Optimization Topics Report

Joshua Gunter
Daniel Snider

December 19, 2017

1 Introduction

Linear programming is a method of mathematical optimization, where the goal is to find the optimal value of a linear objective function over a set of variables, which are constrained by a set of linear inequalities. If all the variables must also be integers, then the problem is referred to as an integer program. While linear programs may be efficiently solved with the use of algorithms such as the Simplex Method, there is no reliably fast method for integer programming problems. In fact, they have been proven to be NP-Hard. Because of this, better methods and algorithms for solving integer programs is an active field of research.

In this study, we investigated a method to speed up the solving of a special set of symmetric integer programs. An integer program (hereafter referred to as an IP) has symmetry if the value of its objective function is invariant under permutations of the coordinate axes. More plainly, swapping around the entries of a vector v describing the solution to an IP doesn't change the value of $f(v)$, where f is the objective function.

When an IP is symmetric in this way we can find a set of orthogonal subspaces in $\mathbb{R}^n$, at least one of which any optimal solution must lie "close to". We can therefore restrict the search space for the optimum by setting up quadratic bounds, restricting how far away an answer can be from the aforementioned subspaces. Exactly how "close to" one of these subspaces the solution has to be is not yet known. The goal of this project is to experimentally test different possible bounds and discover what the effect of adding these bounds are on the runtime of solving IPs using standard solvers. In general, it is expected that with a tighter bound, the solver should be able to find a solution faster than with a looser bound, as there is less space to search through. By experimentally checking various bounds, we can determine whether or not this strategy would be useful once methods to find an exact bound are discovered.

This report consists of four main sections. First, definitions of terms from linear programming, group theory, and geometry required as background. Second we look at the algorithms and methods used to construct special subspaces and integer programs, and compute the additional quadratic constraints. Third we list some of the computational results found. Finally, we give some comments and recommendations for future work.

2 Definitions

2.1 Linear and Integer Programming

Linear programming, or more specifically integer programming played a huge part in the motivation for work which was done. Therefore, in order for the rest of the paper to be fully understood some concepts from linear programming require definition. Linear programming problems occur when one wishes to maximize or minimize a linear function subject to some linear constraints. Observe the following example:

Find z_1 and z_2 which maximize the sum $z_1 + z_2$ subject to the following constraints:

$$z_1 \geq 0$$

$$z_2 \geq 0$$

$$2z_1 + 5z_2 \leq 22$$

$$z_1 + z_2 \leq 6$$

$$3z_1 + z_2 \leq 17$$

As you can see, we have two unknowns, z_1 and z_2 which are bounded with five constraints. We call the special constraints of the form $z_i \geq 0$ nonnegativity constraints. The other three are known as the main constraints. The function which is being minimized (or maximized) is known as the objective function and in this case is: $z_1 + z_2$. Any vector which satisfies all the constraints of a given linear program is a feasible solution. If a vector gives the minimum (or maximum) possible value out of all the feasible solutions it is known as the optimal solution or optimum. If a linear program has no feasible solutions, and therefore no optimum it is called infeasible. Linear programs are good because they're efficiently solvable in both theory and practice.

An integer program in this context, is a linear program where some or all of the variables are restricted to be integers. Integer programs are known to be computationally difficult (NP-Hard as mentioned previously). However the examples which we are running are not only known to be NP-Hard, but also infeasible due to our method of construction of the IPs. We achieved infeasibility by constructing polytopes which we know have integer vertices as the only integer points, then we introduced bounds that severed these vertices, leaving us with an infeasible IP. This means that our optimizer, CPLEX is forced to search every possible point in order to realize that the IP is infeasible.

Typically, when an IP is infeasible the optimizer always has to check every single part of the IP. However, because of the fact the infeasible programs have symmetry as mentioned in the introduction, we can introduce specially constructed quadratic constraints. These quadratic constraints are less than or equal to some constant C, which is currently unknown. As we assume the value of C to be bigger and bigger, the amount of time it takes to determine wheither the program is infeasible gets longer and longer, with the quickest solution being at $C = 0$. This rate isn't monotonic however, and the preliminary results can be found in the Computational Results section later.

