Handling Sub-symmetries in Integer Linear Programming using Activation Handlers

Wessel, Sten

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

Handling Sub-symmetries in Integer Linear Programming using Activation Handlers

Master's Thesis

Sten Wessel

August 11, 2022

Supervisors
dr. rer. nat. Christopher Hojny
dr. ir. Tom Verhoeff

Assessment Committee
dr. rer. nat. Christopher Hojny
dr. ir. Tom Verhoeff
dr. ir. Cor A.J. Hurkens
dr. Wouter Meulemans

Abstract

Symmetry in integer programming formulations is a well-studied topic and several methods have been developed to handle symmetries. The main approach is to cut off symmetric solutions, in order to reduce the number of feasible solutions. During the branch-and-bound process, subproblems at the nodes of the branch-and-bound tree may exhibit symmetries that are different from the symmetries at the root node. These so-called *sub-symmetries* are more difficult to handle, as symmetry-handling constraints need to keep track of when the sub-symmetries become active. Existing methods add symmetry-breaking inequalities to the model, using additional binary variables to encode activation. We propose a new method that dynamically activates sub-symmetry-handling constraints using an *activation handler* extension to the SCIP integer programming solver. This allows the modeller to directly implement routines that check for active sub-symmetries. It eliminates the need to encode this in inequalities and variables in the root problem. We show that this approach is flexible, by showing applications in the graph coloring problem, unit commitment problem, and the two-dimensional knapsack problem. Furthermore, our experimental results show that it is competitive with the existing sub-symmetry handling methods.

Contents

Chapter 1

Introduction

Symmetries occur frequently in integer programming problems, for example when modeling many classical combinatorial optimization problems. Even for relatively small problem instance sizes, large symmetry groups can make the integer program hard to solve with traditional methods. Consider the *graph coloring problem* as an example. We are given a graph and we want to assign a color to each vertex, such that adjacent vertices are not assigned the same color. We want to find a coloring of the graph that uses a minimum number of colors. The colors are represented by the integers $\{1,\dots,K\}$, where K is some sufficiently large upper bound on the chromatic number of the graph. The naming of the colors is irrelevant in a solution to the problem. Indeed, any solution can be turned into an equivalent solution where the colors are re-labeled. This symmetry, that arises from permutations of colors, makes the set of feasible solutions unnecessarily large. For problems that can be modeled as an integer program, symmetries in the problem can therefore have a negative impact on the solving process. To remedy this, we want to exclude symmetric solutions to reduce the size of the feasible set.

The symmetry in the problem partitions the set of feasible solutions into sets of equivalent solutions. The main idea to break the symmetry is to select a *representative* solution for every set of equivalent solutions, and subsequently to restrict the feasible set to contain only representative solutions. There exist various methods for handling such symmetries in integer programming, which we will briefly discuss in Chapter 2.

Problems may also exhibit symmetries that only occur in some subsets of solutions. We again look at the graph coloring problem for an example. Take a subset of the vertices R and two colors c_1 and c_2. Suppose that we have a solution where all the neighboring vertices of R are colored with neither c_1 nor c_2. Then, we can permute the colors c_1 and c_2 within the vertex subset R. These permutations within R are only possible in solutions where indeed this condition on the neighbors of R is satisfied, and thus this symmetry is not present in all solutions. Symmetries that only occur for some solution subsets are called *sub-symmetries* [3].

Handling sub-symmetries can reduce the size of the feasible solution space even further, and thus this can have a positive impact on IP solving times. To handle sub-symmetries, it is necessary to *detect* during the solving process when sub-symmetries become active, i.e., when the subproblem during the solving process only yields solutions in which the sub-symmetry occurs. The existing method for sub-symmetry handling is introduced by Bendotti, Fouilhoux and Rottner [4] achieves detection of active sub-symmetries by introducing a new binary variable in the formulation that indicates activation. The handling of sub-symmetry is based on explicit sub-symmetry-breaking inequalities in the IP formulation. The sub-symmetry-breaking

inequalities are appropriately 'turned on' or 'off' by the binary variable.

In this thesis, we propose a new method for handling sub-symmetries in integer programming. Instead of adding new inequalities to the IP formulation, we instead decouple sub-symmetry handling from the formulation itself. Instead, we allow users to extend the IP solver with an *activation handler*: a dedicated routine that detects when a sub-symmetry becomes active. The activation handler can then be linked to high-level constraints in the solver that break the sub-symmetry.

Our new method is flexible in the kinds of symmetries it can detect and handle. To demonstrate this, we apply our method to handle geometric sub-symmetries in two-dimensional packing problems, for which no previous methods exist in integer programming.

The structure of this thesis is as follows. We first discuss necessary preliminaries and related work in Chapter 2. Then, we introduce our method in the framework for sub-symmetry handling in Chapter 3. In Chapters 4 and 5 we apply our new method to the graph coloring problem and the unit commitment problem, respectively. With experiments on benchmark and generated instances, we directly compare our method to the inequalities-method. In Chapter 6 we apply our method to geometric symmetries and sub-symmetries in the two-dimensional knapsack problem. We describe how geometric symmetries and sub-symmetries can be handled, and apply our activation handler method to benchmark instances. In Chapter 7 we conclude the results in this thesis and provide ideas for further research.

Chapter 2

Preliminaries and related work

2.1 Symmetry in integer linear programming

We start with a brief introduction to symmetry in integer programming. We refer the reader to standard material on permutation groups, such as the book by Rotman [40] and the book chapter on symmetry in integer programming by Margot [32] for a more in-depth overview. For more background on integer linear programming, we refer to the book by Conforti, Cornuéjols and Zambelli [11].

For a positive integer n, let $[n] = \{1, \dots, n\}$ denote the set of positive integers up to and including n. Let Sym_n denote the set of all permutations of the ground set $[n]$. We represent a permutation $\pi \in \mathrm{Sym}_n$ as a bijective function $\pi\colon [n] \to [n]$. For a vector x of length n and $\pi \in \mathrm{Sym}_n$, we slightly abuse notation to let $y = \pi(x)$ denote the vector obtained by permuting the entries of x according to π, i.e.,

$$y = (x_{\pi^{-1}(1)}, \dots, x_{\pi^{-1}(n)}). \tag{2.1}$$

Let $\pi_2 \circ \pi_1$ denote composition of permutations, i.e., $(\pi_2 \circ \pi_1)(x) = \pi_2(\pi_1(x))$. Note that Sym_n forms a group under composition, with the identity permutation id, that maps every number to itself, as the unit element.

Let $c \in \mathbb{R}^n$ be a vector of length n, $b \in \mathbb{R}^m$ a vector of length m, and $A \in \mathbb{R}^{m \times n}$ a matrix of size $m \times n$. Let $x \in \{0,1\}^n$ be a vector of length n of binary decision variables. We then consider the integer linear program (IP) of the form

$$\begin{aligned} \text{minimize} \quad & c^\top x && (2.2)\\ \text{subject to} \quad & Ax \leq b, && (2.3)\\ & x \in \{0,1\}^n. && (2.4) \end{aligned}$$

We also refer to integer linear programs as *integer (linear) programming models* or *models*. It can be the case that there are also variables in the formulation that are not constrained to be binary or integer, in which case we call the program a mixed-integer linear program (MIP). Binary variables are used in many problems that contain symmetry and therefore many symmetry-handling methods are focussed on handling binary variables, and we therefore also consider symmetry handling on binary variables. We define

$$\mathcal{X} = \{x \in \{0,1\}^n \mid Ax \leq b\} \tag{2.5}$$

to denote the set of all *feasible solutions* of the IP, which we will also call the *feasible set* of the IP. A feasible solution $\hat{x} \in \mathcal{X}$ is *optimal* when $c^\top \hat{x} \leq c^\top x$ for all $x \in \mathcal{X}$. A *symmetry* of the integer linear program is defined as a permutation $\pi \in \mathrm{Sym}_n$ such that for any feasible solution $x \in \mathcal{X}$, the vector $\pi(x)$ is also a feasible solution *with the same cost*. The set of all such permutations forms a subgroup of Sym_n

$$G = \{\pi \in \mathrm{Sym}_n \mid \forall_{x \in \mathcal{X}} : \pi(x) \in \mathcal{X} \text{ and } c^\top x = c^\top \pi(x)\} \tag{2.6}$$

which we call the *symmetry group* of the IP. The *orbit* of a given $x \in \{0,1\}^n$ under G is

$$\mathrm{orb}_G(x) = \{\pi(x) \mid \pi \in G\}. \tag{2.7}$$

In other words, two vectors $x, y \in \{0,1\}^n$ are in the same orbit if there exists a permutation in G sending one to the other. Define

$$\mathcal{O} \coloneqq \{\mathrm{orb}_G(x) \mid x \in \mathcal{X}\} \tag{2.8}$$

as the set of all orbits of the feasible solutions $\mathcal{X}$ under G. Note that $\mathcal{O}$ partitions $\mathcal{X}$.

The symmetry group G for a given integer linear program is defined in terms of its feasible set. Computing and therefore handling the full symmetry group G is NP-hard [32]. Instead of considering the full symmetry group G, in practice only some subgroup G^{LP} of *formulation symmetries* is considered that leave the description $Ax \leq b$ and the objective function c invariant [32]. A permutation $\pi \in \mathrm{Sym}_n$ of the variables is a formulation symmetry of the IP when $\pi(c) = c$ and there exists a permutation $\sigma \in \mathrm{Sym}_m$ of the rows of A such that $A_{\sigma(i),\pi(j)} = A_{ij}$ for all $i \in [m]$, $j \in [n]$ and $\sigma(b) = b$, i.e., the rows of A are permuted by σ and the columns by π.

Computing (generators of) the formulation symmetry group can be done by finding graph automorphisms, see Salvagnin [41] for a construction of this graph. The complexity of the graph automorphism problem is currently still unknown. However, several software implementations such as `nauty` [33] and `bliss` [21] exist that are able to find graph automorphisms efficiently in practice. Alternatively, symmetries in the integer linear program can be specified explicitly by the modeler of the problem. The modeler might use structure in the problem to derive additional symmetry that cannot be detected automatically in the formulation. In the remainder, we will mostly consider formulation symmetries in the integer linear programs.

2.2 Solving integer programs

Before discussing the methods that exist to handle symmetry in integer programming, we need to introduce some concepts used in solving integer programs. We will only give a brief introduction, and refer to the book by Conforti, Cornuéjols and Zambelli [11] for a more in-depth overview.

A naive way of solving the program (2.2)–(2.4) is to enumerate all vectors $x \in \{0,1\}^n$ and to pick an optimal solution. This would not be a feasible approach when the number of variables is large. Luckily, algorithms exist that in practice have been proven effective for solving integer programs in practice. For the program (2.2)–(2.4) we can define its *natural linear programming (LP) relaxation* as

$$\min\{c^\top x : Ax \leq b,\ 0 \leq x \leq 1\} \tag{2.9}$$

where we drop the constraint that the variables in x must be integral. The LP relaxation can be solved in polynomial time using the ellipsoid method by Khachiyan [24] or interior-point

method by Karmarkar [23]. However, the older simplex method by Dantzig [12] is most often used as it is faster in practice, even though it does not run in polynomial time asymptotically, according to the currently known pivot rules.

Notice that the value of an optimal solution to the LP relaxation is an upper bound for the optimal solution of the IP. If an optimal solution of the LP relaxation happens to be integral, then it is also a solution of the IP. This observation forms the basis of the *branch-and-bound* (B&B) method for solving integer programs [11]. We start by computing an optimal solution x^* of the LP relaxation. If the solution happens to be integral, then we are done. If not, select a variable x_i^* in the solution of the LP relaxation that has a fractional value f. We then *branch* on this variable, by creating two new IP *subproblems*. In one branch, we set $x_i = 0$, while in the other we branch on $x_i = 1$, and we recursively continue solving the subproblems recursively. This forms an *enumeration tree*, to which we also refer to as the *branch-and-bound* tree. In every node of the B&B tree, the corresponding subproblem is thus defined by sets of *variable fixings*. Next to branching, we can also use the solution objective value to *bound* the optimal solution of the IP. This allows us to prune or cut off nodes when the solution of the LP-relaxation is below the current best solution.

The branch-and-bound method described here forms the basis, but various choices in, e.g., on which variable to branch, are open. Modern solvers implement a number of heuristics and other solving steps to make the solving process faster. One of these steps is symmetry handling, which we will be discussing next.

2.3 Symmetry handling methods

The general idea of handling symmetry in integer linear programs is to choose a *representative* of each orbit of feasible solutions, and to restrict the solution space to these representatives [32]. A common choice of the representative is to choose the *lexicographically maximal* solution in each orbit. A vector $y \in \mathbb{Z}^n$ is *lexicographically greater than* $z \in \mathbb{Z}^n$, denoted by $y \succ z$, if there exists an $i \in [n]$ such that $y_i > z_i$ and $y_j = z_j$ for all $j < i$. We write $y \succeq z$ when the vector y is lexicographically greater than or equal to z. Then, a solution $x \in \mathcal{X}$ is lexicographically maximal in its orbit under G when $x \succeq \pi(x)$ for all $\pi \in G$. A method can be *fully* or *partially* symmetry-breaking when the solution space is exactly or partially restricted to the set of representatives, respectively.

Orbital fixing One of the methods that handles symmetries is *orbital fixing*, introduced by Margot [31], which is based on variable fixings. Let a be a given node of the branch-and-bound tree, and let F_0^a and F_1^a denote the sets of variables that are fixed at 0 and 1 at node a, respectively. Variables may be in F_0^a or F_1^a due to, e.g., branching on the variable in ancestor nodes. Then it might be possible to set further variables to 0, or to even cut off the current node to handle symmetry.

First, we need to introduce the concept of a *stabilizer* from basic group theory. For a subgroup G of Sym_n, the stabilizer of a set $S \subseteq [n]$ for the symmetry group G is defined as

$$\mathrm{stab}_G(S) = \{\pi \in G : \pi(S) = S\}, \tag{2.10}$$

that is, all the permutations in G that leave S invariant. Note that $\mathrm{stab}_G(S)$ is a subgroup of G. The orbital fixing rule is then as follows. For every orbit O of $\mathrm{stab}_G(F_1^a)$:

- If $O \cap F_0^a \neq \emptyset$, then all variables in O can be fixed to 0.

- If $O \cap F_1^a \neq \emptyset$, then all variables in O can be fixed to 1.

In other words, if some variables in the orbit are fixed to 0 or 1, then the other variables in the orbit can be fixed to 0 or 1 as well, respectively.

This method for symmetry handling is purely based on *propagation*, i.e., further reducing bounds or fixing of variables. Other approaches may be to add symmetry breaking inequalities to the formulation.

Orbitopes We will also consider IP formulations where the variables can be organized in a matrix $x = (x_{i,j})_{i\in[n],j\in[m]}$, where the symmetry arises from permutations of the columns of x. That is, we consider Sym_n as the subgroup of the symmetry group of the IP, where the permutations act on the columns of x. We define a solution matrix x to be lexicographically maximal in its orbit when its columns are lexicographically non-increasing. As for vectors of variables, we can break symmetry by choosing lexicographically maximal matrices as the representative solutions. The convex hull of binary matrices with lexicographically non-increasing columns is called the *full orbitope*, as introduced by Kaibel and Pfetsch [22]. Several methods exist for restricting the solution to the full orbitope, such as a propagation-based approach by Bendotti, Fouilhoux and Rottner [3], and separation of inequalities that describe the orbitope, by Hojny and Pfetsch [17].

For more details on symmetry handling methods, we refer to the overview by Margot [32] and a computational survey by Pfetsch and Rehn [38].

2.4 Sub-symmetries in integer linear programming

Consider a subset $Q \subset \mathcal{X}$ of solutions of the formulation (2.2)–(2.4), and consider the *subproblem*

$$\min\{c(x) \mid x \in Q\}$$

with symmetry group G_Q. In general, the group G_Q is different from G. In particular, it may be the case there exist symmetries in subproblems that are not present in G. To see this, consider the following example from Bendotti, Fouilhoux and Rottner [3]. Suppose we have an integer program with solution set $\mathcal{X} = \{x^1, x^2, y\} \subseteq \{0,1\}^3$ with

$$x^1 = (1,0,1), \quad x^2 = (1,1,0), \quad y = (0,1,0).$$

Also suppose that the solutions x^1 and x^2 have the same objective value, i.e., $c(x^1) = c(x^2)$. Define a solution subset $Q \subset \mathcal{X}$ of solutions that have exactly two 1-entries, thus $Q = \{x^1, x^2\}$. Consider now the permutation $\pi_{132} \in \mathrm{Sym}_3$, defined by the mappings $1 \mapsto 1$, $2 \mapsto 3$, $3 \mapsto 2$, i.e., when the permutation is applied to a vector x, it swaps the second and third entry of x. Notice that $\pi_{132}(x^1) = x^2$ and vice versa. Thus, π_{132} is in the sub-symmetry group G_Q, but not in the symmetry group G, since we have $\pi_{132}(y) = (0,0,1) \notin \mathcal{X}$.

We call the permutations in G_Q *sub-symmetries* of the problem [3]. Note that every node of the branch-and-bound tree corresponds to a subproblem, and therefore it can be beneficial to handle the sub-symmetries if they appear frequently during the solving process. It can be expensive however to compute G_Q for every subproblem, but we can leverage sub-symmetry deduced from the problem structure for particular solution subsets Q. In general, we consider predefined solution subsets $\mathbb{S} = \{Q_s \subset \mathcal{X} \mid s \in [q]\}$ for some positive q, that exhibit predefined sub-symmetries. During the branch-and-bound process, a node in the B&B tree corresponds

to a subproblem that may belong to one or more solution subsets Q_s. We then say that the sub-symmetries in G_{Q_s} become *active*, following the terminology in [3].

To handle the sub-symmetries of all Q_s simultaneously, the general idea remains the same: we want to disregard solutions such that for each orbit at least one representative remains. For a solution subset Q_s, let σ_k^s for $k \in [o_s]$ denote the orbits defined by G_{Q_s}. Furthermore, let $\mathcal{O} = \{\sigma_k^s \mid s \in [q], k \in [o_s]\}$ denote the family of all orbits of the considered solution subsets Q_s. Notice now that $\mathcal{O}$ does not partition the solutions of the solution subsets, as the orbits of different solution subsets may overlap. As a consequence, we need to be more careful in choosing the representative solution $r(\sigma)$ for each orbit $\sigma \in \mathcal{O}$, as for a given orbit σ the set $\sigma \setminus \{r(\sigma)\}$ may contain a representative of another orbit $\sigma' \in \mathcal{O}$.

