VRIJE
UNIVERSITEIT
BRUSSEL

STATIC SYMMETRY AND DOMINANCE HANDLING FOR PSEUDO-BOOLEAN OPTIMIZATION

Bachelor Thesis

Daimy Van Caudenberg
daimy.stefanie.van.caudenberg@vub.be
0565495

June 2022

Promotor: Prof. Dr. Bart Bogaerts
Sciences and Bio-engineering Sciences

Contents

Chapter 1

Introduction

Combinatorial search and optimization problems have been an active research field for many years. Since a lot of real-life combinatorial problems can be naturally modelled and often efficiently solved using constraint techniques this field knows many applications. Bruynooghe (1981) defines the combinatorial search problem as finding values for a set of variables where each variable has a known set of possible values and the solution needs to satisfy a given constraint. Examples of such search problems are Boolean Satisfiability (SAT) and the graph colouring problem. The combinatorial optimization problem consists of finding an optimal solution to the same problem given an objective function. Some well-known examples of optimization problems are the knapsack problem and the travelling salesman problem.

Due to its wide applicability, SAT has been researched extensively and now applications in contexts such as planning (Kautz and Selman 1992), software verification (Biere et al. 1999), and cryptology (Massacci and Marraro 2000) exist. The *conflict-driven clause learning* (CDCL) algorithm proposed by Marques-Silva and Sakallah (1999), which itself is an extension of the *Davis-Putnam-Logemann-Loveland*-algorithm (DPLL) by Davis et al. (1962), lies at the heart of almost all of the state-of-the-art SAT solvers available nowadays. These solvers implicitly generate an unsatisfiability proof while searching for the solution for an unsatisfiable problem. When it is known that the best proof for a given problem is long this causes issues. For problems such as the pigeon-hole problem Haken (1985) has shown that exponential lower bounds exist for the unsatisfiability proof. A solution thus is needed.

There are several ways to increase the performance of constraint satisfaction and optimization solvers. A first way to achieve this is the addition of *symmetry handling* techniques. Several tools exist for improving search speed based on symmetries. For instance `BreakId` (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016) and `Shatter` (F. Aloul, Sakallah, et al. 2006) use a static symmetry breaking technique where they modify the given problem during a preprocessing phase to prevent that isomorphic parts of the search tree are visited. Various other tools exploit symmetries dynamically, i.e., during the search. That is, they intervene in the search process, for instance by modifying the choice heuristics to take symmetries into account (Benhamou and Nabhani 2010), by adding new propagators that propagate symmetric information (Devriendt, Bogaerts, De Cat, et al. 2012) or by learning symmetrical versions of learned clauses (Benhamou, Nabhani, et al. 2010). A second way to prune the search space is the exploitation of *dominance relations* as proposed by Chu and Stuckey (2015), this can be viewed as a generalization of symmetry that lends itself to optimization problems.

A third way to increase the performance of the solvers is using other search methods that offer a better way to generate proofs. One such method can be found in *0–1 integer linear*

programming. Here constraints are expressed as linear equalities and inequalities and variables are restricted to the integers 0 and 1. This way of expressing constraints opens up the possibility of using a different search method. Cook et al. (1987) have shown that this method can be exponentially stronger than the method used by the SAT solvers, but it still has the same lower bound as the method used by the SAT solvers mentioned above.

In this thesis, we propose a new symmetry-induced dominance relation together with static breaking formulas for this dominance relation. This hopefully allows for improved pruning that is valid for combinatorial search and optimization problems.

Chapter 2

Preliminaries

In this chapter, we offer a short theoretic introduction on the subjects used in this paper. These preliminaries were based on the work of: Devriendt, Bogaerts, Bruynooghe, and Denecker (2016) and Devriendt, Bogaerts, and Bruynooghe (2017).

