5-2015

Symmetry Detection in Integer Linear Programs

Jonathan David Schrock
University of Tennessee - Knoxville, jschroc1@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Jonathan David Schrock entitled "Symmetry Detection in Integer Linear Programs." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mathematics.

James A. Ostrowski, Major Professor

We have read this thesis and recommend its acceptance:

Abner J. Salgado, Michael W. Berry

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Symmetry Detection in Integer Linear Programs

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Jonathan David Schrock

May 2015

I would like to dedicate this thesis to my parents and the rest of my family for the support they have given me through all of my schooling and beyond.

Acknowledgements

Foremost, I would like to thank my advisor Dr. Jim Ostrowski for his support and guidance on my masters work. He helped me decide on a topic, and was always willing to work with me to understand the background material and make progress on my work.

I would also like to thank the rest of my thesis committee: Dr. Abner Salgado and Dr. Michael Berry for the encouragement and willingness work with me.

ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE REGENTS OF THE UNIVERSITY OF MICHIGAN SHALL NOT BE LIABLE FOR ANY DAMAGES, INCLUDING SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WITH RESPECT TO ANY CLAIM ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE SOFTWARE, EVEN IF IT HAS BEEN OR IS HEREAFTER ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Abstract

Symmetry has long been recognized as a major obstacle in integer programming. Unless properly recognized and exploited, the branch-and-bound tree generated when solving highly symmetric integer programs (IPs) can contain many identical subproblems, resulting in a waste of computational effort. Effective methods have been developed to exploit known symmetry. This thesis focuses on improving methods that compute the symmetry group of an IP. In the literature, computing the symmetry group of an IP is performed by generating a graph with a similar structure as the IP, and then computing the automorphism group of the graph. Unfortunately, these graphs may be much larger than the IP problem, resulting in a possible waste of computational resources. The approach of this thesis is to detect the group of an edge-colored graph generated by the IP. This avoids the need for additional nodes to track coefficients, reducing the search space. Actually finding the group will be done by adapting and expanding a symmetry detection algorithm to work with these edge-colored graphs.

Table of Contents

1 Introduction **1**

1.1 Preliminaries 2

2 Background **4**

2.1 *Saucy* 4

2.1.1 Refinement 5

2.1.2 Problem Breakdown 5

2.1.3 Symmetry Search 6

2.1.4 Orbit Pruning 8

2.1.5 Example 8

2.2 Integer Program to Graph 10

3 Methods **13**

3.1 Modifications to *Saucy* 13

3.1.1 Example 14

3.2 Comparing the Algorithms 15

4 Results **16**

4.1 Graph Sizes 16

4.2 Timing Comparisons 18

4.3 Conclusions 20

Bibliography **22**

Vita **25**

List of Tables

2.1 Walk-through of the search tree for figure 2.1 9

4.1 Size differences between edge-colored and non edge-colored graphs . . 17

List of Figures

2.1 Simple graph to demonstrate *saucy* search 8
2.2 Non edge-colored graph generated by IP in equation 2.1 11
2.3 Edge-colored graph generated by IP in equation 2.1 12

4.1 Timings for problems BI-SC-10 through BI-SC-17 18
4.2 Timings for problems BI-SC-18 through BI-SC-4 19
4.3 Timings for problems BI-SC-5 through qp2 19
4.4 Timings for problems dano3mip_c through mod011_c 20

Chapter 1

Introduction

An integer program (IP) is symmetric if its variables can be permuted without changing the structure of the system. Such IPs often arise when constructing problems in combinatorics or optimization. For example, graph coloring, scheduling of jobs on parallel identical machines, covering design or codes construction are all problems which contain symmetries. Even a modestly sized problem can be difficult to solve using traditional branch-and-bound or branch-and-cut algorithms if it has a large symmetry group. The trouble arises as a result of many subproblems in the enumeration tree being isomorphic, leading to wasted computational effort [11].

However, effective methods have been developed to exploit known symmetry. In the literature, computing the symmetry group of an IP and similar problems is performed by generating a graph with a similar structure, and then computing the automorphism group of the graph [4, 5, 11, 14]. Unfortunately, these graphs may be much larger than the IP problem, resulting in a possible waste of computational resources. This paper focuses on efforts to improve this method of computing the symmetry group of an IP. The approach chosen is to adapt an existing symmetry detection algorithm to work with edge-colored graphs. This would allow an IP to be more directly converted into a graph and reduce the search space for the automorphism group. In particular, the symmetry detection algorithm first proposed

in [7] and updated in [6] was chosen. This algorithm was designed to work efficiently with sparse graphs which is ideal since graphs produced by IPs are sparse.

1.1 Preliminaries

This section provides basic definitions and notation used in this paper. Define $[n]$ to be the set $\{1, 2, ..., n\}$. Let Π^n be the set of all permutations of $[n]$. A permutation $\gamma \in \Pi^n$ is represented in cycle notation. That is $\gamma = (i\ j\ k)$ indicates that $i^\gamma = j$, $j^\gamma = k$, and $k^\gamma = i$. Clearly this can be extended to a cycle permuting any number of elements of $[n]$, and the cycles can be composed. Let ι be the permutation such that $j^\iota = j$. This is the **identity permutation**.

A graph G is an ordered pair (V, E) where V is the set of vertices and E is the the set of edges. A **symmetry** γ is a permutation of V such that $G^\gamma = G$. This is equivalent to $E^\gamma = E$ since $V^\gamma = V$ by definition. A permutation γ **repects a partition** π of V if v and v^γ are in the same cell of π for each $v \in V$. The set of all symmetries of a graph G which respect a partition π is called the **automorphism group** of G under π and is written as $Aut(G)_\pi$. A graph with $V = [n]$ can have as few as one symmetry, ι, and up to $n!$. This group can be represented by a subset of at most $n - 1$ symmetries which **generates** the group through every possible composition of its elements. If B is a subgroup of $Aut(G)_\pi$, then it partitions $Aut(G)_\pi$ into equally sized **cosets**. A theorem of group theory states that each coset of B can be generated by composing a **representative** of the coset with every element of B [6].