To formalize this, Bendotti, Fouilhoux and Rottner [3] introduce *generalized orbits*, which intuitively group intersecting orbits of $\mathcal{O}$ together. To this end, they define

$$G(x) = \bigcup_{Q_s \ni x} G_{Q_s} \tag{2.11}$$

as the set of permutations π such that $\pi(x)$ is a symmetric solution to x. The authors then define a relation on the solutions in $\mathcal{X}$. A solution $x' \in \mathcal{X}$ is said to be in relation with x when there exist $r \in \mathbb{N}$ and permutations $\pi_1, \ldots, \pi_r$ such that

$$\pi_i \in G(\pi_{i-1} \circ \cdots \circ \pi_1(x)) \text{ for all } k \in [r] \text{ and } x' = \pi_r \circ \pi_{r-1} \circ \cdots \circ \pi_1(x). \tag{2.12}$$

That is, a solution x' is in relation with x if there is a sequence of permutations such that each application of a permutation yields a symmetric solution in one of the groups G_{Q_s}, where the final application yields x'. The *generalized orbit* $\mathbb{O}$ *of a solution* x *with respect to* $\{Q_s \mid s \in [q]\}$ is the set of all solutions x' that are in relation with x. Notice that the relation is an equivalence relation, and therefore the generalized orbits partition the solutions $\mathcal{X}$. Also notice that for a generalized orbit $\mathbb{O}$, there exist orbits $\sigma_1 \ldots, \sigma_p \in \mathcal{O}$ such that

$$\mathbb{O} = \bigcup_{i \in [p]} \sigma_i, \tag{2.13}$$

which follows almost directly from the definition. A set of representatives $\{r(\sigma) \mid \sigma \in \mathcal{O}\}$ of each orbit is called *orbit-compatible* when for every generalized orbit $\mathbb{O} = \bigcup_{i \in [p]} \sigma_i$ there exists j such that $r(\sigma_j) = r(\sigma_k)$ for all $k \in [p]$ where $r(\sigma_j) \in \sigma_k$. Such a $r(\sigma_j)$ is called a *generalized representative* of $\mathbb{O}$ (notice that a generalized orbit may have multiple generalized representatives). In other words, generalized representatives must be the representative solution of every orbit it is contained in. This exactly captures our condition for carefully choosing representatives for handling multiple sub-symmetries simultaneously; when we remove all solutions in $\bigcup_{\sigma \in \mathcal{O}}(\sigma \setminus \{r(\sigma)\})$, there remains at least one solution in every generalized orbit.

In the remainder, we consider specific solution subsets Q such that we can characterize for which representatives orbit-compatibility is achieved. For a solution matrix $x \in Q$, let $x(R, C)$ denote the submatrix of x obtained by restricting to rows $R \subseteq [m]$ and columns $C \subseteq [n]$. The symmetry group G_Q is the *sub-symmetric group* with respect to (R, C) if it contains all the permutations of the columns of $x(R, C)$. If G_Q is the sub-symmetric group, then Q is called *sub-symmetric* with respect to (R, C). Now, let $\mathbb{S}$ be a set of solution subsets such that every $Q_s \in \mathbb{S}$ is sub-symmetric with respect to (R_s, C_s). For every orbit σ_s^i of G_{Q_s}, choose the representative $x_s^i \in \sigma_s^i$ such that the submatrix $x_s^i(R_s, C_S)$ is lexicographically maximal in its orbit, i.e., its columns are lexicographically non-increasing. In [3], the authors show the these representatives are orbit-compatible, as formulated in the following lemma.

Lemma 2.1. *The set of lexicographically maximal representatives* x_s^i *is orbit-compatible.*

2.5 Sub-symmetry-breaking inequalities

One way of handling sub-symmetries as described above is by adding inequalities to the model that cut off non-representative solutions, such as the method of Bendotti, Fouilhoux and Rottner [4], which makes use of the result in Lemma 2.1.

Let $\mathbb{S} = \{Q_s \mid s \in [q]\}$ be a set of solution subsets such that each Q_s is sub-symmetric with respect to (R_s, C_s), with $R_s = \{r_1^s, \ldots, r_{|R_s|}^s\}$ and $C_s = \{c_1^s, \ldots, c_{|C_s|}^s\}$ where $r_1^s < \cdots < r_{|R_s|}^s$ and $c_1^s < \cdots < c_{|C_s|}^s$. For every solution subset Q_s, we introduce an integer variable z_s such that $z_s = 0$ if and only if $x \in Q_s$. Let $c_j, c_{j-1} \in C_s$ be two consecutive columns in the submatrix $x(R_s, C_s)$. Then, the partial-sub-symmetry breaking inequality is defined as follows.

$$x_{r_1,c_j} \leq z_s + x_{r_1,c_{j-1}} \qquad \text{where } r_1 = \min R_s. \tag{2.14}$$

Notice that when $z_s \geq 1$, the inequality is trivially satisfied, which means we do not attempt to break the sub-symmetry whenever $x \notin Q_s$. These inequalities combined, for all pairs of consecutive columns, ensure that the first row of the submatrix $x(R_s, C_s)$ is lexicographically non-increasing whenever $z_s = 0$. This only partially breaks the symmetry, as Inequality (2.14) only considers the first row. Indeed, when $x_{r_1,c_j} = x_{r_1,c_{j-1}}$, subsequent rows need to be considered until a *tie-break* row is found. The main idea is to construct a set $\tilde{\mathbb{S}}$ from $\mathbb{S}$ with additional *tie-break subsets* for which Inequality (2.14) breaks the symmetry on rows where the previous rows have equal entries.

The set $\tilde{\mathbb{S}}$ can be defined as

$$\tilde{\mathbb{S}} = \{\tilde{Q}_s(i,j) \mid s \in [q],\, i \in \{1, \ldots, \mid R_s\},\, j \in \{2, \ldots, \mid C_s\}\} \tag{2.15}$$

where

$$\tilde{Q}_s(i,j) = \{x \in Q_s \mid x_{r,c_{j-1}^s} = x_{r,c_j^s} \text{ for all } r \in \{r_1^s, \ldots, r_{i-1}^s\}\}. \tag{2.16}$$

Note that for a solution $x \in \tilde{Q}_s(i,j)$ columns c_{j-1}^s and c_j^s are equal for the rows r_1^s up to r_{i-1}^s. Also note that $\tilde{Q}_s(1,j) = Q_s$, for all $j \in \{2, \ldots, \mid C_s\}$. We refer to Bendotti, Fouilhoux and Rottner [4] for the proof that using $\tilde{\mathbb{S}}$ is full-symmetry breaking of the solution subsets $\mathbb{S}$. The key observation is that tie-break subset $\tilde{Q}_s(i,j)$ corresponds to the sub-symmetry where the columns of the submatrix $(\{r_i^s, \ldots, r_{|R_s|}^s\}, \{c_{j-1}^s, c_j^s\})$ can be permuted. When $x \in \tilde{Q}_s(i,j)$, Inequality (2.14) then indeed breaks the tie to ensure full lexicographical ordering of the columns of x.

Lastly, we note that the number of inequalities that need to be added to the model can be large depending on the chosen $\mathbb{S}$. Also note that each (tie-break) solution subset requires an additional integer variable z_s. However, sometimes z_s can be expressed as a linear function in x, in which case it is not necessary to add the variable z_s explicitly, as instead the linear expression can be used directly. Otherwise, the variable z_s can be expressed using additional inequalities and/or integer variables.

Chapter 3

Dynamic activation of sub-symmetries

In this chapter, we introduce a new approach on handling sub-symmetries in integer linear programming. The general idea is to de-couple detection of when sub-symmetries become active from the IP formulation. Instead, we provide a way for the problem modeller to use custom routines to determine active sub-symmetries. We will call these routines *activation handlers*. An activation handler is coupled with a high-level symmetry-breaking constraint, where the activation handler acts as a guard for the constraint. Whenever a sub-symmetry becomes active, the activation handler *activates* the symmetry-breaking constraint.

As a general outline, we distinguish three steps in sub-symmetry handling methods.

1. Identification of sub-symmetries present in the IP formulation.
2. Detection during the solving process of when sub-symmetries become active.
3. Breaking active sub-symmetries by activating symmetry-breaking constraints.

Throughout this work, we assume that sub-symmetries for the given problem are identified beforehand. That is, we assume that a set of solution subsets $\mathbb{S} = \{Q_s \subseteq \mathcal{X} \mid s \in [q]\}$ is given, with known symmetries that do not exist in the root problem. Here, $\mathcal{X}$ denotes the feasible region of the root problem.

Note that it is possible to detect formulation symmetries in the root problem automatically, as described in Section 2.1, by finding automorphisms in a graph representing the structure of the formulation. Extending this to detecting sub-symmetries would be a non-trivial task, however. It is not feasible to compute the symmetry group of every subproblem, as this would require enumerating all the subproblems, which would take as much time as solving the root problem in its entirety. A possible remedy would be to find additional symmetries that occur after only a few variables have been fixed. This can for example be done with finding *almost symmetries* in the graph representing the problem formulation, as described by Knueven, Ostrowski and Pokutta [25]. However, only a limited number of sub-symmetry subsets can be identified with this approach, as a small number of allowed variable fixings is inherently limiting. Therefore, identifying these sub-symmetries would generally require knowledge of the structure of the modeled problem. We further focus on detecting when the identified sub-symmetries become active and breaking active sub-symmetries.

As already described in Section 2.5, the existing method to handle sub-symmetries is using sub-symmetry-breaking inequalities. In our outline, detecting active sub-symmetries is done by introducing additional binary z-variables in the formulation that indicate activation. Then,

breaking active sub-symmetries is done with explicit inequalities in the formulation, which are turned off and on with the z-variables.

One of the downsides of this approach is that many inequalities are necessary to fully break the symmetry. In addition, sub-symmetry-breaking inequalities for all the identified solution subsets must be added to the root problem, even though they are still 'turned off' as the sub-symmetries are not yet active. Lastly, since the detection and breaking of sub-symmetry is directly encoded in the IP formulation, it requires some effort of the modeler to translate the identified sub-symmetries to explicit linear expressions in the model. We instead now propose a more lightweight method where this translation is not necessary.

3.1 Detecting sub-symmetries with an activation handler

In our method, we provide an extension to the integer programming solver that allows the user to write a custom routine that detects when a sub-symmetry becomes active. The activation handler is called at a node of the branch-and-bound tree, and has access to the current state of the solver. The activation handler returns a YES or NO result, indicating when the sub-symmetry becomes active.

The specific implementation of activation handlers depends on the sub-symmetries that exist in the problem. We refer to Chapters 4, 5 and 6 for more details on the activation handlers used in our applications.

3.2 Activating sub-symmetry breaking constraints

The activation handler is used to activate or de-activate constraints that break the sub-symmetry. Because the activation handler is defined separate from the sub-symmetry breaking constraint itself, the activation handler can be used with any kind of constraint in the solver. We modify constraints such that they can be linked to an activation handler. When linked, the solver first checks the activation handler at every node if the constraint should be handled. Optionally, when the constraint activates, the activation handler may provide additional data to the constraint that describes the activated sub-symmetry. We will see an example of this in the application to the unit commitment problem in Chapter 5.

3.3 Interfacing with SCIP

The SCIP framework is a highly modularized software system with a small core, surrounded by a number of *plugins* that extend the core functionality [1]. In the current framework, many *default* plugins are defined that together form the full functionality of the MIP solver. Plugins are separated into *plugin types*. An important plugin type are the *constraint handlers*, that define classes of constraints that the solver can handle. For example, there exists linear constraints, handling linear inequalities, but also specialized constraints such as orbitope constraints that can be used for breaking symmetries.

We introduce the activation handler as a new plugin type in SCIP. Specific activation handlers can then be created as a new plugin in the framework. This allows each activation handler to maintain its own internal data structure, and the user has full control of how the activation handler behaves. When creating a new activation handler, it is required to implement a function that is the entry point of the activation handler. When called, the current status of the

solving process is available to the activation handler. It then produces a YES or NO result, to indicate whether it activates.

We furthermore extend the constraint handlers such that they can be linked to an activation handler. When linked, the constraint handler calls the entry point of the activation handler and depending on the result the constraint is handled or ignored.

In the following chapters, we will look at three applications where we use our method to handle sub-symmetries. We will define specific activation handlers for the sub-symmetries in these problems and compare the results with the existing inequalities method.

Chapter 4

Application to graph coloring problem

We consider the graph coloring problem (GCP) as an example to illustrate how the framework of Chapter 3 can be applied to handle sub-symmetries. Graph coloring has previously been used to evaluate the sub-symmetry-breaking approach using inequalities in [4]. We replicate the experimental setting as much as possible for our approach to symmetry handling, in order to directly compare the two methods.

Consider an undirected graph $G = (V, E)$ with vertex set $V = [n]$ and edge set $E \subseteq \binom{V}{2}$. A function $c\colon V \to [K]$ is called a *vertex coloring* of G if for any pair $\{i, j\} \in E$ holds that $c(i) \neq c(j)$. For a given vertex $i \in V$, we call $c(i)$ the *color* of i. The minimum number of colors in any vertex coloring of G is called the *chromatic number* of G, denoted by $\chi(G)$.

The *graph coloring problem* for a given graph G asks us to find a vertex coloring that uses a minimum number of colors. The problem is well-known to be NP-hard [15], and has been studied extensively in the literature, see e.g. Malaguti and Toth for an overview [30].

Let the integer K be a given upper bound on $\chi(G)$. Define $x = (x_{i,k})_{i\in V,\ k\in[K]}$ as a matrix of variables, with rows indexed by the vertices V and columns by the colors $[K]$. The variable $x_{i,k}$ indicates that vertex $i \in V$ is assigned color $k \in [K]$. Define furthermore the variables y_k for every $k \in [K]$, indicating that color k is used to color at least one vertex. Then the classical IP formulation for this problem is as follows [10].

$$\text{minimize} \quad \sum_{k=1}^{K} y_k \tag{4.1}$$

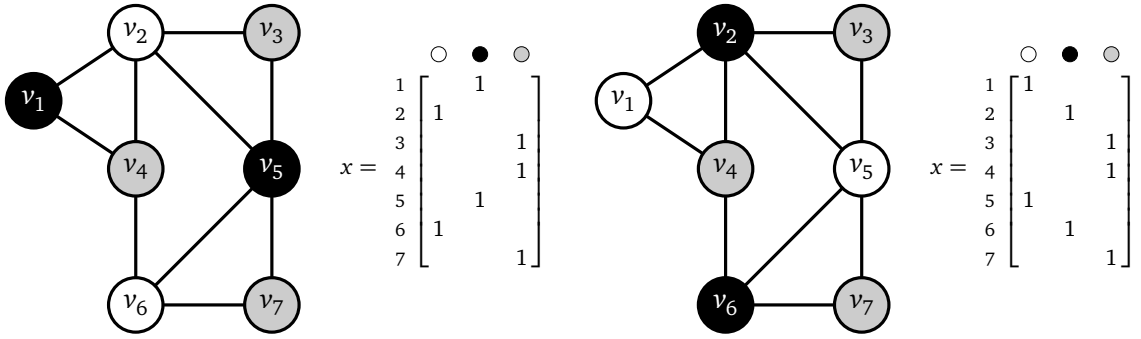
$$\text{subject to} \quad x_{i,k} + x_{j,k} \leq y_k \qquad \forall_{\{i,j\}\in E},\ \forall_{k\in[K]}, \tag{4.2}$$

$$\sum_{k=1}^{K} x_{i,k} = 1 \qquad \forall_{i\in V}, \tag{4.3}$$

$$x_{i,k} \in \{0,1\} \qquad \forall_{i\in V},\ \forall_{k\in[K]}, \tag{4.4}$$

$$y_k \in \{0,1\} \qquad \forall_{k\in[K]}. \tag{4.5}$$

Constraint (4.2) ensures that adjacent vertices cannot both be assigned color k when this color is used, while Constraint (4.3) ensures that every vertex is assigned exactly one color. Note that in any optimal solution for this IP, the variable y_k is zero if and only if color k is not used, since we are minimizing the sum of all y_k. Note that an optimal solution to the integer program is fully determined by x, as from x we can uniquely determine variables y_k corresponding to an optimal solution. We denote the set of all feasible solutions x as $\mathcal{X}_{\text{GCP}}$.



(a) (b)

Figure 4.1: Example graph $G = (V, E)$ on 7 vertices, with vertices colored with the colors 1 (white), 2 (black), and 3 (gray). Figures (a) and (b) show equivalent colorings, where the colors 1 and 2 are permuted between the two figures. The corresponding matrix x has lexicographically ordered columns, and hence is the representative solution in its orbit. Solution (a) is cut-off by symmetry handling.

4.1 Symmetry in the graph coloring problem

In any solution of the graph coloring problem, it is possible to obtain an equivalent solution by permuting color indices, see also Figure 4.1 for an example. In the IP formulation (4.1)–(4.5), colors correspond to the columns of the matrix x, and therefore this symmetry corresponds to permutations of columns of x. In other words, the symmetry group of the IP contains the symmetric group acting on the columns of x. This global symmetry can be handled by restricting the columns of x to be lexicographically non-increasing. There are multiple approaches to add these constraints to the model, but following the construction of [4], we add *column inequalities*. These symmetry-breaking inequalities have been introduced for the GCP by Méndez-Díaz and Zabala [34, 35] and has later been generalized by Kaibel and Pfetsch [22]. In our setting this translates to the inequalities

$$\sum_{k'=k}^{\min\{i,K\}} x_{i,k'} \leq \sum_{i'=k-1}^{i-1} x_{i',k-1} \qquad \forall_{i\in V},\ \forall_{2\leq k\leq \min\{i,K\}}. \tag{4.6}$$

These are a polynomial number of additional inequalities and they fully break the symmetry of the color permutations. These inequalities are usually not explicitly added to the formulation, but instead are separated in the linear-time separation algorithm described in [22]. Since the separation can be done in linear time, it will be faster than adding the inequalities explicitly beforehand.

