Using Isomorphism Pruning
to Solve Symmetric ILPs

Sebastian Raaphorst
University of Ottawa

May 9, 2003
Abstract

One available technique to solve certain classes of problems (particularly those of a combinatorial nature), is to formulate the problem as an integer linear program (ILP) and proceed to use a branch-and-bound or branch-and-cut approach in order to determine the set of optimal solutions.

Unfortunately, while this approach may be elegant, certain problems demonstrate a large number of symmetries or isomorphisms, and with naïve techniques, branch-and-cut may solve equivalent subproblems many times over. Because of this, even moderate-sized instances of some problems may become infeasible to solve using these simple methods.

The aim of this paper is to present the reader with some of the concepts explored by Dr. François Margot to avoid processing these equivalent subproblems, and to investigate and compare two different algorithms for determining the symmetry group of an arbitrary ILP.
## Contents

1 Introduction and motivation ................................................. 2
2 Definitions and notation: Permutations .......................... 2
3 Integer linear programs and the $t-(v, k, \lambda)$-packing design .... 3
4 General approach: Branch-and-cut with isomorphism pruning ...... 5
   4.1 Symmetry groups .................................................. 5
   4.2 Orbits and stabilizers .............................................. 6
   4.3 Isomorphism testing, pruning, and fixing ......................... 6
   4.4 Optimizing via 0-fixing ........................................... 7
   4.5 Isomorphism cuts .................................................. 8
   4.6 Overall strategy ................................................... 8
   4.7 Algorithmic details ............................................... 8
5 Finding the symmetry group of an arbitrary ILP .................... 9
   5.1 Naïve technique .................................................... 9
   5.2 Backtracking / partitioning technique ......................... 10
      5.2.1 Backtracking on $\pi$ ........................................ 11
      5.2.2 Partition schemes and refinement ....................... 11
   5.3 Comparison of the two techniques ............................... 14
A Enumeration trees and the symmetry group for the $2-(5, 3, 1)$-packing design 15
   A.1 Tree A: Naïve branch-and-bound tree ....................... 15
   A.2 The symmetry group for the $2-(5, 3, 1)$-packing design .... 17
   A.3 Tree B: Isomorphism-pruning branch-and-cut tree .......... 19
   A.4 Tree C: Isomorphism-pruning branch-and-cut tree with 0-fixing and isomorphism cuts 20
1 Introduction and motivation

There is a well-known approach in computer science called “branch-and-cut” for solving integer linear programs (ILPs). This is useful, because many problems, particularly in the study of combinatorics, have natural ILP formulations that makes them ideal candidates for this approach.

However, there exists a problem with this technique: certain problems may show a high degree of symmetry (for example, certain groups of objects in design theory demonstrate large isomorphism classes), and traditional branch-and-cut techniques may end up repeating huge amounts of computation many times over, thus rendering the simplistic approach infeasible for even moderate-sized instances.

We do not need to immediately discard a branch-and-cut approach, however. In a series of papers ([2], [3], [4]), available on his website (http://www.ms.uky.edu/~fmargot), Dr. François Margot details his investigations as to how we can use the concept of a symmetry group (detailed later) of an ILP in order to avoid generating and processing equivalent subproblems. This paper examines the concepts Margot has used in order to modify branch-and-cut to be a feasible method of dealing with these problems, as well as two different techniques that can be used to generate the symmetry group for an arbitrary ILP.

It is assumed that the reader has a reasonable knowledge of branch-and-cut techniques. Readers with some experience in elementary group theory and design theory may better understand the presented concepts, but this is not imperative.

Concepts are supplemented with an example: the exploration of using these techniques for discovering the 2-(5, 3, 1)-packing design.

2 Definitions and notation: Permutations

We begin with some definitions that are necessarily to understand the problem and the algorithms.

Definition: Let $I^n = \{1, 2, \ldots, n\}$. Then we define $\Pi^n$ to be the set of all permutations over $I^n$. We call $I^n$ the ground set of $\Pi^n$, and we note that the elements of $\Pi^n$ form a group under composition of permutations.

Notation: A permutation $\pi \in \Pi^n$ is represented by an $n$-vector in the usual way, i.e.:

$$\forall i \in I^n, \pi[i] = \pi(i)$$

(Thus, $\pi[i]$ is the image of $i$ under $\pi$.)

Let $v$ be an $n$-vector. Then $w = \pi(v)$ is the $n$-vector obtained by permuting the coordinates of $v$ with respect to $\pi$:

$$w[\pi[i]] = v[i] \rightarrow w[i] = v[\pi^{-1}[i]]$$

Example: If we have $\pi = [4 2 1 3 0]$ and $v = [2 0 1 4 3]$, and we define $w = \pi(v)$, we have that $w[0] = v[\pi^{-1}[0]] = v[5] = 3$, $w[1] = v[\pi^{-1}[1]] = v[3] = 1$, and so on. We continue, to discover that $w = [3 1 0 4 2]$. 