2.2 Groups and Representations

One of our methods of constraint generation uses the idea of characters. Characters are a concept that require a solid understanding of both Groups and Group Representations. In this section we will try and give sufficient background into Group and Representation Theory, however if further knowledge is needed a group theory textbook could be helpful. The textbook we personally used for this was **??**

Definition 2.1 (Group)**.** A group $G(S, *)$ is an algrebraic structure which consists of a set of elements S and an operation between the elements $*$ which maps any two elements to a third element within S. The group must also satisfy the four group axioms, closure, associativity, identity and invertibility.

This may seem slightly abstract, however it should become clear with a rather mundane example. Consider the group of all integers under addition. If we use the above notation, we'd have a group $G(Z, +)$ where Z is the set of all integers and $+$ is the addition operation. It immediately becomes clear based on the axioms that this group is legitimate. Closure is satisfied because any integer plus another integer yeilds an integer result. Associativity is one of the properties of addition so we can know this to be true as well. Identity can be seen as the number zero, as zero added to any number simply returns that number. The final axiom, Invertibility is also satisfied, as every positive integer has an equivalent negative integer which if you add to it, will result in the identity, which as previously mentioned is zero.

Definition 2.2 (Representation)**.** Let V be a vector space over the field C of complex numbers. Let GL(V) be the group of isomorphisms of V onto itself. Let G be a finite group. A linear representation of G in V is a homomorphism p from the group G into the group GL(V). In other words, we associate with each element s in G an element p(s) of GL(V) in such a way that we have the equality: $p(st) = p(s) \cdot p(t)$ for $s, t \in G$, that implies $p(1) = 1$, $p(s^{-1}) = p(s)^{-1}$. When p is given, we say that V is a representation space of G (or simply, a representation of G).

Representations are really only used as a background to characters as far as this experiment is concerned. Please see subsection 3.1.1 for the full details concerning how irreducible characters are used to create these LPs.

2.3 Core Points

In this section, we will be looking at the definition of what a core point is, and how they relate integer programs. These objects were first studied by [**?**], in the context of [...].

Definition 2.3 (Core Point)**.** A core point of a permutation group G is a point $z \in \mathbb{Z}^n$ such that the orbit polytope $\text{conv}(Gz)$ contains no interior integer points. In other words, $\text{conv}(Gz) \cap \mathbb{Z}^n = Gz$. The set of all core points of G is called the **core set** and is denoted by $\text{core}(G)$.

A very important result from [**?**] is that these core points must lie "close to" the invariant subspaces of their group G. The proof of this is rather technical, so rather than give the full theorem and proof we will sketch out the general idea behind it. First, we define a function $\mu : \mathbb{Z}^n -> \mathbb{R}$ which maps points $z \in \mathbb{Z}^n$ to the minimal distance of a point in the affine hull of Gz from the fixed space $\text{Fix}(G)$. We then construct a minimal bounding ellipsoid E around the orbit polytope $\text{conv}(Gz)$, where $z \in \text{core}(G)$, using results from Theorem 2.2 in [**?**]. We then find a scaled and translated ellipsoid E' constructed from E such that $E' \subseteq \text{conv}(Gz)$, and use this to find conditions for when E' (and by extension $\text{conv}(Gz)$ contain interior integer points. From this exploration we can find a constant bound $C(G)$ for the projection of z onto any of the invariant subspaces of G, such that $z \in \text{core}(G) \implies ||z|_{V_i}|| \leq C(G)$. If $||z|_{V_i}|| \geq C(G)$, then $E' \subseteq \text{conv}(Gz)$ will contain an integer point inside of it, and therefore z can not be a core point.

A natural question is what these points have to do with solving integer programs. Before that question can be answered, let us state the precise definition of a symmtery of an integer programming problem.

