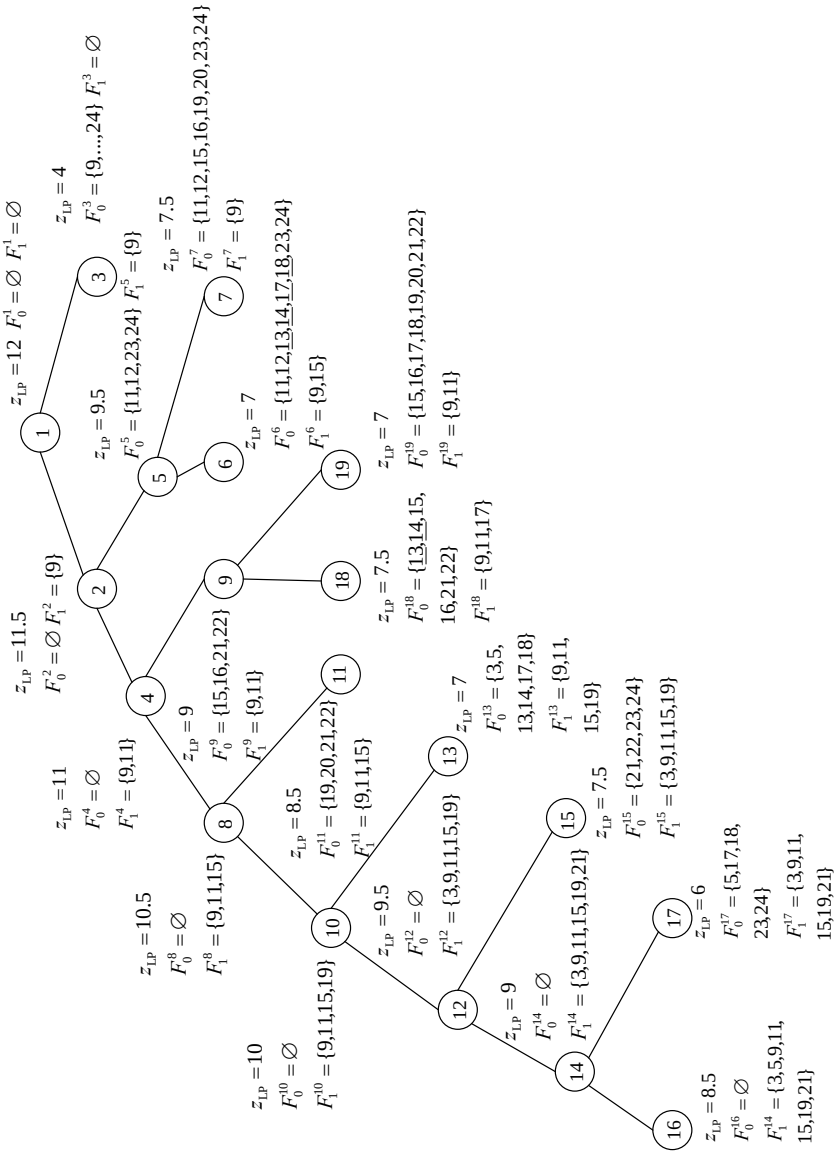


Figure 3.6: Enumeration tree with orbital branching and orbital fixing

in node 4 with respect the symmetry group $\mathcal{G}^2$.

Node 4 was created by fixing x_{11} to 1. Thus, we have $h = 11$ and $\pi' \in \mathcal{G}^2$ can be defined as: $(11, 12)$. Finally, $\tilde{x} = \pi'(\bar{x}) = \pi'(\pi(x))$ is: $\tilde{x}_3 = \tilde{x}_7 = \tilde{x}_9 = \tilde{x}_{11} = \tilde{x}_{15} = \tilde{x}_{17} = \tilde{x}_{19} = \tilde{x}_{23} = 1$, and all other variables set to 0. This is feasible for subproblem 4. Thus, 4 is a focus node for a with respect to solution x.

3.2.2 Reversing Orbital Branching

One of the advantages of orbital branching is that the "right" branch, in which all variables in the branching orbit O are fixed to zero, typically changes the optimal value of the LP relaxation significantly, and the left branch, in which one variable in O is fixed to one, also has a significant impact on the problem. In some classes of ILPs, fixing a variable to zero can have more impact than fixing a variable to one. This is typically true in instances in where the number of ones in an optimal solution is larger than 1/2 the number of variables. In such cases, orbital branching is much more efficient if all variables are complemented, or equivalently if the orbital branching dichotomy (3.2) is replaced by its complement. Margot [55] also makes a similar observation for his isomorphism pruning algorithm, and he solves the complemented versions of such instances. Orbital branching opts for the former way of exploiting this fact, where the "left" branch fixes one variable to zero and orbital fixing fixes variables to one instead of zero.

3.3 Implementation

The orbital branching method has been implemented using the user application functions of MINTO v3.1 [65]. The branching dichotomy of Algorithm 3.1 or 3.2 is implemented in the `appl_divide()` method, and reduced cost fixing is implemented in `appl_bounds()`. The entire implementation, including code, for all the branching rules subsequently introduced in Section 3.3.2, consists of slightly over 1000 lines of code. All advanced ILP features of MINTO were used, including *clique inequalities*, which can be useful for instances of (3.3). In this section, we discuss the features of the implementation that are specific to orbital branching—the computation of the symmetry groups and orbital branching rules.

3.3.1 Using a Subgroup of the Original Symmetry Group

Computation of the symmetry group $\mathcal{G}^a$ is discussed in Section 1.5. All known algorithms that compute the symmetry group of a given graph have worst-case exponential running times. Thus, computing the symmetry group $\mathcal{G}^a$ at each node a may be computationally prohibitive. It is shown via computational results in Section 3.4 that this is often not the case. However, in the case that recomputing the full symmetry group $\mathcal{G}^a$ is too costly, there is an alternative. Orbital branching can use the symmetry group $\mathrm{stab}(F_1^a, \mathcal{G}^r)$, where $\mathcal{G}^r$ is the symmetry group of the root node, to

create orbits at every node in the tree. In this method, the original global symmetry group $\mathcal{G}^r$, is computed once and only the stabilizers are computed at nodes in the tree. Using stabilizers is typically more computationally efficient than re-computing the symmetry groups from scratch. In order to distinguish between the two symmetry groups that could be used in orbital branching at node a, $\mathrm{stab}(F_1^a, \mathcal{G}^r)$ is referred to as the *global symmetry group* (because only the symmetry group found at the root node is used), and $\mathcal{G}^a$ is referred to as the *local symmetry group*.

When using the global symmetry group the decrease in computational overhead for computing orbits comes at a price. As Theorems 3.4 and 3.5 demonstrate, the global group $\mathrm{stab}(F_1^a, \mathcal{G}^r)$ is a subset of the local group $\mathcal{G}^a$. As a result, when using the global group, the branching dichotomy and fixing mechanisms are weaker (as the orbits generated by the global group will be a subdivision of the orbits generated by the local group).

Theorem 3.4 $\mathrm{stab}(F_1^a, \mathcal{G}^r) \subseteq \mathcal{G}^{\hat{a}}$.

Proof Let π be any permutation in $\mathrm{stab}(F_1^a, \mathcal{G}^r)$. The permutation π preserves objective values and feasibility in r. Since $\hat{a} = (F_1^a, \emptyset)$ and π stabilizes the set F_1^a, for any $x \in \mathcal{F}^{\hat{a}}$, $\pi(x)$ is feasible at the root node r and has $\pi(x)_i = 1$ for all $i \in F_1^a$. The solution $\pi(x)$ is also feasible in $\hat{a}$. Thus, $\pi \in \mathcal{G}^{\hat{a}}$. □

Orbital fixing does not change the result of Theorem 3.4. Specifically, if S^a is the set of indices of variables fixed to zero by orbital fixing at node a, then the orbits from the group $\mathcal{G}^{\hat{a}}$ are a subdivision of orbits from the group $\mathcal{G}^{a^*}$, where subproblem $a^* = (F_1^a, F_0^a \cup S^a)$.

Theorem 3.5 $\mathcal{G}^{\hat{a}} \subseteq \mathcal{G}^{a^*}$.

Proof. For any $\pi \in \mathcal{G}^{\hat{a}}$, π preserves objective values and feasibility in $\mathcal{F}^{\hat{a}}$. Note also that $\mathcal{F}^{a^*} \subseteq \mathcal{F}^{\hat{a}}$. For any $x \in \mathcal{F}^{a^*}$, $\pi(x) \in \mathcal{F}^{\hat{a}}$. If $\pi(x) \notin \mathcal{F}^{a^*}$, then there must be some $i \in F_0^a \cup S^a$ with $\pi(x)_i = 1$. However, because $F_0^a \cup S^a$ is a union of orbits, $\pi^{-1}(i)$ must also be in $F_0^a \cup S^a$, but $\pi(x)_i = 1$ only if $x_{\pi^{-1}(i)} = 1$, a contradiction. □

The global symmetry group $\mathcal{G}^r$ for the ILP $\min_{x \in \{0,1\}^n}\{c^T x | Ax \leq b\}$ can be approximated by the formulation group $\mathcal{G}(A, b, c)$. However, as discussed in Chapter 1, the formulation of a problem can have a significant effect on how well $\mathcal{G}(A, b, c)$ approximates $\mathcal{G}^r$. While it is not clear what the best approach is for generating a good formulation to a specific problem instance, it is reasonable to assume that the formulation of the root node was chosen in a way that it represents a reasonable approximation to G^r. However, it is not reasonable to expect all subproblems to be formulated in a way that provides an accurate approximation to the local symmetry group. The set of fixed variables that define the subproblem may render collections of constraints in the subproblem formulation redundant. These redundant constraints may result in a significantly decreased local formulation group. To avoid issues resulting from poor formulations, this chapter focuses on set covering and set packing problems. In both cases, all variables

represented by the sets F_1^a and F_0^a are removed from the formulation for subproblem a. Constraints that include variables i for all $i \in F_1^a$ are also removed. In the set packing case, all variables not in F_1^a that appear in removed constraints are also removed from the subproblem (and included in F_0^a). The symmetry group of node a found by using the subproblem processed in this way will be referred to as $\mathcal{G}(A^a, b^a, c^a)$.

3.3.2 Branching Rules

The orbital branching rule introduced in Section 3.1 leaves significant freedom in choosing the orbit on which to base the branching (Step 2 of Algorithm 3.1). In this section, mechanisms for deciding on which orbit to branch are discussed. A fractional solution $\hat{x}$ and orbits $O_1, O_2, \ldots O_p$ (consisting of all currently free variables) of the orbital partition $\mathcal{O}(\mathcal{G}^a)$ are given as input to the branching decision for the subproblem at node a. Output of the branching decision is an index j^* of an orbit on which to base the orbital branching. Six different branching rules are tested.

Rule 1: Branch Largest The first rule chooses to branch on the largest orbit O_{j^*}:

$$j^* \in \arg \max_{j \in \{1,\ldots,p\}} |O_j|.$$

Rule 2: Branch Largest LP Solution The second rule branches on the orbit O_{j^*}, whose variables have the largest total solution value in the fractional solution $\hat{x}$:

$$j^* \in \arg \max_{j \in \{1,\ldots,p\}} \hat{x}(O_j).$$

Rule 3: Strong Branching The third rule is a strong branching rule. For each orbit j, two tentative children are created and their bounds z_j^+ and z_j^- are computed by solving the resulting linear programs. The orbit j^*, for which the product of the change in linear program bounds is largest, is used for branching:

$$j^* \in \arg \max_{j \in \{1,\ldots,p\}} (|e^T\hat{x} - z_j^+|)(|e^T\hat{x} - z_j^-|).$$

A combination of the bound changes

$$j^* \in \arg \max_{j \in \{1,\ldots,p\}} (3\min(|e^T\hat{x} - z_j^+|, |e^T\hat{x} - z_j^-|) + \max(|e^T\hat{x} - z_j^+|, |e^T\hat{x} - z_j^-|)),$$

was also suggested by [48], but the computational results with the max product of the change were slightly stronger.

Note that if one of the potential children in the strong branching procedure was be pruned, either by bound or by infeasibility, then the bounds on the variables may be fixed to their values on the alternate child node. This is referred

to as *strong branching fixing*, and the computational results in the Appendix report the number of variables fixed in this manner. As discussed at the end of Section 3.2.1, variables fixed by strong branching fixing may allow for additional variables to be fixed by orbital fixing.

Rule 4: Break Symmetry Left: This rule is similar to *strong branching*, but instead of fixing a variable and computing the change in objective value bounds, a variable is fixed and the change in the size of the symmetry group is computed. Specifically, for each orbit j, the size of the symmetry group in the resulting left branch is computed as if orbit j (including variable index i_j) was chosen for branching. Recall $a(j) = (F_1^a \cup \{j\}, F_0^a)$. The orbit that reduces the symmetry by as much as possible:

$$j^* \in \arg\min_{j \in \{1,\dots,p\}} \left(|\mathcal{G}^{a(i_j)}| \right)$$

is chosen for branching.

Rule 5: Keep Symmetry Left This branching rule is the same as **Rule 4**, except that the orbit for which the size of the child's symmetry group would remain the largest:

$$j^* \in \arg\max_{j \in \{1,\dots,p\}} \left(|\mathcal{G}^{a(i_j)}| \right).$$

is chosen for branching.

Rule 6: Branch Max Product Left This rule attempts to branch on a large orbit at the current level while also keeping a large orbit at the second level on which to base the branching dichotomy. For each orbit $O_1, O_2, \dots, O_p$, the orbits $P_1^j, P_2^j, \dots, P_q^j$ of the symmetry group $\mathcal{G}^{a(i_j)}$ of the left child nodes are computed for some variable index $i_j \in O_j$. The orbit j^* for which the product of the orbit size and the largest orbit of the child subproblem is largest:

$$j^* \in \arg\max_{j \in \{1,\dots p\}} \left(|O_j| (\max_{k \in \{1,\dots q\}} |P_k^j|) \right).$$

is chosen for branching.

3.4 Computational Experiments

In this section, empirical evidence of the effectiveness of orbital branching is given. The impact of choosing the orbit on which branching is based is investigated, and the positive effect of orbital fixing is demonstrated. The computations are based on the instances whose characteristics are given in Table 3.1. The instances beginning with `cod` are used to compute maximum cardinality binary error-correcting codes [50]. The instances whose names begin with `cov` are covering design problems [64], the instance `f5` is the "football pool problem" on five matches [33], and the

Name	Variables	Group Size
cod83	256	10,321,920
cod93	512	185,794,560
cod105	1024	3,715,891,200
cov1053	252	3,628,800
cov1054	252	3,628,800
cov1075	120	3,628,800
cov1076	120	3,628,800
cov954	126	362,880
f5	243	933,120
sts45	45	360
sts63	63	72,576
sts81	81	1,965,150,720

Table 3.1: Symmetric Integer Programs

instances `sts` are used to compute the incidence width of the well-known Steiner-triple systems [22]. The `cov` formulations have been strengthened with a number of Schöenheim inequalities, derived by Margot [56]. The `sts` instances typically have roughly 2/3 of the variables equal to one in an optimal solution, so for these instances, the orbital branching dichotomy is reversed, as explained in Section 3.2.2. All instances, save for `f5`, are available from Margot's web site: `http://wpweb2.tepper.cmu.edu/fmargot/lpsym.html`.

The computations were run on machines with AMD Opteron processors clocked at 1.8GHz and having 2GB of RAM. The COIN-OR software `Clp` was used to solve the linear programs at nodes of the branch and bound tree. For each instance, the (known) optimal solution value was set a priori to aid pruning and reduce the random impact of finding a feasible solution in the search (with a tolerance of .05). Nodes were searched in a depth-first fashion. When the size of the maximum orbit in the orbital partitioning was less than or equal to two, nearly all of the symmetry in the problem was eliminated by the branching procedure, and there was little use in performing orbital branching. In this case, we used MINTO's default branching strategy [48]. If orbital branching is not performed at a node, then there is little likelihood that it will be effective at the node's children. In this case, we saved the computational overhead of re-computing the symmetry group, and simply allowed MINTO to choose a branching variable. The CPU time was limited in all cases to four hours, and a limit of 1,000,000 nodes evaluated was imposed.

Table 3.2 shows the results of an experiment designed to compare the performance of the six different orbital branching rules introduced in Section 3.3.2. In this experiment, reduced cost fixing, orbital fixing, and the local symmetry group $\mathcal{G}^a$ were used. The CPU time required (in seconds) for orbital branching to solve each instance in the test suite for the six different is reported. A complete table showing the number of nodes, CPU time, CPU time computing automorphism groups, the number of variables fixed by reduced cost fixing, orbital fixing, strong branching fixing, and the deepest tree level at which orbital branching was performed for a variety of parameter settings is shown in Table 3.7 in the Appendix.

Instance	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6
cod83	11	4	5	6	8	5
cod93	1677	1557	2368	3269	242	399
cod105	239	238	345	255	424	229
cov954	5	4	24	8	17	5
cov1053	103	617	768	346	105	90
cov1054	14400	14400	14400	14400	181	14400
cov1075	69	50	216	14400	210	128
cov1076	14400	14400	14400	14400	1560	14400
f5	64	80	668	42	34	64
sts45	8	8	95	8	8	8
sts63	93	91	1132	1630	161	137
sts81	127	164	13465	3423	434	3371
Times Best	2	6	0	1	5	2

Table 3.2: CPU Time for Orbital Branching Using Local Symmetry Group

Figure 3.7: Performance Profile of Branching Rules

In order to succinctly present the computational results, the performance profiles of Dolan and Moré [14] are used. A performance profile is a relative measure of the effectiveness of one solution method in relation to a group of solution methods on a fixed set of problem instances. A performance profile for a solution method m is essentially a plot of the probability that the performance of m (measured in this case with CPU time) on a given instance in the test suite is within a factor of β of the *best* method for that instance. Methods whose corresponding profile lines are the highest are the most effective. Figure 3.7 shows a performance profile of the results of the first experiment, the CPU times in Table 3.2.

The most effective branching method is **Branching Rule 5**—the method that keeps the size of the symmetry group large on the left branch. (This method gives the "highest" line in Fig. 3.7). In fact, this branching method is the only one that is able to solve all of the instances in the test suite within the four hour time limit. This result is somewhat surprising. Anecdotally, symmetry has long been thought to be a significant hurdle for solving integer programs. One might expect that methods in which symmetry was *removed* as quickly as possible would have been the most effective. The computational results are counter to this intuition. Instead, if effective methods for *exploiting* problem symmetry (like those in orbital branching) are present, the results indicate that keeping a large amount of symmetry in the subproblems may be effective in some cases. As orbital branching does not exploit all symmetry available, it is important to determine if **Branching Rule 5** is effective because it moves the LP bounds quickly or because it is able to exploit more symmetry than other rules. This is discussed in Section 3.5.

A second experiment was aimed at measuring the impact of using the global symmetry group $\mathrm{stab}(F_1^a, \mathcal{G}(A, b, c))$

instead of the local symmetry group $\mathrm{stab}(F_1^a, \mathcal{G}(A, b, c))$, discussed in Section 3.3.1, when making a branching decision. Table 3.3 shows the CPU time (in seconds) that orbital branching, equipped with reduced cost fixing and orbital fixing, required on the instances in the test suite, for the different branching rules employing the global symmetry group.

Instance	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6
cod83	10	3	5	1	1	5
cod93	1677	1556	2361	166	167	396
cod105	237	237	359	234	242	237
cov954	5	4	23	13	6	5
cov1053	103	619	761	280	240	89
cov1054	14400	14400	14400	14400	179	14400
cov1075	55	42	202	14400	152	95
cov1076	14400	14400	14400	14400	1415	14400
f5	64	79	664	44	45	64
sts45	8	8	50	8	8	8
sts63	104	90	101	20	20	81
sts81	29	28	73	39	39	3383
Times Best	1	4	0	6	6	1

Table 3.3: CPU Time for Orbital Branching Using Global Symmetry Group

Again, **Branching Rule 5** was by far the most effective. A side-by-side comparison of Tables 3.2 and 3.3 indicates that, in general, using the global symmetry group is more effective than attempting to exploit symmetry that may only be locally present at a node. Figure 3.8 shows a performance profile comparing the CPU time required to solve the instances using **Branching Rule 5** with both the local and global symmetry groups. Surprisingly, the improved performance of the global symmetry group comes not only from the improved efficiency of the branching calculations, but in many cases, the number of *nodes* is reduced, as shown in Table 3.4. These computational results run counter to Theorem 3.4, which states that orbits from the global symmetry group are a subdivision of orbits from the local group. Since the orbits of the local group are no smaller, one would expect that orbital branching's enumeration tree would also be smaller in this case.