2
3 Integer linear programs and the $t$-$(v, k, \lambda)$-packing design

We introduce the concept of formulating certain problems (particularly in design theory) as integer linear programs (ILPs). While it is assumed that the reader is familiar with ILPs, as a brief review, we examine the standard form for maximization and minimization problems. We define a maximization problem over $n$ variables as follows:

Maximize $c^T \cdot x$

subject to $Ax \leq b$

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

The standard minimization problem is as follows:

Minimize $c^T \cdot x$

subject to $Ax \geq b$

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

In both cases, we assume that $c$ is an $n$-vector, and that $A$ is an $m \times n$ matrix.

As mentioned in the proceeding section, finding subsets of a base set with certain given properties is a typical combinatorial problem that often has a large number of isomorphisms.

As an example, we will investigate the construction of the $2$-$(5, 3, 1)$-packing design.

Definition: A $t$-$(v, k, \lambda)$-BIBD (Balanced Incomplete Block Design) is an ordered pair ($\mathcal{X}, \mathcal{B}$) (where $\mathcal{X}$ is called the set of points, and $\mathcal{B}$ is called the set of blocks), with:

$\mathcal{X} = \{1, 2, \ldots, v\}$

$\mathcal{B} \subseteq \mathcal{P} = \{B \subseteq \mathcal{X} \mid |B| = k\}$

such that every $t$-subset of $\mathcal{X}$ must appear in exactly $\lambda$ blocks of $\mathcal{B}$. We define the set $\mathcal{P}$ (of "potential" blocks) for convenience in future definitions.

Definition: A $t$-$(v, k, \lambda)$-packing design is an ordered pair ($\mathcal{X}, \mathcal{B}$), defined as in the BIBD, albeit with the modified condition that every $t$-subset of $\mathcal{X}$ must appear in at most $\lambda$ blocks of $\mathcal{B}$ and the caveat that we maximize the size of $\mathcal{B}$.

While there are sixteen $2$-$(5, 3, 1)$-packing designs, it is easily verifiable that only one of them is non-isomorphic, $(\mathcal{X}_1, \mathcal{B}_1)$, defined as follows:

$\mathcal{X}_1 = \{1, 2, 3, 4, 5\}$

$\mathcal{B}_1 = \{\{1, 2, 3\}, \{1, 4, 5\}\}$

To formulate this problem as an ILP, we derive the following bijection

$f : \mathcal{P} \rightarrow \mathcal{V} = \{1, 2, \ldots, \binom{v}{k}\}$
(which simply assigns a natural number to every possible block in standard lexicographical order, although any arbitrary ordering can be used). In our example, there are \( \binom{5}{3} = 10 \) possible blocks over our set of points. Here they are, with their corresponding \( f \)-value:

\[
\begin{align*}
\{1, 2, 3\} & \rightarrow 1 \\
\{1, 2, 4\} & \rightarrow 2 \\
\{1, 2, 5\} & \rightarrow 3 \\
\{1, 3, 4\} & \rightarrow 4 \\
\{1, 3, 5\} & \rightarrow 5 \\
\{1, 4, 5\} & \rightarrow 6 \\
\{2, 3, 4\} & \rightarrow 7 \\
\{2, 3, 5\} & \rightarrow 8 \\
\{2, 4, 5\} & \rightarrow 9 \\
\{3, 4, 5\} & \rightarrow 10
\end{align*}
\]

We then define \( \binom{v}{k} \) variables, each corresponding to a block and indicating its presence in or absence from \( \mathcal{B} \).

\[
x_i = \begin{cases} 
0 & : f^{-1}(i) \not\in \mathcal{B} \\
1 & : f^{-1}(i) \in \mathcal{B}
\end{cases}
\]

Similarly to what we did for \( f \) above, we derive a bijection \( g \) between \( t \)-subsets of \( \mathcal{X} \) and \( \{1, 2, \ldots, \binom{v}{t}\} \) as per the standard lexicographical order of \( t \)-subsets.

Then, the natural ILP formulation for a \( t-(v, k, \lambda) \)-packing design is as follows:

\[
\text{Maximize } \sum_{i=1}^{\binom{v}{k}} x_i \\
\text{such that } S_i : \sum_{j \in \mathcal{V}, g^{-1}(i) \subset f^{-1}(j)} x_j \leq \lambda \\
\text{and } I_k : x_k \in \{0, 1\}
\]

where \( S_i \) denotes the \( i \)-th subset constraint (we have \( \binom{v}{t} \) such constraints), and \( I_k \) denotes the \( k \)-th integral constraint (we have \( \binom{v}{k} \) such constraints).

Here, then, is the standard ILP formulation for 2-(5, 3, 1)-packing design, which is obviously a maximization problem:

\[
\text{Maximize } \sum_{i=1}^{\binom{5}{3}=10} x_i
\]
such that  
$S_1 \rightarrow \{1, 2\} : x_1 + x_2 + x_5 \leq 1$
$S_2 \rightarrow \{1, 3\} : x_1 + x_4 + x_5 \leq 1$
$S_3 \rightarrow \{1, 4\} : x_2 + x_4 + x_6 \leq 1$
$S_4 \rightarrow \{1, 5\} : x_3 + x_5 + x_6 \leq 1$
$S_5 \rightarrow \{2, 3\} : x_1 + x_7 + x_8 \leq 1$
$S_6 \rightarrow \{2, 4\} : x_2 + x_7 + x_9 \leq 1$
$S_7 \rightarrow \{2, 5\} : x_3 + x_8 + x_9 \leq 1$
$S_8 \rightarrow \{3, 4\} : x_4 + x_7 + x_{10} \leq 1$
$S_9 \rightarrow \{3, 5\} : x_5 + x_8 + x_{10} \leq 1$
$S_{10} \rightarrow \{4, 5\} : x_6 + x_9 + x_{10} \leq 1$

$x_i \in \{0, 1\} \forall i \in \mathcal{V}$

The branch-and-bound tree for this ILP is shown in appendix A.