Definition 2.4 (IP Symmetry)**.** A **symmetry** of an integer program with a feasible region $P \subset \mathbb{R}^n$ and objective function $f : \mathbb{Z}^n \to \mathbb{R}$ is a permutation $g \in S_n$ such that $gx \in P$ and $f(x) = f(gx)$ for all $x \in P$. The group of all these permutations is the symmetry group $G \leq S_n$ of the integer program.

Now we can state the result connecting these two concepts of core points and integer programs, Theorem 7.3 from [**?**].

Theorem 2.5. Let $\min_{x \in C \cap \mathbb{Z}^n} f(x)$ be a convex integer optimization problem, with a convex function f and a convex set $C \subseteq \mathbb{R}^n$ and a symmetry group G. Then $\min_{x \in C \cap \mathbb{Z}^n} f(x) = \min_{x \in C \cap \text{core}(G)} f(x)$.

The idea behind the proof of this theorem is that if the optimal solution $x \in C \cap \mathbb{Z}^n$ is not a core point, then we can find some integral points inside of $\text{conv}(Gx)$. By choosing an x' from within the integer points $\text{conv}(Gx) \cap \mathbb{Z}^n$ that has minimal norm, we are guaranteed that x' is a core point for G. Since $x' \in \text{conv}(Gz)$, x' is a convex combination of permutations of x, so $x' = tx + (1-t)(gx)$ for some $t \in [0,1], g \in G$. Because f is convex, $f(tx + (1-t)(gx)) \leq tf(x) + (1-t)f(gx) \implies f(x') \leq f(x)$, as $f(x) = f(gx)$. Therefore x' is an optimal solution for the integer program with $x' \in \text{core}(G)$.

Due to this theorem, we can find an optimal solution to any integer program with a symmetry by only looking at the core points of the symmetry group. In some cases, there are a finite number of core points, and we can devise algorithms based on enumerating core points and searching for an optimal solution. However in this project we focused on a set of groups containing infinite core points, in which case enumeration of core points won't help us any. Instead, armed with our additional constraints on the norm of the projection of core points on the invariant subspaces of the group, we can restrict the size of the feasible region when searching for solutions to the integer program. This is the extent of the knowledge we will need about core points for this project.

3 Methods & Algorithms

Now that we know that we can add additional constraints to reduce the size of the feasible region, all we need are algorithms to actually compute all the required data. First, we will examine two methods to construct invariant subspaces of symmetry groups, one method using information from character theory, the second using results about group actions in relation to invariant subspaces. Next, we will look at how we can actually compute a set of core points of group, and use these to construct difficult integer programs to test the effectiveness of this method on. Finally, we will cover how to compute the new quadratic constraints that we can use to solve these problems faster.

3.1 Computing Invariant Subspaces

3.1.1 Irreducible Characters

This method was the primary one used during the project, and makes use of results found in character theory. Many formulas dealing with representations of groups can be transformed into formulas involving their characters, which are much easier to work with computationally.

Definition 3.1 (Character)**.** The **character** $\chi_\rho : G \to \mathbb{C}$ of a representation $\rho : G \to GL_n(\mathbb{C})$ is defined as $\text{tr}(\rho(g))$ for all $g \in G$.

Similarly to how representations can be decomposed into a direct sum of irreducible representations, the character of a representation can be decomposed into a linear combination of irreducible characters. This means we can write any character of a group G as

$$\chi = m_1\chi_1 + m_2\chi_2 + ... + m_k\chi_k,$$

where $m_i \in \mathbb{Z}_{\geq 0}$, and χ_i is the character of the ith irreducible representation of G. We can compute each m_i by taking an inner product of the characters, using the formula:

$$m_i = \frac{1}{|G|} \sum_{g \in G} \chi_i(g)\chi(g^{-1}). \tag{1}$$

From this, we can see that as long as we know the values of the irreducible characters, we can easily compute what the decomposition is for any character.

