Symmetry in MILP's and Orbital Branching

Atharv Saxena 2021ME21055, Srishti 2021ME21113

April 24, 2024

1 Introduction

In Mixed Integer Linear Programming (MILP), symmetry refers to situations where the optimal solution has multiple equivalent solutions due to symmetries in the problem structure. These can arise from various sources, such as the structure of the constraints or the objective function.

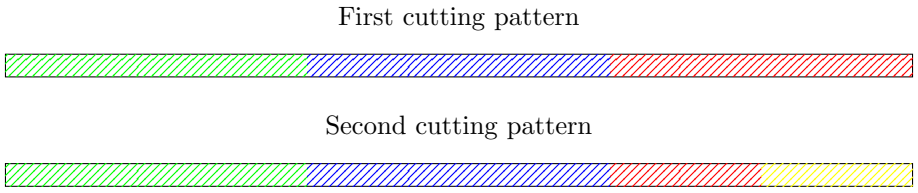
Symmetry can lead to several problems that impact the efficiency and effectiveness of solving the optimization problem. Some of the key problems due to symmetry include increased computational complexity, solution duplicity, degenerate solutions etc.

One approach to counter symmetry is through orbital branching. This method aims to break the symmetry by selecting a special subset of variables called "orbit" and branching on them. By carefully choosing the orbit, orbital branching can effectively reduce symmetry, leading to more efficient MILP solving.

2 Some examples to get started

2.1 Cutting Stock Problem

A very common example for a symmetric problem is the cutting stock problem, where various solutions can yield the same outcome. For instance, cutting a log of length 12 into pieces of 4, 4, and 4 is equivalent to cutting it into pieces of 4, 4, 2 and 2. This redundancy complicates the formulation, increasing computational complexity as the solver must navigate through equivalent solutions.



2.2 Graph Coloring Problem

Given a graph, the goal is to assign colors to its vertices such that no two adjacent vertices share the same color, using the fewest possible colors. Symmetry can occur when different color assignments lead to the same coloring pattern but with the colors swapped. An example of equivalent solutions is given in the following figure.

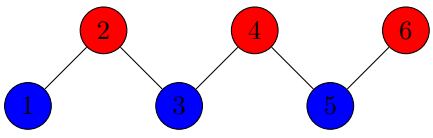


Figure 1: Solution 1

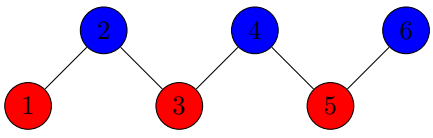


Figure 2: Solution 2

2.3 Travelling Salesman Problem

In the TSP, a salesman must visit a set of cities exactly once and return to the starting city, minimizing the total distance traveled. Symmetry arises when multiple tours have the same total distance because the order of visited cities can be different but still result in the same overall distance.

3 Existing Methods

- **Perturbation**: Slight modification of the ILP coefficients to break symmetry, making the problem easier to solve without significantly affecting the optimal solution. This method is formalized in *Symmetry in Integer Linear Programming, Francois Margot*
- **Graph Isomorphism** This method involves formulating the ILP as a graph, where each node represents a variable or a constraint, and an edge exists between two nodes if the corresponding variable and constraint are related. This method can be seen in *Fran¸cois Margot, Pruning by isomorphism in branch-and-cut*
- **Symmetry-Breaking Inequalities**: Addition of extra constraints to the ILP that exclude non-optimal symmetric solutions, reducing problem size and improving efficiency, as seen in *Symmetry-breaking inequalities for ILP with structured sub-symmetry*
- **Pruning of Search Space and Orbital Branching**: Once the symmetries are detected, they can be used to reduce the search space of the ILP. This is done by partitioning the set of all feasible solutions into equivalence classes or orbits , where two solutions are considered equivalent if one can be obtained from the other by applying a permutation from the symmetry group. These orbits are used to create a valid partitioning of the feasible region which significantly reduces the effects of symmetry while still allowing a flexible branching rule. Search space pruning is given in *Search Space Pruning Constraints Visualization Blake Haugen,Jakub Kurzak.* Orbital branching is formalized in *Orbital Branching JAMES OSTROWSKI, JEFF LINDEROTH*

We will now delve into the specifics and mathematical formulations of orbital branching.

4 Orbital Branching

Orbital Branching is an effective branching method for integer programs containing a great deal of symmetry. The method is based on computing groups of variables that are equivalent with respect to the symmetry remaining in the problem after branching, including symmetry which is not present at the root node. These groups of equivalent variables, called orbits, are used to create a valid partitioning of the feasible region which significantly reduces the effects of symmetry while still allowing a flexible branching rule.