For some positive integer k, consider the linear program

$$\begin{aligned} \min \quad & c^T x \\ & Ax \geq b \\ & x \in [k]^n, \end{aligned} \tag{1.1}$$

where A is an $m \times n$ matrix, b is an m-vector, c is an n-vector, and they all have rational entries. If $\gamma \in \Pi^n$ and $\sigma \in \Pi^m$, then $A(\sigma, \gamma)$ is the matrix resulting from

permuting the rows of A by σ and the columns by γ. Thus, the **symmetry group** of the linear program is the set of permutations

$$\mathcal{G}(A, b) := \{\gamma \in \Pi^n : \exists \sigma \in \Pi^m \text{ such that } A(\sigma, \gamma) = A \text{ and } b^\sigma = b\}.$$

If $\gamma \in \mathcal{G}(A, b)$ and $\hat{x}$ is a feasible solution for the linear program in Equation 1.1, then $\hat{x}^\gamma$ is also feasible [13]. For example, consider the IP

$$\begin{array}{lccccc} \min & x_1 & +x_2 & +x_3 & +x_4 & \\ & x_1 & +x_2 & & & \geq 1 \\ & & x_2 & +x_3 & & \geq 1 \\ & & & x_3 & +x_4 & \geq 1 \\ & x_1 & & & +x_4 & \geq 1 \\ & x_1, & x_2, & x_3, & x_4 & \in \{0, 1\}. \end{array} \tag{1.2}$$

The structure of this problem suggests symmetry in the variables. This program is small enough that the symmetry group can easily be computed. In fact, the symmetry group, $\mathcal{G}(A, b)$, for this program consists of ι, (1 2 3 4), (1 3)(2 4), (1 4 3 2), (1 3), (1 4)(2 3), (2 4), (1 2)(3 4). Showing $\mathcal{G}(A, b)$ to be a group is straightforward. Note, there are 7 solutions to this program: $(1, 0, 1, 0)$, $(0, 1, 0, 1)$, $(1, 1, 1, 0)$, $(1, 1, 0, 1)$, $(1, 0, 1, 1)$, $(0, 1, 1, 1)$, $(1, 1, 1, 1)$. Consider the solution $(1, 0, 1, 0)$. Applying any of the permutations in $\mathcal{G}(A, b)$ produces $(1, 0, 1, 0)$ or $(0, 1, 0, 1)$ which are both solutions. It is a simple procedure to verify this for each of the solutions and permutations. This example comes from [11].

It is $\mathcal{G}(A, b)$ that we are interested in when searching for the symmetries of an IP. Essentially, it contains all permutations of the columns of the constraint matrix which can preserve the original problem. This is equivalent to the permutations of the variables x_i which preserve the IP.

Chapter 2

Background

As stated in the introduction, symmetries of an IP are often found indirectly by searching for the symmetries of a graph related to the problem. These symmetries have been shown to be isomorphic to $\mathcal{G}(A, b)$ [5]. This section will introduce the search method used for this work, and will demonstrate how to convert an IP into a graph.

2.1 *Saucy*

Saucy is software which implements a symmetry detection algorithm for simple, non edge-colored graphs. It is based on a previous program, *nauty*, described in [12]. The aim of these algorithms is to find a generating set for the group $Aut(G)_\pi$ described in section 1.1. The goal of *saucy* was to develop a program which performed better on sparse graphs utilizing sparse data structures [7]. Additional speed-up was gained by recognizing that sparsity could also be exploited in the symmetries themselves. That is, more performance can be attained by expecting that symmetries leave the majority of the vertices fixed [6]. Since the graphs generated by an IP are sparse, *saucy* was a reasonable base for this work.

2.1.1 Refinement

Saucy initially reads in the vertex set of a graph which is partitioned by the colors of the vertices. The first step in detecting symmetries is to differentiate the vertices as much as possible. This is accomplished by using properties of vertices which do not change under symmetry, or **vertex invariants**. A partition π can be **refined**, further subdivided, by partitioning each cell S of π based on these vertex invariants. One such invariant is the **connection function** c. Given a vertex v and an invariant set of vertices S, $c(v, S)$ is the number of elements of S which are neighbors of v. A set of vertices is called invariant if $S^\gamma = S$ for all $\gamma \in Aut(G)_\pi$. This is trivially satisfied by all cells in π. So we can select some cell S of π and calculate $c(v, S)$ for every v, and further partition π based on these values. This produces a finer partition π' without removing any working symmetries. Thus, $Aut(G)_\pi = Aut(G)_{\pi'}$ and the cells of π' can be used to further refine itself. This process continues until the partition can no longer be refined. The partition at this point is called **equitable**. The formal algorithm for this refinement process is based on Hopcroft's algorithm for minimizing the number of states in a finite automaton [2]. It is described in more detail in [7, 12]. The core of the algorithm is using one cell of the partition to attempt to split every other cell. This cell is called the **inducing cell**. To improve performance in *saucy*, the program only attempts to split cells that actually have a connection to the inducing cell. Let S be the inducing cell. This is accomplished by first computing $k(v, S)$, the number of connections vertex v has to S. Then, for each cell T in the partition with at least one connection to S, T is split up based on the values of k.

2.1.2 Problem Breakdown

If refining a partition π results in a **discrete** partition π', every vertex is in its own cell, then $Aut(G)_\pi = \{\iota\}$. This is because the identity is the only permutation which respects such a π'. However, if π' is not discrete, then there exists a cell in π' with more than one element. Let $T = \{v_1, ..., v_j\}$ be such a cell. Call this the **target cell**.