The decomposition of a character χ into a sum of irreducible characters corresponds to the decomposition of the related representation ρ into a direct sum of subrepresentations. We can rewrite ρ into

$$\rho = \rho_1^{(m_1)} \oplus \rho_2^{(m_2)} \oplus ...\rho_k^{(m_k)}.$$

Each of these subrepresentations $\rho_i^{(m_i)}$ corresponds to an invariant subspace V_i, which is what we're really interested in. In [**?**], we can find the following formula for the projection of $\mathbb{C}^n$ onto V_i:

$$P_i = \frac{m_i \chi_i(1)}{|G|} \sum_{g \in G} \chi_i(g^{-1}) \rho(g). \tag{2}$$

The column space of P_i is then a basis for V_i. At this point we have all of the basic tools necessary for computation of invariant subspaces. However these formulas do have a weakness: as they require summing over the all of the group elements, the runtime required can be exorbitant for larger groups. We can make our lives a little easier by doing some computations with conjugacy classes of groups, along with a couple of additional small optimizations.

Definition 3.2 (Conjugacy Classes)**.** For a group G, two elements $a, b \in G$ are said to be **conjugate** if $a = gbg^{-1}$ for some $g \in G$. A **conjugacy class** is a subset of G containing elements that conjugate with one another. The conjugacy class containing an element $g \in G$ is denoted by $Cl(g)$.

This relation can easily be seen to be reflexive, symmetric, and transitive, and is therefore an equivalence relation. As such, we can partition G in a set of disjoint conjugacy classes. This is very useful to us because characters are constant over a conjugacy class, so by dealing with conjugacy classes we can avoid computations summing over every group element. Instead we can sum over the conjugacy classes whenever we are only dealing with characters. We can perform an additional optimization making use of the fact that $\chi(g^{-1}) = \overline{\chi(g)}$, where $\overline{z}$ denotes the complex conjugate of $z \in \mathbb{C}$. This allows us to compute $\chi(g^{-1})$ without needing to know what g^{-1} actually equals. We can now re-express the previous formula for computing the coefficients of the irreducible characters in the decomposition of a character by

$$m_i = \frac{1}{|G|} \sum_{Cl(g) \subseteq G} |Cl(g)| \chi_i(g) \overline{\chi(g)}. \tag{3}$$

This is the equation we will actually be implementing. Unfortunately as for the second part of finding invariant subspaces, computing the projection matrices P_i, due to the fact that the representation itself is being used in the formula we will not be able to make use of conjugacy classes to save time, as representations are definitely not constant over conjugacy classes.

All of these computations were performed using GAP, which has excellent support for computations with characters. Character tables containing information about characters of a group G can be obtained with the function `CharacterTable(G)`, with the values for all irreducible characters of a group found in the attribute `Irr`. The following code block is the GAP implementation of (3).

```
tbl := CharacterTable(G);;
irrs := Irr(tbl);;
reg := RegularCharacters(G,dim);;

m := List(irrs, i -> 0);;
```

```
for i in [1..Size(irrs)] do
    m[i] := Sum([1..Size(SizesConjugacyClasses(tbl))],
                j -> SizesConjugacyClasses(tbl)[j]*irrs[i][j]
                    *ComplexConjugate(reg[j]) ) / Order(G);
od;
```

Here `SizesConjugacyClasses` is a list of the cardinalities of the conjugacy classes of G. Next we have the implementation of (2) in GAP:

```
P := List(irrs, i -> NullMat(dim,dim));;
for i in [1..Size(irrs)] do
    if m[i] <> 0 then
        for g in G do
            irr_index := Position(cclasses,ConjugacyClass(G,g));
            reg_rep := PermutationMat(g,dim);
            P[i] := P[i] + ComplexConjugate(irrs[i][irr_index])
                        *reg_rep;
        od;
        P[i] := m[i]*irrs[i][1]/Order(G) * P[i];
    fi;
od;
```

`Position(Classes, C)` returns the index of a conjugacy class C in the list of conjugacy classes.

Once we've computed the projection matrices P_i, all that's left to do is compute their column spaces. A simple method to achieve this in GAP is to transpose the matrix with `TransposedMat`, and then find the row space of that using the `BaseMat` function. After that, we will have a list of bases of invariant subspaces for a group.