4 General approach: Branch-and-cut with isomorphism pruning

We proceed to investigate Margot’s approach to using branch-and-cut to solve problems with a large number of symmetries.

4.1 Symmetry groups

Definition: Consider

1. a permutation $\pi \in \Pi^n$ such that $\pi(c) = c$ (i.e. $\pi$ fixes the vector of coefficients of the objective function), and

2. a permutation $\sigma \in \Pi^m$ such that $\sigma(b) = b$ (i.e. $\sigma$ fixes the $b$ vector of the constraints)

We denote $A(\pi, \sigma)$ to be the matrix $A$ with its columns permuted by $\pi$ and its rows permuted by $\sigma$. Then, we define the following set:

$$G = \{ \pi \in \Pi^n \mid \exists \sigma \in \Pi^m \text{ with } A(\pi, \sigma) = A \}$$

(Thus, $G$ is the set of all column permutations of $A$ for which we can find row permutations such that $A$ is fixed under the action of both permutations.) We note that $G$ is a group, and particularly, $G \leq \Pi^n$. For an $n$-vector $x$ and any $g \in G$, we note that:

$x$ feasible $\iff g(x)$ feasible

$x$ optimal $\iff g(x)$ optimal

Then we call $G$ the symmetry group of the feasible (and optimal) set of the ILP (or simply symmetry group for brevity).

Many combinatorial problems demonstrate large symmetry groups. Examples of such problems include:
1. looking for families of subsets with certain properties (e.g. $t$-$(v, k, \lambda)$-packing designs)

2. scheduling jobs on parallel, identical machines

In these types of problems, a traditional branch-and-cut approach becomes infeasible for even modest-sized instances; because of the large number of symmetries, our branch-and-cut enumeration tree contains many isomorphic, equivalent subproblems and much of the work is duplicated. For example, if we look at our ILP formulation of the 2-$(5, 3, 1)$-packing design, we can see by the branch-and-cut tree that while only one solution is considered non-isomorphic, we generate all sixteen solutions in that isomorphism class. For larger instances of $v$, this will become even more problematic and such techniques will not be feasible. It is clear that another approach is needed; Margot explores how we can exploit the concept of our symmetry group to avoid wasting time solving these isomorphic subproblems.

From this point forward, we assume that we are given the ILP representing our problem along with its symmetry group $G$. Techniques for calculating the symmetry group of an ILP are detailed in the next section. It is our goal to use the symmetry group $G$ to efficiently prune isomorphic subproblems from the search space and to assist the search by making isomorphism cuts (which will be described later).

4.2 Orbits and stabilizers

We examine the concepts of orbits and stabilizers, which are used extensively in the modified branch-and-cut.

Definitions: Let $S \subseteq I^n$ be a set, and $G \leq \Pi^n$ a permutation group.

- The orbit of $S$ under $G$, denoted $\text{orb}(S, G)$, is the set of all images of $S$ under the permutations of $G$. This is mathematically defined as follows:
  \[ \text{orb}(S, G) = \{ S' \subseteq I^n \mid S' = g(S) \text{ for some } g \in G \} \]

- The stabilizer of $S$ in $G$, denoted $\text{stab}(S, G)$, is the set of all permutations of $G$ that fix $S$. This is mathematically defined as follows:
  \[ \text{stab}(S, G) = \{ g \in G \mid g(S) = S \} \]

It is easy to check that $\text{stab}(S, G) \leq G \leq \Pi^n$.

4.3 Isomorphism testing, pruning, and fixing

We now define the concepts of isomorphism with regards to subproblems in a branch-and-cut enumeration tree, and we detail our isomorphism pruning strategy.

Definitions: Let $a$ be a node of our branch-and-cut enumeration tree. We define the following sets:

- $F_0^a = \{ i \in I^n \mid x_i \text{ fixed to 0 at } a \}$
- $F_1^a = \{ i \in I^n \mid x_i \text{ fixed to 1 at } a \}$
\( F^a = I^a \setminus (F^a_0 \cup F^a_1) \)

We refer to the last set as the indices of the free variables at \( a \) (meaning those variables who have not yet been fixed to 0 or 1).

**Definition:** Given two nodes \( a \) and \( b \) of a branch-and-cut enumeration tree, we say that their associated problems are isomorphic if and only if \( \exists g \in G \) with

- \( g(F^a_0) = F^b_0 \)
- \( g(F^a_1) = F^b_1 \)

We can use this fact directly for the purposes of isomorphism testing, but this approach presents some problems that we would rather avoid. Firstly, it requires us to store a maximal set of non-isomorphic subproblems, and storing and manipulating such a set could be costly. Secondly, it requires us to find the existence of permutations \( g \in G \) (as described above) for many pairs of subproblems, which may also be difficult.