For a symmetry γ of G which respects π', $v_1^\gamma \neq v$ for all $v \notin T$. So, the image of v_1 is an equivalence relation on $Aut(G)_{\pi'}$, that forces a partition of $Aut(G)_{\pi'}$ into j subsets of symmetries. Since the refinement process is not perfect, some of these subsets might be empty. Consider the first subset of permutations γ which fix v_1. This subset can be expressed with vertex partitions by **distinguishing** the vertex v_1. This is done by letting $\hat{\pi} = \pi'$ except with T replaced by $\{v_1\}$ and $T - \{v_1\}$. In this way, we have created a subproblem of the original. To find generators for $Aut(G)_\pi$, we must first find them for $Aut(G)_{\hat{\pi}}$. This recursion terminates once the partition becomes discrete. After finding the generators for $Aut(G)_{\hat{\pi}}$, we search for other symmetries respecting π that map v_1 onto every other element of T. To maintain a polynomial bound on the number generators, only a single symmetry is sought for each $v \in T - \{v_1\}$ that maps v_1 to v. This restriction is justified by recognizing that $Aut(G)_{\hat{\pi}}$ is a subgroup of $Aut(G)_\pi$ and that we are searching for cosets of this subgroup.

2.1.3 Symmetry Search

Let $T = \{v_1, v_2, ...\}$ be a target cell. Suppose we are searching for a representative of the coset containing symmetries γ such that $v_1^\gamma = v_2$. This coset might not exist which requires proof that such a γ does not exist. To do this, *saucy* utilizes a backtracking search over partial permutations. A **partial permutation** ρ is an isomorphism between two partitions of vertices. The mapping ρ is called partial since it might not be well defined. Let π_1 and π_2 be partitions such that $\pi_1^\rho = \pi_2$. If $S_1 \in \pi_1$ and $S_2 \in \pi_2$ such that $S_1^\rho = S_2$, then $v \in S_1$ could map to any vertex in S_2. If they are both singleton sets, the mapping is explicit. Otherwise, it is ambiguous and decisions must be made. This suggests that ρ represents a set of permutations. Suppose T is some target cell of an equitable partition π and we are searching for some γ such that $v_1^\gamma = v_2$. A partial permutation ρ is constructed by creating two new partitions π_1 and π_2. This is accomplished by distinguishing v_1 from T in π_1 and v_2 from T in π_2. Finally, ρ is defined so that $\{v_1\} \mapsto \{v_2\}$ and $T - \{v_1\} \mapsto T - \{v_2\}$ and all

other cells of π_1 map to the corresponding cells in π_2. A partial permutation becomes better specified by propagating the constraints imposed by these mappings. This propagation is performed by partition refinement as discussed earlier. Both π_1 and π_2 can possibly be refined since splitting T might have caused the partitions to no longer be equitable. The partitions are refined simultaneously and every time cells are split, ρ is updated. For cells $S_1 \in \pi_1$ and $S_2 \in \pi_2$, they **divide isomorphically** if, given equivalent inducing cells T_1 and T_2, S_1 has the same number of vertices with no connection to T_1 as S_2 has to T_2, and so on for each possible number of connections.

After refinement completes, ρ is examined. It is possible that some cell in π_1 does not divide isomorphically with its equivalent cell in π_2. This implies that ρ cannot represent a symmetry of the vertices and the algorithm must backtrack. Another possibility is that $S_1 = S_2$ for every nonsingleton set where $S_1^\rho = S_2$. In other words, π_1 and π_2 only differ in their singleton elements. For this, ρ creates a well-defined permutation γ such that $v_1^\gamma = v_2$, and $v^\gamma = v$ otherwise. *Saucy* then attempts to verify $G^\gamma = G$. If successful, the search terminates. If not, the algorithm must backtrack. The last possibility is that π_1 and π_2 differ in at least one pair of equivalent nonsingleton sets. This forces another mapping decision. Some nonsingleton set $T_1 \in \pi_1$ selected as a target cell, and hence $T_1^\rho = T_2$ is selected in π_2. A vertex $v_1 \in T_1$ is also chosen. By construction, any symmetry must map v_1 to an element of T_2. So, the size of the target cells is the branching factor at each point in the search tree. An element v_2 is chosen from T_2, construct the possible mapping $\{v_1\}^\rho = \{v_2\}$ and propagate as before. If the algorithm backtracks to a decision point, an alternative mapping is chosen and the search continues. However, if all possible mappings have failed, *saucy* backtracks again. If it backtracks from the root of the tree, the search ends without finding a symmetry.

2.1.4 Orbit Pruning

The **orbit partition** $\theta(\Gamma)$ of a permutation group Γ is a partition of the set underlying Γ such that, if x and y are in the same cell of $\theta(\Gamma)$, then there exists some $\gamma \in \Gamma$ such that $x^\gamma = y$. This can be utilized in the subproblem decomposition. Suppose π is a vertex partition with some cell $T = \{v_1, v_2, ...\}$ as a target. Let $\hat{\pi}$ be the partition formed by distinguishing v_1, and suppose $Aut(G)_{\hat{\pi}}$ is known. Now search for a representative γ of the coset where $v_1^\gamma = v$ for all $v \in T - \{v_1\}$. As symmetries are found, update θ accordingly. When considering a mapping $v_1^\gamma = v$, if v_1 and v are in the same cell in θ, then there is no need to find such a γ since this indicates that such a mapping exists, and the entire coset can already be generated. The backtracking search for symmetries can also use orbit pruning. Though, θ cannot be used in this situation because symmetries might have been discovered which do not respect vertex partitions at a particular decision point. So, a collection of generators is kept, and the orbit partition of those respecting the vertex partition is constructed and used to check a candidate mapping. This is only done during the uncommon case of needing to backtrack [6].

2.1.5 Example

Consider the simple, vertex colored graph in figure 2.1. This example graph and search tree come from [6]. Table 2.1 shows the breakdown of the *saucy* search.