4.1 Mathematical Formulation

Let Π_n be the set of all permutations of $I_n = \{1, ..., n\}$. Given a permutation $\pi \in \Pi_n$ and a permutation $\sigma \in \Pi_n$, let $A(\pi, \sigma)$ be the matrix obtained by permuting the columns of A by π and the rows of A by σ, i.e. $A(\pi, \sigma) = P_\pi A P_\sigma$, where P_π and P_σ are permutation matrices. The symmetry group G of the matrix A is the set of permutations.

$$G(A) = \{\pi \in \Pi_n | \exists \sigma \in \Pi_m \text{ such that } A(\pi, \sigma) = A\}$$

So, for any $\pi \in G(A)$, if $\hat{x}$ is feasible, then $\pi(\hat{x})$ is also feasible, and has the same objective value

Let Γ be the symmetry group of the matrix A, then for a set $S \subseteq I_n$, the orbit of S with respect to Γ is the set of all subsets of I_n to which S can be sent by permutations in Γ, i.e.,

$$\text{orb}(S, \Gamma) = \{S_0 \subseteq I_n | \exists \pi \in \Gamma \text{ such that } S_0 = \pi(S)\}$$

By definition, if $j \in orb(k, \Gamma)$, then $k \in orb(j, \Gamma)$, i.e. the variable x_j and x_k share the same orbit. Therefore, the union of the orbits

$$O(\Gamma) = \bigcup_{j=1}^{n} \text{orb}(\{j\}, \Gamma)$$

forms a partition of $I^n = \{1, 2, ..., n\}$, which we refer to as the orbital partition of Γ, or simply the orbits of Γ. The orbits encode which variables are "equivalent" with respect to the symmetry Γ.

We characterize a node $a = (F_1^a, F_2^a)$,of the branch-and-bound enumeration tree by the indices of variables fixed to one F_1^a and fixed to zero F_2^a at node a.

4.2 Orbital Branching in Branch and Bound

Let $O \in \Gamma_a$ be an orbit of the symmetry group of the subproblem a. If at least one variable $i \in O$ is going to be one, and they are all "equivalent", then we might as well pick one (say i^x) arbitrarily.

For example, suppose $O = \{p, q, r, s\}$, then the best solution that can be found by branching on variables x_p, x_q and x_r will be the same as by branching on x_s. Thus, we can prune nodes corresponding to x_p, x_q and x_r.

4.3 Finding Symmetry Group

The symmetry group of any matrix $A_{m \times n}$ can be found with a brute force approach. We generate all the possible permutations of I^n, and check if $A(\pi, \sigma) = P_\sigma A P_\pi$. Given below is the code for the same:

```python
import itertools
import numpy as np

def calculate_symmetric_group(A):
    m, n = A.shape
    row_perms = list(itertools.permutations(range(m)))
    col_perms = list(itertools.permutations(range(n)))

    G = set() # symmetry group (pi)
    S = set() # corresponding sigmas
    for pi in col_perms:
        for sigma in row_perms:
            A_permuted = A[np.ix_(pi, sigma)]
            if np.array_equal(A_permuted, A):
                G.add(pi)
                S.add(sigma)
                break

    return G, S
```

For example, the symmetry group of a matrix $A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$ will be the set of permutations $\pi \in \left\{ \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \right\}$ In the first case, the corresponding $\sigma = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$, and we are not doing any changes to matrix A.

In the second case, the corresponding $\sigma = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$. We first swap row 2 and 3, and then swapping column 1 and 3. Matrix A is transformed as follows:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \xrightarrow{\text{Swap row 2 \& 3}} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \xrightarrow{\text{Swap col 1 \& 3}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

5 Example Problem

Let us take a simple example of a knapsack problem

$$\min_{x \in \{0,1\}^{n+1}} \{x_5 \mid 2x_1 + 2x_2 + ... + 2 * x_n + x_{n+1} = 2k + 1\}$$

5.1 Traditional Branch and Bound

This is a fairly simple problem. Let us try to solve it using the branch and bound algorithm implemented in python

```
     def branch_and_bound(c, A, b):
    fixed = []
    for i in range(0,len(c)):
        fixed.append(-1)
    def solve_lp(c, A, b,fixed):
        global count
        count += 1
        temp = c/A
        x = np.zeros(len(c))
        if sum(A*np.ones(len(A))) < b[0]:
            return False
        for i in range(0,len(x)):
            if fixed[i] == 0:
                x[i] = 0
            if fixed[i] == 1:
                x[i] = 1
        if sum(x*A)>b[0]:
            return False
        while True:
            minind = temp.argmin()
            if fixed[minind] == 1 or fixed[minind] == 0:
                temp[minind] = 1000000000
                continue
            p = b-np.sum(A*x)
            p = p[0]
            if p/A[minind] <= 1:
                x[minind] = p/A[minind]
            else:
                x[minind] = 1
            temp[minind] = 1000000000
            if sum(A*x) == b:
                break
        opt = np.sum(c*x)
        return x,opt

    def int_check(x):
```