The limitations of such an approach quickly become obvious, so we avoid it and look at a more elegant and efficient technique. Instead, we divide our subproblems into isomorphism classes, and we select one subproblem from each class and denote it as the representative. Then, at each node, all that we have to do is determine if the subproblem represented by this node is a representative. If it is, we process this node. If it is not, then we can immediately prune this node.

**Definition:** We say that \( S \subseteq I^a \) is a representative if \( S \) is the lexicographically smallest set in \( \text{orb}(S, G) \).

There is one caveat with this simplification of isomorphism testing: at a node \( a \), we are no longer free to branch on any variable \( x_i \) with \( i \in F^a \). Instead, we must now branch on \( x_f \), where:

\[ x_f = \min(F^a) \]

Finally, we detail our isomorphism pruning strategy, which is as follows: at a node \( a \) of our branch-and-cut enumeration tree, if \( F^a_1 \) is not a representative, we then prune the node.

### 4.4 Optimizing via 0-fixing

While the isomorphism pruning technique detailed in the previous section can be used to improve efficiency dramatically, there are other optimizations that we may include that will reduce the amount of computation required. One such technique is called 0-fixing, and this technique was derived with the observation that at a node \( a \) of our branch-and-cut enumeration tree, certain variables can be fixed to 0 without affecting the optimal solution.

The basic algorithm for 0-fixing is as follows:

1. If \( a \) is the child of node \( b \) with branching variable \( x_f \) set to 0, then we set to 0 all variables over:

\[ F^a \cap \text{orb}(f, \text{stab}(F^a_1, G)) \]

(this consists of setting to 0 all free variables in the set of images of \( x_f \) under the set of permutations that fix the variables set to 1 at node \( a \)).
2. If \( F^a = \emptyset \), we return \( n + 1 \). Otherwise, we take \( f = \min(F^a) \) and determine if \( F_1^a \cup \{f\} \) is a representative. If it is, we return \( f \). Otherwise, we set to 0 all variables over:

\[
F^a \cap \text{orb}(f, \text{stab}(F_1^a, G))
\]

(as above, this again consists of setting to 0 all free variables in the set of images of \( x_f \) under the set of permutations that fix the variables to 1 at node \( a \)).

We repeat stage (2) as many times as is possible, until the algorithm returns a value.

### 4.5 Isomorphism cuts

Despite the fact that we now have an isomorphism pruning strategy, we may still be duplicating some of our efforts in the branch-and-cut enumeration tree. We take further steps to ensure that this does not happen via a technique called isomorphism cuts.

At a node \( a \), we define \( H^a = I^n \setminus F_0^a \). Then suppose that \( \exists J \subseteq H^a \) with the representative \( J^* \) in \( \text{orb}(J, \text{stab}(F_1^a, G)) \) (with \( J^* \) lexicographically smaller than \( F_1^a \)).

If some descendant node \( b \) of \( a \) has \( J \subseteq F_1^b \), we want to prune it, because it will correspond to a subproblem that has already been solved on an earlier branch. To ensure that no descendant of \( a \) has this property, we can add the following isomorphism cut to the ILP for the subtree rooted at node \( a \):

\[
\sum_{j \in J} x_j \leq |J| - 1
\]

(This is simply saying that \( J \not\subseteq F_1^b \) for any descendant \( b \) of \( a \)).

### 4.6 Overall strategy

Our overall strategy for a node \( a \) of our branch-and-cut enumeration tree then becomes:

\[
\begin{align*}
r &= 0 & \text{fixing}(a) \\
&\text{repeat until a criterion is met} \\
&\quad \text{solve the LP relaxation} \\
&\quad \text{generate cuts} \\
&\quad \text{if } (r < n + 1) \text{ then} \\
&\quad \quad \text{branch on } x_r
\end{align*}
\]

### 4.7 Algorithmic details

For the sake of brevity, the algorithms suggested by Margot will not be detailed here; implementation details are available in his paper. To briefly summarize, \( G \) is represented by a Schreier-Sims scheme, with a modification in order to incorporate the concept of a base, denoted \( \beta \), which is an arbitrary ordering on the elements of \( I^n \). The base is used primarily to calculate stabilizers over \( F_1^a \).

The rest of this paper will focus on a preliminary algorithm: given an ILP, how can we find its symmetry group? As we will see, for some problems, like 2-\((v, 3, 1)\)-packing designs, this is obvious from the structure of the problem, but for other more complicated ILPs, this may not always be the case.
Three search trees and the symmetry group for the 2-(5, 3, 1)-packing design may be found in appendix A. By examining these trees, we can see the improvement that Margot’s techniques have on even a very small problem.

Tree A represents a typical branch-and-bound approach to solving this problem; we see that the algorithm processes 189 nodes and generates all elements of the isomorphism class.