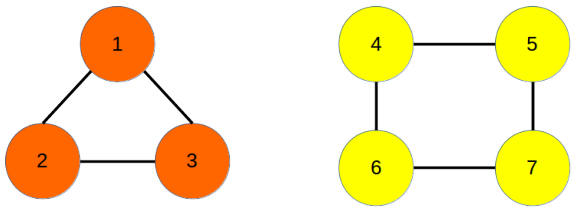

Figure 2.1: Simple graph to demonstrate *saucy* search

In the decomposition, cells in a partition are separated by | and target cells are indicated by an underline. If a partition has a target cell, it is equitable. As stated

Table 2.1: Walk-through of the search tree for figure 2.1

Decomposition	Subproblem 1	Subproblem 2	Subproblem 3	Subproblem 4
[1 2 3|4 5 6 7] ⇓ [1|2 3|4 5 6 7] ⇓ [1|2|3|4 5 6 7] ⇓ [1|2|3|4|5 6 7] ↓ [1|2|3|4|5 6|7] ⇓ [1|2|3|4|5|6|7]	[1|2|3|4|5 6|7] ⇓ [1|2|3|4|5|6|7] [1|2|3|4|6|5|7]	[1|2|3|4 5 6 7] ⇓ [1|2|3|4|5 6 7] [1|2|3|5|4 6 7] ↓ [1|2|3|4|5 6|7] [1|2|3|5|4 7|6] ⇓ [1|2|3|4|5|6|7] [1|2|3|5|4|7|6]	[1|2 3|4 5 6 7] ⇓ [1|2|3|4 5 6 7] [1|3|2|4 5 6 7]	[1 2 3|4 5 6 7] ⇓ [1|2 3|4 5 6 7] [2|1 3|4 5 6 7] ⇓ [1|2|3|4 5 6 7] [2|1|3|4 5 6 7]

above, we are trying to reach a discrete partition, and if the partition is equitable, it cannot be refined further. So a target cell must be chosen and a vertex distinguished. In this decomposition, ⇓ represents partitions formed by distinguishing an element of a target cell, and ↓ represents the refinement of a non-equitable partition.

Section 2.1.2 mentions that distinguishing an element of a target cell creates a subproblem that must be solved. This is indicated in the table. Consider subproblem 1. We initially distinguished 5. Now, symmetries must be discovered which map 5 to every other element of the target cell. In this case, we only need to find γ such that $5^\gamma = 6$. With this partial permutation, the isomorphic partition pair above results. This is the permutation (5 6) which is a symmetry of the graph.

For subproblem 2, search for representatives of the cosets of the symmetry group found in the previous subproblem. These are found by searching for mappings which take 4 into each of the other elements in the target cell. Start by looking for a mapping such that $4^\gamma = 5$. After some refinement, the equitable partition pair above results. Since $\{5, 6\} \neq \{4, 7\}$, a target cell must be chosen, an element distinguished, and the partition further refined. This leads to the discrete partition pair which corresponds

to the permutation (4 5)(6 7). This is a symmetry of the graph. It is unnecessary to search for the representatives of the other two cosets due to orbit pruning.

Subproblem 3 is easier than the previous one. In this case, we are looking for a mapping such that $2^\gamma = 3$. Since the resulting partition pair only differs in its singleton elements, the partial permutation is completely defined. There is no need to further refine this partition pair. We are left with the graph symmetry (2 3).

Finally, we search for a representative of the coset where $1^\gamma = 2$. Distinguishing elements and refining leads to the equitable partition pair at the bottom. Again, these partitions differ only in their singletons. This results in the symmetry (1 2). The coset represented by $1^\gamma = 3$ is removed by orbit pruning. All of this leaves us with the generators $\{(5\ 6), (4\ 5)(6\ 7), (2\ 3), (1\ 2)\}$.

2.2 Integer Program to Graph

When converting an IP to a graph, we will focus on constructing the **restricted IP graph** discussed in [5]. As mentioned in that paper, this restricts the search space to symmetries typically focused on in the literature. Let the set of variables in the IP, $\{x_1, .., x_n\}$, and the set of constraints, $\{c_1, ..., c_m\}$, be the disjoint sets representing a bipartite graph. A vertex representing x_j connects to a vertex representing c_i if the (i, j) position of the constraint matrix A is non-zero. This edge is then colored dependent on the value of this position in the matrix. Finally, the variable vertices can be colored based on their coefficients in the objective function, and the constraint vertices can be colored based on their value. Most algorithms that detect symmetries in graphs work with simple, non edge-colored graphs. Thus, when creating a graph from an IP, edges are not colored, but instead a new vertex is created to represent the coefficient at $A_{i,j}$. These new vertices are then given a color dependent on the coefficient value. So, vertices representing matrix coefficients with the same value receive the same color. To reduce the graph somewhat, we let an edge have an

assumed color for 1. Thus, a vertex is not needed for any coefficient in the constraint matrix equal to 1.

To show construction of the graph from an IP, a simple example will be presented.

$$\begin{array}{lrrrl} \min & x_1 & +x_2 & +x_3 & \\ & 2x_1 & +x_2 & +2x_3 & \geq 1 \\ & x_1 & +2x_2 & +x_3 & \geq 1 \\ & x_1, & x_2, & x_3 & \in \{0,1\}. \end{array} \tag{2.1}$$

Consider the IP in equation 2.1, and create a non edge-colored graph representing it. So, we have vertices for the variables $\{x_1, x_2, x_3\}$, vertices for the constraints $\{c_1, c_2\}$, and vertices for the nonzero coefficients which do not equal 1 $\{A_{1,1}, A_{1,3}, A_{2,2}\}$. Since $A_{1,1} = 2$, there is a connection between the vertices for x_1 and $A_{1,1}$ and the vertices for $A_{1,1}$ and c_1. Similarly, there are connections between x_3 and $A_{1,3}$ and $A_{1,3}$ and c_1. However, $A_{1,2} = 1$ indicates a direct connection from the vertex for x_2 and c_1. Continuing this procedure produces the graph in figure 2.2.