Figure 3.8: Performance Profile of Local versus Global Symmetry Groups

A third comparison worthy of note is the impact of performing orbital fixing, as introduced in Section 3.2.1. Using branching **Rule 5**, each instance in Table 3.1 was run both with and without orbital fixing. Figure 3.9 shows a performance profile comparing the results in the two cases. The results shows that orbital fixing has a *significant* positive impact.

Instance	Local Symmetry	Global Symmetry
cod83	195	25
cod93	1577	1361
cod105	23	11
cov954	449	249
cov1053	3139	9775
cov1054	1249	1249
cov1075	381	381
cov1076	31943	31943
f5	717	1125
sts45	4507	4709
sts63	9993	5533
sts81	83961	6293
Geo. Mean	1651	1081

Table 3.4: Number of Nodes in Orbital Branching Enumeration Tree with Different Symmetry Groups

Figure 3.9: Performance Profile of Impact of Orbital Fixing

The final comparison made in the chapter is between different branch and bound techniques for solving the symmetric test instances. Five different algorithms were compared: the isomorphism pruning algorithm of Margot, orbital branching (using branching **Rule 5** and the global symmetry group), MINTO's default algorithm, CPLEX v11.0 without symmetry handling, and CPLEX v11.0 with symmetry handling. (The symmetry handling was changed in CPLEX by setting the option `symmetry` to 0 or 5 respectively). As orbital branching is implemented in the MINTO framework, the MINTO default results demonstrate the direct impact of orbital branching on the symmetric test instances. Table 3.5 summarizes the results of the comparison. The results for isomorphism pruning are taken directly from Margot's paper using the most sophisticated of his branching rules "BC4" [55]. The paper does not report results on `f5`. The CPLEX results were obtained on an Intel Pentium 4 CPU clocked at 2.0GHz, as this was the only machine on which a CPLEX license was available. Since the results were obtained on three different computer architectures and each used a different LP solver for the child subproblems, the CPU times should be interpreted appropriately. The results clearly show that isomorphism pruning and orbital branching are the most effective methods for these symmetric instances.

3.5 Incomplete Symmetry Removal

This section focuses on how symmetry can remain when using orbital branching. Consider again the PILP, with symmetry group $\mathcal{G}$, associated with Figure 3.1, using the branching ordering shown in Figure 3.10.

At every given node, the symmetry group is computed twice, once to perform orbital fixing, and again to generate

	Isomorphism Pruning		**Orbital Branching**		**MINTO Default**		**CPLEX v11 w/o Sym**		**CPLEX v11 w/Sym**	
Instance	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
cod83	19	33	1	25	14400	346963	89	8351	100	9338
cod93	651	103	167	1361	14400	43305	14400	289928	14400	287998
cod105	2000	15	242	11	14400	4	3025	2188	2611	816
cov954	24	126	6	249	1180	39213	11	1266	11	1266
cov1053	35	111	240	9775	14400	207430	3508	262187	3556	262628
cov1054	130	108	179	1249	14400	32679	14400	101478	14400	94949
cov1075	118	169	152	381	14400	144064	868	20663	1170	21076
cov1076	3634	5121	1415	31943	14400	180147	14400	184600	14400	173900
f5	N/A	N/A	45	1125	14400	176611	1713	111505	1592	107816
sts45	31	513	8	4709	223	58683	18	32980	12	19931
sts63	120	1247	20	5533	14400	1475000	5410	4681239	5488	4805781
sts81	68	199	39	6293	14400	1557850	14400	13425146	14400	13361288

Table 3.5: Comparison of Different Solvers on Test Instances

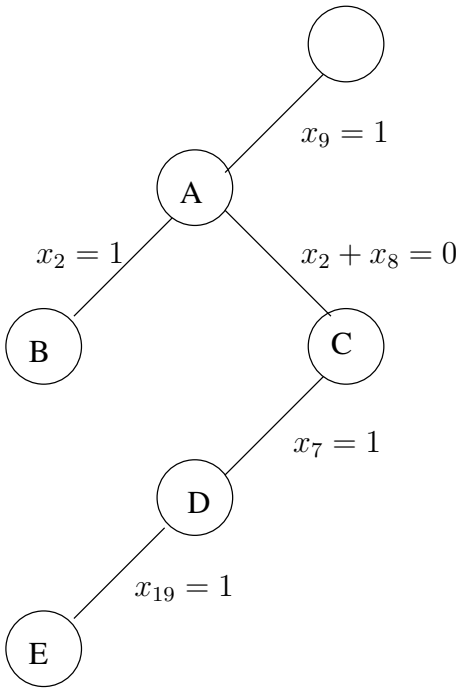


Figure 3.10: Subset of Enumeration Tree

the orbits used for branching. Thus, two graphs are created for every node and the symmetry group of each graph is computed. Figure 3.11 represents both graphs formed by fixing x_9 to 1. Solid vertices and edges in Figure 3.11 represent edges and vertices that appear in both graphs, while hashed edges and vertices are only in the graph used to compute $\mathcal{G}^{\hat{a}}$.

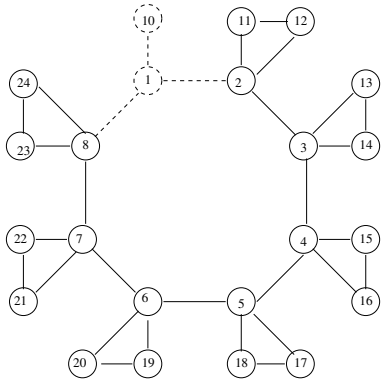


Figure 3.11: Graph of subproblem A

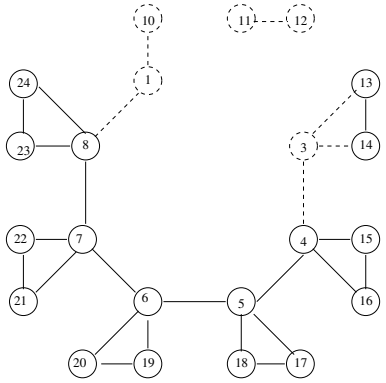


Figure 3.12: Graph of subproblem B

Figure 3.11 represents a case where $\mathrm{stab}(F_1^A, \mathcal{G}) = \mathcal{G}^{\hat{A}} = \mathcal{G}^{A^*}$. This will not be true in general. Node B, shown in Figure 3.12, is more interesting. While $\mathrm{stab}(F_1^B, \mathcal{G}) = \mathcal{G}^{\hat{B}}$, node B is a nontrivial case where $\mathcal{G}^{\hat{B}} \subset \mathcal{G}^{B^*}$. Note that $\mathcal{G}^{B^*}$ contains the reflection permutation $\pi_r = (4,8)(5,7)(15,23)(16,24)(17,21)(18,22)$ and no reflection appears in $\mathcal{G}^{\hat{B}}$. Also worth noting is that π_r is not a newly found symmetry. The permutation π_r is the projection of the permutation

$$(1,3)(4,8)(5,7)(9,13)(10,14)(15,23)(16,24)(17,21)(18,22)$$

found in $\mathcal{G}$.

At node C of Figure 3.13, $\mathrm{stab}(F_1^C, \mathcal{G}) = \mathcal{G}^{\hat{C}} = \mathcal{G}^{C^*}$. Also, no orbital fixing can be done. Node D of Figure 3.14, is another graph where $\mathcal{G}^{\hat{D}} \subset \mathcal{G}^{D^*}$. Here too, no orbital fixing can be done.

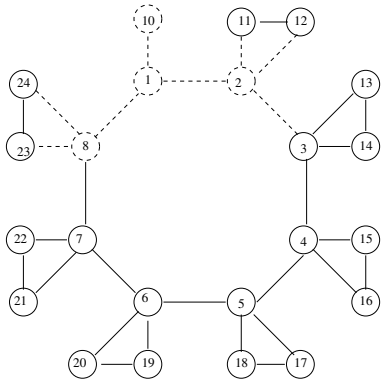


Figure 3.13: Graph of subproblem C

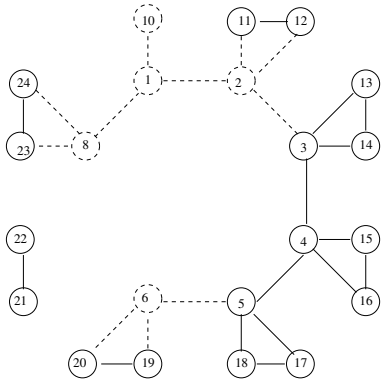


Figure 3.14: Graph of subproblem D

The graph shown in Figure 3.15 is isomorphic to a subgraph of Figure 3.12. This can be seen by rotating the graph 90 degrees clockwise. Any feasible solution in node E will be equivalent to a solution feasible at node B. The variable x_{19} could have been fixed to 0 at node D, but neither orbital branching nor orbital fixing allowed us to do this.

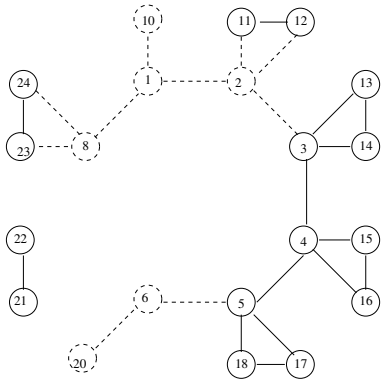


Figure 3.15: Graph of subproblem E

If SBDS (Section 2.4.3) was used instead of orbital branching, the set of constraints $\pi(x_9 + x_2) \leq 1$, for all $\pi \in \mathcal{G}$, would have been added to node C. This can be seen by noting that for any solution with $\pi(x_9 + x_2) = 2$, the permutation $\pi^{-1} \in \mathcal{G}$ maps that solution to a solution in B. Orbital branching implicitly enforces this constraint only for $\pi \in Stab(F_1^C, \mathcal{G})$, as these are the only constraints that can fix variables at the current node. When future variables are fixed to 1 (in this case x_7), orbital branching does not go back in the tree to check either for violated inequalities or for inequalities that may allow us to fix variables (in this case, the inequality $x_7 + x_{19} \leq 1$ at node D). Orbital fixing is a way to capture some of these forgotten inequalities, but orbital fixing's effectiveness can be very dependent on the ordering of the branching. If x_{19} had been chosen to branch on at node C instead of x_7, then orbital fixing would have fixed x_7 to 0, satisfying the inequality $x_7 + x_{19} \leq 1$. Observe that $\mathrm{stab}(\{9, 19\}, \mathcal{G})$ contains the horizontal reflection permutation π_h, and that $\pi_h(7) = 8$, so elements 7 and 8 would have shared the same orbit. Since $x_8 = 0$, orbital fixing would fix x_7 to 0. While changing the branching ordering may avoid problems associated with the particular permutation π_h, it may miss permutations that are handled easily by the original branching decision.

3.5.1 Symmetry Removal by Branching Rule

The branching strategies developed for orbital branching serve two purposes: to exploit symmetry and to move the LP bounds. In this section, the branching rules discussed in Section 3.3.2 are examined in order to determine how effective they are at exploiting symmetry.

In order to test the ability of a branching rule to exploit symmetry, all near-optimal solutions found during orbital branching are counted and the size of the set of solutions is compared to the size of the true set of non-isomorphic solutions. If a particular branching rule removes all symmetry, the collection of solutions generated will have a

cardinality equal to the number of all near optimal non-isomorphic solutions. If an algorithm fails to exploit symmetry, the size of the collection will be much larger. Table 3.5.1 shows the computational results.

Instance	k	Smallest	Largest	Keep Symmetry	Remove Symmetry	Actual
cod83	1	647	3037	376	*	14
cod105	0	2	2	2	2	2
sts27	0	4	7	4	7	1
sts45	1	716	1474	540	1637	37
sts63	1	628	4489	463	179	87
sts81	1	8	8	8	*	2

Table 3.6: Number of Solutions Generated Within k of Optimal

These results from Table 3.5.1 used a time limit of 2 hours. A "*" denotes instances where the branching method was unable to finish. The results indicate that both branching on the smallest orbit and branching to keep symmetry are more effective than branching on the largest orbit and branching to remove symmetry. These results can be supported by considering Theorem 3.2. This theorem states that for any two distinct nodes a and b, no solution feasible in a is isomorphic to a solution feasible in b with respect to the symmetry group of the pair of node's first common ancestor. The branching rule, "Keep Symmetry Left" tries to keep as much symmetry in the tree as possible. Ideally, the first common ancestor of any pair of nodes has a large symmetry group, making is less likely that the pair of nodes contains equivalent solutions.

Branching to keep symmetry is very similar to branching on the smallest orbit. Branching on an orbit of size k decreases the size of the child's symmetry group by at most a multiple of $\frac{1}{k}$. In practice, the symmetry group of the left child will not be much larger than $\frac{1}{k}$ times smaller than the parent's. Therefor, it is likely that branching on the smallest orbit also creates the smallest symmetry group in the left child. Unfortunately, intuition would indicate that branching on small orbits would lead to small changes in the LP bound, as there are fewer variables in the orbit to fix to zero in the right branch.

3.6 Comparison with Other Methods

3.6.1 Isomorphism Pruning

Isomorphism pruning, discussed in Chapter 1 and 4 is a powerful tool to exploit symmetry, however, it requires a rigid branching rule. Orbital branching does not suffer from this inflexibility. By not focusing on pruning *all* isomorphic nodes, but rather eliminating the symmetry through branching, orbital branching offers a great deal more flexibility in the choice of branching entity. Another advantage of orbital branching is that it allows for the use of symmetry group $\mathcal{G}^a$, symmetry *introduced* as a result of the branching process. While using the local group has not been shown to be effective in these studies, there may be classes of problems where symmetry tends to enter the tree, and in these cases,

using the local symmetry group may be more effective.

Both methods allow for the use of traditional integer programming methodologies such as cutting planes and fixing variables based on considerations such as reduced costs and implications derived from preprocessing. In isomorphism pruning, for a variable fixing to be valid, it must be that *all* non-isomorphic optimal solutions are in agreement with the fixing. Orbital branching does not suffer from this limitation. A powerful idea in both methods is to combine the variable fixing with symmetry considerations in order to fix many additional variables. This idea is called *orbit setting* in [55] and *orbital fixing* in this work (see Sec. 3.2.1).

3.6.2 Symmetry Breaking Inequalities

Throughout this thesis we have been concerned with removing variables by fixing variables. This can significantly reduce the number of feasible solutions in a given subproblem that are isomorphic to solutions found elsewhere (to the left) in the tree. As mentioned before, this fixing does not guarantee that all such isomorphic solutions will be eliminated. Orbital branching, as well as isomorphism pruning, only concerns itself with variables that, if fixed to a certain value at a subproblem, will cause all corresponding feasible solutions to be isomorphic.

Neither orbital branching nor isomorphism pruning attempts to actually remove all isomorphic solutions that are feasible at a given subproblem. This is a much more difficult task. Doing so would guarantee no two isomorphic nodes were processed and may actually further decrease the number of nodes needed to solve the problem. It has been observed many times, and in fact it is the basis of SBDS (see section 2.4.3), that if we branch on variable x_i at node a we can add the constraint

$$\sum_{k \in \pi(F_1^a \cup i)} x_k \leq |F_1^a|$$

for every $\pi \in \mathcal{G}^r$ to the right branch. It should be clear that this is a valid branching rule and does in fact remove all feasible integer solutions that are symmetric to solutions to the left of the node from the feasible region. It is also possible to explain orbital branching in terms of this method of symmetry removal.

For instance, suppose variable x_i is chosen to branch on at node a. In the right node $x_i = 0$. Consider any variable x_j that shares an orbit with x_i at a. We know that there is a permutation $\pi \in \mathrm{stab}(F_1^a,\ \mathcal{G}^r)$ that sends i to j, so, for this π the constraint $\sum_{\{k \in \pi(F_1^a \cup i)\}} x_k \leq |F_1^a|$ becomes $\sum_{\{k \in F_1^a \cup j\}} x_k = |F_1^a| + x_j \leq |F_1^a|$, so we have that $x_j = 0$ for every j sharing an orbit with i.

By explaining orbital branching in terms of this method, our proof for the validity of the global version of orbital fixing becomes more clear. Recall that if there exists a mixed-zero-free orbit O_j in $\mathrm{stab}(F_1^a, \mathcal{G}^r)$, then we can fix all variables in O_j to zero. This can be more easily seen by considering the variable x_i in O_i that was fixed to zero first in the enumeration tree, at node b, by the constraint $\sum_{i \in \{F_1^b \cup i\}} x_i \leq |F_1^b|$. For any free variable j in O_i there is a permutation $\pi \in Stab(F_1^a, \mathcal{G}^r)$ sending i to j. We have then that $\sum_{i \in \pi(F_1^b)} x_i = |F_1^b|$ because $F_1^b \subset F_1^a$, and

$\pi(F_1^b) \subset F_1^a$ by π being a stabilizer. That means that the constraint $\sum_{i \in \pi(F_1^a \cup i)} x_i \leq |F_1^a|$ fixes x_j to zero.

Not only does this description of orbital branching give more descriptive proofs to some of the fundamental theorems in this chapter, it also describes orbital branching in such a way that would be easily comparable to SBDS. By branching on orbits and performing orbit setting, orbital branching is able to take advantage of many of the inequalities that are introduced during SBDS without the overhead required by SBDS.

3.7 Summary

In this chapter, we presented a simple way to capture and exploit the symmetry of an integer program when branching. We showed, through a set of experiments, that orbital branching outperforms CPLEX, a state-of-the-art solver, when a high degree of symmetry is present. Orbital branching also seems to be of comparable quality to the isomorphism pruning method of Margot [55] and we will discuss a powerful combination of the two methods in Chapter 4. Further, we feel that the simplicity and flexibility of orbital branching makes it an attractive candidate for further study. In Chapter 5 we discuss ways to extend the orbital branching method on more general types of branching disjunctions.