Tree B represents a branch-and-cut tree with isomorphism pruning, but without isomorphism cuts and 0-fixing. In this case, the algorithm processes 45 nodes of the tree and only generates the non-isomorphic solution, as expected.

Tree C represents the combined approach of all the techniques we have seen thus far. It becomes apparent, when examining this tree, how much impact 0-fixing can have upon the size of the enumeration tree. In this final case, only seven nodes are examined, and as we again expect, only the one non-isomorphic solution is generated.

5 Finding the symmetry group of an arbitrary ILP

The approach that Margot presents us with assumes that we are given the symmetry group of our constraint matrix for the problem we are examining. We investigate two different techniques to calculate this symmetry group, and the time requirements of each.

5.1 Naïve technique

The first approach that we look at simply iterates over all column and row permutations. Here is the basic pseudocode (note that the term identity(\(\mathcal{X}\)) refers to the identity permutation in the permutation group \(\mathcal{X}\)):

\[
\begin{align*}
\pi &= \text{identity}(\Pi^n) \\
\text{do } &
\{ \\
\quad \text{if } (\pi(c) = c) \text{ then} \\
\quad \quad \sigma &= \text{identity}(\Pi^m) \\
\quad \quad \text{do } &
\{ \\
\quad \quad \quad \text{if } (\sigma(b) = b) \text{ then} \\
\quad \quad \quad \quad \text{if } (A(\pi, \sigma) = A) \text{ then} \\
\quad \quad \quad \quad \quad \text{output } \pi \\
\quad \quad \quad \quad \quad \text{break} \\
\quad \quad \quad \sigma &= \text{nextperm}(\Pi^m, \sigma) \\
\quad \quad \} \text{ loop until } (\sigma = \text{identity}(\Pi^m)) \\
\pi &= \text{nextperm}(\Pi^n, \pi) \\
\} \text{ loop until } (\pi = \text{identity}(\Pi^n))
\end{align*}
\]

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

While this may seem infeasible, it is simple to implement, and the initial idea was that it might run in a reasonable amount of time for small \(m\) and \(n\), as is the case with the 2-(5, 3, 1)-
packing design \((m = 10, n = 10)\). However, this proved to be an incorrect assumption; on a PowerMac G4 733 MHz, with 512 MB of RAM, running Mac OS X 10.2, it was found that the algorithm took approximately 1.76 seconds (real time) to examine each column permutation. Given that there are \(10! = 3,628,800\) such permutations, we can easily see that iterating naively over all permutations like this would take an approximated 74.28 days (real time).

Clearly, this is not a reasonable amount of time to calculate the symmetry group of such a simple problem, and larger problems would take ridiculously longer. To give a general idea of the time-growth of this problem, if we estimate similar operation times in the calculation of \(2-(7,3,1)\)-packing designs (which turn out to be \(2-(7,3,1)\)-BIBDs), we discover that our algorithm takes

\[
\left(\binom{7}{2}\right)! \text{ perms} \div \frac{3,628,800 \text{ perms}}{1.76 \text{ s}} \approx 2.48 \times 10^{13} \text{ s} \approx 7.86 \times 10^5 \text{ years}
\]

to examine each column permutation. There are \(\left(\binom{7}{3}\right)! \approx 1.03 \times 10^{40}\) such column permutations, so this na"{"ive} approach would take approximately \(8.12 \times 10^{45}\) years to determine the symmetry group for the standard ILP formulation of \(2-(7,3,1)\)-BIBD.

Instead, it was obvious that a different approach was needed to hope to calculate the symmetry group, which spurred the investigation of the backtracking / partitioning technique.

### 5.2 Backtracking / partitioning technique

The general idea behind the second technique we examine is that if a partial column permutation is unable to fix \(A\), then no extension of the partial permutation is able to fix \(A\). Thus, we build \(\pi\) by backtracking techniques, determining at each step of the backtracking whether or not the possibility of a \(\sigma\) exists such that \(A(\pi, \sigma) = A\).

As demonstrated in the na"{"ive} technique, it is not feasible to consider all row permutations; thus, we use a technique called partition refinement in order to minimize the number of row permutations that we need to try.

So, in essence, when we manage to complete \(\pi\) via backtracking techniques and we have a valid partition refinement, we have found a \(\pi\) and \(\sigma\) (or possibly a family of \(\sigma\)s) that fix \(A\). Thus, we output \(\pi\) and backtrack.

Here is a very general pseudocode for the problem. Curious readers may wish to investigate the source code, which is available on the author’s website (http://www.site.uottawa.ca/~raaphors).

```plaintext
calculate the partition scheme for the matrix \(A\)
\(i = 1\)
while \((i \geq 1)\)
    extend the permutation \(\pi\) at position \(i\)
    if (we were unable to do so) then
        \(i = i - 1\)
        loop
    refine the partitions of \(\sigma\)
    if (\(\sigma\) can no longer be refined to a bijection) then
        loop
    if (\(\pi\) is not a complete permutation) then
        \(i = i + 1\)
```

10
Note that, at every iteration of the loop, we ensure that our partial permutation $\pi$ fixes $c$, and that the family of row permutations represented by our partition refinement is capable of fixing $b$ (if either of these cases fail, we backtrack immediately). Following is a detailed explanation of the various components of the algorithm.