Propositional Logic Let Σ be a set of Boolean variables and $\mathbb{B} = \{\mathbf{f}, \mathbf{t}\}$ the set of Boolean values. For each $x \in \Sigma$, there exist two *literals*; the *positive* literal denoted by x and the *negative* literal denoted by $\neg x$ or $\overline{x}$. We write $\overline{\Sigma}$ for the set of all literals over Σ. A *(partial) assignment* is a set of literals ($\alpha \subseteq \overline{\Sigma}$) such that α contains at most one literal over each variable in Σ. Under assignment α, a literal l is said to be *true* if $l \in \alpha$, *false* if $\neg l \in \alpha$, and *unknown* otherwise. As usual, we often identify true with 1 and false with 0. An assignment is *complete* if it contains exactly one literal over each variable in Σ. The set of all complete assignments is denoted A_v. If an assignment is complete, it can equivalently be viewed as a function $\alpha : \Sigma \mapsto \mathbb{B} \cup \{\mathbf{u}\}$ and extended to literals as $\alpha(\neg x) = \neg\alpha(x)$, where $\neg\mathbf{t} = \mathbf{f}$, $\neg\mathbf{f} = \mathbf{t}$ and $\neg\mathbf{u} = \mathbf{u}$. We will use these two views interchangeably.

Boolean Satisfiability Problem A *clause* is a finite disjunction of literals denoted $c = \bigvee_{i=1}^{k} l_i$, and a *formula* is a finite conjunction of clauses written as $\psi = \bigwedge_{i=1}^{j} c_i$ (assuming formulas are in conjunctive normal form (CNF)). An assignment α *satisfies* a clause c if the clause contains at least one true literal under α ($c \cap \alpha \neq \emptyset$), we write $\alpha \models c$. A CNF formula is *satisfiable* if an assignment exists that satisfies all of the formula's clauses, and it is unsatisfiable otherwise. The SAT problem consists of deciding whether a Boolean formula is satisfiable.

Pseudo-Boolean Solving A *pseudo-Boolean constraint* C is a 0–1 integer linear constraint denoted by $C = \sum_i a_i l_i \bowtie A$, where $\bowtie \in \{\leq, \geq, =, >, <\}$ and $a_i, A \in \mathbb{Z}$. A *pseudo-Boolean formula* F is a conjunction of pseudo-Boolean constraints denoted $F = \bigwedge_{i=1}^{j} C_i$. The sum of the positive literal x and negative literal $\neg x$ equals 1. The *normal form* $C = \sum_i a_i l_i \geq K$ requires the constraint to always be greater-than-or-equal-to and $a_i, K \in \mathbb{N}$. When the constraint is normalized, K is referred to as the *degree (of falsity)* of the constraint. A pseudo-Boolean constraint is satisfied under assignment α when $\sum_i a_i \alpha(l_i) \bowtie A$. A *model* of a pseudo-Boolean formula is an assignment α such that all constraints are satisfied under α. *Pseudo-Boolean Solving* decides whether a pseudo-Boolean formula F is satisfiable, *pseudo-Boolean Optimization* finds satisfying assignments to F that minimize an *objective function* $f : A_v \mapsto \mathbb{Z}$ defined as $f(\alpha) = \sum_i w_i l_i$ with $w_i \in \mathbb{Z}$. If we wish to maximize the objective function we replace $f(\alpha)$ by $-f(\alpha)$. We focus on minimizing the objective function. A pseudo-Boolean Solving problem can

be viewed as a pseudo-Boolean Optimization problem with the objective function $f(\alpha) = 0$ for all assignments α. A clause $c = \bigvee_{i=1}^{k} l_i$ can be expressed as $\sum_{i=1}^{k} l_i \geq 1$. As a result, formulas in CNF are special cases of pseudo-Boolean formulas. From now on we will give our definitions for the more general case.

Group Theoretical Concepts A *permutation* π is a bijection from a set onto itself. We write permutations in disjoint cycle notation: $(abc)(de)$ is the permutation that maps a to b, b to c, c to a and swaps d and e. All other elements are mapped to themselves. A *swap* is a non-trivial permutation that is an involution. Permutations form algebraic groups under the composition relation ($\circ$). The *support* $Supp(\pi)$ of a permutation π is the set of elements $\{x \mid \pi(x) \neq x\}$. The support $Supp(\mathcal{P})$ of a permutation group $\mathcal{P}$ is the union of the supports of permutations in $\mathcal{P}$.