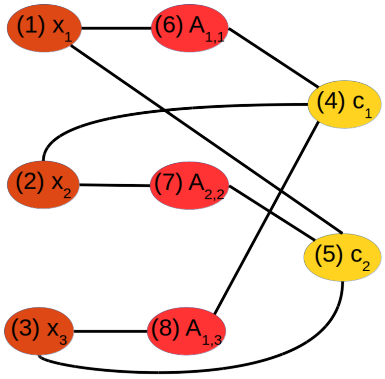

Figure 2.2: Non edge-colored graph generated by IP in equation 2.1

Though this is a small example, it indicates that a graph could easily grow much larger than the original problem with more coefficients. To avoid this, we worked to adjust *saucy* to work with edge-colored graphs. Using 2.1, create an edge-colored graph representing it. This is a more straightforward procedure than the previous example. We only have the sets $\{x_1, x_2, x_3\}$ and $\{c_1, c_2\}$. Like in the non edge-colored

case, $A_{1,2} = 1$ indicates there is a connection between the verticess for x_2 and c_1. The difference comes for the coefficients which are not 1. So for $A_{1,1} = 2$, An edge connects x_1 and c_1 with the edge colored purple. Continue this procedure to produce the graph in figure 2.3.

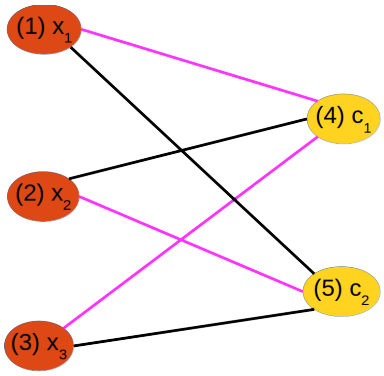

Figure 2.3: Edge-colored graph generated by IP in equation 2.1

To demonstrate symmetry detection of an IP, run the *saucy* search on the graph in figure 2.2. Since the vertices have different colors, we begin with the partition [1 2 3|4 5|6 7 8]. Let $S = \{1, 2, 3\}$ be the initial inducing cell. This splits $\{4, 5\}$ because $k(4, S) = 1$ and $k(5, S) = 2$. However, $k(6, S) = k(7, S) = k(8, S) = 1$. So, the partition becomes [1 2 3|4|5|6 7 8] and $\{4\}$ becomes an inducing cell. We proceed in a similar manner to split $\{1, 2, 3\}$ and $\{6, 7, 8\}$. This leads to the equitable partition [2|1 3|4|5|6 8|7]. That is to say, no cell in the current partition can split another. Choose $\{1, 3\}$ as the target cell and distinguish 1. This gives the partition [2|1|3|4|5|6 8|7]. Choose $\{1\}$ as the inducing cell which splits $\{6, 8\}$. Thus we have the discrete partition [2|1|3|4|5|6|8|7]. Return to the first subproblem and look for a γ such that $1^\gamma = 3$. This will lead to the discrete partition [2|3|1|4|5|8|6|7]. We have an isomorphic partition pair that gives the permutation (1 3)(6 8). Since there are no additional subproblems, the symmetry group for this graph is $\{\iota, (1\ 3)(6\ 8)\}$.

Chapter 3

Methods

As described in section 2.1, *saucy* works with non edge-colored, simple graphs. However, converting an IP into a graph can result in a larger search space than the original problem had. To overcome this issue and attempt to reduce run time, *saucy* was altered to work with edge-colored graphs. Here, we will describe the adjustments made to *saucy* and the methods used to compare the algorithms. All tests and development were performed on a MacBook Pro running OS X 10.8.5 with 8 GB of 1600 MHz DDR3 RAM and a 2.3 GHz Intel Core i7 processor.

3.1 Modifications to *Saucy*

The first thing required for our goal was to make *saucy* accept as input and run with edge-colored graphs. This was easily achieved by adding a structure to hold this data and to test the edge colors when checking the validity of a symmetry. Some additional alterations were needed to make the IO components and other utilities compatible with edge-colored graphs.

The primary alteration made to the algorithm was to attempt to split cells based on the number of connections of a specific color. Conceptually, we altered the k function, described in section 2.1, to differentiate on the number of connections of a specific color. This made it possible to generate finer partitions at each subproblem in

the search tree. Additionally, this removed the need for a vertex to represent the color of an edge since that information was being used directly. Thus, the overall graph size could be greatly reduced. Most of the remaining algorithm was left unchanged including the search tree and supporting methods. This simplified the alterations while leveraging existing code which worked well.

3.1.1 Example

Again, consider the IP described in equation 2.1, and create an edge-colored graph as described in section 2.2. This produces the graph in figure 2.3. Since the vertices have different colors, we begin with the partition [1 2 3|4 5]. Let $S = \{1, 2, 3\}$ be the initial inducing cell. We have two edge colors in this graph, black and purple. Of course in code, these colors are numeric. Since $\{4, 5\}$ is the only other cell in the partition, we attempt to split it. We calculate $k(4, S, \text{purple}) = 2$ and $k(5, S, \text{purple}) = 1$. Clearly, we split into $\{4\}$ and $\{5\}$, and it is not necessary to check black connections in this case. Here, $\{4\}$ becomes our new inducing cell. We proceed in a similar manner to split $\{1, 2, 3\}$. This leads to the equitable partition [1 3|2|4|5]. The cell $\{1, 3\}$ is the only non-singleton in the partition so it is chosen as the target cell. Distinguish 1 from the cell giving the discrete partition [1|3|2|4|5]. So the recursion ends and we return to the first subproblem. Look for a mapping so that $1^\gamma = 3$. This gives [3|1|2|4|5]. Comparing the partitions gives the symmetry (1 3). This symmetry clearly preserves the graph structure and original problem. Since there are no additional subproblems, the symmetry group is $\{\iota, (1\ 3)\}$. Note that this is nearly the same as the result produced by the non edge-colored graph. The symmetry in the vertices representing the variables is the same, which is what we are looking for. The non edge-colored graph produced symmetries which also tracked the edge colors. This can be ignored when applying the symmetry to the original IP.