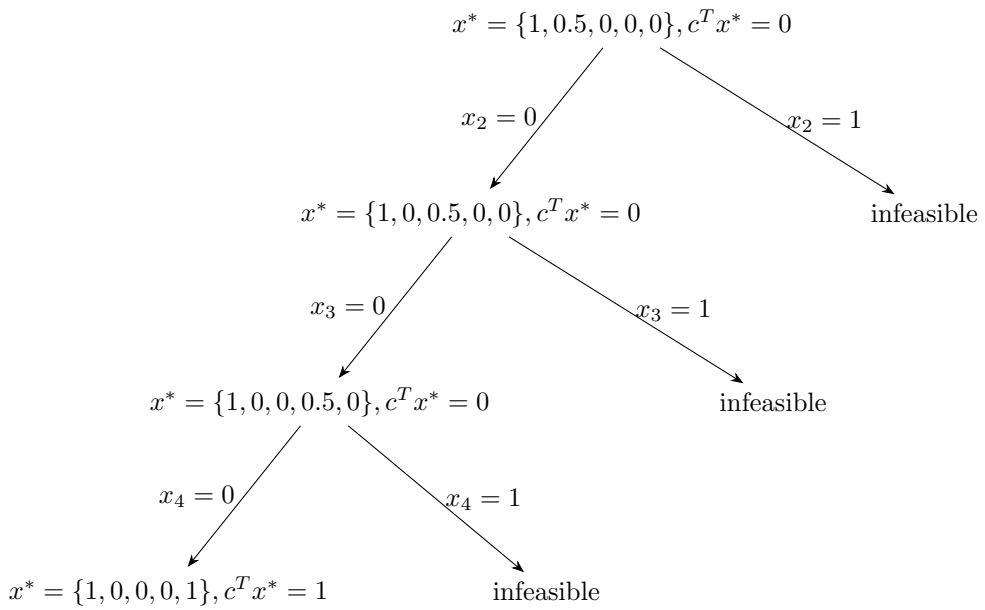
```
        for i in x:
            if i%1 != 0:
                return False
        return True


    def branch(c, A, b, fixed):
        res = solve_lp(c, A, b, fixed)
        if res != False:
            if int_check(res[0]):
                return res
            else:
                for i in range(0,len(c)):
                    if res[0][i]%1 != 0:
                        non_int_index = i
                        break
                fixed[non_int_index] = 0
                res_lower = branch(c, A, b, fixed)
                fixed[non_int_index] = 1
                res_upper = branch(c, A, b, fixed)
                if res_lower == None and res_upper == None:
                    return None
                if res_lower == None:
                    return res_upper
                if res_upper == None:
                    return res_lower
                if res_lower[1] < res_upper[1]:
                    return res_lower
                else:
                    return res_upper
        else:
            return None

    return branch(c, A, b, fixed)
```

For any $k \in Z, 0 \leq k < n$, there will be a fractional solution feasible to the LP relaxation with $x_{n+1} = 0$. Thus, nodes in the branch-and-bound-tree will not be pruned until either k variables are fixed to 1 or $n - k$ variables are fixed to zero. As a result, the enumeration tree will contain at least $min(2^k, 2^{n-k})$ nodes. Data in the phD thesis, *Symmetry in integer programming, James Ostrowski, Lehigh university* shows the results obtained from solving small instances using commercial solver CPLEX with it's advanced features turned off,

n	k	Time(sec)	Nodes
20	9	21.68	352,714
25	6	38.96	657,798
30	6	160.15	2,629,573

Inspection of the enumeration tree, however, reveals that most of the work performed by traditional branch and bound is unnecessary. Many nodes in the tree represent identical subproblems that only need to be solved once.

For a relatively small case, let's say $n = 4$ and $k = 1$, $A = \begin{bmatrix} 2 & 2 & 2 & 2 & 1 \end{bmatrix}$. The enumeration subtree by setting $x_1 = 1$ will be as follows:



The subtree made by setting $x_1 = 0$ and $x_2 = 1$ will consist of similar subproblems, which have to be solved multiple times.

The symmetry group G of A will be the permutation [2 1 3], and thus x_1 and x_2 lie in the same orbit. Hence, we can prune the subtree of $x_1 = 0$ and $x_2 = 1$, which reduces the number of subproblems solved.

5.2 Algorithm for Orbital branching

Let O be an orbit in the orbital partitioning $\mathcal{O}(G(A(F_1^a, F_a^0)))$ and let j and k be two variable indices in O. If $a(j) = (F_1^a \cup \{j\}, F_0^a)$ and $a(k) = (F_1^a \cup \{k\}, F_0^a)$ are the child nodes created when branching on variables x_j and x_k, then $z^*(a(j)) = z^*(a(k))$. The algorithm is formalized as follows

Algorithm 1: Orbital Branching

Input: Subproblem $a = (F_{a1}, F_{a0})$, non-integral solution $\hat{x}$
Output: Two child subproblems b and c

Step 1. Compute orbital partition $O(G(A(F_{a1}, F_{a0}))) = \{O1, O2, ..., Op\}$;
Step 2. Select orbit O_{j^*}, $j^* \in \{1, 2, ..., p\}$;
Step 3. Choose arbitrary $k \in O_{j^*}$. Return subproblems $b = (F_{a1} \cup \{k\}, F_{a0})$ and $c = (F_{a1}, F_{a0} \cup O_{j^*})$;

6 Results and Discussion

In the code we have written, we already have some heuristics and tie breaking mechanisms specific to knapsack problems. Thus, the number of nodes do not increase as exponentially as in general case.

n	Number of Nodes
5	9
20	39
50	99

In a traditional branch and bound algorithm, the number of nodes for $n = 25$ can reach more than $5,00,000$.