The formulation (4.1)–(4.5) also exhibits sub-symmetries, with the following observation from Bendotti, Fouilhoux and Rottner [4]. Consider two colors c_1 and c_2 and a subset of the vertices $R \subseteq V$ such that the neighbors of R, denoted by $N(R)$, are colored with neither c_1 nor c_2. Then, the colors c_1 and c_2 can be permuted within the set of vertices R, see also Figure 4.2 for an example. More formally, consider the solution subset

$$Q^R_{c_1,c_2} = \{x \in \mathcal{X}_{\text{GCP}} \mid x_{i,c_1} = x_{i,c_2} = 0 \quad \forall_{i\in N(R)}\}. \tag{4.7}$$

Then this solution subset is sub-symmetric with respect to $(R, \{c_1, c_2\})$, i.e., the symmetry group of the subproblem corresponding to $Q^R_{c_1,c_2}$ contains the permutations acting on the columns

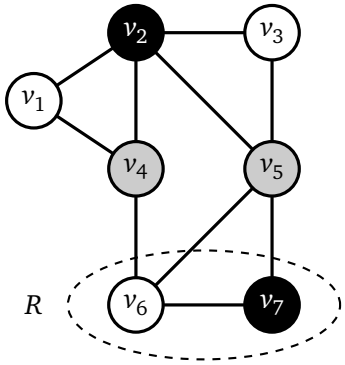


Figure 4.2: Example graph with a zero-neighbor-sub-symmetry, with $R = \{v_6, v_7\}$, $c_1 = 1$ (white), and $c_2 = 2$ (black). We have that $N(R) = \{v_4, v_5\}$ and these vertices are colored with neither c_1 nor c_2. Hence, the colors c_1 and c_2 can be permuted inside R.

of the submatrix $x(R, \{c_1, c_2\})$. These sub-symmetries are referred to as *zero-neighbor-sub-symmetries*. Note that there are an exponential number of sub-symmetric solution subsets, as there are exponentially many subsets of the vertices V. Handling all sub-symmetries might therefore be infeasible, as the added overhead might negatively impact the solving time. Thus in practice we have to make a choice in what solution subsets to consider.

For the sake of comparison, we choose to follow the approach in [4], and therefore we construct the subsets R by first choosing two distinct vertices $s_1, s_2 \in V$ and selecting R to be the vertices non-adjacent to s_1 and s_2, i.e.,

$$R = V \setminus (N(s_1) \cup N(s_2) \cup \{s_1, s_2\}). \tag{4.8}$$

Then, we consider the following solution subset

$$Q_{c_1,c_2}^{s_1,s_2,R} = \{x \in \mathcal{X}_{\text{GCP}} \mid x_{s_1,c_1} = 1,\, x_{s_2,c_2} = 1,\, x_{i,c_1} = x_{i,c_2} = 0\ \forall_{i \in N(R)}\}. \tag{4.9}$$

Thus, in the subsets we consider we will furthermore constrain s_1 and s_2 to be colored by c_1 and c_2, respectively. Define

$$\mathbb{S}_{\text{GCP}} = \{Q_{c_1,c_2}^{s_1,s_2,R} \mid s_1, s_2 \in V,\, s_1 < s_2,\, c_1, c_2 \in [K],\, c_1 < c_2\} \tag{4.10}$$

as the set of all solution subsets that contain zero-neighbor-sub-symmetries that we consider. Note that the size of $\mathbb{S}_{\text{GCP}}$ is in the order of $\mathcal{O}(n^2K^2)$.

4.2 Sub-symmetry-breaking inequalities

We now briefly summarize how sub-symmetry-breaking inequalities are used to break the sub-symmetries in $\mathbb{S}_{\text{GCP}}$. Details of the full derivation are omitted, as our goal is only to compare the resulting inequalities with our novel activation handler approach. We refer to Section 2.3 of [4] for a more detailed derivation.

Let $Q_{c_1,c_2}^{s_1,s_2,R} \in \mathbb{S}_{\text{GCP}}$ be a solution subset. Then, the associated z-variable can be expressed as

$$z = (1 - x_{s_1,c_1}) + (1 - x_{s_2,c_2}) + \sum_{r \in N(R) \setminus N(s_1)} x_{r,c_1} + \sum_{r \in N(R) \setminus N(s_2)} x_{r,c_2}, \tag{4.11}$$

as a direct consequence of the definition of the solution subset in Equation (4.9). Note that we do not need to check whether vertices in $N(s_1)$ and $N(s_2)$ are colored with c_1 and c_2,

respectively, since this is already enforced by constraint (4.2). Therefore, we can omit these vertices from the summations over $N(R)$.

The variable z is used to break the symmetry on the first row of the submatrix $x(R, \{c_1, c_2\})$. For ease of notation, name the vertices of $R = \{v_1, \ldots, v_{|R|}\}$, with $v_1 < \cdots < v_{|R|}$. Since graph coloring is a partitioning problem, we know that at most one of x_{v_1,c_1} and x_{v_1,c_2} can be 1. Hence, we can break the symmetry on the first row by enforcing that $x_{v_1,c_2} = 0$ whenever the sub-symmetry is active. This yields the partial-sub-symmetry-breaking inequality

$$x_{v_1,c_2} \leq z. \tag{4.12}$$

In order fully break the symmetry, we need to extend $\mathbb{S}_{\text{GCP}}$ with additional tie-break sets

$$\tilde{Q}^{s_1,s_2,R}_{c_1,c_2}(r) = Q^{s_1,s_2,R}_{c_1,c_2} \cap \{x \mid x_{i,c_1} = x_{i,c_2} \quad \forall_{i \in v_1,\ldots,v_{r-1}}\}, \tag{4.13}$$

for every $r \in \{2, \ldots, |R|\}$. The corresponding variable $\tilde{z}_r$ can then be expressed as

$$\tilde{z}_r = z + \sum_{i=1}^{r-1} x_{v_i,c_1}. \tag{4.14}$$

The variable $\tilde{z}_r$ is then used in the sub-symmetry-breaking inequality

$$x_{v_r,c_2} \leq \tilde{z}_r. \tag{4.15}$$

Observe that the variables z and $\tilde{z}_r$ have a direct linear expression in the variables x. Hence, it is not necessary to add z and $\tilde{z}_r$ as explicit binary variables to the formulation. Instead, we can substitute every occurrence of z and $\tilde{z}_r$ in the sub-symmetry-breaking inequalities with the expressions (4.11) and (4.14).

The sub-symmetry inequalities are based on the column inequalities in (4.6) that are used for global symmetry-breaking. In fact, by choosing $R = V$ and by omitting s_1 and s_2, the original column inequalities can be derived from the sub-symmetry breaking inequalities, as described in [4].

4.3 Handling sub-symmetry using an activation handler

We now describe how we can apply our framework for handling sub-symmetries from Chapter 3.

Consider a solution subset $Q^{s_1,s_2,R}_{c_1,c_2} \in \mathbb{S}_{\text{GCP}}$, and suppose that during the solving process we are at a node a in the branch-and-bound tree. Let F^a_0 and F^a_1 denote the set of variables fixed to 0 and 1 at node a, respectively. Furthermore let $Q_a \subseteq \mathcal{X}_{\text{GCP}}$ be the solution subset corresponding to the subproblem at a. The sub-symmetry is *active* at a when $Q_a \subseteq Q^{s_1,s_2,R}_{c_1,c_2}$. Since $Q^{s_1,s_2,R}_{c_1,c_2}$ is fully determined by variable fixings: the sub-symmetry is active at a if and only if

$$x_{s_1,c_1}, x_{s_2,c_2} \in F^a_1 \text{ and } x_{i,c_1}, x_{i,c_2} \in F^a_0 \quad \forall_{i \in N(R)} \tag{4.16}$$

holds (see also the definition of $Q^{s_1,s_2,R}_{c_1,c_2}$ in Equation (4.9)).

When the sub-symmetry becomes active, it can be handled by enforcing the columns of the submatrix $x(R, \{c_1, c_2\})$ to be lexicographically ordered. With a slight abuse of notation, denote

the columns of this submatrix by x_{R,c_1} and x_{R,c_2}. Since the submatrix has only two columns, the symmetry-breaking constraint can be stated as

$$x_{R,c_1} \succeq x_{R,c_2}. \tag{4.17}$$

The constraint (4.17) is also known as an *orbisack* constraint, which can be separated in linear time using methods based on what is described by Loos [29], and Hojny and Pfetsch [17]. Separation and propagation routines are implemented in the SCIP solver, and are available as an *orbisack constraint handler* plugin [16].

The implementation of the activation handler in SCIP is then as follows. The variable fixings F_0^a and F_1^a are inspected at node a, sending a YES or NO response indicating whether the sub-symmetry is active. The activation handler is linked to the orbisack constraint that handles the symmetry, such that the orbisack constraint is only handled on a YES response of the activation handler. See Algorithm 1 the pseudocode of the implementation of the *variable fixings activation handler*.

Algorithm 1 Variable fixings activation handler procedure that inspects variable fixings at node a, and checks whether the given variables in sets $\mathcal{F}_0$ and $\mathcal{F}_1$ are all fixed to 0 and 1, respectively.

```
procedure VarFixingsIsActive(a, F_0, F_1)        ▷ F_0,F_1 are sets of variables.
    for all x_{i,j} ∈ F_0 do
        if x_{i,j} ∉ F_0^a then
            return NO

    for all x_{i,j} ∈ F_1 do
        if x_{i,j} ∉ F_1^a then
            return NO

    return YES
```

To handle the zero-neighbor-sub-symmetry at node a, the corresponding call to the activation handler is VARFIXINGSISACTIVE$(a, \mathcal{F}_0, \mathcal{F}_1)$, with sets of variables

$$\mathcal{F}_0 = \{x_{i,c_1} \mid i \in N(R) \setminus N(s_1)\} \cup \{x_{i,c_2} \mid i \in N(R) \setminus N(s_2)\}, \qquad \mathcal{F}_1 = \{x_{s_1,c_1}, x_{s_2,c_2}\}. \tag{4.18}$$

With precomputed sets $\mathcal{F}_0$ and $\mathcal{F}_1$, the activation handler runs in time linear in the size of these sets, as the checking whether $x \in F_0^a$ and $x \in F_1^a$ can be done in constant time in SCIP.

4.4 Implementation details

The IP formulation is constructed in Python 3.10 using the PySCIPOpt interface that exposes the SCIP API in Python. See also Appendix A. The activation handler is implemented in SCIP as a new plugin, and can be added to the model with the PySCIPOpt interface. The activation handler for a sub-symmetry defined by R, s_1 and s_2, the variable fixings activation handler is constructed by providing a lists of references to variables that must be fixed to zero or one, as described in Section 4.3. The activation handler is then linked to an *orbisack constraint* in SCIP that breaks the symmetry. Note that we instantiate separate activation handlers and orbisack constraints for every sub-symmetry.

The formulation is verified by testing the instance input readers and by checking the generated model for small test instances with expected output.

4.5 Experiments on benchmark instances

In order to directly compare our method to the existing inequality-based approach, we replicate the experimental setting from [4] as much as possible.

Experiments are performed on DIMACS graph coloring benchmark instances [42] with upper bound K on the chromatic number computed using the DSatur algorithm [7]. Instances of which results are reported in [4] are included in our instances set. The instances are partitioned into classes, based on the solving time of the best-known algorithm. Class NP-s contains instances solvable in less than a minute. The classes NP-m, NP-h, and NP-? contain instances solvable in less than an hour, a day, and in an unknown amount of time, respectively.

In order to compare symmetry-breaking techniques, we compare the following models for every problem instance:

F	Formulation (4.1)–(4.5) with global-symmetry-breaking column inequalities (4.6), with default SCIP parameters.
F-S0	Formulation (4.1)–(4.5) with global-symmetry-breaking column inequalities (4.6), with SCIP internal symmetry handling turned off.
F-Ineq	Formulation (4.1)–(4.5) with global-symmetry-breaking column inequalities (4.6) and sub-symmetry-breaking inequalities from Section 4.2.
F-Act	Formulation (4.1)–(4.5) with global-symmetry-breaking column inequalities (4.6) and sub-symmetry breaking using orbisack constraints with the activation handler from Section 4.3.

4.5.1 Choices of parameters

For the models F-Ineq and F-Act, we make further choices on what solution subsets from $\mathbb{S}_{\text{GCP}}$ to consider, as a trade-off between computational overhead of the added inequalities and the overall performance gain. The choices are in line with the recommendations from [4].

Pairs of colors For a triple (s_1, s_2, R), the set of solution subsets $\mathbb{S}_{\text{GCP}}$ contains solution subsets for every pair of colors $c_1 < c_2$. Instead of considering all pairs of colors, we only consider pairs of *consecutive* colors c_j and c_{j+1}, for $j \in \{1, \dots, K-1\}$, to reduce computational overhead. Except for instances where

- $|V| \geq 900$ and $K \leq 10$,
- $|E| < 300$, or
- $K < 100$ and $|V|/K < 10$,

we indeed consider all pairs of colors as this is acceptable from the size of the graph, and might possibly lead to more sub-symmetry breaking. The choice of color pairs applies to both F-Ineq and F-Act.

Number of variables in z For F-Ineq, the number of variables that are necessary to express z are limited to

- 30, if $|E|/|V| > 100$,
- 20, if $|E|/|V| > 10$ and $|V| < 200$,

- 20, if $|V| < 200$ and $|E| < 1000$ and $K > 5$, or
- 10, otherwise.

If the expression for z exceeds this limit, the corresponding sub-symmetry is not handled.

Limit on tie-break sets For F-Ineq, the number of tie-break sets added for every solution subset in $\mathbb{S}_{\text{GCP}}$ is limited to

- 1, if $|V| < 100$,
- 2, if $|N| \leq 900$, or
- 3, otherwise.

Note that this may results in only partially breaking the sub-symmetries in F-Ineq.

Number of variable fixings in activation handler The number of variable fixings necessary to check in the activation handler is limited to 50. If the required number of variable fixings exceeds this limit, the corresponding sub-symmetry is not handled.

Total number of sub-symmetries For F-Act, the total number of sub-symmetries is limited to 100 000. For F-Ineq, the total number of sub-symmetry inequalities that we add to the model is limited to 50 000 when $K < 100$, and no limit when $K \geq 100$. When the limit is reached, the further sub-symmetries or sub-symmetry inequalities are not added to the model and remain untreated.

Time limit For all models, instances are solved to optimality with a time limit of 7200 seconds (2 hours). The time limit excludes construction time of the model and the time spent finding the solution subsets that exhibit sub-symmetry. The time limit for model construction and finding sub-symmetries is 14 400 seconds (4 hours).

4.5.2 Results

All experiments are run with the development version of SCIP 7.0.3 with the SoPlex LP-solver [14], on a single core of an Intel Xeon Platinum 8260 CPU, running at 2.4 GHz, with 10.7 GB of RAM.

The results are reported in Table 4.1, 4.2, 4.3, and 4.4, and include the following data.

$\|V\|$	Number of vertices.
$\|E\|$	Number of edges.
K	Upper bound on the chromatic number, as computed by DSatur.
SSBC	Number of sub-symmetry-breaking constraints added in the model. For F-Ineq this is the number of added inequalities. For F-Act this is the number of orbitope constraints.
Init (sec)	The time spent to generate sub-symmetry breaking inequalities and initialize the SCIP model, in seconds.
Nodes	Number of nodes in the B&B tree.
Solving (sec)	Time spent solving the model, in seconds. When the time limit is reached when the model is not solved to optimality, the solving time is equal to 7200.

Table 4.1: GCP experiments on NP-s instances.

Instance	$\|V\|$	$\|E\|$	K	Model	SSBC	Init (sec)	Nodes	Solving (sec)
DSJC125.1	125	736	5	F	0	1.04	2	10.31
				F-S0	0	1.04	2	10.1
				F-Ineq	0	3.88	2	9.76
				F-Act	3832	3.88	2	10.5
1-Insertions_4	67	232	5	F	0	1.40	931 343	7200
				F-S0	0	0.72	1 020 469	5132.06
				F-Ineq	370	2.39	680 419	1030.16
				F-Act	5490	5.67	10 086	58.79
queen9_9	81	2112	10	F	0	4.97	73 050	7200
				F-S0	0	5.96	79 915	7200
				F-Ineq	0	46.02	410 013	7200
				F-Act	0	6.27	75 988	7200
r125.1c	125	7501	46	F	0	56.29	1	7200
				F-S0	0	56.67	1	7200
				F-Ineq	50 000	55.14	1	7200
				F-Act	0	56.18	1	7200
school1	385	19 095	14	F	0	60.49	1	7200
				F-S0	0	68.62	1	7200
				F-Ineq	2106	1377.1	1	7200
				F-Act	2275	273.13	1	7200

Of the 26 instances in total, only 7 instances were solved to optimality within the time limit. The majority of the instances prove too difficult for our model and cannot be used for a comparison of the sub-symmetry handling methods.

A surprising result is that our replication of the sub-symmetry-breaking inequalities model does not show the same performance improvements as reported in [4]. In the generation of the models, there might be a difference in our implementation and the implementation used for the original paper. The order of variables and constraints can have an impact on, e.g., branching decisions and other heuristics, leading to different results. In addition, we were not able to fully replicate their experimental setup of the original paper, as for some models a different number of sub-symmetry-breaking inequalities are generated. Moreover, in the original paper the experiments were performed using the CPLEX 12.8 commercial solver, which might differ significantly from the solving process in SCIP. In order to check whether the difference between CPLEX and SCIP is significant, we have also tested F, F-S0 and F-Ineq on CPLEX 12.8. Although there is a difference in results, also on CPLEX the F-Ineq model does not show a clear performance improvement over F and F-S0. The results for every instance can be found in Appendix B. This shows that the sub-symmetry-breaking inequalities approach is very much dependent on the choices for the parameters in Section 4.5.1, and that our efforts in replicating the exact setup is not sufficient for reaching the same performance gains.

For the instances that are solved to optimality, we see that in 6 out of the 7 times the activation handler method is the most efficient. Especially for the 1-Insertions_4, 2-Insertions_4, and 4-Insertions_3 instances, the solving time is significantly shorter. On other instances, the performance of the other models is rather similar. Overall, the results show that sub-symmetry breaking with the activation handler method can significantly improve the solving process.

Table 4.2: GCP experiments on NP-m instances.

Instance	$\|V\|$	$\|E\|$	K	Model	SSBC	Init (sec)	Nodes	Solving (sec)
5-FullIns_3	154	792	8	F	0	2.10	7611	144.78
				F-S0	0	2.10	7611	152.6
				F-Ineq	7350	8.67	8249	241.39
				F-Act	5040	6.49	9808	333.68
2-FullIns_4	212	1621	6	F	0	2.90	2671	276.56
				F-S0	0	2.90	2671	275.15
				F-Ineq	20	11.87	3597	264.85
				F-Act	5195	10.58	2391	245.67
wap05a	905	43 081	51	F	0	557.77	1	7200
				F-S0	0	557.48	1	7200
				F-Ineq	0	14 400		
				F-Act	0	1131.77	1	7200
4-Insertions_3	79	156	4	F	0	0.31	26 970	44.76
				F-S0	0	0.33	26 970	44.88
				F-Ineq	2250	1.71	22 542	51.17
				F-Act	2280	1.27	4594	11.04
school1_nsh	352	14 612	21	F	0	50.42	1	7200
				F-S0	0	50.34	1	7200
				F-Ineq	1400	1402.1	1	7200
				F-Act	1340	109.53	1	7200
ash608GPIA	1216	7844	8	F	0	92.52	1	7200
				F-S0	0	92.65	1	7200
				F-Ineq	3528	680.26	1	7200
				F-Act	67 732	919.80	1	7200
le450_15a	450	8168	19	F	0	45.16	1	7200
				F-S0	0	45.14	1	7200
				F-Ineq	0	1091.72	1	7200
				F-Act	1980	113.98	1	7200
myciel6	95	755	7	F	0	1.00	1 219 363	7200
				F-S0	0	1.04	1 180 294	7200
				F-Ineq	5130	10.39	650 925	7200
				F-Act	6978	4.64	956 060	7200

It is also clear that the success varies greatly with the instance. This can also be explained by the fact that selecting the solution subsets for sub-symmetry handling has been quite arbitrary. There are too many solution subsets to consider all sub-symmetries during solving. For some instances, we may have been unlucky in the selection of the subsets.