5.2.1 Backtracking on $\pi$

This is a fairly simple concept, and one that will not be examined in great depth (it is assumed that the reader has knowledge of backtracking algorithms). The general idea is that we have a partial permutation over the columns, and we extend it via backtracking techniques. Thus, at some stage of the algorithm, if we have $\pi = (1, 3, 5)$ over the ground set $I^6$, we have the following possible extensions of $\pi$:

$$
\begin{align*}
\pi_1 &= (1, 3, 5, 2) \\
\pi_2 &= (1, 3, 5, 4) \\
\pi_3 &= (1, 3, 5, 6)
\end{align*}
$$

and all of these possibilities will be explored by the backtracking tree.

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

5.2.2 Partition schemes and refinement

*Note:* In the following terminology and the context of the program, we view permutations as different than as defined previously; if we have some permutation $\delta$ such that $\delta[i] = j$, we take this to mean that, under $\delta$, $j \rightarrow i$. It is trivial to change a permutation of this form to a permutation of the previously defined form. The reason for this change of notation is because we want to build our partially permuted matrix from left to right; the motivation behind this will become obvious later on in this section.

*Definition:* A partition scheme is a two-dimensional array of sets, $\mathcal{PS}$, such that $\mathcal{PS}[col, val]$ is the set of positions in which $val$ appears in column $col$ of the matrix $A$. 
Example: For the 2-(5, 3, 1)-packing design, we have the following matrix:

\[
A = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 
\end{bmatrix}
\]

Then (if we index the columns and the rows beginning at 1), \( \mathcal{PS}[1, 0] \) is simply the set of positions in which 0 appears in the first column (namely \{3, 4, 7, 8, 9, 10\}), and \( \mathcal{PS}[1, 1] \) is the set of positions in which 1 appears in the first column (namely \{1, 2, 5\}). Similarly, \( \mathcal{PS}[2, 0] = \{2, 4, 5, 7, 8, 9, 10\} \) and \( \mathcal{PS}[2, 1] = \{1, 3, 6\} \), etc...

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

Initially, when \( \pi \) is empty (\( \pi = () \)), we assume that any row can be mapped to any other row (thus, \( \mathcal{PR}[i] = I^n \ \forall i \in I^m \)). At any stage of the backtracking, if we extend \( \pi \) by \( x \in I^n \) at position \( p \), the family of permutations represented by \( \mathcal{PR} \) may no longer all fix \( A \) under \( \pi \), so we must refine to remove the valid ones.

When \( x \) is added to \( \pi \) at position \( p \), we perform the following refinement step:

\[
\mathcal{PR}[i] = \mathcal{PR}[i] \cap \mathcal{PS}[p, A[i, \pi[p]]] \ \forall i \in I^n
\]

What we are saying here is that for any row \( i \), by fixing another column \( p \) of our matrix (which, in essence, grows the size of each row of our “partially permuted” matrix by 1), we may have reduced the number of rows to which \( i \) can be mapped. Assume that by extending \( \pi \) by \( x \), we “added” value \( k \) to row \( i \) (at position \( p \)). Now, to determine the new set of rows that row \( i \) can be mapped to, we take the intersection of the old set of rows to which it can be mapped with the set of rows of our original matrix \( A \) that contain \( k \) in position \( p \). This is best demonstrated by an example.

Example: Assume we have the matrix \( A \) as defined above for the 2-(5, 3, 1)-packing design.
We begin with $\pi$ empty and add 3 to it, so $\pi = (3)$. Then we perform the following refinements:

$$
\mathcal{PR}[1] = \mathcal{PR}[1] \cap \mathcal{PS}[1, A[1, 3] = 1] \\
= I^m \cap \{1, 2, 5\} \\
= \{1, 2, 5\}
$$

$$
\mathcal{PR}[2] = \mathcal{PR}[2] \cap \mathcal{PS}[1, A[2, 3] = 0] \\
= I^m \cap \{3, 4, 6, 7, 8, 9, 10\} \\
= \{3, 4, 6, 7, 8, 9, 10\}
$$

$$
\mathcal{PR}[3] = \mathcal{PR}[3] \cap \mathcal{PS}[1, A[3, 3] = 0] \\
= I^m \cap \{3, 4, 6, 7, 8, 9, 10\} \\
= \{3, 4, 6, 7, 8, 9, 10\}
$$

$$
\mathcal{PR}[4] = \mathcal{PR}[4] \cap \mathcal{PS}[1, A[4, 3] = 1] \\
= I^m \cap \{1, 2, 5\} \\
= \{1, 2, 5\}
$$

... ...

$$
\mathcal{PR}[5] = \mathcal{PR}[5] \cap \mathcal{PS}[2, A[5, 7] = 1] \\
= \{3, 4, 6, 7, 8, 9, 10\} \cap \{1, 3, 6\} \\
= \{3, 6\}
$$

Now, say we further extend $\pi$ with 7, so $\pi = (3, 7)$. Now our “partially permuted” matrix is as follows:

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

Then our partition refinements become:

$$
\mathcal{PR}[1] = \mathcal{PR}[1] \cap \mathcal{PS}[2, A[1, 7] = 0] \\
= \{1, 2, 5\} \cap \{2, 4, 5, 7, 8, 9, 10\} \\
= \{2, 5\}
$$

$$
\mathcal{PR}[2] = \mathcal{PR}[2] \cap \mathcal{PS}[2, A[2, 7] = 0] \\
= \{3, 4, 6, 7, 8, 9, 10\} \cap \{2, 4, 5, 7, 8, 9, 10\} \\
= \{4, 7, 8, 9, 10\}
$$

... ...

$$
\mathcal{PR}[5] = \mathcal{PR}[5] \cap \mathcal{PS}[2, A[5, 7] = 1] \\
= \{3, 4, 6, 7, 8, 9, 10\} \cap \{1, 3, 6\} \\
= \{3, 6\}
$$

It begins to become obvious at this point; in our “partially permuted” matrix, row 1 corresponds to (10). If we scan the rows over the first two columns of $A$, we see that the entries (10) occur in rows 2 and 5. Thus, row 1 can only be mapped to rows 2 and 5 of our original matrix. Similarly, row 5 corresponds to (01) under the current column permutation, and so we scan our original matrix $A$ to find that rows 3 and 6 begin with (01), so they are the only possible
images of row 5 under a row permutation designed to fix $A$. Note: in this case, it is easy to see that our refinement is not extendable to a bijection; by looking at our partially permuted matrix above, we can see that there is no row that corresponds to (11). The first row of $A$ is (11), and thus it is not possible to find a surjective mapping between our partially permuted matrix and the first two columns of $A$. Thus, since permutations are bijections, our refinement represents the empty set of permutations (i.e. no row permutation exists), and we backtrack immediately.

Note: In our current implementation, surjectivity of mapping families represented by our refinement is not checked. Regardless, the algorithm works all the same, because if a mapping from one set to another of equal cardinality is not surjective, then it either cannot be injective (by simple set theory), or it cannot be a function, and this will be detected at a later point in the generation. I am still contemplating an efficient method to test surjectivity that runs better than $O(m)$.

5.3 Comparison of the two techniques

As shown previously, the naïve technique fails to run in a reasonable amount of time for even very small examples. The table below details the running time of the two different algorithms for several different 2-$(v, 3, 1)$-packing design ILP formulations.

<table>
<thead>
<tr>
<th>$v$</th>
<th>$n$</th>
<th>$m$</th>
<th>Naïve (s, est)</th>
<th>B/P (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Naïve (s, est)</td>
<td>B/P (s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>real</td>
<td>CPU</td>
</tr>
<tr>
<td>5</td>
<td>$\binom{5}{2} = 10$</td>
<td>$\binom{5}{2} = 10$</td>
<td>$6.42 \times 10^6$</td>
<td>$5.22 \times 10^6$</td>
</tr>
<tr>
<td>6</td>
<td>$\binom{6}{3} = 20$</td>
<td>$\binom{6}{2} = 15$</td>
<td>$1.25 \times 10^{24}$</td>
<td>$1.54 \times 10^{24}$</td>
</tr>
<tr>
<td>7</td>
<td>$\binom{7}{3} = 35$</td>
<td>$\binom{7}{2} = 21$</td>
<td>$2.08 \times 10^{53}$</td>
<td>$2.56 \times 10^{53}$</td>
</tr>
</tbody>
</table>

It is not difficult to see, in the case of the symmetry group over these particular designs, that each column permutation $\pi$ represents a point permutation over $v$. For example, the column permutation $\pi = (7 8 1 9 2 3 10 4 5 6)$ corresponds to point permutation $\delta = (5 1 2 3 4)$ (defined in the standard way, meaning 1 \rightarrow 5, 2 \rightarrow 1, etc..., and not the reverse as in the above algorithm). Given this fact, we notice that it is simple to generate the symmetry group for 2-$(v, 3, 1)$-packing designs; we can simply take permutations over the point set and transform them into permutations over the ILP variables. This approach may not always be so easy in practice with more complicated problems.
A Enumeration trees and the symmetry group for the 2-(5, 3, 1)-packing design

A.1 Tree A: Naïve branch-and-bound tree
### A.2 The symmetry group for the 2-(5, 3, 1)-packing design

Note that these permutations follow the notation described in section 5.2.2 of this document and are indexed by 0 instead of 1 (e.g. $\pi = (2 \ 7 \ 8 \ 4 \ 5 \ 9 \ 0 \ 1 \ 6 \ 3)$ implies $\pi(0) = 6$, $\pi(1) = 7$, $\pi(2) = 0$, etc...). The table is continued on the next page.