Example. Consider the permutation group $G = \langle (12), (34) \rangle \cong S_2 \times S_2$, with the regular representation $\rho : G \to GL_4(\mathbb{C})$ mapping permutations to permutation matrices. We have $\rho(()) = I$,

$$\rho((12)) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \ \rho((34)) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \ \rho((12)(34)) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

from which it is easy to see that

$$\chi(()) = 4, \ \chi((12)) = 2, \ \chi((34)) = 2, \text{and } \chi((12)(34)) = 0.$$

We can write this function compactly as an array indexed by group elements $\chi = [4, 2, 2, 0]$. By doing a quick calculation with GAP, we can get that the irreducible characters of this group are

$$\chi_1 = [1, 1, 1, 1], \ \chi_2 = [1, -1, -1, 1]$$

$$\chi_3 = [1, -1, 1, -1],\ \chi_4 = [1, 1, -1, -1].$$

Now we can compute the coefficients of the irreducible character decomposition $\chi = m_1\chi_1 + m_2\chi_2 + m_3\chi_3 + m_4\chi_4$:

$$m_1 = (1 \cdot 4 + 1 \cdot 2 + 1 \cdot 2 + 1 \cdot 0)/4 = 8/4 = 2$$

$$m_2 = (1 \cdot 4 + (-1) \cdot 2 + (-1) \cdot 2 + 1 \cdot 0)/4 = 0/4 = 0$$

$$m_3 = (1 \cdot 4 + (-1) \cdot 2 + 1 \cdot 2 + (-1) \cdot 0)/4 = 4/4 = 1$$

$$m_4 = (1 \cdot 4 + 1 \cdot 2 + (-1) \cdot 2 + (-1) \cdot 0)/4 = 4/4 = 1$$

Therefore $\chi = 2\chi_1 + \chi_3 + \chi_4$. We can now apply Equation (2) to find e.g. P_1:

$$P_1 = \frac{m_1\chi_1(1)}{4}\left(\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} + \begin{bmatrix}0&1&0&0\\1&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} + \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix} + \begin{bmatrix}0&1&0&0\\1&0&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix},\right)$$

$$P_1 = \frac{2}{4} \cdot \begin{bmatrix}2&2&0&0\\2&2&0&0\\0&0&2&2\\0&0&2&2\end{bmatrix} = \begin{bmatrix}1&1&0&0\\1&1&0&0\\0&0&1&1\\0&0&1&1\end{bmatrix}$$

Taking the column space of P_1, we find that $V_1 = \text{span}\{(1,1,0,0),(0,0,1,1)\}$. By the same method we get $V_2 = \text{span}\{(1,-1,0,0)\}$ and $V_3 = \text{span}\{(0,0,-1,1)\}$. Therefore $\text{span}\{(1,1,0,0),(0,0,1,1)\} \oplus \text{span}\{(1,-1,0,0)\} \oplus \text{span}\{(0,0,-1,1)\}$ is a decomposition of $\mathbb{C}^4$ into G-invariant subspaces.

As stated earlier, this was the primary method used in this project, as aside from the symmetric and alternating groups, most groups we looked at had relatively small orders (less than a few hundred thousand). With larger groups, another method is required to solve for their invariant subspaces.

3.1.2 Solving Polynomials

Another method of finding invariant subspaces of a group is to solve a system of quadratic equations. This method scales with number of orbits $O_1, O_2, ...O_k$ of the stabilizer subgroup $Stab_G(1)$ of G, rather than the order of the group, so it is more amenable for large groups such as S_n for $n \geq 11$ than the method relying on character decompositions. However, it relies on the group action acting transitively. Fortunately for this project, all the groups we are concerned with are primitive, and therefore transitive by definition, so we won't run into difficulties applying this algorithm. Before moving onto the method, we'll first need to prove a couple lemmas to use as tools.

Lemma 3.3. Let $G \leq S_n$, with V an invariant subspace of G. $(gx)|_V = g(x|_V)$ for all $g \in G$ and $x \in \mathbb{R}^n$.