Instance	Branching Rule	Time	Nodes	Nauty Calls	Nauty Time	# Fixed by RCF	# Fixed by OF	#Fixed by SBF	Deepest Orbital Level
cod105	Break Symmetry	254.5	17	7	15.0	0	1020	0	7
cod105	Keep Symmetry	423.9	23	10	21.6	216	1228	0	8
cod105	Branch Largest LP	237.9	7	2	4.2	0	0	0	2
cod105	Branch Largest	239.2	9	3	6.4	0	0	0	3
cod105	Max Prod. Orbit Size	229.5	9	3	6.1	1	960	0	3
cod105	Strong Branch	344.7	7	2	4.2	0	1024	1532	2
cod83	Break Symmetry	6.2	143	41	1.3	325	548	0	15
cod83	Keep Symmetry	8.2	195	73	2.3	251	942	0	18
cod83	Branch Largest LP	3.6	57	18	0.5	328	864	0	7
cod83	Branch Largest	10.6	193	14	0.4	233	588	0	7
cod83	Max Prod. Orbit Size	4.8	105	19	0.6	69	642	0	11
cod83	Strong Branch	5.3	21	9	0.4	16	762	412	6
cod93	Break Symmetry	3268.8	37297	557	58.4	106725	6202	0	26
cod93	Keep Symmetry	242.5	1577	303	32.9	11473	2422	0	44
cod93	Branch Largest LP	1557.1	14461	13	1.9	201292	348	0	7
cod93	Branch Largest	1677.3	16439	15	2.2	205636	1060	0	7
cod93	Max Prod. Orbit Size	398.9	3503	59	6.8	41907	704	0	25
cod93	Strong Branch	2367.9	161	79	8.2	437	2400	13478	15
cov1053	Break Symmetry	345.8	15321	800	28.7	0	2418	0	35
cov1053	Keep Symmetry	105.4	3139	520	18.7	0	1696	0	31
cov1053	Branch Largest LP	616.7	20725	61	2.1	0	988	0	19
cov1053	Branch Largest	103.5	3437	55	1.9	0	1094	0	17
cov1053	Max Prod. Orbit Size	90.2	2859	71	2.5	0	1466	0	20
cov1053	Strong Branch	768.4	777	387	14.1	0	2834	16462	43
cov1054	Break Symmetry	14400.0	110116	1	0.2	0	0	0	0
cov1054	Keep Symmetry	181.3	1249	103	18.5	0	454	0	15
cov1054	Branch Largest LP	14400.0	104126	6	1.1	56	88	0	5
cov1054	Branch Largest	14400.0	105500	10	1.7	0	0	0	7
cov1054	Max Prod. Orbit Size	14400.0	104172	12	2.0	0	176	0	8
cov1054	Strong Branch	14400.0	846	430	79.3	0	220	12846	57
cov1075	Break Symmetry	14400.0	408822	1	0.8	862268	0	0	0
cov1075	Keep Symmetry	209.7	381	181	189.8	413	962	0	15
cov1075	Branch Largest LP	49.8	495	22	23.3	1400	520	0	9
cov1075	Branch Largest	68.6	461	43	44.3	1333	900	0	13
cov1075	Max Prod. Orbit Size	128.4	543	98	102.0	1028	1090	0	21
cov1075	Strong Branch	215.5	71	34	37.4	126	92	1858	10
cov1076	Break Symmetry	14400.0	496533	1	0.7	720913	0	0	0
cov1076	Keep Symmetry	1559.9	31943	786	657.0	21902	960	0	20
cov1076	Branch Largest LP	14400.0	498573	20	15.8	631691	222	0	7
cov1076	Branch Largest	14400.0	504396	40	34.0	495631	388	0	9
cov1076	Max Prod. Orbit Size	14400.0	498258	133	110.2	638795	532	0	18
cov1076	Strong Branch	14400.0	4989	2498	2327.4	2798	1256	71682	27
cov954	Break Symmetry	8.4	237	69	4.4	423	272	0	11
cov954	Keep Symmetry	17.3	449	170	11.0	677	948	0	15
cov954	Branch Largest LP	3.8	153	11	0.7	638	0	0	6
cov954	Branch Largest	5.3	249	20	1.2	818	304	0	12
cov954	Max Prod. Orbit Size	4.8	217	18	1.1	699	132	0	11
cov954	Strong Branch	24.0	63	30	1.9	65	160	1724	11
f5	Break Symmetry	42.5	995	117	2.5	3515	1356	0	14
f5	Keep Symmetry	34.5	717	78	1.5	2102	598	0	14
f5	Branch Largest LP	79.8	2573	31	0.6	7660	252	0	8
f5	Branch Largest	64.1	1829	35	0.6	9710	430	0	11
f5	Max Prod. Orbit Size	64.3	1835	38	0.7	9678	418	0	13
f5	Strong Branch	668.2	123	60	1.1	169	736	8610	15
sts45	Break Symmetry	7.6	4571	11	0.7	1	0	0	4
sts45	Keep Symmetry	8.1	4507	16	1.3	2	0	0	6
sts45	Branch Largest LP	7.8	4683	6	0.6	3	0	0	3
sts45	Branch Largest	8.1	4917	4	0.4	1	0	0	2
sts45	Max Prod. Orbit Size	8.1	4917	4	0.4	1	0	0	2
sts45	Strong Branch	94.5	1417	707	43.0	0	0	7984	16
sts63	Break Symmetry	1630.3	666623	10	6.9	720	126	0	43
sts63	Keep Symmetry	160.8	9993	155	135.7	12	0	0	11
sts63	Branch Largest LP	91.4	32627	17	12.6	7	0	0	9
sts63	Branch Largest	92.7	33785	15	9.1	19	0	0	7
sts63	Max Prod. Orbit Size	136.8	31261	73	57.3	48	0	0	10
sts63	Strong Branch	1132.1	3157	1577	913.6	0	0	16858	24
sts81	Break Symmetry	3422.7	1000000	5	2.4	235	0	0	4
sts81	Keep Symmetry	434.1	83961	38	128.0	8	0	0	15
sts81	Branch Largest LP	164.0	25739	20	68.7	5	0	0	13
sts81	Branch Largest	127.0	11323	28	84.6	0	0	0	13
sts81	Max Prod. Orbit Size	3370.8	1000000	1	0.1	200	0	0	0
sts81	Strong Branch	13465.4	11291	5644	12074.9	1	0	62098	30

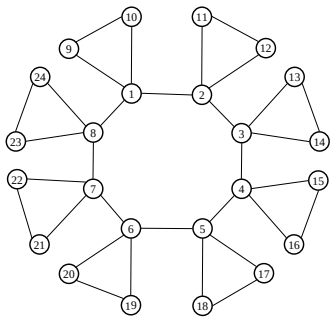
Table 3.7: Performance of Orbital Branching Rules (Local Symmetry) on Symmetric ILPs

Instance	Branching Rule	Time	Nodes	Nauty Calls	Nauty Time	# Fixed by RCF	# Fixed by OF	#Fixed by SBF	Deepest Orbital Level
cod105	Break Symmetry	234.1	11	4	7.1	0	1020	0	4
cod105	Keep Symmetry	242.5	11	4	7.1	0	1020	0	4
cod105	Branch Largest LP	237.2	7	2	3.6	0	0	0	2
cod105	Branch Largest	237.3	9	3	5.5	0	0	0	3
cod105	Max Prod. Orbit Size	237.5	9	3	5.3	1	960	0	3
cod105	Strong Branch	359.1	7	2	3.6	0	1024	1532	2
cod83	Break Symmetry	1.3	25	11	0.3	37	906	0	7
cod83	Keep Symmetry	1.3	25	11	0.3	37	906	0	7
cod83	Branch Largest LP	3.4	57	18	0.4	328	864	0	7
cod83	Branch Largest	10.4	193	14	0.3	233	588	0	7
cod83	Max Prod. Orbit Size	4.6	105	19	0.4	69	642	0	11
cod83	Strong Branch	5.2	21	9	0.3	16	762	412	6
cod93	Break Symmetry	165.7	1361	80	8.4	7397	3378	0	14
cod93	Keep Symmetry	167.2	1361	80	8.4	7397	3378	0	14
cod93	Branch Largest LP	1555.7	14461	13	1.5	201292	348	0	7
cod93	Branch Largest	1677.2	16439	15	1.7	205636	1060	0	7
cod93	Max Prod. Orbit Size	395.7	3503	59	4.1	41907	704	0	25
cod93	Strong Branch	2361.0	161	79	3.8	437	2400	13478	15
cov1053	Break Symmetry	280.5	11271	1276	23.7	0	3454	0	33
cov1053	Keep Symmetry	240.2	9775	248	4.7	0	724	0	25
cov1053	Branch Largest LP	619.3	20903	56	1.1	0	988	0	19
cov1053	Branch Largest	102.6	3437	55	1.2	0	1094	0	17
cov1053	Max Prod. Orbit Size	89.2	2859	71	1.6	0	1466	0	20
cov1053	Strong Branch	761.0	777	387	7.6	0	2830	16464	43
cov1054	Break Symmetry	14400.0	110307	1	0.2	0	0	0	0
cov1054	Keep Symmetry	178.8	1249	103	15.2	0	454	0	15
cov1054	Branch Largest LP	14400.0	104161	6	0.9	56	88	0	5
cov1054	Branch Largest	14400.0	105846	10	1.4	0	0	0	7
cov1054	Max Prod. Orbit Size	14400.0	104184	12	1.7	0	176	0	8
cov1054	Strong Branch	14400.0	846	430	52.7	0	220	12846	57
cov1075	Break Symmetry	14400.0	410572	1	0.8	865517	0	0	0
cov1075	Keep Symmetry	152.2	381	181	133.0	413	962	0	15
cov1075	Branch Largest LP	41.9	495	22	15.7	1400	520	0	9
cov1075	Branch Largest	54.6	461	43	30.6	1333	900	0	13
cov1075	Max Prod. Orbit Size	95.2	543	98	69.2	1028	1090	0	21
cov1075	Strong Branch	201.6	71	34	23.9	126	92	1858	10
cov1076	Break Symmetry	14400.0	495919	1	0.7	719961	0	0	0
cov1076	Keep Symmetry	1415.0	31943	786	516.2	21902	960	0	20
cov1076	Branch Largest LP	14400.0	496393	20	13.1	628579	222	0	7
cov1076	Branch Largest	14400.0	504849	40	26.2	496164	388	0	9
cov1076	Max Prod. Orbit Size	14400.0	497593	133	86.5	637905	532	0	18
cov1076	Strong Branch	14400.0	5280	2642	1692.9	2971	1288	76298	27
cov954	Break Symmetry	12.7	373	150	7.1	632	524	0	13
cov954	Keep Symmetry	6.2	249	42	1.9	748	48	0	11
cov954	Branch Largest LP	3.6	153	11	0.5	638	0	0	6
cov954	Branch Largest	5.0	249	20	0.9	818	304	0	12
cov954	Max Prod. Orbit Size	4.6	217	18	0.8	699	132	0	11
cov954	Strong Branch	23.5	63	30	1.3	65	160	1724	11
f5	Break Symmetry	44.5	1125	292	4.6	2983	2994	0	17
f5	Keep Symmetry	44.6	1125	292	4.6	2983	2994	0	17
f5	Branch Largest LP	79.5	2573	31	0.4	7660	252	0	8
f5	Branch Largest	63.8	1829	35	0.4	9710	430	0	11
f5	Max Prod. Orbit Size	63.9	1835	38	0.5	9678	418	0	13
f5	Strong Branch	664.4	123	60	0.4	169	736	8610	15
sts45	Break Symmetry	7.9	4709	16	0.8	0	0	0	6
sts45	Keep Symmetry	7.8	4709	16	0.7	0	0	0	6
sts45	Branch Largest LP	7.8	4683	6	0.5	3	0	0	3
sts45	Branch Largest	8.1	4917	4	0.4	1	0	0	2
sts45	Max Prod. Orbit Size	8.1	4917	4	0.4	1	0	0	2
sts45	Strong Branch	49.9	1287	642	3.2	0	148	7150	16
sts63	Break Symmetry	20.1	5533	93	5.5	1	308	0	11
sts63	Keep Symmetry	20.1	5533	93	5.6	1	308	0	11
sts63	Branch Largest LP	90.1	36579	16	1.7	19	32	0	9
sts63	Branch Largest	103.8	43349	14	1.7	17	32	0	7
sts63	Max Prod. Orbit Size	81.1	30133	53	4.6	47	176	0	8
sts63	Strong Branch	101.4	1377	687	10.5	0	676	6710	24
sts81	Break Symmetry	39.1	6293	112	14.3	0	670	0	17
sts81	Keep Symmetry	38.9	6293	112	14.3	0	670	0	17
sts81	Branch Largest LP	27.9	5649	41	6.0	0	562	0	14
sts81	Branch Largest	28.9	5823	46	5.7	0	410	0	14
sts81	Max Prod. Orbit Size	3382.5	1000000	1	0.1	200	0	0	0
sts81	Strong Branch	73.5	573	285	19.8	0	1112	2514	22

Table 3.8: Performance of Orbital Branching Rules (Global Symmetry) on Symmetric ILPs

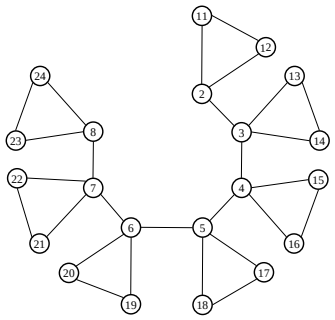
Figure 3.16: Example 3.1.3: Structure of Subproblems and Orbits in Orbital Branching.

1



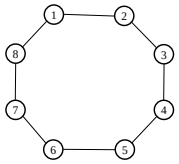
$\{1, 2, 3, 4, 5, 6, 7, 8\}$
$\{9, 10, 11, 12, 13, 14, 15, 16, 17,$
$18, 19, 20, 21, 22, 23, 24\}$

2



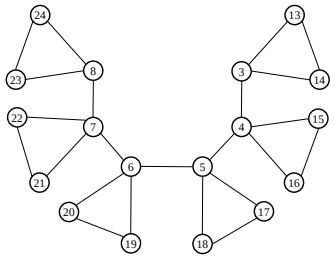
$\{2, 8\}\{3, 7\}\{4, 6\}\{5\}$
$\{11, 12, 23, 24\}\{13, 14, 21, 22\}$
$\{15, 16, 19, 20\}\{17, 18\}$

3



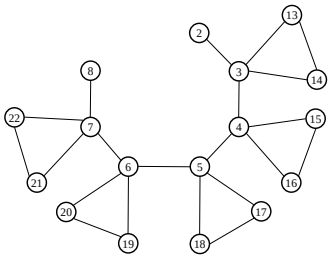
$\{1, 2, 3, 4, 5, 6, 7, 8\}$

4



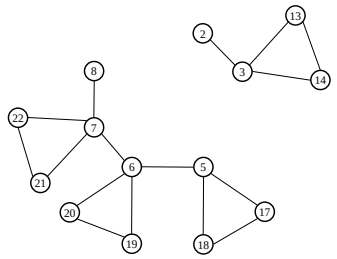
$\{3, 8\}\{4, 7\}$ $\{5, 6\}$
$\{13, 14, 23, 24\}$
$\{15, 16, 21, 22\}\{17, 18, 19, 20\}$

5



$\{2, 8\}\{3, 7\}\{4, 6\}\{5\}$
$\{13, 14, 21, 22\}\{15, 16, 19, 20\}$
$\{17, 18\}$

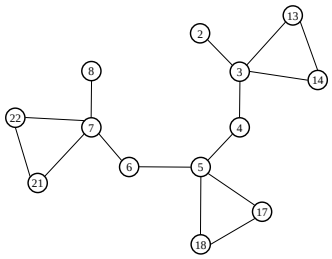
6



$\{2\}\{3\}\{5\}\{8\}\{6,7\}$

$\{13,14\}\{17,18\}\{19,20,21,22\}$

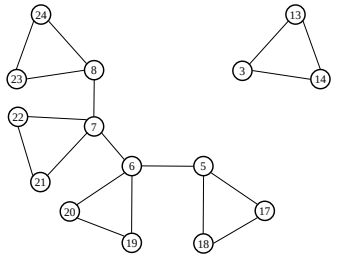
7



$\{2,8\}\{3,7\}\{4,6\}\{5\}$

$\{13,14,21,22\}\{17,18\}$

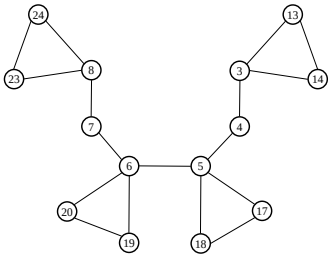
8



$\{3,13,14\}\{5,8\}\{6,7\}$

$\{17,18,23,24\}\{19,20,21,22\}$

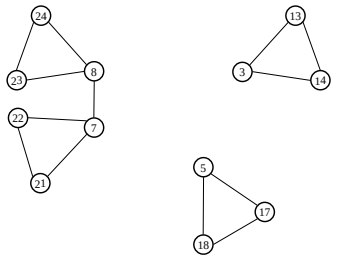
9



$\{3,8\}\{4,7\}\ \{5,6\}$

{13,14,23,24}{17,18,19,20}

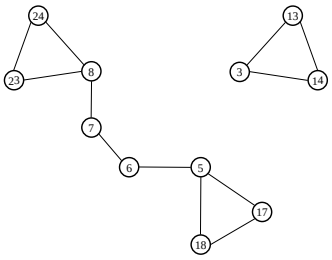
10



$\{3,13,14,5,17,18\}\{7,8\}$

$\{21,22,23,24\}$

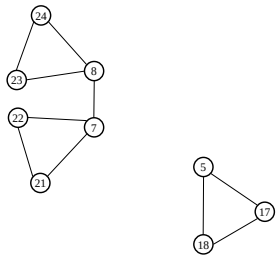
11



$\{3,13,14\}\{5,8\}\{6,7\}$

{17,18,23,24}

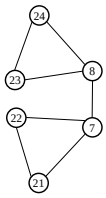
12



$\{5, 17, 18\}\{7, 8\}$

$\{21, 22, 23, 24\}$

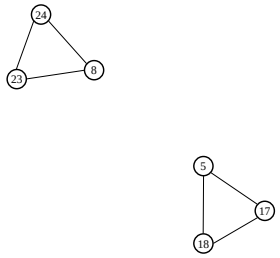
13



$\{7, 8\}$

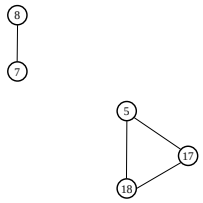
$\{21, 22, 23, 24\}$

14



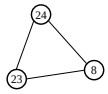
$\{5, 17, 18, 23, 24\}$

15



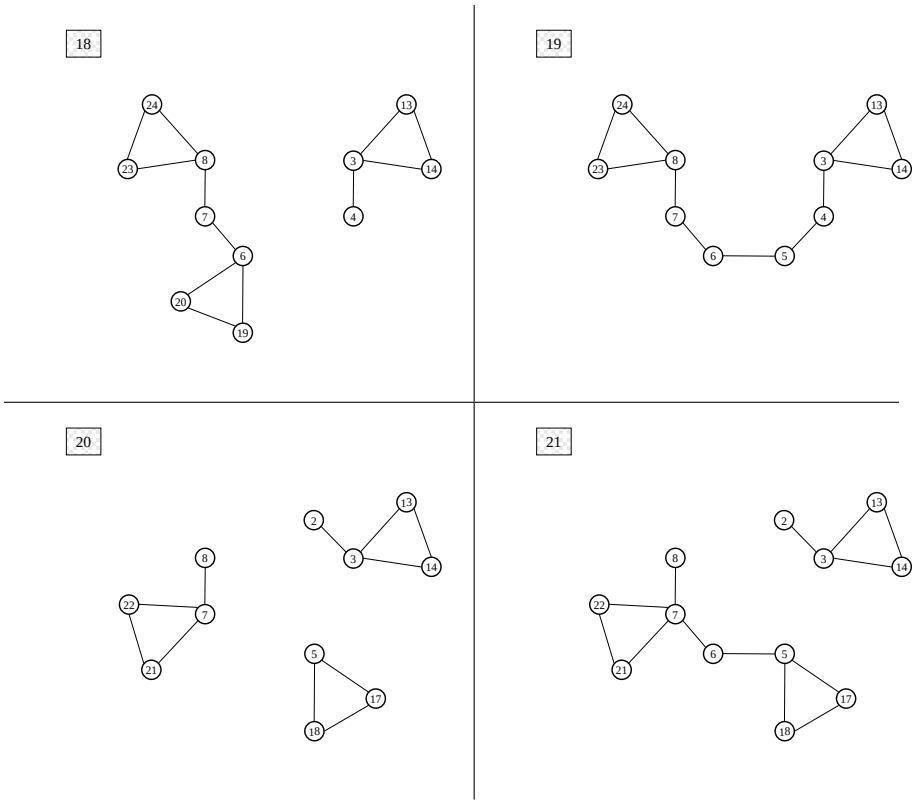
$\{5, 17, 18\}\{7, 8\}$

16



$\{8, 23, 24\}$

17


18
24
23
8
7
6
20
19
13
3
14
4
19
24
23
8
7
6
5
4
3
13
14
20
8
22
7
21
2
13
3
14
5
17
18
21
2
13
3
14
8
22
7
21
6
5
17
18

Chapter 4

Flexible Isomorphism Pruning

Orbital branching, introduced in Chapter 3, is an easily-implemented method of mitigating many of the negative effects of symmetry. However, orbital branching does not completely remove the effects of symmetry. Specifically, the feasible region searched by a branch-and-bound algorithm using orbital branching may not be a minimal fundamental domain of the integer linear program. Isomorphism pruning, developed for ILP by François Margot [54] [55], is a method that reduces the feasible region to a minimal fundamental domain for a general ILP instance. This fundamental domain is based on lexicographic ordering. Isomorphism pruning is able to avoid numerical issues caused by including lexicographic constraints by providing a clever way to search for violated constraints. In this chapter, we present isomorphism pruning in a way that can be combined with orbital branching.

Restricting the feasible region to a minimal fundamental domain is important for many reasons. Most obviously, a smaller feasible region often leads to smaller branch-and-bound trees. Furthermore, if the fundamental domain is minimal, then no isomorphic subproblems are evaluated. This minimality guarantee is especially important if one wishes to generate all non-isomorphic optimal solutions to an instance. This is a significant advantage over orbital branching. To ensure that the set of solutions generated in orbital branching is non-isomorphic, each solution generated must be compared to all other solutions to check for isomorphism.