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
1 & 0 & 2 & 3 & 5 & 4 & 6 & 8 & 7 & 9 \\
2 & 0 & 1 & 4 & 5 & 3 & 7 & 8 & 6 & 9 \\
3 & 0 & 1 & 2 & 6 & 7 & 8 & 3 & 4 & 5 & 9 \\
4 & 0 & 1 & 2 & 6 & 8 & 7 & 3 & 5 & 4 & 9 \\
5 & 0 & 1 & 2 & 7 & 8 & 6 & 4 & 5 & 3 & 9 \\
6 & 0 & 2 & 1 & 4 & 3 & 5 & 7 & 6 & 8 & 9 \\
7 & 0 & 2 & 1 & 7 & 6 & 8 & 4 & 3 & 5 & 9 \\
8 & 0 & 3 & 4 & 1 & 2 & 5 & 6 & 7 & 9 & 8 \\
9 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
2 & 0 & 1 & 4 & 3 & 5 & 7 & 6 & 8 & 9 \\
3 & 0 & 1 & 2 & 6 & 7 & 8 & 3 & 4 & 5 & 9 \\
4 & 0 & 1 & 2 & 6 & 8 & 7 & 3 & 5 & 4 & 9 \\
5 & 0 & 1 & 2 & 7 & 8 & 6 & 4 & 5 & 3 & 9 \\
6 & 0 & 2 & 1 & 4 & 3 & 5 & 7 & 6 & 8 & 9 \\
7 & 0 & 2 & 1 & 7 & 6 & 8 & 4 & 3 & 5 & 9 \\
8 & 0 & 3 & 4 & 1 & 2 & 5 & 6 & 7 & 9 & 8 \\
9 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
4 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
5 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
6 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
7 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
8 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
9 & 0 & 3 & 4 & 6 & 7 & 9 & 1 & 2 & 5 & 8 \\
\end{array}
\]
| 6 0 7 1 8 2 3 9 4 5 | 7 0 6 2 8 1 4 9 3 5 | 8 1 6 2 7 0 5 9 3 4 |
| 6 0 7 3 9 4 1 8 2 5 | 7 0 6 4 9 3 2 8 1 5 | 8 1 6 5 9 3 2 7 0 4 |
| 6 1 8 0 7 2 3 9 5 4 | 7 2 8 0 6 1 4 9 5 3 | 8 2 7 1 6 0 5 9 4 3 |
| 6 1 8 3 9 5 0 7 2 4 | 7 2 8 4 9 5 0 6 1 3 | 8 2 7 5 9 4 1 6 0 3 |
| 6 3 9 0 7 4 1 8 5 2 | 7 4 9 0 6 3 2 8 5 1 | 8 5 9 1 6 3 2 7 4 0 |
| 6 3 9 1 8 5 0 7 4 2 | 7 4 9 2 8 5 0 6 3 1 | 8 5 9 2 7 4 1 6 3 0 |
| 6 7 0 8 1 2 9 3 4 5 | 7 6 0 8 2 1 9 4 3 5 | 8 6 1 7 2 0 9 5 3 4 |
| 6 7 0 9 3 4 8 1 2 5 | 7 6 0 9 4 3 8 2 1 5 | 8 6 1 9 5 3 7 2 0 4 |
| 6 8 1 7 0 2 9 3 5 4 | 7 8 2 6 0 1 9 4 5 3 | 8 7 2 6 1 0 9 5 4 3 |
| 6 8 1 9 3 5 7 0 2 4 | 7 8 2 9 4 5 6 0 1 3 | 8 7 2 9 5 4 6 1 0 3 |
| 6 9 3 7 0 4 8 1 5 2 | 7 9 4 6 0 3 8 2 5 1 | 8 9 5 6 1 3 7 2 4 0 |
| 6 9 3 8 1 5 7 0 4 2 | 7 9 4 8 2 5 6 0 3 1 | 8 9 5 7 2 4 6 1 3 0 |
| 9 3 6 4 7 0 5 8 1 2 | 9 3 6 4 7 0 5 8 1 2 | 9 3 6 4 7 0 5 8 1 2 |
| 9 3 6 5 8 1 4 7 0 2 | 9 4 7 3 6 0 5 8 2 1 | 9 4 7 3 6 0 5 8 2 1 |
| 9 4 7 5 8 2 3 6 0 1 | 9 5 8 3 6 1 4 7 2 0 | 9 5 8 3 6 1 4 7 2 0 |
| 9 5 8 4 7 2 3 6 1 0 | 9 6 3 7 4 0 8 5 1 2 | 9 6 3 7 4 0 8 5 1 2 |
| 9 6 3 8 5 1 7 4 0 2 | 9 7 4 6 3 0 8 5 2 1 | 9 7 4 6 3 0 8 5 2 1 |
| 9 7 4 8 5 2 6 3 0 1 | 9 8 5 6 3 1 7 4 2 0 | 9 8 5 6 3 1 7 4 2 0 |
| 9 8 5 7 4 2 6 3 1 0 | 9 8 5 7 4 2 6 3 1 0 | 9 8 5 7 4 2 6 3 1 0 |
A.3 Tree B: Isomorphism-pruning branch-and-cut tree
A.4 Tree C: Isomorphism-pruning branch-and-cut tree with 0-fixing and isomorphism cuts

```
x1=1  x1=0  [1]
x2=1  [1]  [1]
   x2=0  (-> x3=x4=x5=x7=x8=0)
   [1,2]  [1]
   x6=1  x6=0  (-> x9=x10=0)
   [1,6]  [1]
       X
```

x6=0
References