Table 4.3: GCP experiments on NP-h instances.

Instance	$\|V\|$	$\|E\|$	K	Model	SSBC	Init (sec)	Nodes	Solving (sec)
DSJC125.5	125	3891	24	F	0	12.06	1	7200
				F-S0	0	13.16	1	7200
				F-Ineq	0	1467.84	1	7200
				F-Act	0	13.73	1	7200
DSJC125.9	125	6961	57	F	0	45.43	1	7200
				F-S0	0	45.60	1	7200
				F-Ineq	50 000	86.65	1	7200
				F-Act	0	46.24	1	7200
DSJC250.9	250	27 897	100	F	0	317.15	1	7200
				F-S0	0	339.92	1	7200
				F-Ineq	0	2292.82	1	7200
				F-Act	0	327.00	1	7200
DSJR500.5	500	58 862	134	F	0	1948.55		7200
				F-S0	0	1063.48	1	7200
				F-Ineq	0	14 400		
				F-Act	0	1641.44	1	7200
flat300_28_0	300	21 695	48	F	0	291.28	1	7200
				F-S0	0	237.51	1	7200
				F-Ineq	0	14 400		
				F-Act	0	185.85	1	7200
r250.5	250	14 849	72	F	0	148.09	1	7200
				F-S0	0	143.80	1	7200
				F-Ineq	50 000	5149.98	1	7200
				F-Act	0	284.68	1	7200

Table 4.4: GCP experiments on NP-? instances.

Instance	$\|V\|$	$\|E\|$	K	Model	SSBC	Init (sec)	Nodes	Solving (sec)
r125.5	125	3838	36	F	0	18.09	1	7200
				F-S0	0	18.42	1	7200
				F-Ineq	50 000	182.06	1	7200
				F-Act	0	19.94	1	7200
3-FullIns_4	405	3524	7	F	0	10.66	2590	7200
				F-S0	0	10.84	2601	7200
				F-Ineq	0	58.29	1953	7200
				F-Act	9984	53.25	4636	7200
1-FullIns_5	282	3247	6	F	0	5.73	18 666	7200
				F-S0	0	5.78	34 281	7200
				F-Ineq	0	38.51	43 478	2947.65
				F-Act	200	13.82	39 189	2840.47
2-Insertions_4	149	541	5	F	0	1.41	161 276	7200
				F-S0	0	1.05	850 298	7200
				F-Ineq	352	4.48	2 011 601	7200
				F-Act	5040	8.14	760	127.13
myciel7	191	2360	8	F	0	4.06	107 866	7200
				F-S0	0	4.05	108 199	7200
				F-Ineq	18 228	74.78	23 528	7200
				F-Act	11 564	20.72	74 910	7200
flat300_20_0	300	21 375	38	F	0	162.15	1	7200
				F-S0	0	227.80	1	7200
				F-Ineq	0	14 400		
				F-Act	0	155.49	1	7200
flat300_26_0	300	21 633	39	F	0	112.59	1	7200
				F-S0	0	113.03	1	7200
				F-Ineq	0	14 400		
				F-Act	0	154.27	1	7200

Chapter 5

Application to unit commitment problem

We now apply our framework to the unit commitment problem, in which production units have to satisfy total production demand over a time period, where the goal is to minimize total cost that arise from active production units. This problem and its variants are for example used to model energy production by power plants. We will consider the variant where every production unit has a minimum uptime and downtime, called the *Min-up/min-down Unit Commitment Problem* (MUCP).

Suppose we are given a time horizon $\mathcal{T} = \{1, \dots, T\}$ and non-negative demands D_t that need to be satisfied at every time $t \in \mathcal{T}$. Furthermore assume we are given a set $\mathcal{U}$ of production units, where each unit $j \in \mathcal{U}$ has a minimum production capacity $P^j_{\min}$ and a maximum production capacity $P^j_{\max}$. Let $|\mathcal{U}| = n$. Production units can be either *up* or *down* at every time $t \in \mathcal{T}$. When a unit j is up, its production is between $P^j_{\min}$ and $P^j_{\max}$ and it must remain up for at least L^j time steps. Similarly, when a unit j is down, it must remain down for at least ℓ^j time steps. Without loss of generality, we assume $L^j \leq T$ and $\ell^j \leq T$. We furthermore have for every unit j a start-up cost c^j_0, a fixed cost c^j_f for every time step the unit is up, and a production cost c^j_p proportional to its production. The goal is to find a *production schedule* satisfying the production demand at every time step and the min-up and min-down constraints, while minimizing the total cost.

It has been shown that the MUCP is strongly NP-hard for general T and n [2]. MUCP remains NP-hard even in special cases of *unit-cost*, where $c^j_0 = c^j_p = 0$ and $c^j_f = 1$ for all $j \in \mathcal{U}$, and *unit-power*, where $P^j_{\min} = P^j_{\max} = P$ for all $j \in \mathcal{U}$ [2]. Furthermore, the problem can be solved in polynomial time for fixed n, showing that the number of production units has a large impact on the complexity of the problem [2].

We can express the MUCP as an integer linear program as follows [5, 37, 39]. For every production unit $j \in \mathcal{U}$ and time step $t \in \mathcal{T}$, we consider the following variables. Let the variable $x_{t,j} \in \{0, 1\}$ indicate whether unit j is up at time t, $u_{t,j} \in \{0, 1\}$ whether unit j starts up at time t, and $p_{t,j} \in \mathbb{R}_+$ the production of unit j at time t. We will also refer to these variables as the matrices $x = (x_{t,j})_{t\in\mathcal{T}, j\in\mathcal{U}}$, $u = (u_{t,j})_{t\in\mathcal{T}, j\in\mathcal{U}}$ and $p = (p_{t,j})_{t\in\mathcal{T}, j\in\mathcal{U}}$. We then

get the following formulation.

$$\text{minimize} \quad \sum_{j\in\mathcal{U}}\sum_{t\in\mathcal{T}}\left(c_f^j x_{t,j} + c_p^j p_{t,j} + c_0^j u_{t,j}\right) \tag{5.1}$$

$$\text{subject to} \quad \sum_{t'=t-L^j+1}^{t} u_{t',j} \le x_{t,j} \qquad \forall_{j\in\mathcal{U}},\ \forall_{t\in\{L^j,\dots,T\}}, \tag{5.2}$$

$$\sum_{t'=t-\ell^j+1}^{t} u_{t',j} \le 1 - x_{t-\ell^j,j} \qquad \forall_{j\in\mathcal{U}},\ \forall_{t\in\{\ell^j,\dots,T\}}, \tag{5.3}$$

$$\sum_{j\in\mathcal{U}} p_{t,j} \ge D_t \qquad \forall_{t\in\mathcal{T}}, \tag{5.4}$$

$$p_{t,j} \ge P_{\min}^j x_{t,j} \qquad \forall_{j\in\mathcal{U}},\ \forall_{t\in\mathcal{T}}, \tag{5.5}$$

$$p_{t,j} \le P_{\max}^j x_{t,j} \qquad \forall_{j\in\mathcal{U}},\ \forall_{t\in\mathcal{T}}, \tag{5.6}$$

$$u_{t,j} \le x_{t,j} - x_{t-1,j} \qquad \forall_{j\in\mathcal{U}},\ \forall_{t\in\{2,\dots,T\}}, \tag{5.7}$$

$$x_{t,j} \in \{0,1\} \qquad \forall_{j\in\mathcal{U}},\ \forall_{t\in\mathcal{T}}, \tag{5.8}$$

$$u_{t,j} \in \{0,1\} \qquad \forall_{j\in\mathcal{U}},\ \forall_{t\in\mathcal{T}}. \tag{5.9}$$

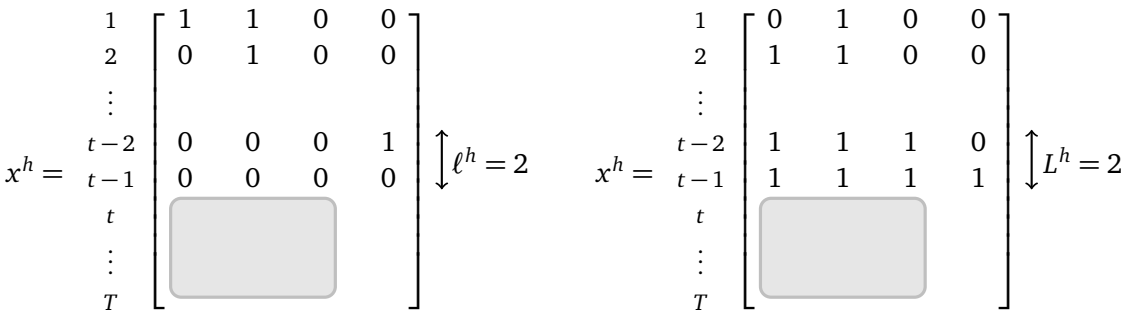
The constraints (5.2) and (5.3) ensure the minimum uptime and downtime, respectively. Constraints (5.5) and (5.6) enforce that the production is between the capacity bounds when the unit is up, while also ensuring that the production is zero when the unit is down. The constraint (5.4) ensures that the demand is satisfied at every time step, while the constraint in (5.7) defines when a unit is starting up. We define $\mathcal{X}_{\text{MUCP}}$ as the set of all feasible matrix solutions x.

5.1 Symmetry in the unit commitment problem

Symmetries are present globally in the MUCP when production units have identical properties, i.e., units where the properties $(P_{\min}, P_{\max}, L, \ell, c_0, c_f, c_p)$ are equal. To make this more explicit, we partition the production units into H *types*, where a type $h \in \{1,\dots,H\}$ consists of n_h identical units that we denote by $\mathcal{U}_h = \{j_1^h,\dots,j_{n_h}^h\}$. For a type h, we slightly abuse notation to denote its properties as $(P_{\min}^h, P_{\max}^h, L^h, \ell^h, c_0^h, c_f^h, c_p^h)$. We can then also partition the matrix variable x into H matrices $x^h = (x_{t,j})_{t\in\mathcal{T}, j\in\mathcal{U}_h}$ for every type of production unit, and similarly for matrices u and p. The production units within each type are identical, and we can hence permute their production schedules. This corresponds to permuting the columns of x^h, u^h and p^h, provided that the same permutation is applied to all three matrices.

In order to break the symmetry arising from permutations of units of the same type, note that the variables u^h are fully determined by x^h, hence breaking the symmetry on x^h will also break the symmetry on u^h. One possible way of breaking the symmetry is to restrict x^h to the full orbitope for binary matrices of size $T \times n_h$, i.e., by imposing that the columns of x^h are lexicographically non-increasing.

The MUCP also exhibits sub-symmetries, as described by Bendotti, Fouilhoux and Rottner [4]. They call a production unit $j \in \mathcal{U}$ *ready to start up* at some time $t \in \mathcal{T}$ if the unit has been down continuously for at least the minimum downtime ℓ^j. In other words, when $x_{t',j} = 0$ for all $t' = t-\ell^j,\dots,t-1$ and $t \ge \ell^j + 1$. Now, suppose there are at least two units $j_1,\dots,j_k \in \mathcal{U}^h$ of type h that are all ready to start up at some time t. Then, their production schedules can be permuted from time t onwards, regardless of their schedule up to time t. This thus



(a) (b)

Figure 5.1: Example of sub-symmetry in MUCP. (a) At time t, three identical units of type h have been down for at least $\ell^h = 2$ time steps, hence their schedules starting from t can be permuted (highlighted in gray). (b) At time t, three identical units of type h have been up for at least $L^h = 2$ time steps, hence their schedules starting from t can be permuted. Note that in both cases, the schedules need not be equal at any time before $t-2$.

defines a sub-symmetry where the columns of the submatrix $x(\{t,\dots,T\},\{j_1,\dots,j_k\})$ can be permuted. Analogously, one can identify sub-symmetries for two units ready to shut down at some time $t \in \mathcal{T}$. These sub-symmetries are referred to as the *start-up* and *shut-down sub-symmetries*, respectively. See also Figure 5.1 for an example.

5.2 Sub-symmetry-breaking inequalities

We now describe the method of Bendotti, Fouilhoux and Rottner of handling the start-up and shut-down sub-symmetries using inequalities [4]. For the details of the derivation, we refer to the original paper.

Let $h \in [H]$ be a type of identical units $\mathcal{U}^h = \{j_1^h,\dots,j_{n_h}^h\}$. As described in Section 5.1, we can define sub-symmetries for *at least* two units that are ready to start up or shut down at some time t. As a limitation of using inequalities, permutations between more than two units cannot be handled with a single inequality. Instead, solutions are defined for *consecutive* pairs of units j_k^h, j_{k+1}^h. Thus, consider for every $k = 1,\dots,n_h-1$ the solution subsets

$$\begin{aligned}\check{Q}_{k,h}^t = \{x \in \mathcal{X}_{\text{MUCP}} \mid x_{t',j} = 0 \text{ for all } t' = t-\ell^h,\dots,t-1,\ j = j_k^h, j_{k+1}^h\} \\ \text{for all } t \geq \ell^h + 1,\end{aligned} \tag{5.10}$$

$$\begin{aligned}\hat{Q}_{k,h}^t = \{x \in \mathcal{X}_{\text{MUCP}} \mid x_{t',j} = 1 \text{ for all } t' = t-L^h,\dots,t-1,\ j = j_k^h, j_{k+1}^h\} \\ \text{for all } t \geq L^h + 1,\end{aligned} \tag{5.11}$$

where sub-symmetries of $\check{Q}_{k,h}^t$ and $\hat{Q}_{k,h}^t$ are the start-up and shut-down sub-symmetries, respectively. These sub-symmetries correspond to permutations of the two columns of the submatrix $x(\{t,\dots,T\},\{j_k^h, j_{k+1}^h\})$. It can be shown that no additional tie-break sets are necessary, and

$$\begin{aligned}\mathbb{S}_{\text{MUCP}} = \{\hat{Q}_{k,h}^t \mid t \in \mathcal{T},\ h \in [H],\ k \in \{1,\dots,n_h-1\},\ t \geq L^h+1\} \\ \cup \{\check{Q}_{k,h}^t \mid t \in \mathcal{T},\ h \in [H],\ k \in \{1,\dots,n_h-1\},\ t \geq \ell^h+1\}\end{aligned} \tag{5.12}$$

satisfies the condition to be full-sub-symmetry breaking.

The variables $\check{z}^t_{k,h}$ and $\hat{z}^t_{k,h}$, indicating whether $x \in \check{Q}^t_{k,h}$ and $x \in \hat{Q}^t_{k,h}$, respectively, can then be expressed directly as

$$\check{z}^t_{k,h} = \sum_{t'=t-\ell^h}^{t-1} x_{t',j} + \sum_{t'=t-\ell^h}^{t-1} x_{t',j'} \qquad \text{for } t \geq \ell^h + 1, \tag{5.13}$$

$$\hat{z}^t_{k,h} = \sum_{t'=t-L^h}^{t-1} \left[1 - x_{t',j}\right] + \sum_{t'=t-L^h}^{t-1} \left[1 - x_{t',j'}\right] \qquad \text{for } t \geq L^h + 1, \tag{5.14}$$

where $j = j^h_k$ and $j' = j^h_{k+1}$, for notational convenience. This leads to sub-symmetry-breaking inequalities

$$x_{t,j'} \leq \check{z}^t_{k,h} + x_{t,j} \qquad \text{if } t \geq \ell^h + 1, \tag{5.15}$$

$$x_{t,j'} \leq \hat{z}^t_{k,h} + x_{t,j} \qquad \text{if } t \geq L^h + 1. \tag{5.16}$$

Note that since the z-variables have a linear expression in x, we need not to add the z-variables to the model explicitly; we can simply replace every occurrence by the linear expression in the sub-symmetry-breaking inequalities.

The inequalities (5.15) and (5.16) can be strengthened to the following inequalities.

$$u_{t,j'} \leq \left\lceil x_{t-\ell^h,j} + \sum_{t'=t-\ell^h+1}^{t-1} u_{t',j} \right\rceil + x_{t,j} \qquad \text{if } t \geq \ell^h + 1, \tag{5.17}$$

$$w_{t,j} \leq \left\lceil 1 - x_{t-L^h,j'} + \sum_{t'=t-L^h+1}^{t-1} w_{t',j'} \right\rceil + 1 - x_{t,j'} \qquad \text{if } t \geq L^h + 1, \tag{5.18}$$

where $w_{t,j} = x_{t-1,j} - x_{t,j} + u_{t,j}$ denotes whether unit j shuts down at time t. We refer to [4] for the derivation of the strengthened inequalities.

5.3 Handling sub-symmetry using an activation handler

We now describe how we can apply the activation handler framework to handle start-up and shut-down sub-symmetries. For the MUCP, we adopt a more general approach in detecting all active sub-symmetries at once. This is different from the approach for the GCP in Section 4.3, where we registered separate activation handlers for every individual solution subset. In this case, our activation handler does not only give a YES or NO answer, but instead returns all the submatrices in x containing active sub-symmetries.

Let a be a node of the branch-and-bound tree, and let F^a_0 and F^a_1 denote the sets of variables of that are fixed to 0 or 1 at node a, respectively. Let $Q_a \subseteq \mathcal{X}_{\text{MUCP}}$ denote the feasible solution set of the subproblem at node a. We are looking for production units that according to the variable fixings at node a are ready to start up or ready to shut down. For ease of presentation, we assume that all production units $\mathcal{U}$ are identical. In the more general case where we have multiple types of production units, we can simply apply our method to the unit types separately.