Symmetry From now on we only consider permutations π of the set of literals $\overline{\Sigma}$ of a given formula that commute with negation i.e., $\pi(\neg l) = \neg\pi(l)$ for all $l \in \overline{\Sigma}$. We extend π to various semantic objects in the expected way:

- pseudo-Boolean Constraints $\pi(\sum_i a_i l_i \geq A) = \sum_i a_i \pi(l_i) \geq A$,
- to pseudo-Boolean formulas: $\pi(C_1 \wedge \ldots \wedge C_n) = \pi(C_1) \wedge \ldots \wedge \pi(C_n)$,
- to objective functions: $\pi(f)(\alpha) = f(\pi(\alpha))$
- and to assignments: $\pi(\alpha) = \{\pi(l) \mid l \in \alpha\}$.

A *semantic symmetry* π of a formula F over Σ is a permutation over $\overline{\Sigma}$ that *preserves satisfaction to* F; i.e. $\alpha \models F$ if and only if $\pi(\alpha) \models F$. A permutation π of $\overline{\Sigma}$ is a *syntactic symmetry* of a propositional formula F over Σ if π fixes the formula; i.e. $\pi(F) = F$. It is easy to see that this last condition added to the commutation with negation guarantees that π maps assignments to assignments, preserving satisfaction to F. Since this type of symmetry, i.e. syntactic symmetry, can be detected with relative ease, typically only this type of symmetry is exploited.

Chapter 3

Symmetry

Symmetries often occur when sets of variables in a problem are interchangeable. The most prototypical example for this phenomenon is the *pigeonhole problem*. All the pigeons (or holes) are exchangeable and each of these swaps represents a symmetric version of the problem. Solving the pigeon hole problem with a traditional Boolean SAT solver for high values of n is known to be impossible in reasonable time (F. Aloul, Ramani, et al. 2002).

If we are aware that a problem exhibits symmetry, we can use it to our advantage. In some cases these symmetries might be known which opens up the possibility of using them. In other cases we are not aware of them a priori, and as a result these symmetries have to be detected.

First of all we will discuss symmetry detection. Next we will make the distinction between *breaking* and *non-breaking* techniques as well as *dynamic symmetry handling* and *static symmetry breaking*. This distinction will help us give an overview of the state of the art of symmetry handling.

3.1 Symmetry Detection

A common way to detect the symmetries of a given (pseudo-)Boolean formula is to transform the instance into a coloured graph that represents an unordered, simplified version of the syntax tree of the formula. When this is done with some care, we can ensure a one-to-one correspondence between the syntactic symmetries of the formula and the automorphism groups of the constructed graph. Tools such as `Nauty` (McKay and Piperno 2014), `Saucy 3.0` (Katebi et al. 2010) and `Bliss` (Junttila and Kaski 2007) can compute these automorphism groups. F. A. Aloul et al. (2003) propose the following method to convert (pseudo-)Boolean formulas to graphs.

A formula with V variables, C clauses and P pseudo-Boolean constraints is transformed into a graph as follows:

- Two vertices per variable represent the positive and negative literals. Each such pair is connected by an edge.
- Any CNF clause is treated as follows: two-literal clauses are represented by an edge connecting the literals, other clauses are represented by a new vertex representing the clause connected to the literal vertices.
- Clause vertices get one colour, literal vertices another.
- Literals in a pseudo-Boolean constraint P_i are organized as follows:

  - The literals in P_i are sorted by coefficient value and literals with the same coefficient are grouped.
  - For each group of literals, L_j a single vertex $X_{i,j}$ (the coefficient vertex) is created. Edges are then added to connect each literal to its literal group.
  - A different colour is used for each distinct coefficient value encountered in the formula.

- Each pseudo-Boolean constraint P_i is represented by a vertex Y_i. Edges are added to connect Y_i to each of the coefficient vertices $X_{i,j}$.
- The vertices Y_i are coloured according to the constraint's right-hand sided value b. Each unique b value indicates a new colour.