3.2 Comparing the Algorithms

To compare the algorithms, a set of readily available IPs was selected from [3, 10, 8]. An attempt was made to collect IPs of differing sizes. Additionally, only problems with multiple coefficients were considered since these are the only problems where an improvement was expected. Once selected, each problem was converted into a simple graph and an edge-colored graph as described in 2.2. To do this, the IPs were represented in MPS format to make them easier to work with. For simplicity, the chosen problems already existed in this format. A description of the MPS format can be found at [1]. From this representation, a problem could be read into a conversion program which utilized *COIN-OR*, [9], to access the coefficient matrix, objective function, and contraints of the IP in order to build the graphs.

Chapter 4

Results

The primary goals of this work were to reduce the size of the graph resulting from an IP and attempt to improve the runtime of the symmetry detection algorithm. As stated earlier, the approach was to create an easier, more direct conversion from IPs to graphs using edge-colored graphs. Then, an existing symmetry detection algorithm was modified to work with these graphs and leverage the additional data available through the colorings. Both goals were at least partially achieved.

4.1 Graph Sizes

Table 4.1 lists various problems used to compare the algorithms and the differences in their sizes between edge-colored and non edge-colored variants. As can be seen, the edge-colored graphs are smaller and significantly so in some cases. For example, `arki001_c` has 21217 vertices in the original graph construction, but only 2436 in the edge-colored version. Clearly, this reduction in size decreases the number of cells in a vertex partition in the search tree as discussed in section 2.1. So, there are fewer cells to compare when determining the connections between them.

Table 4.1: Size differences between edge-colored and non edge-colored graphs

	Edge-colored		Not edge-colored	
Problem	Vertices	Edges	Vertices	Edges
BI-SC-10	1632	6622	7366	12356
BI-SC-11	1502	5872	6551	10921
BI-SC-12	1559	6348	7056	11845
BI-SC-13	1566	6461	7176	12071
BI-SC-14	1566	6461	7176	12071
BI-SC-15	1553	6231	6933	11611
BI-SC-16	1624	6496	7232	12104
BI-SC-17	1554	6299	7002	11747
BI-SC-18	1594	6409	7132	11947
BI-SC-19	1594	6409	7132	11947
BI-SC-2	1641	6864	7617	12840
BI-SC-20	1625	6581	7324	12280
BI-SC-21	1616	6477	7211	12072
BI-SC-22	1563	6501	7213	12151
BI-SC-3	1629	6544	7285	12200
BI-SC-4	1616	6477	7211	12072
BI-SC-5	1617	6509	7244	12136
BI-SC-6	1617	6488	7223	12094
BI-SC-7	1619	6494	7231	12106
BI-SC-8	1622	6572	7312	12262
BI-SC-9	1630	6559	7301	12230
mod010_c	2801	11203	2956	11358
qp1	1327	97	1374	144
qp2	1327	97	1374	144
dano3mip_c	17075	79655	77535	140115
mitre_c	12778	39704	31062	57988
mkc_c	8736	17038	17553	25855
rentacar_c	16341	41842	42767	68268
arki001_c	2436	20439	21217	39220
p2756_c	3511	8937	10263	15689
swath_c	7689	34965	28172	55448
mod011_c	15438	22254	22129	28945

4.2 Timing Comparisons

To produce timings for the algorithms, each non edge-colored graph was run through *saucy* 1000 times, and likewise the edge-colored graphs were run through the modified algorithm. The data sets were collected by a Python script that utilized *matplotlib* to produce plots comparing the timings of the original *saucy* next to the modified algorithm. The timing results can be seen in the following figures. For each chosen IP, the range of run times is presented with the median displayed at the top for easier comparison. Additionally, the number of different coefficients in the IP is recorded next to the name of the IP.

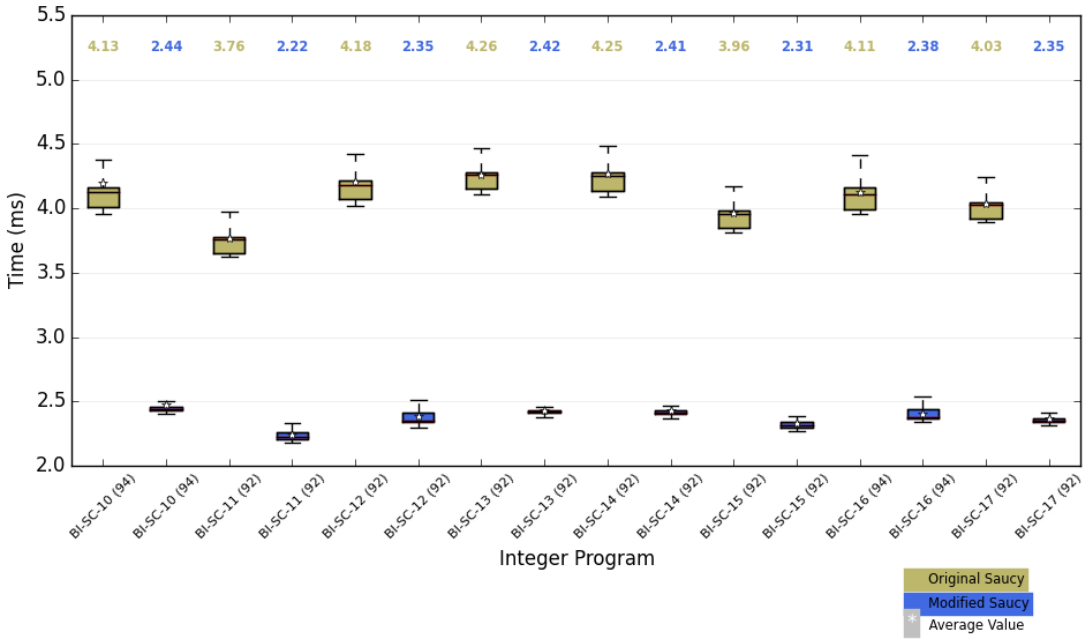

Figure 4.1: Timings for problems BI-SC-10 through BI-SC-17

As the plots suggest, the modified version of the algorithm generally runs faster than the original on IPs with different coefficients. However, as can be seen in figures 4.3 and 4.4, there are a couple IPs for which this doesn't hold. Specifically, the modified algorithm performs worse than the original for mod010_c and mod011_c.