Isomorphism pruning can be a very powerful tool for solving symmetric integer programs, but it suffers from a lack of flexibility in the choice of branching variable. Isomorphism pruning branching only on individual variables while orbital branching branches orbits, allowing for the fixing of additional variables. This fact might seem like a considerable drawback for isomorphism pruning. However, fixing algorithms performed at individual nodes such as orbital fixing (setting) are able to easily fix all variables that would have been fixed by the orbital branching disjunction at the child node. A major difference, then, between orbital branching and isomorphism pruning is that the information provided by fixing variables via symmetry considerations is not used to select the variable on which to branch. Even

if such information was available, due to the branching restrictions of isomorphism pruning, the information would be used only infrequently.

Section 4.1 demonstrates how to remove the branching restrictions for isomorphism pruning. In general, branching strategies can drastically affect the solution times for integer programs, so the hope is that a similar improvement can be achieved by relaxing strict branching requirements for isomorphism pruning. In addition to detailing the updated isomorphism pruning algorithm, Section 4.3 examines different branching rules that can be used in conjunction with isomorphism pruning in order to best exploit information provided by symmetry. Section 4.1 shows a basic algorithm that tests for violated lexicographic inequalities. Section 4.2 discusses implementation issues and presents an algorithm similar to orbital fixing that fixes variable by using information provided by symmetry. Section 4.3 gives computational results.

4.1 Isomorphism Pruning

The basic concepts of isomorphism pruning were introduced in Section 2.4.1. To summarize, the idea of isomorphism pruning is to restrict the feasible region of an ILP to a minimal fundamental domain based on a lexicographic ordering enforced by the linear inequalities

$$\sum_{i=1}^{n} 2^{n-i} x_i \geq \sum_{i=1}^{n} 2^{n-i} \pi(x_i) \; \forall \; \pi \in \mathcal{G}(ILP). \tag{4.1}$$

A key theorem in [54] offers a way to test for violated inequalities if branching variables are chosen by the minimum indexed branching (MIB) rule. MIB is a branching rule that always chooses the free variable with the smallest index for branching. Given the MIB branching rule, for any node $a = (F_1^a, F_0^a)$ in the enumeration tree, if there is a $\pi \in \mathcal{G}(ILP)$ with $\sum_{i \in F_1^a} 2^{n-i} \pi(x_i) > \sum_{i \in (F_1^a)} 2^{n-i} x_i$, then at least one constraint of (4.1) is violated. In [55], Margot provides a new, slightly more flexible branching rule called the ranked branching rule. Inequalities defining the minimal fundamental domain, as well as tests for violated inequalities were also updated to accommodate this new branching rule. Before this work can be described, some notation must be introduced.

4.1.1 The Rank and Lexicographic Ordering

Isomorphism pruning requires a total order on the elements of $\{0,1\}^n$ using a relation $\preceq$. Let $\mathcal{G}(ILP)$ be the symmetry group of the ILP. If $\preceq$ induces a total order, then for each $x \in \{0,1\}^n$, one and only one element $y \in \mathrm{orb}(x, \mathcal{G}(ILP))$ satisfies the inequalities $y \preceq \pi(x)$ for all $\pi \in \mathcal{G}(ILP)$. This set of inequalities $x \preceq \pi(x)$ for all $\pi \in \mathcal{G}$ defines a minimal fundamental domain of $\{0,1\}^n$ with respect to $\mathcal{G}(ILP)$. In the work Margot [54] $\preceq$ is the relation defining a lexicographic ordering. The ranked branching rule is based on using a different relation to define the total order.

A fundamental domain (but not necessarily minimal) can be defined by using a quasi-order on $\{0,1\}^n$. For any function $R : \{1,\ldots,n\} \to \mathbb{N}$, the relation $x \lesssim_R y$ holds if and only if $\sum_i 2^{n-R(i)}x_i \geq \sum_i 2^{n-R(i)}y_i$. Because $\lesssim_R$ produces a quasi-order on the set $\{0,1\}^n$, the constraints

$$x \lesssim_R \pi(x)\ \forall \pi \in \mathcal{G} \tag{4.2}$$

define a fundamental domain because every orbit has at least one element that satisfies (4.2). The fundamental domain is not necessarily minimal on $\{0,1\}^n$ with respect to $\mathcal{G}$. Note that because $\lesssim$ is a quasi-order, there may be $x \in \{0,1\}^n$ and $y \in \mathrm{orb}(x,\mathcal{G})$, $x \neq y$, with $x \lesssim y$ and $y \lesssim x$. In this case, both x and y are isomorphic solutions that may satisfy inequalities (4.2). However, if $\lesssim_R$ is total order on $\{0,1\}^n$, then the constraints (4.2) produce a minimal fundamental domain. In fact, as long as $\lesssim_R$ defines a total order on the set $\mathrm{orb}(x,\mathcal{G})$ for every $x \in \{0,1\}^n$, the fundamental domain is minimal.

Throughout this chapter a function $R : \{1,\ldots,n\} \to \mathbb{R}$ will be referred to as a *rank*. R is the *complete rank* if R is an invertible function mapping elements of the set $\{1,\ldots,\ n\}$ to $\{1,\ldots,\ n\}$. If R is a complete rank, $\lesssim_R$ induces a total order on $\{0,1\}^n$, so to distinguish $\lesssim_R$ from a quasi-order, $\lesssim_R$ will be written as $\preceq_R$. If R is the identity function, we write the ordering $\preceq_R$ as $\preceq_e$ and call it the *standard ordering*.

The relation $\lesssim_R$ acts on sets $F \subseteq \{1,\ldots,n\}$ and $G \subseteq \{1,\ldots,n\}$ in the natural way: $F \lesssim_R G$ holds if and only if $\sum_{i\in F} 2^{n-R(i)} \geq \sum_{i\in G} 2^{n-R(i)}$. A set $G = \sigma(F)$ for some $\sigma \in \mathcal{G}$ with

$$G \lesssim_R \pi(F)\ \forall \pi \in \mathcal{G}$$

is the *smallest-image* of F with respect to $\lesssim_R$ and $\mathcal{G}$. If G is the smallest-image of F with respect to $\lesssim_R$ and $\mathcal{G}$, then G is also its own smallest-image. A rank function R acts on a set $F \subseteq \{1,\ldots,n\}$ by:

$$R(F) = \{R(i)|\ \forall i \in F\}$$

Theorem 4.1 *$\pi(F) \lesssim_R F$ for some $\pi \in \mathcal{G}$ for some rank function R if and only if $R(\pi(F)) \preceq_e R(F)$.*

Proof: $\pi(F) \lesssim_R F \Leftrightarrow \sum_{i\in\pi(F)} 2^{n-R(i)} > \sum_{i\in F} 2^{n-R(i)} \Leftrightarrow \sum_{i\in R[\pi(F)]} 2^{n-i} > \sum_{i\in R[F]} 2^{n-i} \Leftrightarrow R(\pi(F)) \preceq_e R(F)$. □

4.1.2 The Rank and Isomorphism Pruning

Margot is able to provide some flexibility in the branching decision of isomorphism pruning by restricting the feasible region to the minimal fundamental domain using a *ranked branching rule*. The ranked branching rule is a method for dynamically creating a complete rank (inducing a total order) using the branching decisions. Let $a = (F_1^a, F_0^a)$, with

$|F_1^a| + |F_0^a| = d$ be the deepest node currently explored in the search tree. Further, let $\mathcal{S}(d) = \{i_1,\ i_2,\ \ldots,\ i_d\}$ be the (ordered) indices of the variables fixed by branching decisions. The rank, M^d for Margot's ranked branching rule is

$$M^d(i_j) = j \text{ for } j = 1, \ldots, d$$

$$M^d(k) = n + 1 \text{ for } k \notin \mathcal{S}(d).$$

The ranked branching rule requires that the variable with the lowest rank is chosen to branch on. If node a, of depth d, is not the deepest node in the current tree, then there is a free variable x_i with $M(i) = d$. In this case, x_i is chosen for branching. If a is the deepest node in the tree, then there are no such elements of rank d. In this case, any of the free variables (all having rank $n + 1$) may be chosen for branching.

Example Figure 4.1 shows a branch-and-bound tree generated by the ranked branching rule for a problem with $n = 6$ variables. The rank M^2, defined by the nodes up to depth 2 in the tree, is:

$$M^2(6) = 1$$

$$M^2(3) = 2$$

$$M^2(i) = 7\ \forall i \in \{1,\ 2,\ 4,\ 5\}.$$

Because node M was the first node of depth 4 to branch, all nodes of depth 4, L, N, O, and P, must branch on x_4. The rank M^6 is

$$M^2(6) = 1$$

$$M^2(3) = 2$$

$$M^2(5) = 3$$

$$M^2(1) = 4$$

$$M^2(4) = 5$$

$$M^2(2) = 6$$

The total order defined by the relation $\preceq_{M^n}$ is only known when a node in the tree of depth n has been processed. While we are solving a subproblem a with depth d we may not know the constraints

$$x \preceq_{M^n} \pi(x)\ \forall \pi \in \mathcal{G}(ILP) \tag{4.3}$$

explicitly because we do not yet know M^n. However, if at least one of the constraints generated by the quasi-order defined by M^d,

$$x \lesssim_{M^d} \pi(x)\ \forall \pi \in \mathcal{G}(ILP), \tag{4.4}$$

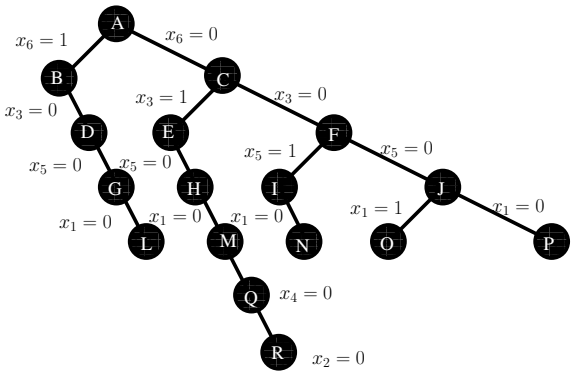


Figure 4.1: Ranked Branching Rule

is violated, then there must be a constraint in (4.3) that is violated. Theorem 4.2 is a key theorem of Margot [55].

Theorem 4.2 *If there is a node a of depth d for which at least one of the constraints* (4.4) *is violated, then at least one inequality* (4.3) *is violated.*

The ranked branching rule restricts the feasible region to the fundamental domain defined by constraints (4.3). As a result of Theorem 4.2, node a can be pruned if one of the constraints (4.4) is violated. Note that isomorphism pruning with the minimum index branching rule is a special case of isomorphism pruning with the ranked branching rule; specifically, it is the case in which the rank function is the identity. The inequalities in (4.3) and (4.4) are called *rank inequalities* generated by M^n and M^d, respectively.

4.1.3 Relaxing Depth-Dependent Rank

The minimal fundamental domain generated using the ranked branching rule on a complete rank function R can also be described as follows. Let $\mathcal{T}(R)$ be a graph representing the branch-and-bound tree generated by the branching decisions implied by R, where nodes are only pruned if the LP relaxation is infeasible (not including the constraints (4.3)), and not by bound. Each node at depth n in $\mathcal{T}(R)$ has all variables fixed and therefore corresponds to a feasible integral solution of the ILP. The tree is ordered by letting the branch $x_i = 1$ be the left branch. The fundamental domain defined by R consists only of the leftmost element of each collection of equivalent feasible solutions. This is an important property of the tree that can be exploited in order to remove the restrictions of minimum indexed branching and the ranked branching rule.

The feasible region of any node in the tree may contain equivalent solutions. Many of the equivalent solutions may satisfy constraints generated by the rank function M^d. Via branching or inequalities, one cannot expect to remove all the symmetry in the current node, but the implicit constraints added by branching (by updating the rank from M^d to M^{d+1}) remove symmetry between the child nodes. This is formalized in Theorem 4.3.

Theorem 4.3 *Let x and y be feasible at node a with $y \in \mathrm{orb}(x, \mathcal{G}(ILP))$. If x is feasible in the left child of a and y is feasible in the right child, then y will violate a rank inequality generated by M^{d+1}. In fact, this is true for all ranks M^k with $k \geq d+1$.*

Proof

$$\sum_{i=1}^{n} 2^{n-M^k[i]} x_i \geq \sum_{i \in F_1^a} 2^{n-M^k[i]} + 2^{n-(d+1)} \geq \sum_{i \in F_1^a} 2^{n-M^k[i]} + \sum_{i=d+2}^{n} 2^{n-i} \geq \sum_{i=1}^{n} 2^{n-M^k[i]} y_i$$

□

Theorem 4.3 implies that adding the constraints $x \lesssim_{M^{d+1}} \pi(x)\ \forall \pi \in \mathcal{G}(ILP)$ to the child nodes are enough to ensure that no feasible solution in the right child is equivalent to a feasible solution in the left. Therefore, restricting the branching decisions for descendents of the right child based on branching decisions in descendents of the left child is not necessary. This insight allows for the defining of a ranked branching rule that removes the restrictions on branching variables that arise in isomorphism pruning.

The *relative rank* for node a with $|F_1^a| + |F_0^a| = d$, M^a, is similar to rank M^d in that it tracks the order in which variables were branched on from the root node to a. However, unlike the ranked branching rule, nodes at the same depth do *not* necessarily have the same history of variables fixed by branching. Now, the set $\mathcal{S}(a) = \{i_1, i_2, \ldots, i_d\}$, a function of the node, contains the (ordered) indices of the variables fixed by branching decisions from the root node to the node a. The relative rank M^a is

$$M^a(i_j) = j \text{ for } j = 1, \ldots, d$$

$$M^a(k) = n+1 \text{ for } k \notin \mathcal{S}(a).$$

As with $\lesssim_{M^d}$, $\lesssim_{M^a}$ induces only a quasi-order on $\{0,1\}^n$. However, as Theorems 4.4 and 4.5 will demonstrate, the collection of quasi-orders corresponding to all nodes in the tree can be used to define a minimal fundamental domain. At every node a in the tree, the following constraints are enforced implicitly:

$$x \lesssim_{M^a} \pi(x)\ \forall \pi \in \mathcal{G}(ILP) \tag{4.5}$$

Theorem 4.4 *Let $\mathcal{T}'(B)$ be the enumeration tree generated by any branching rule B where nodes are pruned only if the resulting linear relaxation, with the rank constraints* (4.5)*, is infeasible. Nodes are not pruned by bound. The set of feasible solutions corresponding to nodes at depth n in $\mathcal{T}'(B)$ is a fundamental domain of the set of all integral feasible solutions of the ILP with respect to $\mathcal{G}(ILP)$.*

Proof: The theorem is proven by contradiction. Suppose there is a feasible solution x for which no members of $\mathrm{orb}(x, \mathcal{G})$ are feasible at any nodes of $\mathcal{T}'(B)$ at depth n. All solutions in $\mathrm{orb}(x, \mathcal{G})$, then, must be feasible in a node that was pruned by isomorphism. WLOG, let x be an element in $\mathrm{orb}(x, \mathcal{G})$ that was feasible in the leftmost such

node and call that node a. Node a was pruned because there exists a $\pi \in \mathcal{G}(ILP)$ with $\pi(F_1^a) \prec_{M^a} F_1^a$. Let i be the smallest element that is in only one of $M^a[F_1^a]$ or $R_r^a[\pi(F_1^a)]$. Because $\pi(F_1^a) \prec_{M^a} F_1^a$, it must be that $i \in M^a[\pi(F_1^a)]$. Let node b be ancestor node of a where variable x_i was branched on. Because $i \in F_0^a$, node a is to the right of node b. By our choice of i, we have $\pi(x)_j = 1$ for all $j \in F_1^c$ and $\pi(x)_j = 0$ for all $j \in F_0^c$. We also have $\pi(x)_i = 1$, so $\pi(x)$ is to the left of c, contradicting our choice of x. □

Theorem 4.5 *The set of all solutions feasible to subproblems corresponding to nodes at depth n in $\mathcal{T}'(B)$ is a minimal fundamental domain of the set of feasible solutions with respect to symmetry group $\mathcal{G}(ILP)$.*

Proof: Again, suppose not. Let $\pi \in \mathcal{G}$ send some feasible solution x to another feasible solution $\pi(x)$, neither of which are pruned in $\mathcal{T}'(B)$. Let z be the node $\mathcal{T}'(B)$ of depth n corresponding to solution x. Let c be the first common ancestor of x and $\pi(x)$, and assume $\pi(x)$ is to the left of x in $\mathcal{T}'(B)$. As a result of Theorem 4.3, $\pi(x) \prec_{M^a} x$ implies $\pi(x) \prec_{M^z} x$, so node z is not in $\mathcal{T}'(B)$. □

Unfortunately, removing all solutions that violate one of the inequalities (4.5) is not practical. It is only practical to test whether the set of fixed variables implies a violated inequality. As a result, it is possible that an optimal solution to the LP relaxation at a node may not satisfy (4.5). There may be cases in which a node would have been pruned by bound if all constraints in (4.5) had been added to the LP relaxation, but it is not pruned with isomorphism pruning. This can happen if the node contains some solutions that are elements of the fundamental domain, but all optimal and near optimal solutions to the LP relaxation violate constraints (4.5). Completely removing all solutions of a given node that violate constraints (4.5) from the feasible region creates too many numerical issues for it to be a worthwhile strategy. Nevertheless, there are opportunities to remove some solutions that violate inequalities (4.5) from the feasible region of a given subproblem via fixing variables. Section 4.2.2 will discuss this strategy.

4.2 Implementation

4.2.1 The Smallest-Image Set function in GAP

The testing required to prune nodes in the enumeration tree by isomorphism pruning requires computational algebra algorithms not found in integer programming software. Instead, we use the computational algebra package GAP [23]. The GAP algorithm that is used to test for violated inequalities of the form (4.5) is described in Section 4.2.1. We describe a variable fixing algorithm in Section 4.2.2. The algorithm used to test for violated rank inequalities is designed to be used only once. As a result, many computations are repeated when we test for violated inequalities at every node in the tree. In Section 4.2.3 we will discuss ways to speed up the testing algorithm. The algorithm we use is based on code written by Steve Linton for the $\mathrm{SmallestImageSet}$ function [49] in the GAP package GRAPE

[24]. To explain the two enhancements to the $\mathrm{SmallestImageSet}$ function specific to isomorphism pruning, we must describe $\mathrm{SmallestImageSet}$ in some detail in Section 4.2.1.

The GAP function `SmallestImageSet` takes as input the set F and the symmetry group $\mathcal{G}$ and outputs the unique set $F' \in \mathrm{orb}(F, \mathcal{G})$ with

$$F' \preceq_e \pi(F) \; \forall \pi \in \mathcal{G}.$$

The `SmallestImageSet` function can be used to prune nodes by isomorphism if the Minimum Indexed Branching rule is used.

Example Let $\mathcal{G}$ be the group generated by the permutations in Table 4.1.

π
(3, 6)(4, 7)(5, 8)
(1, 2)(4, 5)(7, 8)
(1, 3)(2, 6)(5, 7)
(1, 9)(3, 4)(6, 7)

Table 4.1: Generators of $\mathcal{G}$

Table 4.2 gives a collection of sets and their associated smallest-image with respect to $\mathcal{G}$. We also give a permutation that maps each set to its smallest-image.

F	Smallest-Image	Permutation Mapping F to Smallest-Image
(5)	(1)	(1, 6, 5)(2, 9, 3, 4, 7, 8)
(2,6)	(1,3)	(1, 7, 6, 3, 5,2)(4, 8, 9)
(1,6)	(1,3)	(2, 7, 9, 4)(3, 5, 8, 6)
(1,6,7)	(1,2,4)	(1, 2, 8, 7)(4, 9, 5, 6)
(1,2,6)	(1,2,3)	(3, 6)(4, 7)(5, 8)
(1,2,6,7,9)	(1,2,3,4,5)	(1, 5, 6)(2, 3, 7)(4, 8, 9)

Table 4.2: Examples of the `SmallestImageSet` Function

The `SmallestImageSet` function is not able to test for pruning using any relation other than $\preceq_e$. For an arbitrary complete rank R, the inputs to `SmallestImageSet` must be altered to use the relation $\preceq_R$. To alter the input, variables are identified not by their index, but by their rank, so R is a mapping from an *index space* to a *rank space*. The standard relation $\preceq_e$ can then be used to relate elements in the rank space.

Example Let R be such that:

Table 4.2 shows the mapping of the input sets in Table 4.2 from index space to rank space.

The symmetry group must also be altered to account for the variable mapping. For a complete rank function R, a

R(1) = 1
R(2) = 2
R(3) = 7
R(4) = 4
R(5) = 6
R(6) = 9
R(7) = 3
R(8) = 5
R(9) = 8.