We start by defining for every $t \in \{\ell + 1, \ldots, T\}$,

$$\check{S}^a_t = \{j \in \mathcal{U} \mid x_{t',j} \in F^a_0 \text{ for all } t' \in \{t - \ell, \ldots, t - 1\}\}. \tag{5.19}$$

That is, $\check{S}_t^a$ are the production units that are *fixed* to be ready to start up at node a. In other words, for a unit $j \in \check{S}_t^a$, we have that this unit is ready to start up at time t *in every solution of* Q_a. For every subset $\check{S}_t^a$ for which $|S_t^a| \geq 2$, the corresponding start-up sub-symmetry—where the schedules of units S_t^a can be permuted from time t can be permuted—becomes active. Hence, the symmetry correspond to column permutations of the submatrix $x(\{t,\dots,T\},\check{S}_t^a)$. In order to break the symmetry, we activate an orbitope constraint at node a that restricts the columns of $x(\{t,\dots,T\},\check{S}_t^a)$ to be lexicographically non-increasing. In SCIP, this is done with the *orbitope constraint handler* implementing routines for propagation, described by Bendotti, Fouilhoux and Rottner [3], and separation, described by Hojny and Pfetsch [17].

We can do the same for the shut-down sub-symmetries. In that case, define production-unit sets for every $t \in \{L+1,\dots,T\}$,

$$\hat{S}_t^a = \{j \in \mathcal{U} \mid x_{t',j} \in F_1^a \text{ for all } t' \in \{t-L,\dots,t-1\}\}, \tag{5.20}$$

as the production units that are fixed at node a to be ready to shut down at time t. Analogously to the above, every subset $\hat{S}_t^a$ with $|\hat{S}_t^a| \geq 2$ corresponds to a shut-down symmetry, which we break by activating an orbitope constraint for the submatrix $x(\{t,\dots,T\},\hat{S}_t^a)$ at node a.

It remains to describe how to find all active sub-symmetries at node a efficiently. As illustrated in Figure 5.1, sub-symmetries are defined by *towers* of zeros or ones in the matrix x. By iterating over the time horizon, it is possible to find these towers in $\mathcal{O}(nT)$ time. We use a dynamic programming approach, where the main idea is to fill two tables $\mathrm{up}(t,j)$ and $\mathrm{down}(t,j)$ that indicate the current uptime and downtime for a unit j at time t, according to the variable fixings at a. The table entries satisfy the following recursive expressions

$$\mathrm{up}(t,j) = \begin{cases} \mathrm{up}(t-1,j)+1 & \text{if } t>1 \text{ and } x_{t-1,j} \in F_1^a, \\ 0 & \text{otherwise,} \end{cases} \tag{5.21}$$

$$\mathrm{down}(t,j) = \begin{cases} \mathrm{down}(t-1,j)+1 & \text{if } t>1 \text{ and } x_{t-1,j} \in F_0^a, \\ 0 & \text{otherwise.} \end{cases} \tag{5.22}$$

We can fill this table by increasingly iterating over the time horizon and in any order over the production units. When we have completely filled a table row for some time t, we can check if there are multiple units that have an uptime of at least L or a downtime of at least ℓ. If that is the case, we report the that the sub-symmetry in the submatrix with rows and column indices $(\{t,\dots,T\},\check{S}_t^a)$ or $(\{t,\dots,T\},\hat{S}_t^a)$ has become active.

The activated submatrices are passed to the orbitope constraint handler in SCIP, which we have slightly modified to support propagation of the *suborbitopes* defined by the submatrices. Separation for suborbitopes is not implemented. For completeness, pseudocode for this *suborbitope activation handler* algorithm is given in Algorithm 2.

With this approach, we only need to add a single activation handler to the model in SCIP, linked to a single orbitope constraint. The orbitope constraint is able to handle propagation of the suborbitope constraints for the submatrices reported by the activation handler. This allows for smaller overhead for sub-symmetry handling than the approach for the GCP, where multiple activation handlers were added for separate orbisack constraints.

Algorithm 2 Activation handler procedure that determines whether MUCP start-up or shut-down sub-symmetries occur at node a. A set of submatrix indices I is returned indicating that the production schedules in these sub-matrices can be permuted.

procedure SUBORBITOPEISACTIVE(a, x, $\mathcal{T}$, $\mathcal{U}$, L, ℓ)
  $I \leftarrow \{(\mathcal{T},\mathcal{U})\}$ ▷ Global symmetry on the whole matrix x is always active
  $\text{up}_j \leftarrow 0 \quad \forall_{j\in\mathcal{U}}$
  $\text{down}_j \leftarrow 0 \quad \forall_{j\in\mathcal{U}}$

  for all $t \in \mathcal{T}$ **do**
    $\check{S} = \{j \in \mathcal{U} \mid \text{down}_j \geq \ell\}$
    if $|\check{S}| \geq 2$ **then** ▷ Multiple units are down for at least the minimum downtime at t
      $I \leftarrow I \cup \{(\{t,\ldots,T\},\check{S})\}$

    $\hat{S} = \{j \in \mathcal{U} \mid \text{up}_j \geq L\}$
    if $|\hat{S}| \geq 2$ **then** ▷ Multiple units are up for at least the minimum uptime at t
      $I \leftarrow I \cup \{(\{t,\ldots,T\},\hat{S})\}$

    for all $j \in \mathcal{U}$ **do**
      if $x_{t,j} \in F_0^a$ **then** ▷ Update the current downtime for unit j
        $\text{down}_j \leftarrow \text{down}_j + 1$
      else
        $\text{down}_j \leftarrow 0$
      if $x_{t,j} \in F_1^a$ **then** ▷ Update the current uptime for unit j
        $\text{up}_j \leftarrow \text{up}_j + 1$
      else
        $\text{up}_j \leftarrow 0$
  return I

5.4 Implementation details

The IP formulation is constructed in Python 3.10 using the PySCIPOpt interface that exposes the SCIP API in Python. See also Appendix A. The activation handler is implemented in SCIP as a new plugin, and can be added to the model with the PySCIPOpt interface. For every type $h \in [H]$ of production units for which $|\mathcal{U}^h| \geq 2$, we add a single activation handler to the model. Upon creation, the activation handler is supplied with the matrix x^h of variables, the minimum uptime L^h, and the minimum downtime ℓ^h. The activation handler not only produces a YES or NO result to indicate whether a sub-symmetry is active, but also provided a (linked) list of submatrices of x^h where the sub-symmetry is active.

The activation handler is linked to an *orbitope constraint* in SCIP that breaks the symmetry. The orbitope constraint is modified such that it can handle suborbitope constraints for the submatrices returned by the activation handler.

The formulation is verified by testing the instance input readers and by checking the generated model for small test instances with expected output.

5.5 Experiments on generated instances

We compare the sub-symmetry handling methods using experiments on generated instances of the MUCP. We are thankful to the authors of [4] for sharing their instances, allowing for a direct comparison of both methods using the SCIP solver.

The following models are tested for every problem instance:

F	Formulation (5.1)–(5.9) with default SCIP parameters.
F-S0	Formulation (5.1)–(5.9) with SCIP internal symmetry handling turned off.
F-Ineq	Formulation (5.1)–(5.9) with sub-symmetry-breaking inequalities (5.17)–(5.18).
F-Act	Formulation (5.1)–(5.9) sub-symmetry breaking using the (sub)orbitope constraint with the activation handler from Section 5.3.

5.5.1 Choices of parameters

Instances were generated for $n \in \{20, 30, 60\}$ and $T \in \{48, 96\}$. Patterns for the production demand and production units are derived from the dataset in [8]. To make sure the instances exhibit symmetries, parameters for units are randomly generated and duplicated d times, with d randomly selected in the closed interval $[1, n/f]$. The parameter $f \in \{2, 3, 4\}$ is the *symmetry factor*, where a lower symmetry factor indicates larger groups of identical production units in expectation. Symmetry factor $f = 4$ is only considered for instances with $n = 60$. For every triple (n, T, f), 20 instances are generated. The results in Section 5.5.2 show the average over these 20 instances.

For all models, instances are solved to optimality with a time limit of 3600 seconds (1 hour). The time limit excludes construction time of the model and the time spent finding the solution subsets that exhibit sub-symmetry. Model construction time is negligible compared to the model solving time, however.

Table 5.1: MUCP experiments on instances with 20 production units.

n	T	f	Model	Opt	Avg opt nodes	Avg opt time (sec)
20	48	2	F	20	2506	12.99
			F-S0	20	5423	32.72
			F-Ineq	20	1606	38.98
			F-Act	20	7	6.38
		3	F	20	133	6.05
			F-S0	20	260	5.89
			F-Ineq	20	3	9.19
			F-Act	20	43	5.79
20	96	2	F	19	28 792	177.61
			F-S0	15	11 509	57.42
			F-Ineq	20	808	107.79
			F-Act	20	1513	51.92
		3	F	18	1066	23.64
			F-S0	17	15 478	64.23
			F-Ineq	19	1535	52.59
			F-Act	19	1141	37.65

5.5.2 Results

All experiments are run with the development version of SCIP 7.0.3 with the SoPlex LP-solver [14], on a single core of an Intel Xeon Platinum 8260 CPU, running at 2.4 GHz, with 10.7 GB of RAM.

The results are reported in Table 4.1, 5.1, 5.2, and 5.3, and include the following data.

n	Number of production units.
T	Size of time horizon.
f	Symmetry factor.
Opt	Number of generated instances solved to optimality.
Avg opt nodes	Average number of nodes in the B&B tree of the instances solved to optimality.
Avg opt time (sec)	Average spent solving the model, in seconds, of the instances solved to optimality.

Small instances of the MUCP with 20 production units can be solved quite efficiently by all models. For both F-Ineq and F-Act however, the average number of nodes in the branch-and-bound tree is generally smaller for the instances solved to optimality. In addition, both sub-symmetry-breaking methods solve more instances to optimality compared to the methods F and F-S0. F-Act is furthermore faster in solving time of almost all the small instances. The same trend is visible for the instances with 30 units, where the activation handler method now also solves more instances to optimality than the sub-symmetry-breaking method with inequalities. This is especially visible for the instances with $(n, T, f) = (3, 96, 2)$, where not only significantly more instances are solved to optimality, but the activation handler method also shows a large reduction in nodes of the B&B-tree and solving time.

Table 5.2: MUCP experiments on instances with 30 production units.

n	T	f	Model	Opt	Avg opt nodes	Avg opt time (sec)
30	48	2	F	20	2342	31.24
			F-S0	19	51 327	194.91
			F-Ineq	20	416	60.75
			F-Act	20	152	12.64
		3	F	19	40 081	142.06
			F-S0	13	6628	30.92
			F-Ineq	20	484	57.94
			F-Act	20	1795	36.05
30	96	2	F	17	2185	88.52
			F-S0	13	16 246	236.22
			F-Ineq	16	2979	684.54
			F-Act	19	528	87.06
		3	F	13	13 291	175.51
			F-S0	10	9445	128.38
			F-Ineq	19	2009	553.61
			F-Act	20	2397	199.92

For the larger instances with 60 units, the activation handler method clearly shows that it is more efficient than the other methods. Overall, many more instances can be solved to optimality, except for one instance with $(n, T, f) = (60, 48, 3)$, for which only F-Ineq is able to solve it within the time limit. For high-symmetry instances with $f = 2$, the activation handler method solves significantly more instances to optimality, compared to the inequalities method. When increasing the symmetry factor, this difference between F-Act and F-Ineq becomes smaller. The activation handler furthermore shows a significant improvement on the solving time, compared to the inequalities method. A possible explanation for this is that many sub-symmetry-breaking inequalities may be responsible for more overhead, having a negative impact on the solving time.

Table 5.3: MUCP experiments on instances with 60 production units.

n	T	f	Model	Opt	Avg opt nodes	Avg opt time (sec)
60	48	2	F	16	2416	189.69
			F-S0	12	134 170	473.38
			F-Ineq	13	7025	714.69
			F-Act	20	2419	226.05
		3	F	19	9015	171.83
			F-S0	11	64 166	261.57
			F-Ineq	19	3312	251.97
			F-Act	18	6104	107.67
		4	F	17	3087	75.28
			F-S0	9	24 237	140.79
			F-Ineq	19	1360	329.80
			F-Act	20	587	67.15
60	96	2	F	10	13 210	495.17
			F-S0	7	165 289	451.40
			F-Ineq	7	4433	1586.34
			F-Act	16	1524	393.22
		3	F	4	9411	187.87
			F-S0	4	20 450	173.11
			F-Ineq	9	6308	1685.11
			F-Act	16	2685	377.77
		4	F	4	45 752	461.75
			F-S0	2	1583	125.85
			F-Ineq	19	5706	1264.64
			F-Act	18	2536	428.07

Chapter 6

Application to geometric symmetries in two-dimensional packing problems

Consider a packing problem, where the goal is to pack objects in one or more containers where the objects are not allowed to overlap. Packing problems also exhibit permutation symmetries, for instance equivalent objects or containers that are interchangeable; these symmetries can for example be handled with orbisack or orbitope constraints. However, geometric symmetries arising from the shape of the container cannot be handled by these methods. In this section, we introduce a new symmetry handling constraint for such geometric symmetries and extend this to geometric sub-symmetries as well.

We describe our methods for two-dimensional orthogonal packing problems, where we are given rectangular containers and rectangle objects that need to be packed inside the containers. Several variants of rectangle packing exist, which differ in the considered objective function, whether all objects need to be packed, or the number and shapes of available containers. The first typology for two-dimensional packing and cutting problems was given by Dyckhoff [13] in 1990, which was further extended by Lodi, Martello and Vigo [27]. The surveys of Lodi, Marthello and Monaci [26] and Lodi, Martello and Vigo [28] give an overview of various approximation, heuristic, or exact approaches for several variants of two-dimensional packing. Iori, de Lima, Martello, Miyazawa and Monaci [19] provide a more recent review of exact algorithms for two-dimensional packing problems.

In the remainder of this chapter, we consider the *two-dimensional knapsack problem* (2D-KP), where each of the given rectangles to be packed has a profit. The goal is to find a packing of rectangles in a given container that maximizes the profit. We choose the knapsack variant as in this problem we are not only looking for a feasible packing, but indeed an optimal one in terms of profit. We anticipate that symmetry handling has a greater effect on these type of problems. In problems where the goal is to find, e.g., only a feasible packing of all rectangles, the solving process finishes much sooner and cutting off symmetric solutions might have less of an impact.

An instance of the two-dimensional knapsack problem specifies a rectangular container in which rectangles need to be packed. The container has integer width W and height H. Furthermore, we are given a set of rectangles $\mathcal{R}$ with integer widths w_i, heights h_i, and non-negative profits $p_i \in \mathbb{R}_+$ for every $i \in \mathcal{R}$. Rectangles may be selected to be packed in the container, where they must be placed at integer coordinates, may not overlap with other rectangles, and may not cross the container boundaries. Furthermore, rectangles are placed *axis-aligned* in

the container and rotations are not allowed. An optimal solution to the problem maximizes the sum of profits of the rectangles packed in the container. Several integer programming formulations for this problem exist. We use an *absolute placement* approach, where we model the location of a rectangle in the container with (integer) variables for the x and y-coordinate of the bottom-left *reference point* of the rectangle.

The formulation is based on the formulations by Onodera, Taniguchi and Tamaru [36], and Chen, Lee and Shen [9], adapted for the knapsack variant of the problem. Let $s_i \in \{0,1\}$ denote whether rectangle $i \in \mathcal{R}$ is selected to be packed. The variables $x_i, y_i \in \mathbb{Z}$ denote the location of the bottom-left corner of rectangle $i \in \mathcal{R}$ in the container, when it is selected to be placed. Furthermore, for distinct rectangles $i, j \in \mathcal{R}$, let $\ell_{ij} \in \{0,1\}$ denote whether rectangle i is placed *left* of rectangle j in the container. Similarly, let b_{ij} denote whether rectangle i is placed *below* of rectangle j in the container. We then get the following formulation.

$$\text{maximize} \quad \sum_{i \in \mathcal{R}} p_i s_i \tag{6.1}$$

$$\text{subject to} \quad 0 \le x_i \le (W - w_i) s_i \qquad \forall_{i \in \mathcal{R}} \tag{6.2}$$

$$0 \le y_i \le (H - h_i) s_i \qquad \forall_{i \in \mathcal{R}} \tag{6.3}$$

$$\ell_{ij} = 1 \implies x_i + w_i \le x_j \qquad \forall_{i,j \in \mathcal{R},\ i \ne j} \tag{6.4}$$

$$b_{ij} = 1 \implies y_i + h_i \le y_j \qquad \forall_{i,j \in \mathcal{R},\ i \ne j} \tag{6.5}$$

$$\ell_{ij} + \ell_{ji} + b_{ij} + b_{ji} + (1 - s_i) + (1 - s_j) \ge 1 \qquad \forall_{\{i,j\} \in \binom{\mathcal{R}}{2}} \tag{6.6}$$

$$\ell_{ij} + \ell_{ji} + b_{ij} + b_{ji} \le s_i \qquad \forall_{\{i,j\} \in \binom{\mathcal{R}}{2}} \tag{6.7}$$

$$\ell_{ij} + \ell_{ji} + b_{ij} + b_{ji} \le s_j \qquad \forall_{\{i,j\} \in \binom{\mathcal{R}}{2}} \tag{6.8}$$

$$x_i,\ y_i \in \mathbb{Z} \qquad \forall_{i \in \mathcal{R}} \tag{6.9}$$

$$s_i \in \{0,1\} \qquad \forall_{i \in \mathcal{R}} \tag{6.10}$$

$$\ell_{ij},\ b_{ij} \in \{0,1\} \qquad \forall_{i,j \in \mathcal{R},\ i \ne j} \tag{6.11}$$

The constraints (6.2)–(6.3) ensure that the rectangle is placed within the bounds of the container. Moreover, when a rectangle i is not selected, the constraints ensure that x_i and y_i are set to zero, in order to prevent unnecessary branching on these variables for unpacked rectangles. Constraints (6.4)–(6.6) ensure that two rectangles i and j do not overlap when they are both selected. In particular, Constraint (6.6) ensures that if both rectangle i and j are selected to be packed, then rectangle i is placed either to the left, right, top, or bottom of rectangle j. The constraints (6.4) and (6.5) ensure that this is in fact enforced. We formulate these constraints as *indicator constraints*. When a solver does not support indicator constraints, we can alternatively formulate them completely linear as *big-M* constraints:

$$x_i + w_i \le x_j + M(1 - \ell_{ij}), \tag{6.12}$$

$$y_i + h_i \le y_j + M'(1 - b_{ij}), \tag{6.13}$$

where M and M' are sufficiently large constants, for example, one can define $M := W + \max_i w_i$ and $M' := H + \max_i h_i$.

For any pair of rectangles i and j, constraints (6.7) and (6.8) ensure that the overlap constraints are turned off when i or j is not selected. We define $\mathcal{X}_{\text{2D-KP}}$ as the set of all feasible solutions (s, x, y).