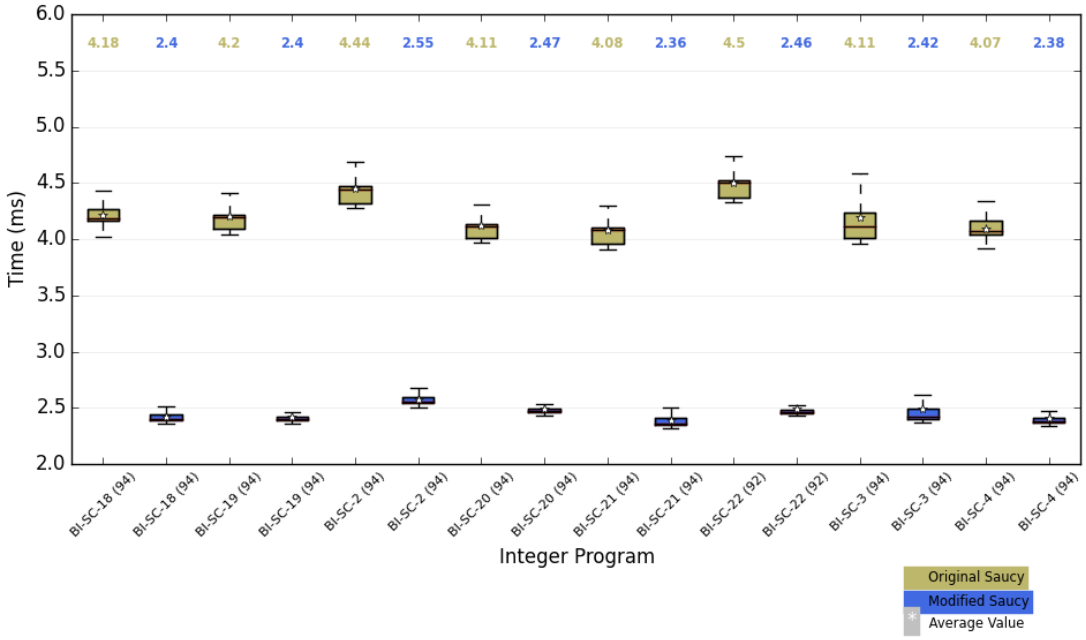

Figure 4.2: Timings for problems BI-SC-18 through BI-SC-4

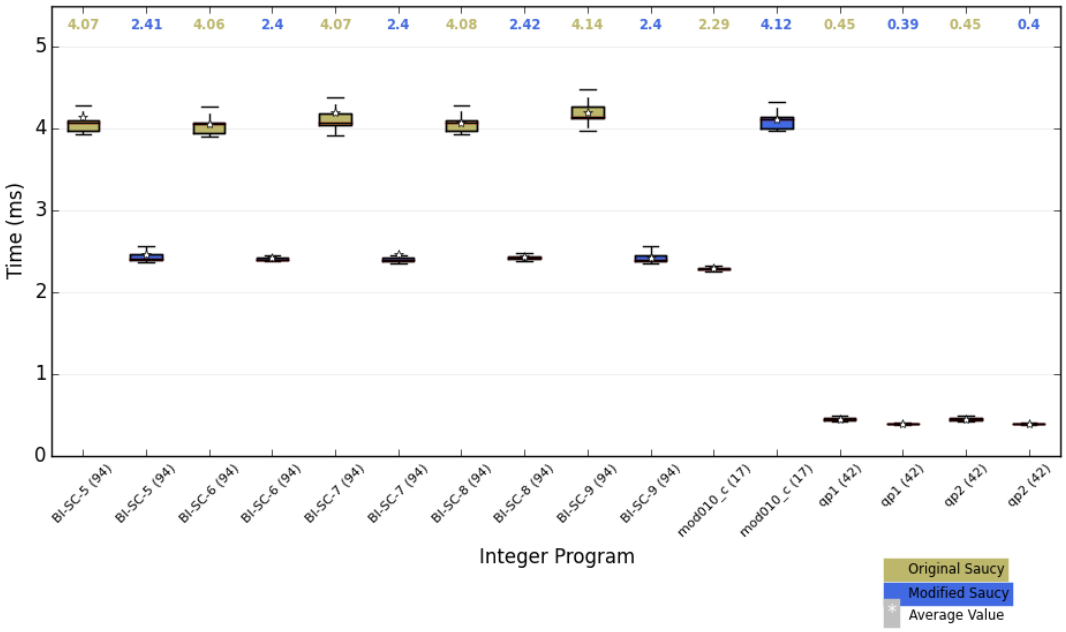

Figure 4.3: Timings for problems BI-SC-5 through qp2

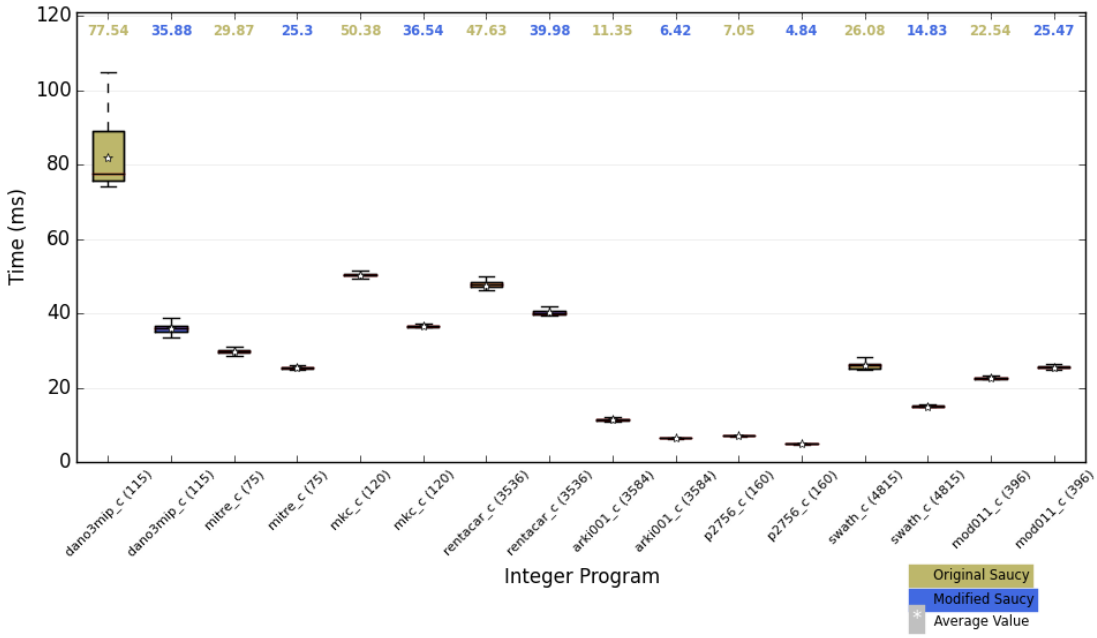

Figure 4.4: Timings for problems dano3mip_c through mod011_c

There are also a few problems which have only a slight improvement such as qp1 and qp2 in figure 4.3.

4.3 Conclusions

Note, when attempting to solve an IP, symmetry detection is sometimes performed at each node in a branch-and-bound tree. So, for large problems, the time spent searching for symmetries adds up. Thus, even though the search time is in milliseconds, this can become an issue quickly. The modified algorithm generally appears to halve the time spent searching. This could provide a benefit over time.

In the previous section, the plots indicated that the modified algorithm performed better than the original overall. Though, there were IPs where this was not the case. The reason for this discrepency is not currently known. This indicates that there are additional factors governing the performance beyond the graph size and number of coefficients. These factors could have to do with the difference in size between the

edge-colored and non edge-colored graphs or with the structure of the IP. Consider the IPs mod010_c and mod011_c. In table 4.1, neither mod010_c or mod011_c change size much between the edge-colored and non edge-colored versions. This suggests that the timing issue could be that the difference between sizes is insufficient to result in speed-up. However, qp1 and qp2 similarly do not change much but there is a slight improvement with the modified algorithm. Most likely, the structure of mod010_c and mod011_c prevents the algorithm from gaining any improvement with the modifications. That is, the colored edges are arranged in such a way that they never cause a cell to split. So all the overhead of attempting to split with a specific color is accrued without the benefit of finer partitions at each search tree node. This would indicate that the modified algorithm can be beneficial with these kinds of IPs, but that additional analysis might be needed to determine whether a specific problem would benefit. This is an issue that could be investigated further. Additionally, there are more mundane issues which can be addressed in the code. A focus on efficiency in memory usage and code design in the refinement methods could lead to greater improvement over the original algorithm.

Bibliography

[1] MPS file format: industry standard. http://pic.dhe.ibm.com/infocenter/cplexzos/v12r5/topic/com.ibm.cplex.zos.help/FileFormats/topics/MPS_synopsis.html, April 2013. 15

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1 edition edition, January 1974. 5

[3] Pietro Belotti, Franois Margot, and Andrea Qualizza. Linear programming relaxations of quadratically constrained quadratic programs. *Tepper School of Business*, August 2009. 15