Set	Rank of Set
(5)	(6)
(2,6)	(2,9)
(1,6)	(1,9)
(1,6,7)	(1,9,3)
(1,2,6)	(1,2,9)
(1,2,6,7,9)	(1,2,3,8,9)

Table 4.3: Mapping from Index Space to Rank Space

permutation $r \in S^n$ with $r(i) = R(i)$ is created for all $i \in \{1, \ldots, n\}$. The group $\mathcal{G}_r = r \circ \mathcal{G} \circ r^{-1}$ is the conjugate of $\mathcal{G}$ by r. The GAP function `ConjugateGroup` performs this operation.

Example Table 4.4 shows the conjugated permutations using the rank R of the generators given in Table 4.1 from Example 4.2.1. The old and new generators are given in Table 4.4.

Original Generator	Conjugate Generator
(3, 6)(4, 7)(5, 8)	(3,4)(5,6)(7,9)
(1, 2)(4, 5)(7, 8)	(1,2)(3,5)(4,6)
(1, 3)(2, 6)(5, 7)	(1,7)(2,9)(3,6)
(1, 9)(3, 4)(6, 7)	(1,8)(3,9)(4,7)

Table 4.4: Conjugating a Symmetry Group

A Corollary to Theorem 4.1 is the following:

Corollary 4.6 *For a complete rank R, $x \preceq_R \pi(x)$ for all $\pi \in \mathcal{G}$, if and only if $R(x) \preceq_e R(\pi(x))$ for all $\pi \in \mathcal{G}_R$.*

The conjugation required to map $\mathcal{G}$ from index space to rank space requires a complete rank. However, the rank associated with nodes of the tree that are not of depth n are not complete ranks. It suffices to arbitrarily assign each variable a unique rank from $\{d + 1, \ldots, n\}$, to form a complete rank. In what follows, we assume that the variables have already been mapped to rank space and the group conjugated, so that only the relation $\prec_e$ is used. Because $\prec_e$ is assumed, the relation will not be specifically stated.

The `SmallestImageSet` function requires finding the set of permutations in $\mathcal{G}$ that obeys a set of permutation constraints, $\mathcal{P} = \{i_1 \rightarrow j_1,\ i_2 \rightarrow j_2,\ \ldots,\ i_l \rightarrow j_l\}$. Generating the set of permutations in $\mathcal{G}$ that satisfy $\mathcal{P}$ is done by an iterative process. Initially $\pi = e$. At every iteration $k \leq l$, a permutation $\sigma \in \mathrm{stab}(j_1,\ j_2,\ \ldots,\ j_{k-1}, \mathcal{G})$ with $\sigma(\pi(i_k)) = j_k$ is found and π is updated to $\pi \leftarrow \sigma \circ \pi$. The collection of permutations $\mathrm{stab}(j_1,\ j_2,\ \ldots,\ j_l, \mathcal{G}) \circ \pi$ contains all permutations that satisfy the permutation constraint $\mathcal{P}$. Note that $\mathrm{stab}(j_1,\ j_2,\ \ldots,\ j_l, \mathcal{G}) \circ \pi \subset \mathcal{G}$ is not necessarily a subgroup (it may not contain the identity permutation). The key thing to note is that, while there may be several permutations that map $\pi(i_k)$ to j_k, the choice of σ does not affect the final collection of permutations. The basic *permutation generation* method is formalized in Algorithm 4.1.

Algorithm 4.1 Generating Permutations Satisfying Constraints

Input:	Group $\mathcal{G}$, set of constraints $\mathcal{P}$.
Output:	Set $\mathcal{S} \subseteq \mathcal{G}$ satisfying $\mathcal{P}$.
Step 1.	Let $\Gamma = \mathcal{G},\ \pi = e$
Step 2.	For every $(i_k \rightarrow j_k) \in \mathcal{P}$:
Step 2a.	Find $\sigma \in \Gamma$ with $\sigma(i_k) = j_k$. If none exists return $\emptyset$
Step 2b.	Update: $\Gamma = \mathrm{stab}(j_k,\ \Gamma)$, $\pi = \sigma \circ \pi$
Step 3.	Return $\Gamma \circ \pi$

The GAP `SmallestImageSet` function uses a branching tree to construct a permutation that maps F to its smallest-image. The branching decision at each node imposes a permutation constraint ($i_k \rightarrow j_k$ for some $i_k \in F$). As such, each node a at depth k is identified by a set of k permutation constraints, $\mathcal{P}_a = \{i_1 \rightarrow j_1,\ \ldots, i_k \rightarrow j_k\}$. Each permutation at node a must map the element i_h to the element j_h for all permutation constraints at node a as well as, of course, be in $\mathcal{G}$.

Using Algorithm 4.1, the collection $\mathcal{S}^a \subset \mathcal{G}$ of feasible permutations at node a may be generated. In the permutation tree, the set of currently unmapped elements in F^a, called $\mathrm{remset}^{\mathrm{a}}$, is examined. Let $m^a = \min\{\pi(f) | f \in \mathrm{remset}^{\mathrm{a}},\ \pi \in \mathcal{S}^a\}$ be the minimum that some element on $\mathrm{remset}^{\mathrm{a}}$ can be mapped to using permutations in $\mathcal{S}^a$. There may be multiple elements in remset^a that can be mapped to m^a. For every element $i \in \mathrm{remset}^a$ that can be mapped to m^a using a permutation in $\mathcal{S}^a$, the algorithm creates a child node with the additional permutation constraint $i \rightarrow m^a$.

For node a at depth k, if m^a is greater than $F[k+1]$, then every permutation $\pi \in \mathcal{S}^a$ maps F to a lexicographically larger set, i.e., $F \prec_e \pi(F)$. In this case, node a can be pruned. If, $m^a < F_{k+1}$, then children of a are formed with the additional constraint $(i \rightarrow m^a)$ for some $i \in \mathrm{remset}^a$. Every permutation feasible at a that satisfies this constraint maps the set F to a lexicographically smaller set. Thus, if $m^a < F_{k+1}$, then F is not a smallest-image. It is not necessary to know the smallest-image of F for isomorphism pruning, only that F is not it. Therefore, any node a such that $m^a < F_{k+1}$ can be pruned by isomorphism. A proof of correctness for `SmallestImageSet` is given in Linton [49].

Example Consider the graph in Figure 4.2.

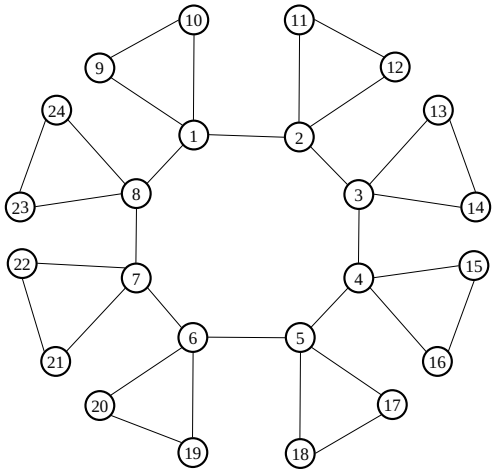


Figure 4.2: `SmallestImageSet` Example

The symmetry group of Figure 4.2, $\mathcal{G}$, is generated by clockwise rotations of 45 degrees,

$$rotate_{45} = (1,\ 2, \ldots,\ 8)(9,\ 11, \ldots,\ 23)(10,\ 12, \ldots,\ 24),$$

as well as reflecting about the line intersecting nodes 1 and 5,

$$reflect_{1,5} = (2,8)(3,7)(4,6)(11,24)(12,23)\ldots(15,20)(16,19).$$

Let F be the set $\{1,\ 3,\ 4\}$. We wish to find the smallest-image of F with respect to the group $\mathcal{G}$. The search tree is shown in Figure 4.3.

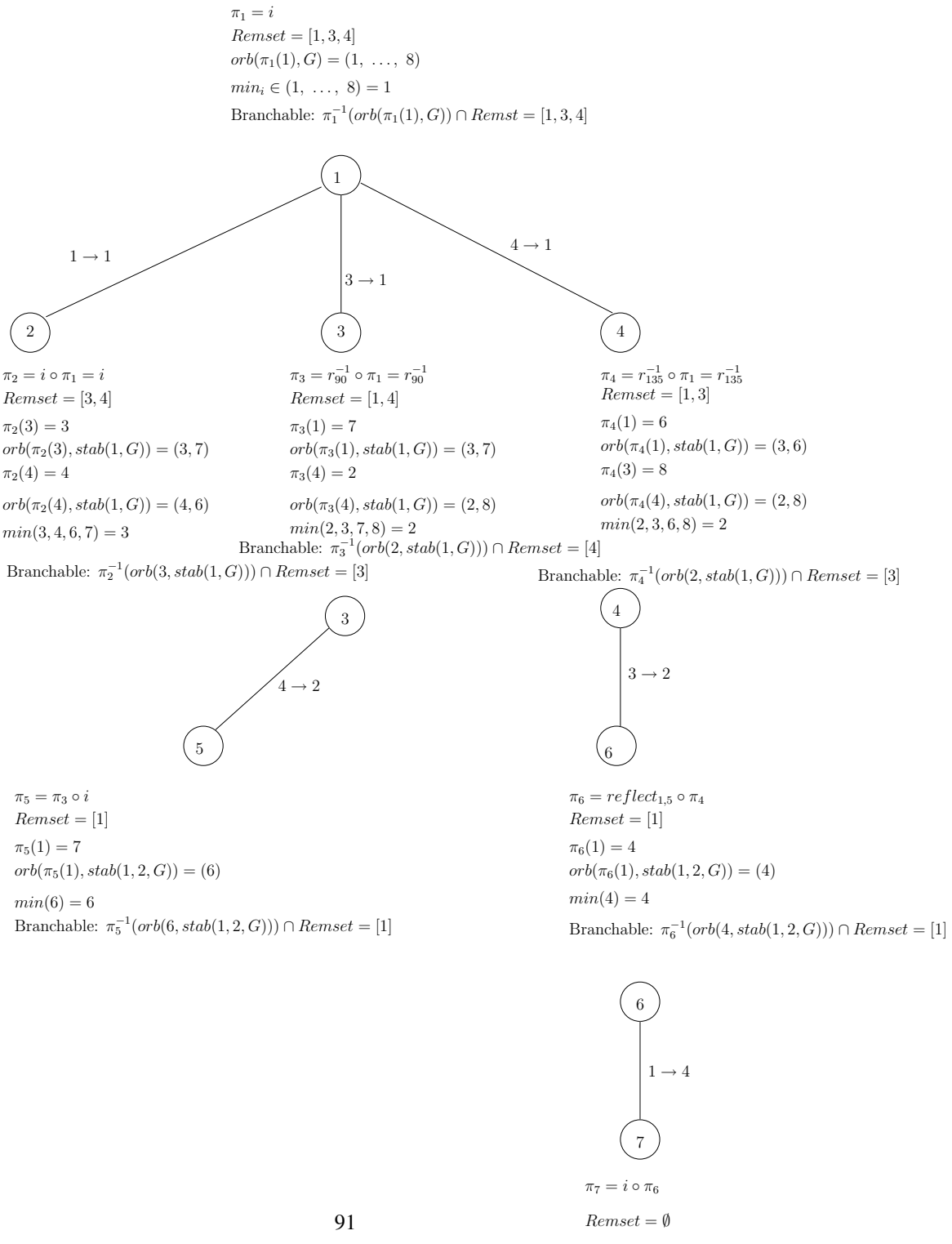


Figure 4.3: Permutation Tree for `SmallestImageSet` Example

At node 1 of the tree, elements 1, 3, and 4 all share the same orbit, $(1,\ \ldots,\ 8)$. The smallest element of the orbit is 1, so the branching decision is based on which element's image will be 1. Since all elements in F can be mapped to 1, three children are created.

Node 2 is formed by adding the permutation constraint $1 \rightarrow 1$. The identity permutation, e, is feasible at node 1 and maps element 1 to element 1, so $\pi_2 = e$. The smallest element that either element 3 or element 4 can be mapped to is 3, so no permutation in $\mathrm{stab}(1, \mathcal{G})$ can map F to a set that is lexicographically smaller than the set $[1, 3]$.

Node 3 is formed by adding the permutation constraint $3 \rightarrow 1$. The permutation $r_{90}^{-1} = {r_{45}^2}^{-1}$ is feasible in the root node and also maps element 3 to element 1, $\pi_3 = r_{90}^{-1}$. The permutation π_3 maps element 1 to 7 and element 4 to 2, so the algorithm must find the smallest element that is either in $\mathrm{orb}(\{7\}, \mathrm{stab}(\{1\}, \mathcal{G})) = \{3, 7\}$ or $\mathrm{orb}(\{2\}, \mathrm{stab}(\{1\}, \mathcal{G})) = \{2, 8\}$. Only element 4 can be mapped to element 2 (by the permutation $e \circ \pi_3$), so only one child node is created. If `SmallestImageSet` was being used for isomorphism pruning, then the search would be terminated at this point because $m^3 = 2 < F_2 = 3$. The permutation $e \circ \pi_3$ maps the set $[1,\ 3,\ 4]$ to a lexicographically smaller set $[1,\ 2,\ 7]$. To find the smallest-image, the algorithm must continue, but because $m^3 = 2 < 3 = m^2$, node 2 can be pruned.

The processing of node 4 is similar to node 3. The element $m^4 = 2$, so no nodes can be pruned. Only one element in remset^4 can be mapped to 2, using $reflect_{1,5}$, so there is only one child of node 4. Node 6 has $\pi_6 = reflect_{1,5} \circ \pi^4$.

Node 5 can be pruned after processing node 6 because π_6 maps element 1 to element 4, i.e., $m^6 = 4 < m^5 = 6$. Node 7 is then formed by including the permutation constraint $1 \rightarrow 4$ and yields the permutation π_7 that maps the set $[1,\ 3,\ 4]$ to $[1,\ 2,\ 4]$. At this point the tree is complete. The set $[1,\ 2,\ 4]$ is the smallest image of F with respect to $\mathcal{G}$.

4.2.2 Smallest-Image Fixing

During the call to `SmallestImageSet`, a set of variables that can be fixed to zero may be identified. This operation is called *smallest-image fixing*. This section discusses how smallest-image fixing is performed.

Node a, of the tree created by the `SmallestImageSet` function, is defined by sets of permutation constraints. For any node a of depth k in the tree, if F is its own smallest-image, then no element in remset^a can be mapped to an element i with $i < F_{k+1}$.

Theorem 4.7 *Suppose MIB was used as a branching rule throughout the branch-and-bound tree. Let x_j be the next variable chosen for branching by MIB. If, at node a of the permutation tree, there is a permutation $\pi \in \mathcal{S}^a$ with $\pi(j) < F_{k+1}$, then the set $F \cup \{j\}$ is not its own smallest-image. As a result, if x_j was branched on, the subproblem formed by setting x_j to one would be pruned by isomorphism. Thus, x_j may be fixed at 0.*

Proof: Because $\pi \in \mathcal{S}^a$, π maps elements of F to $F_1,\ F_2, \ldots,\ F_k$.

$2^{\pi(F_1)}+2^{\pi(F_2)}+\ldots+2^{\pi(F_{|F|})}+2^{\pi(j)} \geq 2^{F_1}+2^{F_2}+\ldots+2^{F_k}+2^{\pi(j)} > 2^{F_1}+2^{F_k}+\ldots+2^{F_k}+2^j+\sum_{i=j+1}^{n} 2^i$.
So, $\pi(F \cup \{j\}) \prec_e F \cup \{j\}$. □

Theorem 4.7 can be easily extended to general branching strategies. The importance of variable x_j in Theorem 4.7 is that it was the variable with the smallest free index. With flexible branching, any variable chosen for branching is given the smallest free rank. Thus, any free variable x_j, if there is a node a in the permutation tree such that there exists a $\pi \in \mathcal{S}^a$ with $\pi(j) < F_{k+1}$, then the set $F \cup \{j\}$ will not be its own smallest-image with respect to the rank vector associated with the current branch-and-bound node.

Example Consider again the graph in Figure 4.2 and assume that one wanted to find the largest independent set in the graph. Let Figure 4.3 be the current branch-and-bound tree. For the sake of simplicity, assume for this problem that MIB is used as a branching rule. Using MIB ensures the rank of an element is equal to the element itself. In his case, no conjugation of the symmetry group is needed, so ranks are omitted below.

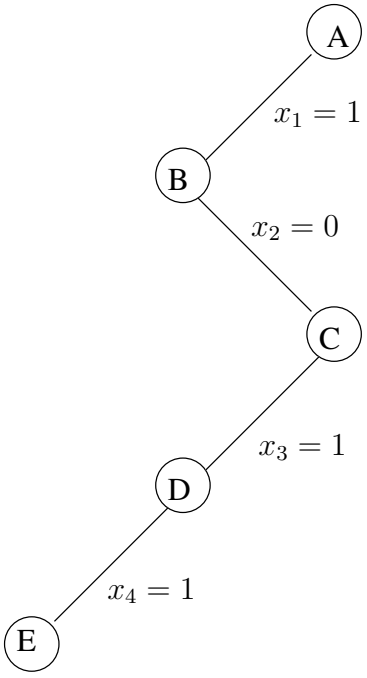


Figure 4.4: Branch and Bound Tree for Isomorphism Fixing Example

We know from Example 4.2.1 that the set $F = [1,\ 3,\ 4]$ is not its own smallest-image with respect to the symmetry group $\mathcal{G}$, so node E can be pruned. Smallest-image fixing, applied at node D, fixes x_4 to zero, eliminating the need to create node E. Consider the tree in Figure 4.5, required to show that the set $F^D = [1, 3]$ is its smallest image.

If G is a descendent of D, then $F_1^D \prec_e F_1^G$. As a result, for any two elements i and j in F^G, if F^G is its own smallest-image, then there does not exist a permutation mapping $\{i, j\}$ to $\{1, 2\}$. This is precisely the information that can be used to fix additional variables by isomorphism fixing. Consider node 2 in Figure 4.5. Since node 2 has

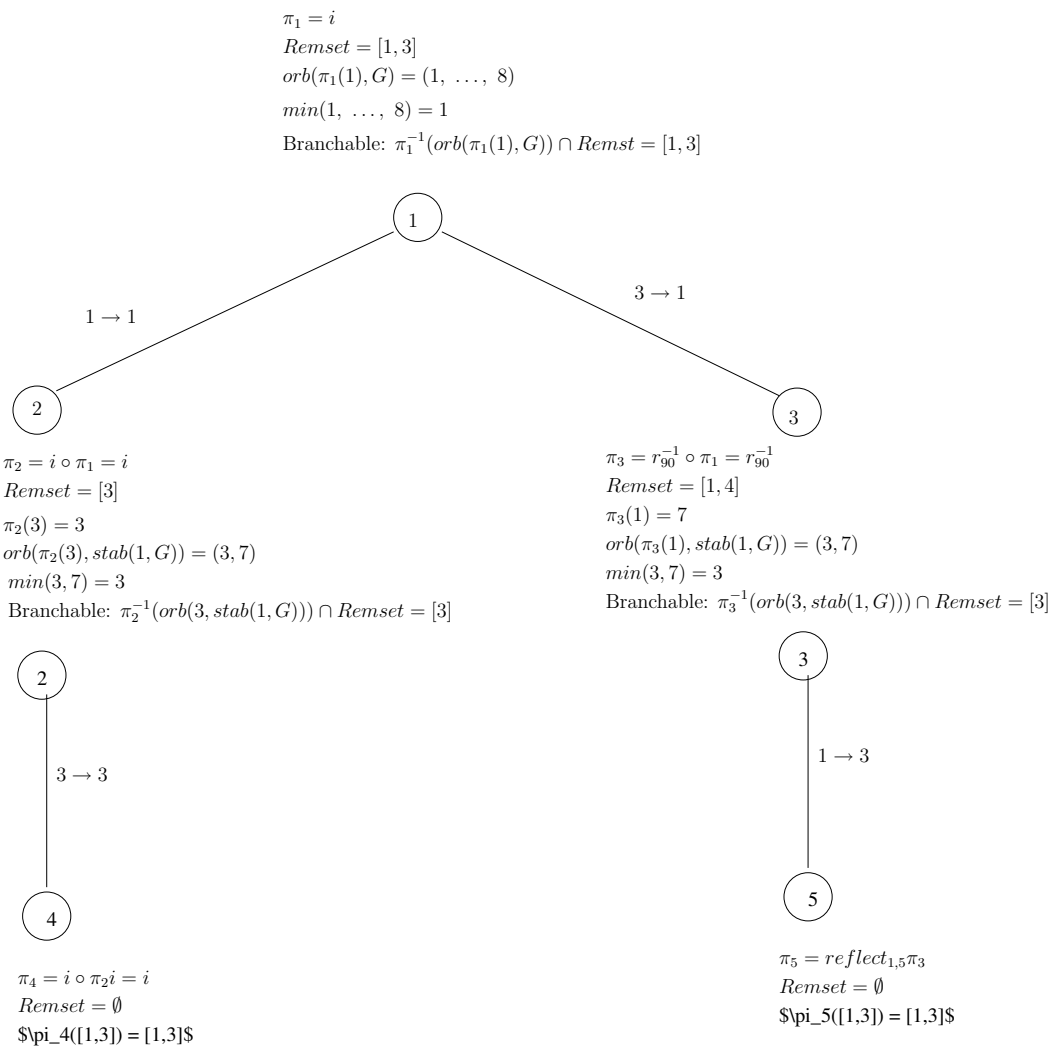


Figure 4.5: Permutation for Isomorphism Fixing Example

the permutation constraint $1 \rightarrow 1$, if F^G is its own smallest-image, then F^G must not contain any element that can be mapped to the element 2 by permutations feasible at node 2, (i.e., $\mathrm{stab}(1, \mathcal{G}) \circ e$). Because $\mathrm{stab}(1, \mathcal{G}) \circ e$ contains a permutation that maps element 8 to element 2, we can fix x_8 to zero at the current node in the branch-and-bound tree. Interestingly, the fixing of x_8 to 0 would also have occurred with orbital branching at node B in the branching tree, as depicted in Figure 4.4 .