This method guarantees that the automorphism groups of the created graph have a one-to-one correspondence with the syntactic symmetries of the formula.

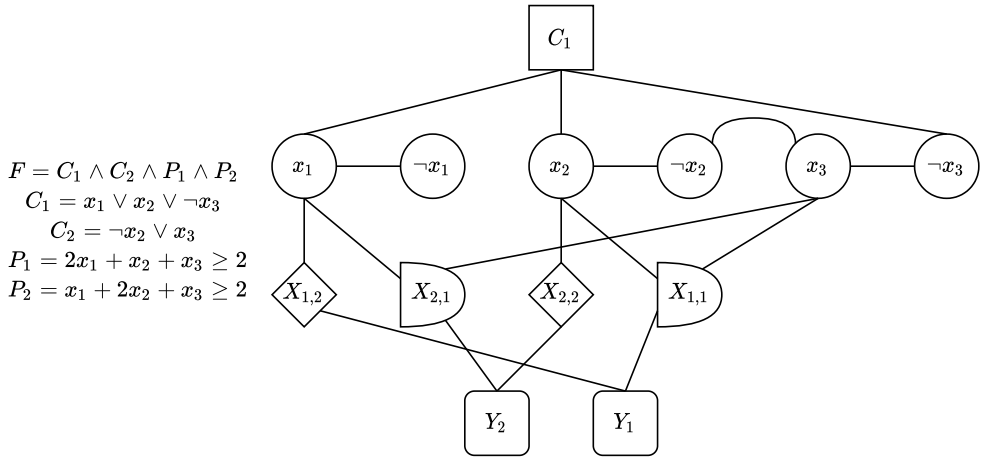


Figure 3.1: Example of a formula F and the constructed graph. The different vertex shapes represent different colours.

3.2 Symmetry Handling

The goal of all symmetry handling techniques is to cut out parts of the search tree that are symmetrical to other parts of it. First, we make the distinction between techniques that *break* the symmetries and those that do not. We call a technique *breaking* if the cut-out parts potentially contain solutions of the original problem and *non-breaking* if this cannot occur. In other words, non-breaking techniques keep all models of the formula intact, while breaking techniques can eliminate models, so long as the pruning of the search tree that they do happens in a satisfiability-preserving manner. This implies that breaking techniques are not suitable for model-counting, they might also cause issues when using domain-specific heuristics since the target of the heuristic might be pruned during the search.

Next, we make a distinction between *static* and *dynamic* techniques. Static techniques break symmetries during a preprocessing phase while dynamic techniques intervene during the search algorithm. Static techniques have the advantage that symmetry breaking constraints are added

to the existing instance during a preprocessing phase, this instance can then be passed to a solver of choice. Dynamic techniques on the other hand are dependent upon the solver itself. Table 3.1 contains some examples of different symmetry handling techniques, ordered according to these distinctions.

	Breaking	Non-Breaking
Static	Global symmetry breaking, `BreakID`, `Shatter`...	$\emptyset$
Dynamic	Local symmetry breaking, Effective symmetry breaking...	Symmetrical learning, SEL, SLS, SP...

Table 3.1: Symmetry Handling Techniques

First, we discuss *static symmetry breaking*. This technique is implemented as *global symmetry breaking* and can be found in tools such as `Shatter` (F. Aloul, Sakallah, et al. 2006) and `BreakID` (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). The technique consists of first detecting the symmetry groups of the instance and next, adding symmetry breaking constraints to the instance before handing it over to a solver. A common way to generate these symmetry breaking constraints is by constructing *Lex-Leader Constraints*. This method, proposed by Crawford et al. (1997), offers a generic way for deriving a complete and sound set of symmetry breaking constraints. A symmetry breaking constraint S is *sound* if it preserves at least one assignment from each symmetry class and it is called *complete* if it preserves at most one assignment from each class. The Lex-Leader constraints are defined as follows by Devriendt, Bogaerts, Bruynooghe, and Denecker (2016):

Definition 1 (Lex-Leader Constraint)**.** Let ψ be a formula over Σ, π a symmetry of ψ, $\leq_{lex}$ an order on Σ and $\leq_\alpha$ the lexicographic order induced by $\leq_{lex}$ on the set of Σ-assignments. A formula LL_π over an extension of Σ is a lex-leader constraint for π if for each Σ-assignment α, α can be extended to a model of LL_π if and only if $\alpha \leq_\alpha \pi(\alpha)$.