Proof. Let $\mathbb{R}^n = V \oplus W$. We can therefore decompose a vector $x \in \mathbb{R}^n$ into a direct sum $x = v \oplus w = v + w$, $v \in V$, $w \in W$. Permutations act linearly, therefore $gx = gv + gw$, and as V and W are invariant subspaces of G, $gv \in V$ and $gw \in W$. Therefore $gx = gv \oplus gw$, so $g(x|_V) = gv = (gx)|_V$. □

Lemma 3.4. Let $G \leq S_n$, with V an invariant subspace of G. For all $z \in \mathbb{Z}^n$, $\mathrm{Stab}_G(z) \leq \mathrm{Stab}_G(z|_V)$.

Proof. Let $g \in \mathrm{Stab}_G(z) = \{g \in G : gz = z\}$. Given that $\mathbb{R}^n = V \oplus W$, then by Lemma (3.3), $g(z|_V + z|_W) = g(z|_V) + g(z|_W) = (gz)|_V + (gz)|_W = z|_V + z|_W$. Now $(gz)|_V + (gz)|_W = z|_V + z|_W \implies (gz)|_V - z|_V = z|_W - (gz)|_W$, where $(gz)|_V - z|_V \in V$ and $z|_W - (gz)|_W \in W$. This implies that $(gz)|_V - z|_V \in V \cap W$, which only contains the 0 vector. Therefore $(gz)|_V - z|_V = 0 \implies g(z|_V) = z|_V \implies g \in \mathrm{Stab}_G(z|_V)$ $\forall g \in \mathrm{Stab}_G(z)$, $\therefore \mathrm{Stab}_G(z) \leq \mathrm{Stab}_G(z|_V)$. □

Let $V \subseteq \mathbb{R}^n$ be an invariant subspace of a transitive group $G \leq S_n$. Then consider the vector $e^{(1)}|_V$. Because G acts transitively, $G(e^{(1)}|_V) = \mathrm{span}\{V\}$. Therefore if we can compute $e^{(1)}|_V$, we can compute a spanning set for V. Since $\mathrm{Stab}_G(e^{(1)}) \leq \mathrm{Stab}_G(e^{(1)}|_V)$ by Lemma (3.4), $e^{(1)}|_V$ is constant under the orbits of $\mathrm{Stab}_G(e^{(1)})$. Let $O_1, O_2, ..., O_k$ be the set of orbits of $\mathrm{Stab}_G(e^{(1)})$. Then

$$e^{(1)}|_V = \sum_{i=1}^{k} \alpha_i \mathbb{1}_{O_i},$$

where $\alpha_i \in \mathbb{R}$, and $\mathbb{1}_{O_i}$ is the characteristic vector of the orbit O_i equal to $\bigoplus_{g \in O_i} ge^{(1)}$. Then let $\{g_1, g_2, ..., g_n\} \subset G$ be a transversal of $\mathrm{Stab}_G(e^{(1)})$ (meaning that $g_j(e^{(j)}) = e^{(1)}$ for each g_j). Then by applying Lemma 5.10 from [**?**], we get that

$$\alpha_i = \langle e^{(1)}|_V, e^{(j)} \rangle = \langle e^{(1)}|_V, e^{(j)}|_V \rangle = \langle e^{(1)}|_V, g_j e^{(1)}|_V \rangle = \langle \sum_{i=1}^{k} \alpha_i \mathbb{1}_{O_i}, g_j \sum_{i=1}^{k} \alpha_i \mathbb{1}_{O_i} \rangle.$$

As there are k α_i's, we get a set of k quadratic equations with k unknowns. By solving these equations for α_i and then substituting the solutions we get back into our first equation we can compute $e^{(1)}|_{V_i}$ for each invariant subspace V_i. Then as stated earlier, we only need to compute $G(e^{(1)}|_{V_i})$ and select linear indepedent vectors to find a basis for the corresponding subspace V_i.