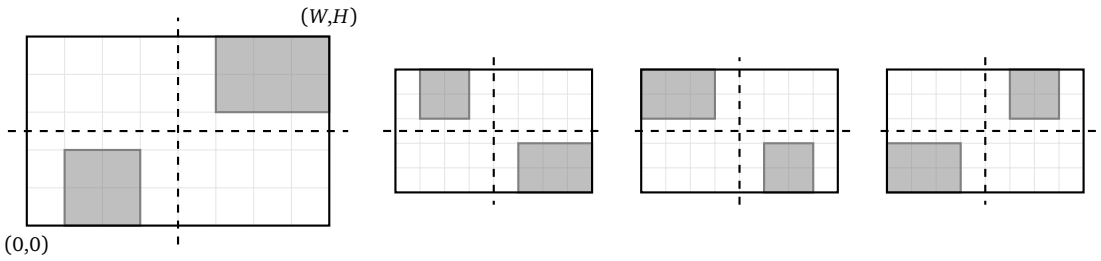


Figure 6.1: Example two-dimensional packing (partial) solution, with two rectangles placed inside the container. The two symmetry axes of the container are indicated with dashed lines. In its orbit, this solution has three equivalent solutions, displayed on the right. These solutions can be obtained by reflecting in the horizontal and/or vertical symmetry axes of the container.

6.1 Symmetries in two-dimensional packing problems

The two-dimensional knapsack problem exhibits multiple types of symmetries. Rectangles with identical width, height and profit can be permuted in any solution. In addition, symmetries in this problem also arise from geometric symmetries of the container. Because rectangles are placed axis-aligned within the container, any solution can be reflected horizontally and/or vertically to obtain an equivalent solution. See also Figure 6.1 for an example. Note that this type of symmetries does not correspond to permutations of variables in the formulation (6.1)–(6.11). However, the approach to breaking the symmetry stays the same: we want to restrict the solution space to a smaller representative set, such that each orbit has at least one solution in the representative set. To choose representative solutions, we propose a method based on which parts of the container are covered with rectangles in a given solution.

6.2 Handling symmetry using covering vectors

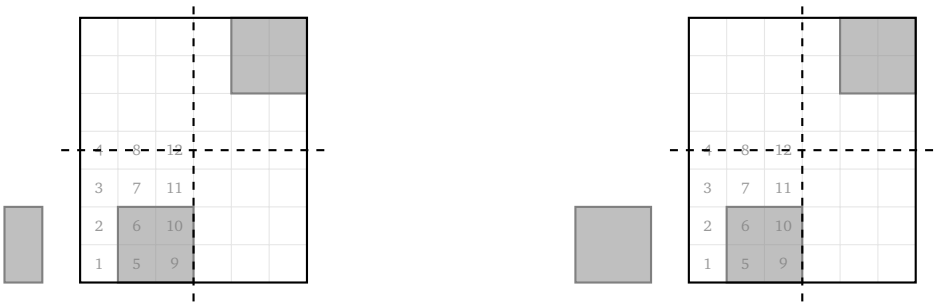
We first partition the area of the container into unit-sized *cells*, which are represented by the coordinates of the bottom-left corner of the cell,

$$\mathcal{C} = \{(i,j) \mid i \in \{0,\dots,W-1\},\ j \in \{0,\dots,H-1\}\}.$$

Let a be a node of the B&B-tree, where $Q_a \subseteq \mathcal{X}_{\text{2D-KP}}$ denotes the set of feasible solutions of the subproblem at node a. We can use the variable fixings at node a to determine whether cells are covered. For example, let $i \in \mathcal{R}$ be a rectangle of width 1 and height 1, and suppose that the variables s_i, x_i and y_i are fixed to 1, 2 and 3 at node a, respectively, then this rectangle covers the cell $(2,3)$ in every solution in Q_a. We call a rectangle i *fixed* at node a when the variables s_i, x_i, and y_i are fixed at some value at a.

Since we will compare the 4 symmetric orientations of the container, we only need to consider the cells at the bottom-left of the container in any given orientation. That is, we only consider the cells $\{(i,j) \in \mathcal{C} \mid i \leq \lceil W/2 \rceil,\ j \leq \lceil H/2 \rceil\}$. Then, based on the variable fixings at node a, we can define a partially-filled *covering vector* c^a with entries indexed by the cells, according to bottom-to-top, left-to-right ordering. The covering vector entries are either 1, 0, or empty (denoted by $*$) depending on the variable fixings at a. The entry for a cell is

- 1, if there is a fixed rectangle at a that covers the cell,
- 0, if the cell cannot be covered in any solution in Q_a,



(a) (b)

Figure 6.2: Partial solutions of the 2D-KP, where the unplaced rectangle is free to be placed anywhere in the container. (a) The unplaced rectangle fits anywhere, thus the associated covering vector does not have any zeros as entries. (b) The unplaced rectangle cannot cover cells 1 and 2 at any placement, hence the associated covering vector has zero entries for cells 1 and 2.

- $*$ (empty), if the cell is covered by a rectangle in some solutions in Q_a.

Note that if an entry is empty, it might still become 1 or 0 at a node in the subtree of a.

We illustrate this with an example. Suppose at the current node in the B&B-tree we have two fixed rectangles and one rectangle that is still free to be placed anywhere in the container; see Figure 6.2a. The order of the bottom-left cells of the container is indicated in the figure. The unplaced rectangle fits anywhere in the container and hence can still cover any cell. Therefore, the covering vector becomes

$$c^\top = \begin{bmatrix} * & * & * & * & 1 & 1 & * & * & 1 & 1 & * & * \end{bmatrix}. \tag{6.14}$$

In Figure 6.2b the situation is similar, except that the unplaced rectangle cannot be placed to cover cells 1 and 2. Therefore, the associated covering vector becomes

$$c^\top = \begin{bmatrix} 0 & 0 & * & * & 1 & 1 & * & * & 1 & 1 & * & * \end{bmatrix}. \tag{6.15}$$

Note that solutions at the leaves of the B&B-tree have a covering vector where every entry is 0 or 1. We can use the covering vector to define representative solutions for the geometric symmetry in the container. Every solution is in an orbit of (at most) 4 other solutions, each with a corresponding covering vector. To break the geometric symmetry, we choose as representatives the solutions that have a lexicographically maximal covering vector among the other covering vectors in the orbit. Note that this is similar to symmetry-breaking in the GCP and MUCP, although here the covering vector is not explicitly part of the IP formulation.

At a given node a in the branch-and-bound tree, we already want to check whether this subproblem only yields solutions that are not lexicographically in its orbit, such that we can cut off this node and its subtree from the solution process. We can do this in the following manner. We first construct the covering vector c^a corresponding to the partial solution at a. Then, we compare c^a to the three other covering vectors, obtained from the partial solutions that are symmetric to the partial solution at a. We call these c^a_{H}, c^a_{V}, and c^a_{HV}, belonging to the partial solutions obtained from reflection in the horizontal axis, vertical axis, and both axes of the container, respectively. Node a is cut off when c^a cannot be made lexicographically maximal among $\{c^a, c^a_{\mathrm{H}}, c^a_{\mathrm{V}}, c^a_{\mathrm{HV}}\}$, for any replacement of empty ($*$) entries with 0 or 1. We can cut off

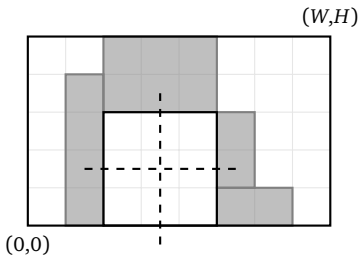


Figure 6.3: This particular placement of four rectangles in the container defines a (rectangular) subregion, indicated by its horizontal and vertical symmetry axes.

the node in this case, as this ensures that every solution in the subtree of a has a covering vector that is not lexicographically maximal in its orbit.

For an example, consider again the partial solution displayed in Figure 6.2a. The covering vectors of this partial solution is given in Equation (6.14). The covering vectors of its symmetric counterparts are

$$c_{\mathrm{H}}^{\top} = \begin{bmatrix} * & * & * & * & * & * & * & * & * & * & * & * \end{bmatrix}, \tag{6.16}$$

$$c_{\mathrm{V}}^{\top} = \begin{bmatrix} * & * & * & * & * & * & * & * & * & * & * & * \end{bmatrix}, \tag{6.17}$$

$$c_{\mathrm{HV}}^{\top} = \begin{bmatrix} 1 & 1 & * & * & 1 & 1 & * & * & * & * & * & * \end{bmatrix}. \tag{6.18}$$

Notice that the covering vector c from Equation (6.14) can still be made lexicographically maximal within $\{c, c_{\mathrm{H}}, c_{\mathrm{V}}, c_{\mathrm{HV}}\}$, by setting the first two empty entries to 1. Hence, if this partial solution appears at a node, we cannot produce a cutoff by geometric symmetry. If however the situation is as displayed in Figure 6.2b, we would be able to cut off the node. Indeed, the corresponding covering vector, in Equation (6.15) starts with a 0-entry, while c_{HV} starts with a 1-entry.

6.3 Geometric sub-symmetries

The problem also features sub-symmetries when smaller *subregions* are formed by the placement of rectangles. We define a subregion $C \subseteq \mathcal{C}$ as a subset of the container cells satisfying:

- the cells in C form a single connected area within the container (the boundary is connected), and
- the boundary of C overlaps everywhere with edges of rectangles outside of C, or container edges.

Note that a subregion need not be completely empty, and that the container itself is also a subregion.

In the remainder, we only consider rectangular subregions that form inside the container. The methods we describe for handling sub-symmetries in rectangular subregions can be extended for subregions with different shapes, but it will be convenient to describe it for rectangular regions as the shape is the same as the container itself. See Figure 6.3 for an example rectangular subregion formed by the placement of rectangles in a partial solution.

Because a subregion is bounded by outside rectangles or container edges, observe that if a rectangle is placed inside of a subregion, then it must be fully contained in the subregion.

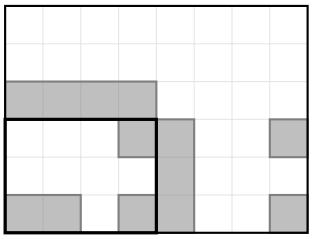


Figure 6.4: An example where simultaneously handling nested subregions can lead to conflicting cutoff decisions. In this example, there is no combination of orientations of the container and the subregion for which both covering vectors are lexicographically maximal in their orbits.

Hence, the subregion defines a smaller container region, and hence we can handle symmetries in the subregion in the same way as for the large container. We can again define a (partially-filled) covering vector for the subregion, and check whether the covering vector can be made lexicographically maximal among the symmetric orientations of the subregion.

We have to be careful when handling symmetries in multiple subregions simultaneously. When cutting off a node based on geometric sub-symmetry, we must make sure we do not cut off all representative solutions for a different subregion. When subregions are disjoint, it is easy to see that handling both sub-symmetries does not yield any conflicting decisions. However, subregions are in general overlapping. For example, the container itself will always overlap with other subregions.

Figure 6.4 displays a situation where handling the container and a nested subregion simultaneously leads to conflicting node-cutoff decisions. Suppose that there are no rectangles unplaced in the solution in Figure 6.4. Note that the covering vector of the subregion is lexicographically maximal, but the covering vector of the container is not. In fact, there does not exist an orientation of both the container and the subregion for which both covering vectors are lexicographically maximal. But this would mean that we cut off this solution in every orientation and we are left with no representative solution.

To remedy this, we define criteria that are sufficient for selecting simultaneously active subregions at every node such that no conflicting cutoff decisions are made. We first need the following definition.

Definition 6.1 (Fully symmetric)**.** *Let C be a subregion in a solution at a given node a in the branch-and-bound tree with covering vector c^a, with respect to the subregion C. We call C* fully symmetric *at node a when the covering vectors c^a, c^a_H, c^a_V and c^a_{HV} are all equal.*

Furthermore, we call a subregion *admissible* at a when the node should not be cut off based on the symmetries in the subregion (i.e., the subregion covering vector can be made lexicographically maximal in its orbit). Note that a fully-symmetric subregion is admissible.

Assume that we have a *subregion activation oracle* that determines a set of *active subregions* for every node of the branch-and-bound tree.

Lemma 6.1. *For a given node a of the branch-and-bound tree, we denote by K_a all the subregions present at a, and by $K^*_a \subseteq K_a$ the set of active subregions at a, as determined by the oracle. The following properties of the oracle are sufficient to make conflict-free cutoff decisions:*

(i) For every node a, the active subregions K^*_a are pairwise disjoint.

(ii) For a node a that is not the root node, let $\bar{K}^*_a$ denote the subregions that are active in the ancestor nodes of a. Let b be the direct parent node of a. Then every newly active subregion $C \in K^*_a \setminus K^*_b$ is either fully contained in some $C' \in K^*_b$, or disjoint from $\bar{K}^*_a$.

(iii) For a given node a with parent b, every newly active subregion $C \in K^*_a \setminus K^*_b$ is fully symmetric at a and every other subregion $C \in K^*_a \cap K^*_b$ is admissible at a.

Proof. Let S be an optimal solution for a given instance of the two-dimensional knapsack problem. Let $\mathcal{T}$ denote the branch-and-bound tree for this instance where a subregion activation oracle satisfying (i)–(iii) selects active subregions. We will show that there exists a solution S' that is equivalent to S (w.r.t. symmetries in subregions) that is found in a leaf of $\mathcal{T}$, showing the lemma. Note that we assume no other symmetries are handled in $\mathcal{T}$, to avoid possible conflicts with other symmetry-handling techniques.

Consider the following procedure for finding S'. Let $r_1 = r$ be the root of the tree $\mathcal{T}$ and let $S_1 = S$. Starting at $i = 1$, define $\mathcal{T}_i$ as the subtree of $\mathcal{T}$ rooted at r_i and truncated up to the nodes a where $K^*_a = K^*_{r_i}$, i.e., only the nodes where the oracle selects the same active subregions as for the subtree root node r_i. Define $\mathrm{orb}_{K^*_{r_i}}(S_i)$ as the set of solutions that can be obtained from S_i by applying symmetry in the subregions $K^*_{r_i}$. Let a_i be a leaf node of $\mathcal{T}_i$ for which the covering vector is compatible with some $S'_i \in \mathrm{orb}_{K^*_{r_i}}(S_i)$. If a_i is a leaf node in $\mathcal{T}$, stop and let $S' = S'_i$, showing the lemma. Otherwise, let r_{i+1} be a child node of a_i in $\mathcal{T}$ for which the covering vector is compatible with S'_i, and repeat.

We will now show that this procedure always identifies S'. First note that S' is obtained from symmetry operations on the subregions present in S. That is, from reflecting the placed rectangles in the symmetry axes of the subregions. Also note that the procedure must terminate, as $\mathcal{T}$ contains finitely many nodes. Hence it remains to show that a_i, r_i and S_i indeed exist and are well-defined for every iteration of the procedure.

For $i = 1$, note that $K^*_{r_1}$ is either empty or only contains the container region. This makes the set $\mathrm{orb}_{K^*_{r_1}}(S_1)$ well-defined, as these subregions are indeed present in the solution $S_1 = S$. For the existence of a_1, consider extending $\mathcal{T}_1$ to $\mathcal{T}'_1$ by expanding the remaining subproblem nodes where in every node of $\mathcal{T}'_1$ the active subregions remain exactly $K^*_{r_1}$. Clearly, as the active subregions are distinct, some solution $S'_1 \in \mathrm{orb}_{K^*_{r_1}}(S_1)$ must be found in $\mathcal{T}'_1$. But then there must exist a leaf of $\mathcal{T}_1$ for which the covering vector is compatible with S'_1. If a_1 is not a leaf in $\mathcal{T}$, then the direct child nodes of a_1 have active subregions that are different from the active subregions at a_1, by construction. By property (iii), the newly active subregions are fully symmetric, and hence a child r_2 of a_1 with a compatible covering vector exists. Especially note that r_2 is not pruned in $\mathcal{T}$, as the subregions $K^*_{r_2}$ are all admissible at r_2 by property (iii).

For $i > 1$, first note that the subregions $K^*_{r_i}$ exist in the solution S_i, ensured by the selection of r_i in the $(i-1)$-th iteration of the procedure, making $\mathrm{orb}_{K^*_{r_i}}(S_i)$ well-defined. Extend $\mathcal{T}_i$ to $\mathcal{T}'_i$ by expanding the remaining subproblem nodes where in every node of $\mathcal{T}'_i$ the active subregions remain exactly $K^*_{r_i}$. Suppose there is no leaf a_i of $\mathcal{T}_i$ for which the covering vector is compatible with some $S'_i \in \mathrm{orb}_{K^*_{r_i}}(S_i)$. Then there must be a newly active subregion C in $K^*_{r_i}$ for which the covering vector is not compatible with any of $\mathrm{orb}_{K^*_{r_i}}(S_i)$. Property (ii) and (iii) ensure that if an optimal solution R can be found in $\mathcal{T}'_i$, then any equivalent solution R' obtained from R using symmetries of C can also be found in $\mathcal{T}'_i$. Note that (by the previous iteration of the procedure) S_i can be found in $\mathcal{T}'_i$, reaching a contradiction. □

With this result, we have established sufficient conditions for which conflict-free sub-symmetry handling is possible. In the next section, we will describe how we detect subregions at a node in the B&B-tree, and how we activate these subregions satisfying the properties of Lemma 6.1.

6.4 Detecting and activating subregions

In general, we cannot handle symmetries in all the subregions present at a. We therefore consider the set $\mathcal{K}_a^* \subseteq \mathcal{K}_a$ of *active* subregions at a that satisfy the conditions of Lemma 6.1.

We will describe a method to find empty, rectangular subregions in the partial solution at every node a. Note that empty subregions are fully symmetric, and therefore can be used as newly added subregions at a node.

Thus, let the integer container width W and height H be given, as well as a set of rectangles $\mathcal{R}$ that are fixed in the partial solution. Every rectangle $r \in \mathcal{R}$ has a width w_r, height h_r and location of the bottom-left vertex (x_r, y_r) inside of the container. All parameters are integer, and we furthermore assume that the rectangles are placed such that they are not overlapping.

We determine the empty rectangular subregions using a *sweep line algorithm*. The sweep line algorithm approach is a common method for design geometric algorithms, see for example [6] for more information and common applications. The idea is to imagine a vertical line that swept across the container from left to right. During the sweep, we maintain a data structure, called the *status*, that stores which intervals on the sweep line are not covered by a rectangle in $\mathcal{R}$. In addition, we keep track of whether an interval in the status represents an empty rectangular subregion up to the sweep line. The status only changes when the sweep line meets the left or right boundary of a rectangle. Therefore, it is not necessary to re-compute the status at every coordinate change, we can restrict ourselves to a set of *events*, sorted in the event queue $\mathcal{Q}$, that we construct from the problem input. In case of a right edge of a rectangle, a new interval may be formed, an existing interval may be extended, or two intervals may be joined. For a left edge, intervals may fully close (forming a subregion), shrink, or split. When intervals are extended or shrunk, the subregion it represents may no longer be rectangular. When intervals are split or joined, the subregion it represents is no longer empty.