Variables can also be fixed via smallest-image fixing at node 3 in Figure 4.5. A variable that can be mapped to 2 using the symmetry group $\mathrm{stab}(1, G) \circ \pi_3$ can be fixed to zero. In addition, $\pi_3 \circ \mathrm{orb}(2, \mathrm{stab}(1, G)) = [2, 4]$, so x_4 can be fixed to zero. This fixing would have been done by orbital fixing at node C. However, it is possible to find variables that can only be fixed using this method, and not by orbital branching or orbital fixing.

4.2.3 Speedups to `SmallestImageSet`

Smallest-image fixing gives information that can be exploited at every node in the branch-and-bound tree to speed up the call to `SmallestImageSet`. At every node a that is not pruned, F_1^a is its smallest-image. Also, for any currently free variable x_i, if $F \cup \{i\}$ is *not* its smallest-image, then the permutation mapping $F \cup \{i\}$ to a smaller image must send i to an element in F. If this is not the case, x_i would have been fixed by smallest-image fixing. When testing if $F \cup \{i\}$ is its own smallest-image, the permutation constraint $i \rightarrow F_j$ can be used as the initial branching disjunction for each $j \in \{1, \ldots, |F|\}$. This will make the permutation tree smaller and avoid processing nodes in the permutation tree that are identical to nodes in the parent's permutation tree. We have not implemented this change to Linton's implementation of `SmallestImageSet` in GAP.

4.3 Computational Experiments

In this section, we give empirical evidence of the effectiveness of combining flexible isomorphism pruning with orbital branching. Because of the flexibility introduced to isomorphism pruning in this chapter, orbital branching can be used in conjunction with isomorphism pruning. We investigate the impact of choosing the orbit on which branching is based, and we demonstrate the positive effect of fixing based on information learned while computing a set's smallest-image. The computations are based on the instances whose characteristics are given in Table 4.5. Most of these instances are described in Chapter 3.

The computations were run on machines with AMD Opteron processors clocked at 1.8GHz and having 2GB of RAM. The COIN-OR software `Clp` was used to solve the linear programs at nodes of the branch and bound tree. For each instance, the (known) optimal solution value was set a priori to aid in pruning and reduce the random impact of finding a feasible solution in the search. Nodes were searched in a depth-first fashion.

One major drawback of having a rigid branching rule is that free variables that are given integer values in the LP

Name	Variables	Group Size
cod83	256	10,321,920
cod93	512	185,794,560
cod105	1024	3,715,891,200
codbt05	243	933,120
codbt15	486	
codbt42	144	
codbt61	192	
codbt71	384	
codbt80	256	
cov1053	252	3,628,800
cov1054	2252	3,628,800
cov1075	120	3,628,800
cov1076	120	3,628,800
cov954	126	362,880
sts45	45	360
sts63	63	72,576
sts81	81	1,965,150,720

Table 4.5: Symmetric Integer Programs

relaxation may be chosen for branching. When this happens, the LP relaxation in at least one of the child nodes does not change. The effect of branching on integer solutions was investigated. Table 4.6 compares the size of the tree using the MIB rule with a slightly relaxed rule that branches on the minimum-indexed fractional variable.

As Table 4.6 shows, the flexibility of branching only on fractional variables can significantly reduce the size of the branch-and-bound tree. For these computations, only reduced-cost fixing and fixing based on symmetry were used to set variables at nodes throughout the branch-and-bound tree. More advanced algorithms for fixing may lessen the impact of flexible branching as variables that are set to a value do not need to be branched on even in the MIB rule.

The branching rules used for these computations are described in Section 3.3.2.

Rule 1: Branch Largest

Rule 2: Branch Largest IP Solution

Rule 3: Strong Branching

Rule 4: Break Symmetry Left

Rule 5: Keep Symmetry Left

Table 4.7 shows the results of an experiment designed to compare the performance of the five different branching rules introduced in Section 3.3.2. In this experiment, reduced cost fixing and smallest-image fixing were used, and the CPU time required (in seconds) for both isomorphism pruning with the minimum index rule and flexible isomorphism pruning are reported.

Instance	Min Index	Min Fractional Index
	Nodes	Nodes
cod(10,5)	11	7
cod(8,3)	23	23
cod(9,3)	257	271
codbt(0,5)	1199	1065
codbt(1,5)	825	791
codbt(4,2)	889	591
codbt(6,1)	29	29
codbt(7,1)	1	1
codbt(8,0)	78	158
cov(10,5,3)	603	357
cov(10,5,4)	401	319
cov(10,7,5)	347	391
cov(10,7,6)	19911	13523
cov(9,5,4)	525	239
sts45	3085	1311
sts63	4866	3081
sts81	485	445

Table 4.6: Flexibility in Min Index Branching

		Rule 1		**Rule 2**		**Rule 4**		**Rule 4**		**Rule 5**	
Instance	Size	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
cod	(8,3)	8	33	8	35	23	23	15	53	17	81
cod	(9,3)	285	1211	464	2611	-	-	1060	1459	400	537
codbt	(0,5)	100	1511	40	183	538	77	376	1765	115	679
codbt	(3,3)	3	7	3	7	10	7	5	13	9	19
codbt	(4,2)	7	243	5	73	20	29	26	359	25	497
codbt	(6,1)	2	9	2	9	4	7	4	13	7	29
codbt	(8,0)	11	13	12	25	17	11	15	19	16	23
cov	(9,5,4)	9	257	6	103	12	31	?	?	10	143
cov	(10,5,3)	35	559	134	2193	122	107			130	1331
cov	(10,7,5)	39	407	19	81	137	45			40	333
sts	45	13	2515	16	3307	38	859	52	4015	17	1733
sts	63	62	5901	57	5563	147	1987	175	6463	55	3721
sts	81	743	395	758	419	73	341	2715	589	356	689

Table 4.7: Comparison of Branching Rules

		SI Fixing	**No SI Fixing**
Instance	Size	Nodes	Nodes
cod	(8,3)	33	33
cod	(9,3)	1211	1347
codbt	(0,5)	1511	1569
codbt	(1,5)		955
codbt	(4,2)	243	247
codbt	(6,1)	9	9
codbt	(8,0)	13	25
cov	(9,5,4)	257	269
cov	(10,5,3)	559	571
cov	(10,7,5)	407	433
sts	45	2515	2905
sts	63	5901	6709
sts	81	395	503

Table 4.8: Impact of Smallest-Image Fixing

The most effective branching method is **Rule 1**, the method that branches on the largest orbit. Recall that for the computational results with orbital branching alone in Chapter 3, this rule was only middling. This is not surprising, given how much symmetry remained in the tree with this rule (see Table 3.4). The ability to remove symmetry through other methods, in this case isomorphism pruning, gives **Rule 1** the advantage of fixing more variables (as a result of branching on larger orbits) without an increase in number of equivalent nodes. Also, this branching rule requires less time than methods such as **Rule 3** and **Rule 5**. **Rule 3** universally produced the smallest trees, but the time required at each node is much greater than that of other branching rules. This would indicate that branching rules that attempt to approximate strong branching would be effective. Most importantly, Figure 4.7 shows that different branching rules can have a significant affect on solution times. Different problem classes may require different branching rules, and as a result of the flexibility given to the branching decisions, these rules can be implemented.

In Table 4.8 the impact of smallest-image fixing on the size of the enumeration tree is examined. For this test, all problems are solved using the branching rule **Rule 1**. In the case where smallest-image fixing is turned off, orbital fixing is still used. The set of variables fixed by orbital fixing will always be a subset of the variables fixed by smallest-image fixing, so the study is meant to show the marginal benefit of smallest-image fixing over orbital fixing (the power of orbital fixing has been shown in Chapter 3). These results show that, especially in larger trees, smallest-image fixing can have a significant impact on the size of the branch-and-bound tree. Recall that this fixing is done using information obtained by the call to the `SmallestImageSet` function. smallest-image fixing requires no significant additional computational effort when using the `SmallestImageSet` function.

Chapter 5

Constraint Orbital Branching

In Chapter 3, we presented orbital branching as a way to branch on variables. Orbital branching does not actually branch on variables. Instead, orbital branching branches on the disjunction "either the orbit contains a non-zero variable *or* all variables in the orbit are zero" (see 3.2). Branching on this particular disjunction allows us to then fix variables; constraints enforcing the disjuncion do not need to be included. In this sense, Chapter 3 can be seen as a special case of the work presented in this chapter, where we examine the effects of branching on more general disjunctions.

Exploiting the symmetry in a problem by branching on more general disjunctions of this form can often be significantly strengthened by exploiting certain types of embedded subproblem structures. Specifically, if the disjunction on which the branching is based is such that relatively few non-isomorphic feasible solutions may satisfy one side of the disjunction, then portions of potential feasible solutions may be enumerated. The original problem instance is then partitioned into smaller, more tractable problem instances. As an added benefit, the smaller instances can then be solved easily in parallel. A similar technique has been recently employed in an ad-hoc fashion in [46] in a continuing effort to solve an integer programming formulation for the football pool problem. This work poses a general framework for solving difficult, symmetric integer programs in this fashion.

The power of constraint orbital branching is demonstrated by solving to optimality, for the first time, a well-known integer program that computes the incidence widths of a Steiner Triple System with 135 elements and with 243 elements. Previously, the largest instance solved in this family contained 81 elements [53]. The generality of the constraint orbital branching procedure is further illustrated by an application to the construction of minimum cardinality covering designs. In this case, the previously best known bounds from the literature are easily reproduced.

In Section 5.1, the constraint orbital branching method is presented and proved to be a valid branching method. Section 5.2 discusses the properties of good disjunctions for the constraint orbital branching method. Section 5.3

describes our computational experience with the constraint orbital branching method, and conclusions are given in Section 5.5.

5.1 Constraint Orbital Branching

The primary focus of this chapter is on set covering problems of the form

$$\min_{x\in\mathcal{F}}\{e^T x\},\ \text{with}\ \mathcal{F} \stackrel{\text{def}}{=} \{x\in\{0,1\}^n \mid Ax \geq e\}, \tag{5.1}$$

where $A \in \{0,1\}^{m\times n}$ and e is a vector of ones of conformal size. The restriction of our work to set covering problems is mainly for notational convenience, but also of practical significance, since many important set covering problems contain a great deal of symmetry.

Constraint orbital branching is based on the following simple observations. If $\lambda^T x \leq \lambda_0$ is a valid inequality for (5.1) and $\pi \in \mathcal{G}$, then $\pi(\lambda)^T x \leq \lambda_0$ is also a valid inequality for (5.1). In constraint orbital branching, given an integer vector $(\lambda, \lambda_0) \in \mathbb{Z}^{n+1}$, we will branch on a base disjunction of the form

$$(\lambda^T x \leq \lambda_0) \vee (\lambda^T x \geq \lambda_0 + 1),$$

simultaneously considering all symmetrically equivalent forms of $\lambda x \leq \lambda_0$. Specifically, the branching disjunction is

$$\left(\bigvee_{\mu\in\mathrm{orb}(\mathcal{G},\lambda)} \mu^T x \leq \lambda_0\right) \vee \left(\bigwedge_{\mu\in\mathrm{orb}(\mathcal{G},\lambda)} \mu^T x \geq \lambda_0 + 1\right).$$

Further, by exploiting the symmetry in the problem, it is only necessary to consider one representative problem for the left portion of this disjunction. That is, either the "equivalent" form of $\lambda x \leq \lambda_0$ holds for one of the members of $\mathrm{orb}(\mathcal{G},\lambda)$, or the inequality $\lambda x \geq \lambda_0 + 1$ holds for all of them. This is obviously a feasible division of the search space. Theorem 5.1 demonstrates that for any vectors $\mu_i, \mu_j \in \mathrm{orb}(\mathcal{G},\lambda)$, the subproblem formed by adding the inequality $\mu_i^T x \leq \mu_0$ is equivalent to the subproblem formed by adding the inequality $\mu_j^T x \leq \mu_0$. Therefore, we need to keep only one of these representative subproblems, pruning the $|\mathrm{orb}(\mathcal{G},\lambda)| - 1$ equivalent subproblems.

Theorem 5.1 *Let a be a generic subproblem and $\mu_i, \mu_j \in \mathrm{orb}(\mathcal{G},\lambda)$. Denote by b the subproblem formed by adding the inequality $\mu_i^T x \leq \mu_0$ to a and by c the subproblem formed by adding the inequality $\mu_j^T x \leq \mu_0$ to a. Then, $z^*(b) = z^*(c)$.*

Proof. Let x^* be an optimal solution of b. WLOG, we can assume that $z^*(b) \leq z^*(c)$. Since μ_i and μ_j are in the same orbit, there exists a permutation $\sigma \in \mathcal{G}$ such that $\sigma(\mu_i) = \mu_j$. By definition of $\mathcal{G}$, $\sigma(x^*)$ is a feasible solution to the subproblem with an objective value of $z^*(b)$. For any permutation matrix P we have that $P^T P = I$. Since x^* is

in b, $\mu_i^T x^* \leq \mu_0$. We can rewrite this as $\mu_i^T P_\sigma^T P_\sigma x^* \leq \mu_0$, or $(P_\sigma \mu_i)^T P_\sigma x^* \leq \mu_0$. This implies that $\mu_j P_\sigma x^* \leq \mu_0$, so $\sigma(x^*)$ is in c. This implies that $z^*(c) \leq z^*(b)$. By our assumption, $z^*(c) = z^*(b)$. □

The basic *constraint orbital branching* algorithm is formalized in Algorithm 5.1.

Algorithm 5.1 Constraint Orbital Branching

Input:	Subproblem a.
Output:	Two child subproblems b and c.
Step 1.	Choose a vector of integers λ of size n and an integer λ_0
Step 2.	Compute the orbit of λ, $\mathcal{O} = \{\mu_1, \ldots, \mu_p\}$.
Step 3.	Choose arbitrary $\mu_k \in O$. Return subproblems b with $\mathcal{F}(b) = \mathcal{F}(a) \cap \{x \in \{0,1\}^n : \mu_k^T x \leq \lambda_0\}$ and c with $\mathcal{F}(c) = \mathcal{F}(a) \cap \{x \in \{0,1\}^n : \mu_i^T x \geq \lambda_0 + 1, i = 1, \ldots, p\}$

As for standard branching on constraints, the critical choice in Algorithm 5.1 is in choosing the inequality (λ, λ_0) [42]. When dealing with symmetric problems, the embedded subproblem structure can be exploited to find strong branching disjunctions, as described in the next section.

5.2 Strong Branching Disjunctions, Subproblem Structure, and Enumeration

Many important families of symmetric integer programs are structured such that small instances from the family are embedded in larger instances. Such families include Steiner Triple System problems and Covering Design problems. In this case, the problem often shows a block-diagonal structure with identical blocks and some linking constraints, as expressed in Figure **??**.

$$\min e^T x^1 + e^T x^2 + \ldots + e^T x^r$$

s.t.

$$\begin{pmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \\ D_1 & D_2 & \ldots & D_r \end{pmatrix} \begin{pmatrix} x^1 \\ x^2 \\ \vdots \\ \\ x^r \end{pmatrix} \geq e$$

$$x^i \in \{0,1\}^n, i = 1, \ldots, r$$

The subproblem $z = \min_{x\in\{0,1\}^n}\{e^T x|\ Ax \geq e\}$, denoted by P, is often computationally manageable and can be solved to optimality in a reasonable amount of time. Constraint orbital branching allows us to exploit its optimal value z. The first step consists of choosing an index $i \in \{1,\ldots,r\}$ and enforcing the constraint $e^T x^i \geq z$, which obviously does not cut off any optimal solution to the original problem. Then, the new constraint is used as branching disjunction by letting $\lambda = [0_n,\ldots,\lambda_i,\ldots 0_n]$, $\lambda_i = e_n$ and $\lambda_0 = z$. The resulting child subproblems have interesting properties, as explainded below.

Left subproblem In the left child, the constraint $\lambda x \leq z$ is added. Since $\lambda x^i \geq z$ also holds, this is equivalent to $\lambda x^i = z$. Therefore, the projection of every feasible solution in the left subproblem onto the set $\{i_1,\ i_2,\ \ldots,\ i_n\}$ coincides with the set of the solutions of P with an objective value equal to z. Let $\{x_1^*, x_2^*, \ldots, x_l^*\}$ be the set of such (optimal) solutions. One can solve the left subproblem by dividing the subproblem into l smaller subproblems. Each of the l new subproblems is associated with a solution x_j^*, for $j = 1,\ldots,l$. Precisely, each child j is generated by fixing $x^i = x_j^*$. This yields two relevant benefits. First, the resulting integer programs are easier than the original. Second, the collection of subproblems are completely independent and can be solved in parallel. Of course, this option is viable only if the number of optimal solutions of P is reasonably small. Otherwise, one can select an index $k \neq i$ and choose $e_n^T x^k \geq z$ as a branching disjunction. Section **??** shows how to address this "branching or enumerating" decision for well-known difficult set covering problems.

However, a more insightful observation can lessen the number of subproblems to be evaluated as children of the left subproblem. Suppose that there is a symmetry group $\mathcal{G}(P) \subseteq \Pi^n$ with the property that any two solutions in P that are isomorphic with respect to $\mathcal{G}(P)$ generate subproblems that are isomorphic with respect to $\mathcal{G}$. If such a group exists, then one can limit the search in the left subproblem only to the children corresponding to solutions x_j^* that are non-isomorphic with respect to $\mathcal{G}(P)$.

The group $\mathcal{G}(P)$ is created as follows. Let $I = \{i\cdot n+1,\ \ldots,\ (i+1)n\}$ be the column indices representing the elements of x^i. First, compute the group $\mathrm{stab}(I,\mathcal{G})$. Note that this group is in $\Pi^{r\times n}$, but our primary interest is in how each $\pi \in \mathrm{stab}(I,\mathcal{G})$ permutes the n elements in I. For this reason, we *project* $\mathrm{stab}(I,\mathcal{G})$ onto I. Every permutation $\pi \in \mathrm{stab}(I,\mathcal{G})$ can be expressed as a product of two smaller permutations, ϕ and γ, where ϕ permutes the elements in I and γ permutes the elements not in I. We can write this as $\pi = (\phi,\gamma)$. The projection of $stab(I,\mathcal{G})$ onto I, $\mathcal{G}\downarrow_I$, contains all ϕ such that there exists a γ with $(\phi,\gamma) \in \mathrm{stab}(I,\mathcal{G})$. Note that permutations not in $\mathrm{stab}(I,\mathcal{G})$ cannot be projected in this way, so it is unambiguous to describe this set as $\mathcal{G}\downarrow_I$.