The code for this algorithm was written using Sage, as it can solve systems of quadractic equations, and make calls to GAP for all the group computations. First, we compute the stabilizer and stabilizer orbits of the group, compute the number of distinct orbits, and build a list of symbolic variables:

```
def invariant_subspaces(G, dim):
    e_v = [0]*dim
    stab = gap.Stabilizer(G,1)
```

```
orbs = gap.Orbits(stab,[1..dim])
k = len(orbs)
a = list(var('a_%d' % (i+1)) for i in range(k))
```

Next, we compute the formula for $e^{(1)}|_V$ symbolically:

```
for i in range(k):
    ivec = index_vector(orbs[i+1],dim)
    for j in range(dim):
        e_v[j] = e_v[j] + a[i]*ivec[j]
```

Compute the transversal of the stablizer along with the permutation matrices corresponding to each g_j within the transversal:

```
trans = gap.RightTransversal(G,stab)
pmats = [Matrix(gap.PermutationMat(x,dim).sage())
    for x in trans]
```

We then set up our system of equations as an inner product between $e^{(1)}|_V$ and $g_j e^{(1)}|_V$, solve it, and substitute the solutions into our original symbolic $e^{(1)}|_V$ vector, getting a list of projections $e^{(1)}|_{V_i}$ for each invariant subspace V_i:

```
e_v_vec = vector(e_v)
eqns = [(a[i] == e_v_vec.inner_product(pmats[i]*e_v_vec))
    for i in range(k)]
sols = solve(eqns, a, solution_dict=True)
candidates = [[QQ(e.subs(sol)) for e in e_v]
    for sol in sols]
```

Then we construct a group of permutation matrices M corresponding to our permutation group G, and compute the orbits of M on each projection:

```
M = gap.Group([gap.PermutationMat(p,dim)
    for p in gap.GeneratorsOfGroup(G)])
vec_orbs = gap.Orbits(M, candidates)
return vec_orbs
```

The only thing that is left to do in this algorithm is compute bases for each of these lists of vectors. Since our other method worked for all the groups we cared about, we primarily used this method for verifying that the results found in each method agreed with one another. However if further work was to be performed, this second algorithm would likely be better to build off of as it should compute results within a reasonable time for many more groups than the first algorithm.

3.2 Constructing Core Points & Integer Programs

In order to test out the effectiveness of the added constraints, we'll need to construct integer programs with symmetries that are difficult to solve with standard solvers. For

this purpose, we can use the orbit polytopes of core points of groups. Since the vertices of these polytopes are permutations of a vector, they will have the corresponding group symmetry. In addition, as the orbit polytope of a core point contains no integer points aside from the vertices, if we can cut off these vertices then the integer program corresponding to this polytope will be infeasible. This will ensure that a solver won't by chance rapidly solve the problem, and will have to fully explore the branch and bound tree.

So to find integer programs we can use to test our methods, we'll need to find core points. This is non-trivial, but from [**?**] we have some tools for this. In particular, there is a certain class of groups (those whose representations can be decomposed into at least two rational subspaces, excluding the fixed space) which have an infinite number of non-isomorphic core points, along with a simple method to generate them. Much of the reasoning depends on points that have globally minimal projection.

Definition 3.5 (Globally Minimal Projection)**.** A point $z \in \mathbb{Z}$ has **globally minimal projection** with respect to a subspace $V \subseteq \mathbb{R}^n$ if $||z|_V|| \leq ||z'||_V||$ for all $z' \in \text{aff}(Gz) \cap \mathbb{Z}^n$.

So if a point has globally minimal projection onto a subspace, it is the closest point in the affine hull of Gz to that subspace.

3.3 Quadratic Constraints

From

3.4 Solving Integer Programs with CPLEX

CPLEX is an optimizer made by IBM. It was designed to optimize and solve both LPs and IPs. It can read in our IPs from LP files which are just text files with the .lp file extension and special formatting. IBM has a guide to formatting the LP files however as we spent lots of time working through errors with the format we will outline some specific formatting details for the IP's we're trying to generate.