Before we discuss the events and status in detail, we first define

$$\mathcal{X} = \{x_r \mid r \in \mathcal{R}\} \cup \{x_r + w_r \mid r \in \mathcal{R}\}$$

as the set of x-coordinates where a left or a right edge of a rectangle is located. We then identify the following events for the sweep line procedure:

- For every rectangle $r \in \mathcal{R}$ a *rectangle left* event at the point (x_r, y_r).
- For every rectangle $r \in \mathcal{R}$ a *rectangle right* event at the point $(x_r + w_r, y_r)$.
- For every $x \in \mathcal{X}$, a *container top* event at the point (x, H).

The events are placed in a queue $\mathcal{Q}$, sorted lexicographically on the following:

1. The x-coordinate of the point where the event occurs.
2. The type of event: *rectangle right* before *rectangle left* before *container top*
3. The y-coordinate of the point where the event occurs.

The status data structure maintains a sorted collection of (disjoint) intervals that denote the parts of the sweep line that do not intersect with a rectangle. For an interval $I = (a, b)$ in

the status, we also maintain the x-coordinate x_1^I at which the interval was created, and the x-coordinate x_2^I at which the interval is currently closing. When the interval fully closes, it defines an empty rectangular subregion characterized by its bottom-left and top-right vertices at (x_1^I, a) and (x_2^I, b), respectively. In addition, in the status we maintain a subset S of intervals that represent an empty rectangular subregion up to the sweep line.

After constructing the event queue, we initialize the status with the single interval $(0, H)$ at starting x-coordinate 0, indicating that the full height of the container is not covered by a rectangle. We then perform the sweep line procedure as follows. While $\mathcal{Q}$ is not empty, extract the first event from $\mathcal{Q}$ and handle the event with the following case distinction:

- The event is a *rectangle left* event for the rectangle $r \in \mathcal{R}$.

  Let $(y_r, y_r + h_r)$ be the left edge of the rectangle. We process every interval $I = (a, b)$ in the status that intersects with the left edge. Then either (see also Figure 6.5):

  - The left edge completely covers I. We remove I from the status and output the subregion $((x_1^I, a), (x_r, b))$ if $I \in S$.
  - $a = y_r$ and $b > y_r + h_r$. The interval is starting to close. We update the left boundary of the interval I to $y_r + h_r$ and set $x_2^I = x_r$.
  - In any other case, the interval I is not properly closed. We shrink or split accordingly to exclude the range of the left edge from I, and we remove the resulting intervals from S.

- The event is a *rectangle right* event for the rectangle $r \in \mathcal{R}$. Let $(y_r, y_r + h_r)$ be the right edge of the rectangle. Then either (see also Figure 6.6):

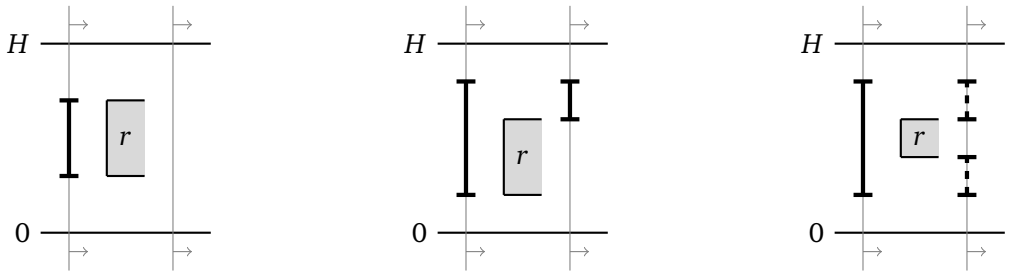
  - The rectangle edge does not share an endpoint with any of the intervals in the status. We then add the interval $I = (y_r, y_r + h_r)$ to the status, add I to S, and set $x_1^I = x_r$, $x_2^I = \infty$.
  - The rectangle edge shares only its lower endpoint with an interval $I = (a, b)$ in the status, for which $x_1^I = x_r$. We update I to be the interval $(a, y_r + h_r)$.
  - The rectangle edge shares only its lower endpoint with an interval $I = (a, b)$ in the status, for which $x_1^I < x_r$. We update I to be the interval $(a, y_r + h_r)$ and remove I from S.
  - The rectangle edge shares its upper endpoint with an interval I in the status. We update I to be the interval $(a, y_r + h_r)$ and possibly merge it with other intervals in the status when they now intersect, and remove I from S.

- The event is a *container top* event at x-coordinate x.

  Remove all intervals I from S that are only partially closed, i.e., $x_2^I = x$.

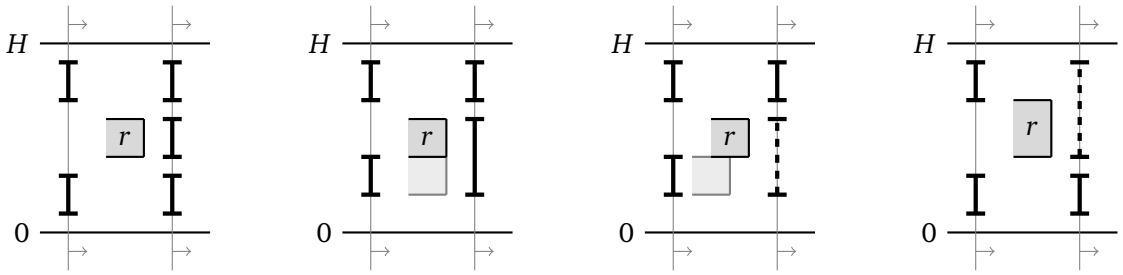
When the queue becomes empty, the remaining intervals in the status will properly close at the right boundary of the container. Hence, for any remaining interval $I = (a, b)$ in S, we output the subregion $((x_1^I, a), (W, b))$.

With the above algorithm, we can thus find empty rectangular subregions in a node a of the B&B tree. The activation handler activates newly found subregions in a greedy fashion. Let b be the parent node of a, with active subregions K_b^*. Note that the newly discovered subregions are either fully contained or disjoint from the subregions in K_b^*, because the new subregions



(a) (b) (c)

Figure 6.5: Cases for a rectangle left event. The status is displayed before and after the event is handled. A dashed interval I indicates that the interval does not represent an empty rectangular subregion ($I \notin S$). In case (a), a subregion is outputted if the interval is in S.



(a) (b) (c) (d)

Figure 6.6: Cases for a rectangle right event. The status is displayed before and after the event is handled. A dashed interval I indicates that the interval does not represent an empty rectangular subregion ($I \notin S$).

are empty. All new subregions are activated at node a, while subregions K_b^* are de-activated when necessary, i.e., to ensure that the set of active subregions at a is pairwise disjoint.

The activation handler is linked to the geometric-symmetry-breaking constraint that is described in Section 6.2. The activation handles provides the constraint handler with the active subregions, and the constraint handler applies the geometric-symmetry-breaking method to all the active subregions.

6.5 Implementation details

The empty rectangular subregion detection algorithm is implemented as follows within SCIP. The status data structure maintains the intervals in a binary search tree, such that it is efficient to find intervals that contain a given x coordinate. The event queue is implemented as a priority queue, provided in the SCIP framework. For every node in the branch-and-bound tree, the active subregions are cached in a hashmap data structure, of which an implementation is available in the SCIP framework. Since nodes in the B&B tree may not be expanded in-order due to heuristic *node selection* in the solver, we need to store active subregions for every node.

The IP formulation is constructed in Python 3.10 using the PySCIPOpt interface that exposes the SCIP API in Python. See also Appendix A. We add a single activation handler to the model. Upon creation, the activation handler is supplied with the variables s, x, y, and arrays containing the widths and heights of the rectangles in the problem instance.

The formulation is verified by testing the instance input readers and by checking the generated model for small test instances with expected output. Furthermore, the activation handler routine for detecting subregions is verified for small instances and edge-cases to see whether all empty rectangular subregions are detected.

6.6 Experiments on benchmark instances

We compare the global and sub-symmetry-handling methods using experiments on benchmark instances from `2DPackLib` [20, 19].

We solve every instance with the following models:

F	Formulation (6.1)–(6.11), with orbitope constraints for rectangles with identical properties, and with default SCIP parameters.
F-Glob	Formulation (6.1)–(6.11), with orbitope constraints for rectangles with identical properties, and geometric symmetry constraint for the container only.
F-Act	Formulation (6.1)–(6.11), with orbitope constraints for rectangles with identical properties, and greedy subregion activation handler for geometric sub-symmetries.

Note that we do not include a comparison with a model with SCIP internal symmetry handling turned off, as internal symmetry handling does not detect the geometric symmetries in the problem. Also note that we do not compare with a sub-symmetry-breaking inequalities approach, as encoding symmetry handling with a covering vector in inequalities would not be feasible. In addition, it is not obvious how one could encode the compatibility of active subregions with this approach.

6.6.1 Choices of parameters

The `2DPackLib` library provides many benchmark instances for various two-dimensional packing problems. For our experiments, we used the instances listed as originally proposed for the two-dimensional knapsack problem. We only include instances with at most 40 rectangles, as the other instances are too large to be solved by our absolute placement approach. These instances also have a large container size, which makes these instances too large to be solved by this method. This results in a benchmark set of 155 instances.

For all models, instances are solved to optimality with a time limit of 7200 seconds (2 hours). The time limit excludes construction time of the model.

6.6.2 Results

All experiments are run with the development version of SCIP 7.0.3 with the SoPlex LP-solver [14], on a single core of an Intel Xeon Platinum 8260 CPU, running at 2.4 GHz, with 10.7 GB of RAM.

The results are reported in Table 6.1, 6.2, 6.3, and 6.4, and include the following data.

W	Width of the container.
H	Height of the container.
$\lvert\mathcal{R}\rvert$	Number of rectangles.
Nodes	Number of nodes in the B&B tree.
Cutoffs	Number of nodes cut off by the geometric constraint handler.
Solving (sec)	Time spent solving the model, in seconds. When the time limit is reached when the model is not solved to optimality, the solving time is equal to 7200.

We exclude from the tables the 4 instances for which all models reach the time limit and where the geometric symmetry constraint does not lead to any cutoffs, or where every model solves the instance at the root node. Of the 44 reported instances, there are 17 instances that cannot be solved to optimality by any of the tested models within the time limit.

For the small instances with less than 20 rectangles, almost all instances can be solved quite efficiently. Because of the relatively short solving times and relatively small number of nodes in the B&B-tree, not many nodes are cut off by the geometric symmetry constraint. For the NGCUT8 instance, the global symmetry and sub-symmetry-breaking models show an improvement in solving time compared to the basic formulation. Note that for this instance F-Glob produces more cutoffs and also performs slightly better in solving time than F-Act. This shows that a greedy strategy for selecting subregions is not always optimal, as it might not lead to more symmetry breaking. In addition, the added overhead of finding subregions might have a negative impact on the total solving time.

For the larger instances with 20 up to 30 rectangles, there are more instances that reach the time limit for all models. Although for some of these instances, a number of cutoffs is produced by the geometric symmetry constraint, leading to more explored nodes within the time limit. The NGCUT3 instance is the only instance that shows an improvement in solving time, but only for handling geometric symmetries in the container. With the sub-symmetry activation handler, the geometric symmetry constraint does produce more cutoffs, but this also leads to more nodes in the B&B-tree and an increase in solving time.

The instances in our test set with 40 rectangles are similar. Neither the global symmetry

Table 6.1: 2D-KP experiments on instances with less than 20 rectangles.

Instance	W	H	$\lvert\mathcal{R}\rvert$	Model	Nodes	Cutoffs	Solving (sec)
NGCUT1	10	10	10	F	29		0.31
				F-Glob	30	4	0.31
				F-Act	29	0	0.31
GCUT01	250	250	10	F	21		0.52
				F-Glob	26	1	0.48
				F-Act	21	0	0.52
GCUT05	500	500	10	F	70		0.45
				F-Glob	61	0	0.45
				F-Act	61	0	0.46
GCUT09	1000	1000	10	F	889		2.43
				F-Glob	1472	0	3.11
				F-Act	1472	0	3.1
NGCUT8	20	20	13	F	91 826		127.06
				F-Glob	72 969	195	94.48
				F-Act	82 663	119	97.94
NGCUT10	30	30	13	F	153		0.66
				F-Glob	153	0	0.66
				F-Act	153	0	0.66
NGCUT5	15	10	14	F	2682		5.69
				F-Glob	1277	0	4.17
				F-Act	1277	0	4.15
NGCUT6	15	10	15	F	7801		22.96
				F-Glob	9192	5	26.02
				F-Act	7568	2	21.1
NGCUT11	30	30	15	F	2776		8.46
				F-Glob	3638	2	11.21
				F-Act	3504	0	11.73
CGCUT1	15	10	16	F	3 951 063		7200
				F-Glob	3 858 597	19 431	7200
				F-Act	4 151 982	6079	7200
NGCUT2	10	10	17	F	3042		9.37
				F-Glob	2839	86	9.51
				F-Act	3670	56	11.12
NGCUT9	20	20	18	F	3212		13.71
				F-Glob	2643	2	11.72
				F-Act	2626	3	11.79

Table 6.2: 2D-KP experiments on instances with 20 up to 30 rectangles.

Instance	W	H	$\lvert\mathcal{R}\rvert$	Model	Nodes	Cutoffs	Solving (sec)
GCUT02	250	250	20	F	185 073		379.12
				F-Glob	137 870	0	270.05
				F-Act	137 870	0	267.79
GCUT06	500	500	20	F	19 408		41.1
				F-Glob	17 799	0	41.15
				F-Act	17 799	0	42.3
GCUT10	1000	1000	20	F	8662		18.28
				F-Glob	9847	0	20.62
				F-Act	9847	0	18.59
NGCUT3	10	10	21	F	33 646		127.01
				F-Glob	19 245	416	90.55
				F-Act	50 400	609	175.77
NGCUT12	30	30	22	F	60 570		182.8
				F-Glob	61 421	1145	229.52
				F-Act	66 564	1208	221.68
OF1	70	40	23	F	2 033 687		7200
				F-Glob	2 275 828	5807	7200
				F-Act	2 077 645	2640	7200
OF2	70	40	24	F	1 684 682		7200
				F-Glob	1 758 726	211	7200
				F-Act	1 480 654	174	7200
OKP2	100	100	30	F	1 153 563		7200
				F-Glob	1 191 333	986	7200
				F-Act	1 230 733	838	7200
OKP3	100	100	30	F	1 326 269		7200
				F-Glob	1 341 175	544	7200
				F-Act	1 335 507	66	7200
GCUT03	250	250	30	F	1 361 251		7200
				F-Glob	1 666 992	15	7200
				F-Act	1 663 830	0	7200
GCUT07	500	500	30	F	963 350		3191.06
				F-Glob	1 286 249	2	4774.89
				F-Act	1 341 709	0	4825.99
GCUT11	1000	1000	30	F	1 160 608		7200
				F-Glob	1 265 385	5	7200
				F-Act	1 416 771	0	7200

Table 6.3: 2D-KP experiments on NGCUTFS1 instances with 40 rectangles.

Instance	W	H	$\lvert\mathcal{R}\rvert$	Model	Nodes	Cutoffs	Solving (sec)
NGCUTFS1-1	100	100	40	F	9933		118.25
				F-Glob	12 503	1	126.66
				F-Act	11 020	0	112.87
NGCUTFS1-2	100	100	40	F	650 424		7200
				F-Glob	611 775	78	7200
				F-Act	595 290	2	7200
NGCUTFS1-3	100	100	40	F	423 316		4119.12
				F-Glob	411 893	245	3873.32
				F-Act	609 040	341	7200
NGCUTFS1-4	100	100	40	F	223 752		1571.9
				F-Glob	286 282	413	1834.68
				F-Act	206 845	28	1431.4
NGCUTFS1-5	100	100	40	F	157 626		1208.29
				F-Glob	70 378	15	644.04
				F-Act	79 311	5	730.05
NGCUTFS1-6	100	100	40	F	135 400		1149.67
				F-Glob	152 315	3	1511.28
				F-Act	165 095	3	1612.16
NGCUTFS1-7	100	100	40	F	24 589		330.51
				F-Glob	21 932	11	295.98
				F-Act	40 291	0	476.09
NGCUTFS1-8	100	100	40	F	53 494		481.59
				F-Glob	51 431	6	474.85
				F-Act	68 282	6	664.51
NGCUTFS1-9	100	100	40	F	555 221		7200
				F-Glob	700 884	145	7200
				F-Act	628 762	38	7200
NGCUTFS1-10	100	100	40	F	213 792		1775.05
				F-Glob	149 921	42	1293.97
				F-Act	134 010	23	1133.33

Table 6.4: 2D-KP experiments on NGCUTFS2 instances with 40 rectangles.

Instance	W	H	$\lvert\mathcal{R}\rvert$	Model	Nodes	Cutoffs	Solving (sec)
NGCUTFS2-1	100	100	40	F	576 581		5653.68
				F-Glob	771 155	144	7200
				F-Act	323 487	7	2388.74
NGCUTFS2-2	100	100	40	F	576 192		7200
				F-Glob	622 171	102	7200
				F-Act	674 402	2	7200
NGCUTFS2-3	100	100	40	F	659 546		7200
				F-Glob	669 965	1473	7200
				F-Act	606 261	13	7200
NGCUTFS2-4	100	100	40	F	735 221		7200
				F-Glob	654 895	2168	7200
				F-Act	570 977	710	7200
NGCUTFS2-5	100	100	40	F	836 105		7200
				F-Glob	567 698	838	7200
				F-Act	646 141	11	7200
NGCUTFS2-6	100	100	40	F	659 227		7200
				F-Glob	551 479	285	7200
				F-Act	587 115	20	7200
NGCUTFS2-7	100	100	40	F	948 178		7200
				F-Glob	863 214	2502	7200
				F-Act	934 965	5534	7200
NGCUTFS2-8	100	100	40	F	224 112		1834.8
				F-Glob	346 272	800	2380.48
				F-Act	130 831	0	1023.59
NGCUTFS2-9	100	100	40	F	640 287		7200
				F-Glob	771 390	941	7200
				F-Act	729 805	117	7200
NGCUTFS2-10	100	100	40	F	790 319		7200
				F-Glob	751 006	343	7200
				F-Act	746 614	33	7200

handling in the container nor the sub-symmetry handling in the subregions shows increased performance on all instances. However, for the instances NGCUTFS1-5 and NGCUTFS1-10 both F-Glob and F-Act solve the instance faster and also shows a clear reduction in nodes in the B&B-tree. For NGCUTFS2-1, only F-Act shows a performance improvement while F-Glob could not solve the instance within the time limit. The number of node cutoffs is small for these instances, hence it is likely that the internal heuristics in SCIP also partly explain the performance gain in these instances. This becomes also clear from the results of the NGCUTFS2-8 instance, where F-Act solves the instance significantly faster, while there were no nodes cut off due to geometric symmetries.