Theorem 5.2 *The projection of $\mathcal{G}$ onto I, $\mathcal{G}\downarrow_I$, is a subset of $\mathcal{G}(P)$.*

Proof. Let $\phi \in \mathcal{G}\downarrow_I$. Let x be any optimal solution of P. By definition, x and $\phi(x)$ are isomorphic with respect to $\mathcal{G}\downarrow_I$. Consider the subproblems formed by setting $x^i = x$ (subproblem a) and $x^i = \phi(x)$ (subproblem b). By

Figure 5.1: Example Graph

definition, there is a $\gamma \in \Pi^{n-I}$ with $\pi = (\phi, \gamma) \in \mathcal{G}$.

Let x^* be any integer feasible solution in a. By definition of permutation, we know that $\pi(x^*)$ is feasible at the root node. Also π sends x^i to $\phi(x^i)$. Since b differs from the root node only by the constraint $x^i = \phi(x^i)$, we have that $\pi(x^*)$ is in b. To conclude, any solutions to P that are isomorphic with respect to $\mathcal{G} \downarrow_I$ will generate subproblems that are isomorphic. □

Corollary 5.3 *The left subproblem can be decomposed into a set of restricted subproblems associated with the optimal solutions to P that are non-isomorphic with respect to $\mathcal{G} \downarrow_I$.*

In practice, non-isomorphic optimal solutions of symmetric problems often represent a small portion of all the optimal solutions. In this case, enumerating the left subproblem becomes very computationally efficient, as shown in the case studies of Section **??**.

Right subproblem In the right branch, the constraints $\mu^T x \geq \lambda_0 + 1$, for all $\mu \in \mathrm{orb}(\mathcal{G}, \lambda)$, are added. If $|\mathrm{orb}(\mathcal{G}, \lambda)|$ is fairly large, then the LP bound is increased considerably.

The whole branching process can be iterated at the right child. In fact, the constraint $e_n^T x^i \geq z + 1$ can be exploited as a branching disjunction. In this case all the solutions to P with value $z + 1$ should be enumerated to solve the new left branch.

Example:

Consider the graph $G = (V, E)$ of Figure 5.1 and the associated vertex cover problem

$$\min_{x \in \{0,1\}^{|V|}} \left\{ e^T x \mid x_i + x_j \geq 1 \quad \forall (i,j) \in E \right\}.$$

Its optimal solution has value 10 and is supposed to be known. The coefficient matrix A shows a block diagonal structure with three blocks, corresponding to the incidence matrices of the 5-holes induced by vertices $\{1, \ldots, 5\}$,

$\{6,\ldots,10\}$ and $\{11,\ldots,15\}$ respectively. Therefore, the i-th subproblem, $i \in \{0,1,2\}$, has the form

$$P: \min\ x_{5i+1} + x_{5i+2} + x_{5i+3} + x_{5i+4} + x_{5i+5}$$

s.t.

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{5i+1} \\ x_{5i+2} \\ x_{5i+3} \\ x_{5i+4} \\ x_{5i+5} \end{pmatrix} \geq e$$

$$x \in \{0,1\}^5$$

The group $\mathcal{G}(A)$ contains 60 permutations in Π^{15} and is generated by the following permutations:

$$\pi^1 = (2,\ 5)(3,\ 4)(7,\ 10)(8,\ 9)(12,\ 15)(13,\ 14) \qquad \pi^2 = (6,\ 11)(7,\ 12)(8,\ 13)(9,\ 14)(10,\ 15)$$

$$\pi^3 = (1,\ 2)(3,\ 5)(6,\ 7)(8,\ 10)(11,\ 12)(13,\ 15) \qquad \pi^4 = (1,\ 6)(2,\ 7)(3,\ 8)(4,\ 9)(5,\ 10)$$

$\mathcal{G}(P)$ can be created by projecting $\mathcal{G}(A)$ on the variables of the first block (i.e., $x_1,\ldots,x_5$). It consists of 10 permutations in Π^5 which are generated by $(2,\ 5)(3,\ 4)$, and $(1,\ 2)(3,\ 5)$.

The optimal solution to P has value 3 and there is only one non-isomorphic cover of size 3 (for instance, $x_1 = 1$, $x_2 = 1$ and $x_4 = 1$). At the root node we branch on the disjunction $\lambda = (1,1,1,1,1,0,\ldots,0)$, $\lambda_0 = 3$. Then, in the left subproblem, the constraint $x_1 + x_2 + x_3 + x_4 + x_5 \leq 3$ is added, while in the right subproblem, the constraints $x_1 + x_2 + x_3 + x_4 + x_5 \geq 4$, $x_6 + x_7 + x_8 + x_9 + x_{10} \geq 4$ and $x_{11} + x_{12} + x_{13} + x_{14} + x_{15} \geq 4$ are enforced.

Since P has a unique, non-isomorphic optimal solution, searching the left child amounts to solving only one subproblem with $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, $x_4 = 1$ and $x_5 = 0$. Its optimal value is 10 and the subproblem can be fathomed. On the right branch, the lower bound increases to 12 and also that subproblem can be fathomed.

If a classic variable-branching dichotomy is applied, it results in a much larger enumeration tree (15 subproblems versus 3).

In the general case of unstructured problems, finding effective branching disjunctions may be difficult. Potential disjunctions can be generated as follows. First, select a subset of variable indices I and project the problem's feasible region onto I. For any choice of λ, find $z_I = \min\{\sum_{i\in I} \lambda_i x_i | x \in Proj_I \mathcal{F}\}$. The constraint $\sum_{i\in I} \lambda_i x_i \geq z_I$ is a valid cut for the original problem and could be branched on. If enumeration is used to solve the left subproblem and the subproblem contains too many solutions, some elements of I could be removed and a new branching disjunction created.

5.3 Case Study:Steiner Triple Systems

A Steiner Triple System of order v, denoted by STS(v), consists of a set S with v elements, and a collection $\mathcal{B}$ of triples of S with the property that every pair of elements in S appears together in a unique triple of $\mathcal{B}$. Kirkman [44] showed that STS(v) exists if and only if $v \equiv 1$ or $3 \mod 6$. A covering of a STS is a subset C of the elements of S such that $C \cap T \neq \emptyset$ for each triple $T \in \mathcal{B}$. The *incidence width* of a STS is its smallest-sized covering. Fulkerson et al. [22] suggested the following integer program to compute the incidence width of a STS(v):

$$\min_{x \in \{0,1\}^v} \{e^T x \mid A_v x \geq 1\},$$

where $A_v \in \{0,1\}^{|\mathcal{B}| \times v}$ is the incidence matrix of the STS(v). Fulkerson et al. [22] created instances based on STS of orders $v \in \{9, 15, 27, 45\}$, and posed these instances as a challenge to the integer programming community. The instance STS(45) was not solved until five years later by H. Ratliff, as reported by Avis [5].

The instance of STS(27) was created from STS(9) and STS(45) was created from STS(15) using a "tripling" procedure described in [32]. We present the construction here, since the symmetry induced by the construction is exploited by our method in order to solve larger instances in this family. For ease of notation, let the elements in STS(v) be $\{1, 2, \ldots v\}$, with triples $\mathcal{B}_v$. The elements of STS($3v$) are the pairs $\{(i,j) \mid i \in \{1,2,\ldots,v\}, j \in \{1,2,3\}\}$, and its collection of triples is denoted as $\mathcal{B}_{3v}$. Given STS(v), the Hall construction creates the blocks of STS($3v$) in the following manner:

- $\{(a,k),(b,k),(c,k)\} \in \mathcal{B}_{3v} \quad \forall \{a,b,c\} \in \mathcal{B}_v,\ \forall k \in \{1,2,3\}$,
- $\{((i,1),(i,2),(i,3)\} \in \mathcal{B}_{3v} \quad \forall i \in \{1,\ldots,v\}$,
- $\{(a,\pi_1),(b,\pi_2),(c,\pi_3)\} \in \mathcal{B}_{3v} \quad \forall \{a,b,c\} \in \mathcal{B}_v,\ \forall \pi \in \Pi^3$.

Feo and Resende [17] created two new instances, STS(81) and STS(243), using this construction. STS(81) was first solved by Mannino and Sassano [53] 12 years ago, and it remains the largest problem instance in this family to be solved. STS(81) is also easily solved by the isomorphism pruning method of Margot [54] and the orbital branching method of Chapter 3, but neither of these methods seem capable of solving larger STS(v) instances. Karmarkar et al. [43] introduced the instance STS(135), which they build by applying the tripling procedure to the STS(45) instance of Fulkerson et al. [22]. [67] have reported the best known solutions to both STS(135) and STS(243), having values 103 and 198 respectively. Using the constraint orbital branching method, we have been able to solve STS(135) to optimality, establishing that 103 is indeed the incidence width.

$$A_{3v} = \begin{bmatrix} A_v & 0 & 0 \\ 0 & A_v & 0 \\ 0 & 0 & A_v \\ I & I & I \\ D_1 & D_2 & D_3 \end{bmatrix},$$

The incidence matrix, A_{3v}, for an instance of STS$(3v)$ generated by the Hall construction has the form shown in Figure 5.3, where A_v is the incidence matrix of STS(v) and the matrices D_i have exactly one "1" in every row. Note that A_{3v} has the block-diagonal structure that was discussed in Section 5.2, so it is a natural candidate on which to apply the constraint orbital branching method. Furthermore, the symmetry group Γ of the instance STS$(3v)$ created in this manner has a structure that can be exploited.

Specifically for STS(135), let $\lambda \in \mathbb{R}^{135}$ be the vector $\lambda = (e_{45}, 0_{90})^T$ in which the first 45 components of the vector are 1, and the last 90 components are 0. It is not difficult to see that the following 12 vectors $\mu_1, \ldots \mu_{12}$ all share an orbit with the point λ. (This fact can also be verified using a computational algebra package such as GAP [23]).

	$1-15$	$16-30$	$31-45$	$46-60$	$61-75$	$76-90$	$91-105$	$106-120$	$121-135$
μ_1	e	e	e	0	0	0	0	0	0
μ_2	0	0	0	e	e	e	0	0	0
μ_3	0	0	0	0	0	0	e	e	e
μ_4	e	0	0	e	0	0	e	0	0
μ_5	e	0	0	0	e	0	0	0	e
μ_6	e	0	0	0	0	e	0	e	0
μ_7	0	e	0	e	0	0	0	e	0
μ_8	0	e	0	0	e	0	0	e	0
μ_9	0	e	0	0	0	e	e	0	0
μ_{10}	0	0	e	e	0	0	0	e	0
μ_{11}	0	0	e	0	e	0	e	0	0
μ_{12}	0	0	e	0	0	e	0	0	e

We will use this orbit to create an effective constraint orbital branching dichotomy. We also use the fact that branching on the disjunction

$$(\lambda x \leq K) \vee (\mu^T x \geq K+1) \; \forall \mu \in \text{orb}(G, \lambda)$$

allows us to enumerate coverings for STS$(v/3)$ in order to solve the left-branch of the dichotomy.

Table 5.1: Computational Statistics for Solution of STS(135)

(a) Solutions of value K for STS(45)

(K)	# Sol
30	2
31	246
32	9497
33	61539
	71,284

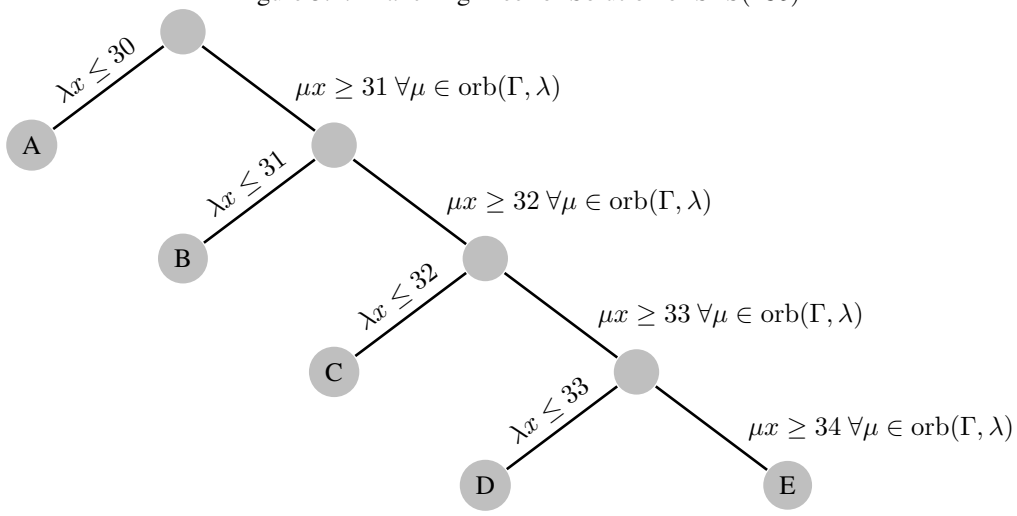
(b) Statistics for STS(135) IP Computations

K	Total CPU Time (sec)	Simplex Iterations	Nodes
30	538.01	2,501,377	164,720
31	90790.94	255,251,657	13,560,519
32	2918630.95	8,375,501,861	306,945,725
33	6243966.98	25,321,634,244	718,899,460
	9.16×10^6	3.36×10^{10}	1.04×10^9

5.3.1 STS135

This section presents computational results that prove the optimality of the cardinality 103 covering of STS(135). The optimal solution to STS(45) has a value of 30. Figure 5.2 shows the branching tree used by the constraint orbital branching method for solving STS(135). The node E in Figure 5.2 is pruned by bound, as the solution of the linear programming relaxation at this node is 103.

Figure 5.2: Branching Tree for Solution of STS(135)



A variant of the (variable) orbital branching algorithm of Ostrowski et al. [68] can be used to obtain a superset of all non-isomorphic solutions to an integer program whose objective value is better than a prescribed value K. The method works in a similar fashion to the method proposed by Danna et al. [10]. Specifically, branching and pruning operations are performed until *all* variables are fixed (nodes may not be pruned by integrality). All leaf nodes of the resulting tree are feasible solutions to the integer program whose objective value is $\leq K$. Using this algorithm, a superset of all non-isomorphic solutions to STS(45) of value 33 or less was enumerated. The enumeration required 10 CPU hours on a 1.8GHz AMD Opteron Processor and resulted in 71,284 solutions. The number of solutions for each value of K is shown in Table 5.1(a).

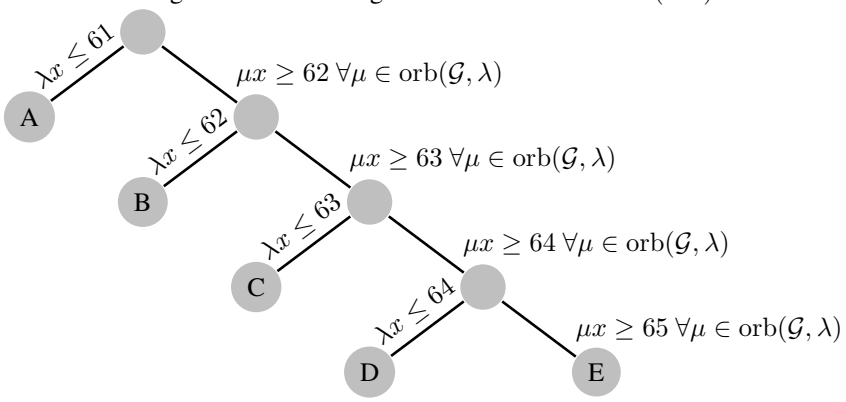
For each of the 71,284 enumerated solutions to STS(45), the first 45 variables of the STS(135) integer programming instance for that particular node were fixed. For example, the node B contains the inequalities $\mu x \geq 31\ \forall \mu \in \mathrm{orb}(\Gamma, \lambda)$, and the bound of the linear programming relaxation is 93. In order to establish the optimality of the covering of cardinality 103 for STS(135), each of these 71,284 90-variable integer programs must be solved to establish that no solution of value smaller than 103 exists. The integer programs are completely independent, so it is natural to consider solving them on a distributed computing platform. The instances were solved on a collection of over 800 computers running the Windows Operating System at Lehigh University. The computational grid was created using the Condor High Throughput Computing software [51], so the computations were run on processors that would have otherwise been idle. The commercial package CPLEX (v10.2) was used to solve all the instances, and an initial upper bound of value 103.1 was provided to CPLEX as part of the input to all instances. Table 5.1(b) shows the aggregated statistics for the computation. The total CPU time required to solve all 71,284 instances was roughly 106 CPU days, and the wall clock time required was less than two days. The best solution found during the search had value 103[1], thus establishing that the incidence-width of STS(135) is 103.

5.3.2 STS(243)

The power of constraint orbital branching on Steiner triple systems is further demonstrated by proving, for the first time, that the incidence width of STS(243) is 198. Similar to Section **??**, the branching disjunction we use for STS(243) is based on the optimal solution to smaller STS problems, in this case, STS(81). The 363 vectors sharing an orbit with $\lambda = (e_{81}, 0_{162})$ can be easily generated by GAP [23].

Figure 5.3 show the branching tree for STS(243) generated by constraint orbital branching by using disjunctions based on STS(81).

Figure 5.3: Branching Tree for Solution of STS(243)



To solve STS(243) in a manner similar to STS(135) we need to enumerate all non-isomorphic solutions to STS(81)

[1] In fact, two solutions of value 103 were found, but they were isomorphic

Node	K	# Sol	**Nodes**	**CPU Time**	**Avg Root LP**
A	61	1	95605	5279	191.18
B	62	1	116985	25975	194
C	63	53	6166988	2690150	197.78
D	64	967	967	1874	> 200
		1022	6,380,545	2,723,278	

Table 5.2: Number of non-isomorphic solutions to STS(81)

with values 61 to 64. As shown in table 5.2, there are 1022 such solutions.

The subproblem generated by each solution was solved using MINTO with orbital branching. The total CPU time required to solve all 1022 subproblems generated by solutions of STS81 was roughly 31 CPU days. Solving them in parallel, however, took only one day. For each subproblem we used an upper bound of 198.01. We found 4 solutions of STS(243) with value 198, but they were all isomorphic.

5.4 Case Study: Covering Designs

A (v, k, t)-*covering design* is a family of subsets of size k, called k-subsets. These subsets are chosen from a ground set V of cardinality v, such that every subset of size t chosen from V is contained in one of the members of the family of subsets of size k. The number of members in the family of k-subsets is the covering design's *size*. The covering number $C(v, k, t)$ is the minimum size of such a covering. Let $\mathcal{K}$ be the collection of all k-sets of V, and let $\mathcal{T}$ be the collection of all t-sets of V. An integer program to compute a (v, k, t)-covering design can be written as

$$\min_{x \in \{0,1\}^{|\mathcal{K}|}} \{e^T x \mid Bx \geq e\}, \tag{5.2}$$

where $B \in \{0,1\}^{|\mathcal{T}| \times |\mathcal{K}|}$ has element $b_{ij} = 1$ if and only if t-set i is contained in k-set j, and the decision variable $x_j = 1$ if and only if the jth k-set is chosen in the covering design.

Numerous theorems exist that give bounds on the size of the covering number $C(v, k, t)$. An important theorem that is needed to generate a useful branching disjunction for the constraint orbital branching method is due to Schönheim [75]. For a given subset $U \subseteq V$ of the ground set elements, let $\mathcal{K}(U)$ be the collection of all the k-sets of V that contain U. Margot [55] shows that the following inequality, which he calls a *Schönheim* inequality, is valid, provided that $|U| = u$ is such that $1 \leq u \leq t - 1$:

$$\sum_{i \in \mathcal{K}(U)} x_i \geq C(v - u, k - u, t - u). \tag{5.3}$$

The *Schönheim* inequalities substantially increase the value of the linear programming relaxation of (5.2).

A second important observation is that the symmetry group G for (5.2) is such that the characteristic vectors of all u-sets belong to the same orbit: if $|U'| = |U|$, then $\chi_{\mathcal{K}(U')} \in \mathrm{orb}(G, \chi_{\mathcal{K}(U)})$. These two observations taken together

indicate that the Schönheim inequalities (5.3) are good candidates for constraint orbital branching. On the left branch, the constraint

$$\sum_{i \in \mathcal{K}(U)} x_i \leq C(v-u, k-u, t-u)$$

is enforced. To solve this node, all non-isomorphic solutions to the $(v-u, k-u, t-u)$-covering design problem can be enumerated. For each of these solutions, an integer program in which the corresponding variables in the (v, k, t)-covering design problem are fixed can be solved.

On the right branch of the constraint-orbital branching method, the constraints

$$\sum_{i \in \mathcal{K}(U')} x_i \geq C(v-u, k-u, t-u) + 1 \qquad \forall U' \in \mathrm{orb}(G, \chi_{\mathcal{K}(U)})$$

may be imposed. These inequalities can significantly improve the linear programming relaxation.