[4] David Bremner, Mathieu Dutour Sikiri, Dmitrii V. Pasechnik, Thomas Rehn, and Achill Schrmann. Computing symmetry groups of polyhedra. *LMS Journal of Computation and Mathematics*, 17(01):565–581, January 2014. 1

[5] Richard Bdi, Katrin Herr, and Michael Joswig. Algorithms for highly symmetric linear and integer programs. *Mathematical Programming*, 137(1-2):65–90, February 2013. 1, 4, 10

[6] Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 149–154, New York, NY, USA, 2008. ACM. 2, 4, 8

[7] P.T. Darga, Mark H. Liffiton, K.A Sakallah, and IL. Markov. Exploiting structure in symmetry detection for CNF. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 530–534, July 2004. 2, 4, 5

[8] Miguel A. Lejeune and Franois Margot. Integer programming solution approach for inventory-production-distribution problems with direct shipments. *International Transactions in Operational Research*, 15(3):259–281, May 2008. 15

[9] R. Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, January 2003. 15

[10] Franois Margot. Testing cut generators for mixed-integer linear programming. *Mathematical Programming Computation*, 1(1):69–95, July 2009. 15

[11] Franois Margot. Symmetry in integer linear programming. In Michael Jnger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 647–686. Springer Berlin Heidelberg, January 2010. 1, 3

[12] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981. 4, 5

[13] James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. Orbital branching. *Mathematical Programming*, 126(1):147–178, January 2011. 3

[14] Jean-Franois Puget. Automatic detection of variable and value symmetries. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, number 3709 in Lecture Notes in Computer Science, pages 475–489. Springer Berlin Heidelberg, January 2005. 1

Vita

Jonathan Schrock was born in Chattanooga, TN to Randy and Karen Schrock. He has a younger and older sister: Sarah and Rachel respectively. He attended Wheatfield Elementary School in Wheatfield, IN followed by Kouts Middle and High School in Kouts, IN. After graduating, he began studying Mathematics and Computer Science at Taylor University in Upland, IN. Jonathan worked on several research projects with faculty at Taylor as well as a REU at the University of Illinois Urbana/Champaign. The projects covered topics such as knot theory, network mapping, and large scale power systems. Additionally, he worked as a tutor and teaching assistant during his time at Taylor. Jonathan obtained a Bachelor of Science degree from Taylor University in May 2011 in Mathematics and Computer Science. After which he worked for the Department of Defense before beginning work at Oak Ridge National Laboratory and graduate school at the University of Tennessee, Knoxville. He is studying mathematics at UT, and working on projects in quantum computing and graph generation at ORNL.