Our results also show that for some instances the overhead of sub-symmetry handling can have a large negative impact on the total solving time. For example, the activation handler method for NGCUTFS1-3 reaches the time limit, while F-Glob and F could be solved in significantly less time. It shows that there is a trade-off between the added computational overhead of symmetry handler compared to its benefit. Moreover, this trade-off largely depends on the specific problem instance.

6.6.3 Conclusion and future improvements

Overall, the results show that for specific instances geometric symmetry and sub-symmetry handling can have a positive impact on solving two-dimensional packing problems. However, the performance gain is significantly less than what we achieved for the GCP and MUCP experiments. In part, this can be explained by the fact that the geometric symmetries that we consider have smaller orbits. For every solution, there are only at most three other equivalent solutions, because of the two symmetry axes. Therefore, breaking the symmetry has less of an impact in reducing the solution space.

In addition, the covering vector approach is only able to cut off nodes when it is certain the covering vector is not lexicographically maximal in its orbit for all solutions in the subtree. This for example means that if there exists a still unplaced rectangle of small size, then this rectangle might possibly be placed at many locations in the container. This prevents zero-entries from occurring in the covering vector, which also prevents the possibility of cutting off the current node.

For geometric-sub-symmetry handling, a clear limitation of the approach is the compatibility of the subregions. In particular this means that when during the solving process we switch to handling sub-symmetries, the global symmetry in the container cannot be handled anymore. This is in clear contrast with the suborbitopes constraints in, e.g., the MUCP problem, where all sub-symmetries can be handled simultaneously. As can be seen in the experimental results, activating new subregions greedily might not always be the optimal choice. This is also motivated by the fact that subregions are empty at the time they are activated, meaning that first rectangles need to be placed in the new subregions before cutting off the node is possible. This might not always happen, e.g., when a subregion is too small for an unplaced rectangle to be placed inside.

Finding subregions and handling geometric symmetries in these problems does introduce a computational overhead. The gain in performance from cutting off nodes based on geometric symmetries does not always outweigh the introduced overhead, as our experiments show.

For future improvements, different heuristics for activating newly discovered subregions can be explored, instead of activating new subregions greedily. Additionally, a different method of detecting subregions, including non-empty or non-rectangular subregions might improve the

number of node cutoffs and speed up the solving process. However, this would also increase subregion detection times, as there potentially exist many more of these subregions. Again, there is a trade-off to be made here, and no approach would work perfectly on every instance. Our experiments show that sub-symmetry breaking efforts for geometric symmetries might in some cases be beneficial. Further exploring the possibilities of handling geometric symmetries with activation handlers may lead to better solving times for other instances as well.

Chapter 7

Conclusion and future work

We have presented a new method for handling sub-symmetries in integer programming. Our method enables the modeler of the problem to extend the integer programming solver with custom routines that are dedicated to detecting when sub-symmetries become active during the solving process. Compared to the existing method for sub-symmetry handling, this does not require encoding these sub-symmetries explicitly in the integer programming formulation. Next to the fact that detecting sub-symmetries with custom routines is more straightforward to implement for the user, there are a number of other advantages to this approach.

Firstly, handling sub-symmetry with explicit inequalities increase the size of the IP formulation, already at the root of the branch-and-bound tree. The sub-symmetry-breaking inequalities need to be added to the root problem, even though they are 'switched off' when the corresponding sub-symmetry is not yet active. This introduces overhead for the solver, as the inequalities also end up in LP relaxations, or might play a role in solving heuristics or branching decisions. Moreover, the additional inequalities may hide the problem structure from the solver. This can have a negative impact on solving times, as the specific techniques leveraging the problem structure might not be employed by the solver. With an activation handler, the sub-symmetry breaking constraints are only handled in the model when they become active. This makes the IP formulation more lightweight at nodes of the B&B-tree, shifting the additional overhead to the activation handler routine itself.

We have compared the performance of our new method with the sub-symmetry-breaking inequalities method for the graph coloring problem and the min-up/min-down unit commitment problem. The results show that our activation handler approach is competitive with the existing methods and can for specific instances show a significant performance gain in terms of total solving time. Especially for the MUCP, the overhead of the activation handler is low, as only a single activation handler is added to the model that detects all the active sub-symmetries at once.

An additional benefit is that our approach is flexible. For sub-symmetry handling, we are not restricted to use explicit inequalities for handling symmetry. Instead, the activation handler can be linked to high-level constraints, such as the orbitope constraint in SCIP, for which both specialized propagation and separation routines can be used to break the symmetry.

The flexibility is also demonstrated by applying our method to geometric sub-symmetry handling in the two-dimensional knapsack problem. We have defined a new approach to handle geometric symmetries in two-dimensional packing problems, both globally and for subregions. The covering vectors that are used to handle symmetry are not explicitly encoded in the

IP formulation, nor are the subregions that we possibly want to consider for sub-symmetry-breaking. Therefore, explicit inequalities can consequently not be used for sub-symmetry handling. Moreover, the restrictions on the activation of subregions to ensure conflict-free decisions cannot be efficiently encoded in the formulation, as this would require many more additional binary variables.

By introducing activation handlers as a separate plugin type in the SCIP solver, we have set up a framework for users to define their own activation handlers that can be used for other types of problems. In our applications, we made an effort to make the activation handlers re-usable in a broader context. The variable-fixings activation handler that was used for the GCP can be applied to any problem where sub-symmetries are defined by variable fixings only. The suborbitope constraint handler that was used for the MUCP detects patterns in variable fixings of a given matrix of variables, and activates symmetries in corresponding submatrices. The pattern for the MUCP was defined by 'towers' of 0- or 1-fixings of variables, but different patterns might be applicable for other types of problems. For future research, investigating what kind of general patterns define sub-symmetries in other problems might lead to a more general implementation of a suborbitope activation handler.

Additionally, a domain specific language might be beneficial for users to express to the solver when sub-symmetries become active. Currently, the API of the activation handlers fulfills this role, but in extending to more problems and types of sub-symmetries, a DSL can help the user in leveraging sub-symmetry handling more easily.

Lastly, we note that by introducing activation handlers as a separate plugin type, it can also be used outside of symmetry handling. We leave it to future research to explore where dynamically activating and de-activating parts of the solver during the solving process might be beneficial. A potential application would be in mixed-integer non-linear programming, where for example an activation handler can detect when a non-linear constraint becomes convex, for which additional solving strategies may be activated.

Bibliography

[1] Tobias Achterberg. “SCIP: solving constraint integer programs”. In: *Mathematical Programming Computation* 1.1 (2009), pp. 1–41. DOI: 10.1007/s12532-008-0001-1.

[2] Pascale Bendotti, Pierre Fouilhoux, and Cécile Rottner. “On the complexity of the unit commitment problem”. In: *Annals of Operations Research* 274.1 (2019), pp. 119–130. DOI: 10.1007/s10479-018-2827-x.

[3] Pascale Bendotti, Pierre Fouilhoux, and Cécile Rottner. “Orbitopal fixing for the full (sub-)orbitope and application to the unit commitment problem”. In: *Mathematical Programming* 186.1 (2021), pp. 337–372. DOI: 10.1007/s10107-019-01457-1.

[4] Pascale Bendotti, Pierre Fouilhoux, and Cécile Rottner. “Symmetry-breaking inequalities for ILP with structured sub-symmetry”. In: *Mathematical Programming* 183.1 (2020), pp. 61–103. DOI: 10.1007/s10107-020-01491-4.

[5] Pascale Bendotti, Pierre Fouilhoux, and Cécile Rottner. “The min-up/min-down unit commitment polytope”. In: *Journal of Combinatorial Optimization* 36.3 (2018), pp. 1024–1058. DOI: 10.1007/s10878-018-0273-y.

[6] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*. 3rd ed. Springer, 1997. DOI: 10.1007/978-3-540-77974-2.

[7] Daniel Brélaz. “New methods to color the vertices of a graph”. In: *Communications of the ACM* 22.4 (1979), pp. 251–256. DOI: 10.1145/359094.359101.

[8] Miguel Carrión and José M. Arroyo. “A computationally efficient mixed-integer linear formulation for the thermal unit commitment problem”. In: *IEEE Transactions on power systems* 21.3 (2006), pp. 1371–1378. DOI: 10.1109/TPWRS.2006.876672.

[9] Chin-Sheng Chen, Shen-Ming Lee, and Q.S. Shen. “An analytical model for the container loading problem”. In: *European Journal of operational research* 80.1 (1995), pp. 68–76. DOI: 10.1016/0377-2217(94)00002-T.

[10] Pablo Coll, Javier Marenco, Isabel Méndez Díaz, and Paula Zabala. “Facets of the graph coloring polytope”. In: *Annals of Operations Research* 116.1 (2002), pp. 79–90. DOI: 10.1023/A:1021315911306.

[11] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. *Integer programming*. Vol. 271. Graduate Texts in Mathematics. Springer, 2014. DOI: 10.1007/978-3-319-11008-0.

[12] George B. Dantzig. “Maximization of a linear function of variables subject to linear inequalities”. In: *Activity analysis of production and allocation* 13 (1951), pp. 339–347.

[13] Harald Dyckhoff. “A typology of cutting and packing problems”. In: *European Journal of Operational Research* 44.2 (1990), pp. 145–159. DOI: 10.1016/0377-2217(90)90350-K.

[14] Gerald Gamrath et al. *The SCIP Optimization Suite 7.0*. ZIB-Report 20-10. Zuse Institute Berlin, 2020. URL: http://nbn-resolving.de/urn:nbn:de:0297-zib-78023.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. San Francisco, CA, United States: W. H. Freeman, 1979. ISBN: 0-7167-1045-5.

[16] Ambros Gleixner et al. *The SCIP Optimization Suite 5.0*. ZIB-Report 17-61. Zuse Institute Berlin, 2017. URL: http://nbn-resolving.de/urn:nbn:de:0297-zib-66297.

[17] Christopher Hojny and Marc E. Pfetsch. "Polytopes associated with symmetry handling". In: *Mathematical Programming* 175.1 (2019), pp. 197–240. DOI: 10.1007/s10107-018-1239-7.

[18] IBM Corporation. *IBM ILOG CPLEX Optimization Studio CPLEX 12.8 User's Manual*. 2017. URL: https://www.ibm.com/docs/en/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/usrcplex.pdf.

[19] Manuel Iori, Vinícius L. de Lima, Silvano Martello, Flávio K. Miyazawa, and Michele Monaci. "Exact solution techniques for two-dimensional cutting and packing". In: *European Journal of Operational Research* 289.2 (2021), pp. 399–415. DOI: 10.1016/j.ejor.2020.06.050.

[20] Manuel Iori, Vinícius L. de Lima, Silvano Martello, and Michele Monaci. "2DPackLib: a two-dimensional cutting and packing library". In: *Optimization Letters* 16.2 (2022), pp. 471–480. DOI: 10.1007/s11590-021-01808-y.

[21] Tommi Junttila and Petteri Kaski. "Engineering an efficient canonical labeling tool for large and sparse graphs". In: *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*. Ed. by David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick. SIAM, 2007, pp. 135–149.

[22] Volker Kaibel and Marc E. Pfetsch. "Packing and partitioning orbitopes". In: *Mathematical Programming* 114.1 (2008), pp. 1–36. DOI: 10.1007/s10107-006-0081-5.

[23] Narendra Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 1984, pp. 302–311. DOI: 10.1145/800057.808695.

[24] Leonid G. Khachiyan. "A polynomial algorithm in linear programming". In: *Soviet Mathematics Doklady*. Vol. 20. 1979, pp. 191–194. URL: http://mi.mathnet.ru/eng/dan/v244/i5/p1093.

[25] Ben Knueven, Jim Ostrowski, and Sebastian Pokutta. "Detecting almost symmetries of graphs". In: *Mathematical Programming Computation* 10.2 (2018), pp. 143–185. DOI: 10.1007/s12532-017-0124-3.

[26] Andrea Lodi, Silvano Martello, and Michele Monaci. "Two-dimensional packing problems: A survey". In: *European journal of operational research* 141.2 (2002), pp. 241–252. DOI: 10.1016/S0377-2217(02)00123-6.

[27] Andrea Lodi, Silvano Martello, and Daniele Vigo. "Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems". In: *INFORMS Journal on Computing* 11.4 (1999), pp. 345–357. DOI: 10.1287/ijoc.11.4.345.

[28] Andrea Lodi, Silvano Martello, and Daniele Vigo. "Recent advances on two-dimensional bin packing problems". In: *Discrete Applied Mathematics* 123.1-3 (2002), pp. 379–396. DOI: 10.1016/S0166-218X(01)00347-X.

[29] Andreas Loos. “Describing orbitopes by linear inequalities and projection based tools”. PhD thesis. Otto-von-Guericke-Universitaet Magdeburg, 2010.

[30] Enrico Malaguti and Paolo Toth. “A survey on vertex coloring problems”. In: *International transactions in operational research* 17.1 (2010), pp. 1–34. DOI: 10.1111/j.1475-3995.2009.00696.x.

[31] François Margot. “Exploiting orbits in symmetric ILP”. In: *Mathematical Programming* 98.1 (2003), pp. 3–21. DOI: 10.1007/s10107-003-0394-6.

[32] François Margot. “Symmetry in Integer Linear Programming”. In: *50 Years of Integer Programming 1958–2008*. Ed. by Michael Jünger et al. Springer Berlin Heidelberg, 2010. Chap. 17, pp. 647–686. DOI: 10.1007/978-3-540-68279-0_17.

[33] Brendan D. McKay and Adolfo Piperno. “Practical graph isomorphism, II”. In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112. DOI: 10.1016/j.jsc.2013.09.003.

[34] Isabel Méndez-Díaz and Paula Zabala. “A branch-and-cut algorithm for graph coloring”. In: *Discrete Applied Mathematics* 154.5 (2006), pp. 826–847. DOI: 10.1016/j.dam.2005.05.022.

[35] Isabel Méndez-Díaz and Paula Zabala. “A Polyhedral Approach for Graph Coloring1”. In: *Electronic Notes in Discrete Mathematics* 7 (2001), pp. 178–181. DOI: 10.1016/S1571-0653(04)00254-9.

[36] Hidetoshi Onodera, Yo Taniguchi, and Keikichi Tamaru. “Branch-and-bound placement for building block layout”. In: *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* 76.7 (1993), pp. 15–26. DOI: 10.1002/ecjc.4430760702.

[37] James Ostrowski, Miguel F. Anjos, and Anthony Vannelli. “Modified orbital branching for structured symmetry with an application to unit commitment”. In: *Mathematical Programming* 150.1 (2015), pp. 99–129. DOI: 10.1007/s10107-014-0812-y.

[38] Marc E. Pfetsch and Thomas Rehn. “A computational comparison of symmetry handling methods for mixed integer programs”. In: *Mathematical Programming Computation* 11.1 (2019), pp. 37–93. DOI: 10.1007/s12532-018-0140-y.

[39] Deepak Rajan and Samer Takriti. *Minimum Up/Down Polytopes of the Unit Commitment Problem with Start-Up Costs*. IBM Research Report. IBM Research Division, 2005. URL: https://dominoweb.draco.res.ibm.com/reports/rc23628.pdf.

[40] Joseph J. Rotman. *An Introduction to the Theory of Groups*. 4th ed. Vol. 148. Graduate Texts in Mathematics. Springer, 1984. DOI: 10.1007/978-1-4612-4176-8.

[41] Domenico Salvagnin. “A dominance procedure for integer programming”. Master’s thesis. University of Padova, Padova, Italy, 2005.

[42] *Vertex Coloring - Graph Coloring Instances*. URL: https://sites.google.com/site/graphcoloring/vertex-coloring (visited on 2022-07-13).

Appendix A

Links to implementation

The implementation of the activation handlers described in this thesis is done on a fork of SCIP, available at `https://gitlab.tue.nl/master-project-sten-wessel/scip`.

The models used for the experiments and the experimental setup is done in Python. The repository is available at `https://gitlab.tue.nl/master-project-sten-wessel/experiments`. Between Python and SCIP, the PySCIPOpt library takes care of the communication. A fork of the PySCIPOpt library that supports the activation handler is available at `https://gitlab.tue.nl/master-project-sten-wessel/pyscipopt`.

Appendix B

Graph coloring problem experiments on CPLEX

This table shows the results of running the models F, F-S0 and F-Ineq on CPLEX 12.8 [18] for the graph coloring problem, as described in Section 4.5.2.

Instance	Model	Nodes	Solving (sec)
1-FullIns_5	F	97 125	4325.08
	F-S0	97 125	1350.5
	F-Ineq	19 938	2011.43
1-Insertions_4	F	938 345	811.77
	F-S0	938 345	456.72
	F-Ineq	606 896	928.59
2-FullIns_4	F	8455	245.6
	F-S0	8455	85.27
	F-Ineq	2158	272.61
2-Insertions_4	F	216 705	7200
	F-S0	268 197	7200
	F-Ineq	273 544	7200
4-Insertions_3	F	795 928	737.37
	F-S0	795 928	207.02
	F-Ineq	857 324	803.69
5-FullIns_3	F	9468	427.4
	F-S0	9468	84.15
	F-Ineq	8090	344.99
DSJC125.1	F	11	77.04
	F-S0	11	43.07
	F-Ineq	11	92.65
DSJC125.5	F	0	7200
	F-S0	14 214	7200
	F-Ineq	6	7200
DSJC125.9	F	0	7200

Instance	Model	Nodes	Solving (sec)
	F-S0	0	7200
	F-Ineq	0	7200
DSJC250.9	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
DSJR500.5	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
ash608GPIA	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
flat300_20_0	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
flat300_26_0	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
flat300_28_0	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
le450_15a	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
myciel6	F	2 721 057	7200
	F-S0	3 736 327	7200
	F-Ineq	1 106 806	7200
myciel7	F	214 104	7200
	F-S0	789 085	7200
	F-Ineq	65 586	7200
queen9_9	F	1 564 253	7200
	F-S0	1 464 225	7200
	F-Ineq	1 597 926	7200
r125.1c	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
r125.5	F	11 555	7200
	F-S0	14 173	7200
	F-Ineq	0	7200
r250.5	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
school1	F	0	7200

Instance	Model	Nodes	Solving (sec)
	F-S0	0	7200
	F-Ineq	3	7200
school1_nsh	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200
wap05a	F	0	7200
	F-S0	0	7200
	F-Ineq	0	7200