5.4.1 Computational Results

We will demonstrate the applicability of constraint orbital branching using the Schönheim inequalities with an application to the $(11, 6, 5)$-covering design problem. Nurmela and Östergård [66] report an upper bound of $C(11, 6, 5) \leq 100$, and Applegate et al. [4] were able to show that $C(11, 6, 5) \geq 96$. Using the constraint orbital branching technique, we are also able to easily obtain the bound $C(11, 6, 5) \geq 96$, and ongoing computations are aimed at further sharpening the bound. The covering design numbers $C(10, 5, 4) = 51$, $C(9, 4, 3) = 25$, and $C(8, 3, 2) = 11$ are all known [31], and this knowledge is used in the branching scheme.

The branching tree used for the $(11, 6, 5)$-covering design computations is shown in Figure 5.4. In the figure, node D is pruned by bound, as the value of its linear programming relaxation is > 100. The nodes A, B, and C will be solved by enumerating solutions to a (v, k, t)-covering design problem of appropriate size. For node A, $(10, 5, 4)$-covering designs of size 51 are enumerated; for node B, $(9, 4, 3)$-covering designs of size ≤ 26 are enumerated; and for node C, $(8, 3, 2)$-covering designs of size ≤ 11 are enumerated. Table 5.3 shows the number of solutions at each node, as well as the value of the linear programming relaxation $z(\rho)$ of the parent node. The size 51 $(10, 5, 4)$-covering designs are taken from Margot [56], and the other covering designs are enumerated using the variant of the orbital branching method outlined in Section 5.3.1.

Since the value of the linear programming relaxation of the parent of node B is 95.33, if none of the 40 integer programs created by fixing the size 51 $(10, 5, 4)$-covering design solutions at node A of Figure 5.4 has a solution of value 95, then immediately, a lower bound of $C(11, 6, 5) \geq 96$ is proved. The computation to improve the lower bound for each of the 40 IPs to 95.1 required only 8789 nodes and 10757.5 CPU seconds on a single 2.60GHz Intel

Figure 5.4: Branching Tree for $C(11, 6, 5)$

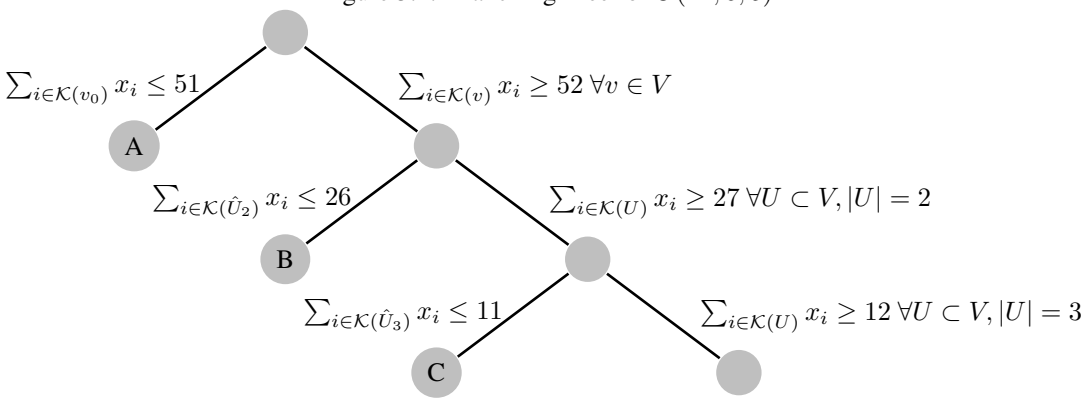


Table 5.3: Node Characteristics

Node	# Sol	$z(\rho)$
A	40	93.5
B	782,238	95.33
C	11	99

Pentium 4 CPU.

It is interesting to note that an attempt to improve the lower bound of $C(11, 6, 5)$ by a straightforward application of the variable orbital branching method of Chapter 3 was unable to improve the bound higher than 94, even after running several days and eventually exhausting a 2GB memory limit. The results on specific classes of problems show that the generality of constraint orbital branching does appear to be useful to solve larger symmetric problems.

5.5 Summary

In this chapter, we generalized the work for branching on orbits of variables (Chapter 3) to branching on orbits of constraints (constraint orbital branching). Constraint orbital branching can be especially powerful if the problem structure is exploited to identify a strong constraint on which to base the disjunction. Enumerating all partial solutions that might satisfy the constraint gives rise to an effective partition of the original problem. Using this methodology, we are, for the first time, able to establish the optimality solution for STS(135) and STS(243).

Chapter 6

Conclusions

Most ILPs do not contain any symmerty. However, symmetry is present in standard formulations of important classes of ILP problems such as graph coloring, covering design, error correcting code, and vehicle routing problems. Methods of solving these problems that are not able to exploit the symmetry have little hope of cracking problems large enough to have some real world significance.

Prior research on exploiting symmetry has relied on either adding problem-specific constraints to the ILP formulation or using computational algebra techniques to identify and remove symmetry. Adding problem-specific constraints can be very helpful. However, aside from the constraints not being applicable to general problems, these methods are rarely able to exploit all of the symmetry contained in a problem. As a result, much of the recent research has focused on more advanced methods of exploiting symmetry.

There have been few algorithms developed to fully exploit symmetry for general ILPs. Instead, many methods focus on how to fully exploit symmetry in problems with a specific structure. Some classes of problems have been found where symmetry breaking can be done in polynomial time. Unfortunately, there are not many such classes. The only previous general algorithms for solving symmetric integer linear programs are isomorphism pruning, SBDS, and SBDD. All three are effective at exploiting symmetry, but have drawbacks. Generation and storage of all constraints generated by SBDS can be very difficult for problems with large symmetry groups. SBDD requires the solution of multiple graph isomorphism problems for each node in the branch-and-bound tree. Also, SBDD is not able to prune some nodes that would be pruned by SBDS or isomorphism pruning. Isomorphism pruning, the only method implemented specifically for ILP, imposes undesirable restrictions on branching.

The main objective of this thesis was to determine better ways of exploiting symmetry for general ILPs. Specifically, it was to create algorithms for general ILPs that are adept at exploiting symmetry without restricting branching decisions. In fact, moreso than the algorithms listed above, this thesis examines how to use branching as a tool to

exploit symmetry.

6.1 Orbital Branching

Orbital branching is a simple way to detect and exploit the symmetry of an integer program when branching. Branching disjunctions are based on sets of equivalent variables, not individual variables. These disjunctions take into account fixing that can be done as a result of symmetry and in doing so create a much stronger disjunction. Orbital branching also allows for orbital fixing, a powerful method for fixing variables at nodes throughout the branch-and-bound-tree as a result of information provided by symmetry. Implemented in MINTO, orbital branching outperforms CPLEX, a state-of-the-art solver, when a high degree of symmetry is present. While it is less effective than isomorphism pruning at removing symmetry, orbital branching places no restrictions on branching decisions. This is important, as the results from Chapter 3 show that different branching methods have a significant affect on overall computation time.

Chapter 3 presents different branching strategies developed specifically for orbital branching. An interesting finding of the computational results presented for orbital branching is that branching methods that aim to preserve symmetry are shown to be effective. This goes against intuition primarily because preserving symmetry often requires branching on small orbits, branching which does not fix as many variables as other branching strategies. This result, however, may be particular to this method of exploiting symmetry.

In addition to its simplicity and flexibility, orbital branching is also able to recognize and exploit symmetry that enters the tree as a result of branching. While the computational tests have not been favorable, this method deserves further study, as the phenomena may be common in specific problem classes.

6.2 Flexible Isomorphism Pruning

François Margot's work on adapting isomorphism pruning for ILP problems was undoubtedly seminal, as it was the first algorithm that completely exploited symmetry present in a general ILP. As important as this work was, however, the method had some drawbacks. Mainly, isomorphism pruning requires the use of a rigid branching rule. Different branching strategies can have an enormous impact on overall solution times in general ILPs, and this is also true in ILPs with a large degree of symmetry.

Isomorphism pruning fully exploits the identified symmetry by providing a way to test whether a given node is symmetric to others already explored. Such nodes can be pruned. In addition to pruning nodes, isomorphism pruning offers powerful tools to fix variables using symmetry information.

Chapter 4 presents a version of isomorphism pruning that removes all branching restrictions. This version requires no additional computational efforts beyond previous versions. Removing branching restrictions allows isomorphism

pruning to be combined with orbital branching. In Chapter 4 we present computational results of the combined method using the branching rules developed in Chapter 3 for orbital branching. The results show that isomorphism pruning combined with orbital branching is a very powerful tool for exploiting symmetry.

Also in Chapter 4 a new method for fixing variables, smallest image fixing, is detailed. This method is more effective at fixing variables than orbital fixing, described in Chapter 3.

6.3 Constraint Orbital Branching

In some cases, exploiting symmetry, using the methods already described, is not enough to solve highly symmetric ILPs. Particularly, for combinatorial optimization problems, ILP formulations of highly symmetry problems are often very poor and have large integrality gaps. However, combinatorial problems often contain structures that can be exploited along with the symmetry. Constraint orbital branching accomplishes this. By generalizing orbital branching to allow the use of general linear disjunctions for branching, constraint orbital branching uses disjunctions based on optimal solutions to embedded subproblems. These disjunctions can be very effective at closing the integrality gap. Also, these disjunctions allow us to branch on optimal solutions to the subproblems. The power of this method is shown in Chapter 5 by proving, for the first time, the optimal solutions to STS(135) and STS(243).

6.4 Future Work

The branching rules discussed in Chapters 3 and 4 show the power of orbital branching and the importance of a flexible branching rule. However, more advanced rules may be needed for increasingly difficult problems. Also, specific classes of problems may require individually tailored branching rules.

One potential bottleneck for the work presented in Chapters 3 and 4 comes from the computational algebra techniques used. These techniques were not originally designed to be used in an enumeration tree format. For instance, calls to the SmallestImageSet function at a parent node and its child often perform a lot of identical computations. Section 4.2.3 mentions ways to reduce this overlap, however, more work is needed to reduce the time spent on this function.

Bibliography

[1] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2004.

[2] Aloul, Sakallah, and Markov. Efficient symmetry breaking for boolean satisfiability. *IEEETC: IEEE Transactions on Computers*, 55:549–558, 2006.

[3] R. Anbil, R. Tanga, and E.L. Johnson. A global approach to crew-pairing optimization. *IBM Systems Journal*, 31:71–78, 92.

[4] D. Applegate, E. Rains, and N. Sloane. On asymmetric coverings and covering numbers. *Journal of Combinatorial Designs*, 11:218–228, 2003.

[5] D. Avis. A note on some computationally difficult set covering problems. *Mathematical Programming*, 8: 138–145, 1980.

[6] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. *Constraints*, 7:333–349, 2002.

[7] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization (Athena Scientific Series in Optimization and Neural Computation, 6)*. Athena Scientific, February 1997. ISBN 1886529191.

[8] RE. Bixby, M. Fenelon, Z. Gu, and E. Rothberg. Mixed-integer programming: A progress report. In Martin Grotschel, editor, *Handbook of Constraint Programming*, pages 309–326. Society for Industrial and Applied Mathematic, 2004.

[9] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proc. of the Intl. Conf. on Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.

[10] E. Danna, M. Fenelon, Z. Gu, and R. Wunderling. Generating multiple solutions for mixed integer programming problems. In M. Fischetti and D. Williamson, editors, *IPCO 2007: The Twelfth Conference on Integer Programming and Combinatorial Optimization*, pages 280–294. Springer, 2007.

[11] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[12] M. Desrochers and F. Soumis. A column generation approach to the urban transit crew scheduling problem. *Transportation Science*, 23:1–13, 1989.

[13] I. M//enendez D/'iaz and P. Zabala. A polyhedral approach for graph coloring. *Electronic Notes in Discrete Mathematics*, 7:1–4, 2001.

[14] Elizabeth Dolan and Jorge Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[15] Y. Dumas, M. Desrochers, and F. Soumis. The pickup and delivery problem with time windows. *European Journal of Operations Research*, 54:7–22, 1991.

[16] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference,*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001. ISBN 3-540-42863-1. URL `http://link.springer.de/link/service/series/0558/bibs/2239/22390093.htm`.

[17] T. A. Feo and G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

[18] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. *Lecture Notes in Computer Science*, 2470:462–??, 2002. ISSN 0302-9743. URL `http://link.springer-ny.com/link/service/series/0558/bibs/2470/24700462.htm;http://link.springer-ny.com/link/service/series/0558/papers/2470/24700462.pdf`.

[19] Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2001. ISBN 3-540-42863-1.

[20] P. Foggia, C. Sansone, and M. Vento. A preformance comparison of five algorithms for graph isomorphism. *Proc. 3rd IAPR-TC15 Workshop Graph-Based Representations in Pattern Recognition*, pages 188–199, 2001.

[21] Eric J. Friedman. Fundamental domains for integer programs with symmetries. In Andreas W. M. Dress, Yinfeng Xu, and Binhai Zhu, editors, *Combinatorial Optimization and Applications, First International Conference,*

COCOA 2007, Xi'an, China, August 14-16, 2007, Proceedings, volume 4616 of *Lecture Notes in Computer Science*, pages 146–153. Springer, 2007. ISBN 978-3-540-73555-7. URL http://dx.doi.org/10.1007/978-3-540-73556-4_17.

[22] D. R. Fulkerson, G. L. Nemhauser, and L. E. Trotter. Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triples. *Mathematical Programming Study*, 2: 72–81, 1973.

[23] *GAP—Groups, Algorithms, and Programming, Version 4.4*. The GAP Group, 2004. http://www.gap-system.org.

[24] *GRAPE—GRaph Algorithms using PErmutation groups version 4.3*. The GAP Group, 2006. http://www.gap-system.org.

[25] Gent, Harvey, and Kelsey. *Groups and Constraints: Symmetry Breaking during Search*. 2002.

[26] Ian Gent, Steve Linton, and Barbara Smith. Symmetry breaking in the alien tiles puzzle. Technical report, University of St. Andrews, October 19 2000. URL http://citeseer.ist.psu.edu/373393.html;http://www.apes.cs.strath.ac.uk/reports/apes-22-2000.pdf.

[27] Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 415–430. Springer, 2002. ISBN 3-540-44120-4.

[28] Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Generic SBDD using computational group theory. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2003. ISBN 3-540-20202-1.

[29] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, pages 329–376. Morgan Kaufman, 2006.

[30] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.

[31] D. Gordon, G. Kuperberg, and O. Patashnik. New constructions for covering designs. *Journal of Combinatorial Designs*, 3:269–284, 1995.

[32] M. Hall. *Combinatorial Theory*. Blaisdell Company, 1967.

[33] H. Hamalainen, I. Honkala, S. Litsyn, and P. Östergård. Football pools—A game for mathematicians. *American Mathematical Monthly*, 102:579–588, 1995.

[34] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[35] Warwick Harvey. Symmetry breaking and the social golfer problem. In *Sym-Con-01: Symmetry in Constraints*, October 31 2001.

[36] Derek F. Holt. *Handbook of Computational Group Theory (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, January 2005.

[37] Rotman J.J. *An Introduction to the Theory of Groups*. Springer, 4th ed. edition, 1994.

[38] David Joslin and Amitabha Roy. Exploiting symmetry in lifted CSPs. In *AAAI/IAAI*, pages 197–202, 1997.

[39] V. Kaibel and M.E. Pfetsch. Packing and partitioning orbitopes. *Mathemathical Programming*, 114:1–36, 2008.

[40] V. Kaibel, M. Peinhardt, and M.E. Pfetsch. Orbitopal fixing. In *IPCO 2007: The Twelfth Conference on Integer Programming and Combinatorial Optimization*, pages 74–88. Springer, 2007.

[41] L. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6:366–422, 1960.

[42] M. Karamanov and G. Cornuéjols. Branching on general disjunctions. *submitted*, 2005.

[43] N. Karmarkar, K. Ramakrishnan, and M.Resende. An interior point algorithm to solve computationally difficult set covering problems. *Mathematical Programming, Series B*, 52:597–618, 1991.

[44] T. P. Kirkman. On a problem in combinations. *Cambridge and Dublin Mathematics Journal*, 2:191–204, 1847.

[45] Grove L.C. and Benson C.T. *Finite Reflection Groups*. Springer, 1985.

[46] J. Linderoth, F. Margot, and G. Thain. Improving bounds on the football pool problem via symmetry reduction and high-throughput computing. Submitted, 2007.

[47] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies in mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[48] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies in mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[49] Steve Linton. Finding the smallest image of a set. In *International Conference on Symbolic and Algebraic Computation*, pages 229–234. ISSAC, 2004.

[50] S. Litsyn. An updated table of the best binary codes known. In V. S. Pless and W. C. Huffman, editors, *Handbook of Coding Theory*, volume 1, pages 463–498. Elsevier, Amsterdam, 1998.

[51] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997.

[52] Luks and Roy. The complexity of symmetry-breaking formulas. *ANNALSMAI: Annals of Mathematics and Artificial Intelligence*, 41, 2004.

[53] Carlo Mannino and Antonio Sassano. Solving hard set covering problems. *Operations Research Letters*, 18(1), July 13 1995.

[54] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming*, 94:71–90, 2002.

[55] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming, Series B*, 98:3–21, 2003.

[56] F. Margot. Small covering designs by branch-and-cut. *Mathematical Programming*, 94:207–220, 2003.

[57] F. Margot. Symmetry in integer linear programming. *Tepper Working Paper*, E37, 2008.

[58] A. Martin, T. Achterberg, and T. Koch. Miplib 2003. Technical Report 05-28, ZIB,, 2005.

[59] B. D. McKay. *Nauty User's Guide (Version 1.5)*. Australian National University, Canberra, 2002.

[60] A. Mehrotra and M. A. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8:344–354, 1996.

[61] Russell D. Meller, Venkat Narayanan, and Pamela H. Vance. Optimal facility layout design. *Oper. Res. Lett*, 23 (3-5):117–127, 1998.

[62] I. Méndez-Díaz and P. Zabala. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5):826–847, 2006.

[63] Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1–2):133–163, 2001.

[64] W. H. Mills and R. C. Mullin. Coverings and packings. In *Contemporary Design Theory: A Collection of Surveys*, pages 371–399. Wiley, 1992.

[65] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[66] K. J. Nurmela and P. Östergård. Upper bounds for covering designs by simulated annealing. *Congressus Numerantium*, 96:93–111, 1993.

[67] Michiel A. Odijk and Hans van Maaren. Improved solutions to the Steiner triple covering problem. *Information Processing Letters*, 65(2):67–69, 29 January 1998.

[68] J. Ostrowski, J. Linderoth, F. Rossi, and S. Smriglio. Orbital branching. In *IPCO 2007: The Twelfth Conference on Integer Programming and Combinatorial Optimization*, volume 4517 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2007.

[69] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Pub., 1998.

[70] Justin Pearson. Symmetry breaking in constraint satisfaction with graph-isomorphism: Comma-free codes. In *AMAI*, 2004.

[71] Cameron P.J. *Permutation Groups*. London Mathematical Society, 1999.

[72] J.-F. Puget. On the satisfiability of symmetrical constrainted satisfaction problems. In J. Komorowski and Z. W. Raś, editors, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS'93)*, volume 689 of *LNAI*, pages 350–361, Trondheim, Norway, June 1993. Springer Verlag.

[73] Jean-Francois Puget. Symmetry breaking using stabilizers. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2003.

[74] Jean-Francois Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.

[75] J. Schönheim. On coverings. *Pacific Journal of Mathematics*, 14:1405–1411, 1964.

[76] E. C. Sewell. A branch-and-bound algorithm for the stability number of a sparse graph. *INFORMS Journal on Computing*, 10:438–447, 1998.

[77] H. D. Sherali and J. C. Smith. Improving zero-one model representations via symmetry considerations. *Management Science*, 47(10):1396–1407, 2001.

[78] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *ECAI*, pages 249–253, 1998.

[79] P.H. Vance. *Crew scheduling, cutting stock, and column generation: solving huge integer programs.* PhD thesis, Georgia Institute of Technology, 1993.

[80] P.H. Vance, C. Barnhart, E.L. Johnson, and G.L. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3:111–130, 1994.

[81] P.H. Vance, C. Barnhart, E.L. Johnson, and G.L. Nemhauser. Airline crew scheduling: a new formulation and decomposition algorithm. *Operations Research*, 45:188–200, 1997.

[82] Douglas B. West. *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, 2000.