SAGE output uses '*' as the operator for multiplication between variables and coefficients within the constraints. The LP file format does not recognize '*' as a multiplication operator unless it's a monomial within a quadratic constraint. Therefore upon pasting the SAGE output into the LP all the instances of '*' must be replaced with a space.

Another issue that was encountered was that the SAGE output contained fractions as coefficients which had to be converted to floating point. This loss of accuracy was enough to not correctly construct the polytope, and it was taking trivial amounts of time to realise infeasibility. This problem was solved by scaling the coefficients by the value of their denominator. Upon doing this, we had integer coefficients and therefore lost no accuracy.

Adding the quadratic constraints also requires the manipulation of sage output, this time the fractional coefficient should be relatively easy to represent as floating point

values. However, in order to keep the quadratic constraints in the proper LP form, one can't just replace the fractional number with the floating point equivalent. If we concatenate an '*' onto the end of our fractional number we can create a new string. By replacing all instances of the new string we constructed with our converted floating point with a space concatenated to the end we can quickly and easily make functioning quadratic constraints within the LP file format. This step is necessary because the quadratic constraints need the '*' to combine variables to form monomials. This can be seen just above.

One of the major hurdles encounted when attempting to construct the infeasible LP files are the implicit non-negativity constraints CPLEX imposes if we do not specify a lower bound. We arbitrarily chose the negative equivalent of the upper bound as the lower bound, that way we allow the optimizer to look into the negative numbers as well for integer solutions. Once we fixed this issue as well as all of the above issues, the problem which we had generated took an unreasonable amount of time to solve. Which is exactly what we were trying to accomplish.

Once you have a functioning LP file, getting CPLEX to solve your linear program is nearly trivial. On debian based systems simply start from your installation of CPLEX, and navigate to your bin folder keeping in mind that the folder in the bin will be named depending on your operating system and CPLEX installation. Once inside the innermost folder you should see an executable named 'cplex'. Place any and all LP's you wish to test into the same folder as the CPLEX executable. Execute CPLEX like any other executable, and use the CPLEX read command to read in an LP. Type "opt" to get CPLEX to attempt to optimize the infeasible LPs we created.

At this point CPLEX will output many things

4 Computational Results

None yet, but just got Ace-net account up, may go to the school ASAP to start runnning the LPs.

5 Recommendations

Due to the time constraints of only having a term to work on the project, we didn't get to perform as many experiments to see the change (or lack thereof) in runtime when introducing the quadratic constraints as we would've liked to do. Much of the time was spent solving problems in both algorithm and data formatting. Now that we have a reliable method of building these infeasible systems, it would not be very difficult to run more experiments on various different groups. The results presented in this paper are simply preliminary results, however if our work was extended and more experimentation was done by a third party, we may be able to conclusively say wheither the runtimes were improved on or not when the LPs were subjected to the quadratic constraints.

Some ways to improve the experiment would be to prepare more for C than what we used, by doing the whole range from 1 - 100 rather than just specific values you will end up with more accurate data and a smoother plot. Larger sample sizes are just better in general, as they allow you to more accurately observe trends in the data like runtime increase/decrease. Not only should we plot more of the various potential C values, but also plot the effect of changing C values for a couple different groups. Currently we have functional LP files for the 15-2 and 16-10 groups but they both have 2 subgroups and so it ends up being 4 different groups each with 1 file without quadratic constraints and 2 with the quadratic constraints (marked with a Q).

The work we've been doing could have fringe applications in research optimizer technology, in the case that the results indicate that some value of C makes the optimizer realize the problem is infeasible quicker than it does when it runs without the quadratic constraint. Unfortunately, due to time constraints, we only ended up getting minor calculations done, although all of the setup is done and well documented. This should aid any researcher who desires to fully complete the experiment. Also we will include a link to a repository containing our LP files and the SAGE code we used to get to the point we are at now (which will be located in the appendix). Using that repository, and the contents of this document should be sufficient to replicate our starting conditions.