In other words, Lex-Leader constraints select the smallest assignment α of each symmetry class according to a fixed order $\leq_{lex}$. Each assignment whose image under π is smaller according to this order is eliminated. Adding these constraints for each permutation π restricts the search tree to only one search branch per symmetry class. Optimizations can be made by altering the encoding of the constraints or by exploiting the algebraic properties of the symmetry group.

Next, we discuss *dynamic symmetry breaking* techniques. One such technique is *local symmetry breaking* as implemented by Benhamou and Nabhani (2010). The technique consists of extending static symmetry breaking's symmetry detection and elimination to so-called *local symmetries*. This is done by using the automorphism groups of the graphs of the sub-formulas defined at each node of the search tree as opposed to using the formula of the instance as was done with static symmetry breaking. By using this technique the search algorithm's heuristic is adapted to take symmetries into account. Effective symmetry breaking (Metin et al. 2018) is also a dynamic breaking technique. Here, an extra component keeps track of the current search branch and forces a backjump when this search branch turns out to be redundant. This backjump is forced by adding an "efficient symmetry breaking" clause to the formula, based on the static breaking clauses. Since these techniques are breaking techniques as well, models could be pruned from the search tree.

Next, we discuss *dynamic non-breaking* techniques. *Symmetrical learning* uses failed search branches to generate constraints that prevent the search from visiting symmetric versions of these branches. Since these techniques only prune the symmetries of failed branches, all models are

preserved. These techniques are thus suitable for model-counting problems. Learning all symmetrical clauses is infeasible due to the high amount of possible clauses to learn (Heule et al. 2005). So the real difficulty is to make a good a selection of clauses to learn. *Symmetry propagation* (Devriendt, Bogaerts, De Cat, et al. 2012), *symmetrical learning scheme* (Benhamou, Nabhani, et al. 2010) and *symmetrical explanation learning* (Devriendt, Bogaerts, and Bruynooghe 2017) all offer different solutions. Devriendt, Bogaerts, and Bruynooghe (2017) have shown that symmetrical explanation learning outperforms the other symmetrical learning techniques mentioned above. Breaking techniques still outperform non-breaking techniques but symmetrical explanation learning has almost closed the gap in performance between both techniques as shown by Devriendt, Bogaerts, and Bruynooghe (2017).

Lastly, since non-breaking techniques rely on learning from failed search branches to prevent pruning out models, this can not be done during a preprocessing phase. This implies that no *static non-breaking* techniques exist, this is represented by $\emptyset$ in Table 3.1. From now on we will always refer to static symmetry handling as static symmetry breaking.

Chapter 4

Dominance

The symmetry handling techniques discussed in the previous section all focus on *decision* problems, but what if we focus our attention on *optimization* problems as well? These problems consist of a formula F and an objective function f. This implies that using symmetry breaking techniques is no longer an option if these techniques possibly prune optimal branches from the search tree. This is why we introduce *dominance relations*. These dominance relations describe pairs of assignments, where one of the assignments dominates the other when that assignment is better than the other in some "suitable sense". That "suitable sense" can be comparing the optimization term of the problem, or it can be something more complex. Similar to symmetries, these relations can be used to prune the search tree. Dominance relations can be exploited to add formulas that prevent the search from exploring non-dominant assignments, which we will call *Static Dominance Breaking*.

4.1 Dominance Relations

First, we take a closer look at what these relations are. We define a dominance relation $\preceq$ as follows:

Definition 2. Let Σ be a set of variables, F a (pseudo-)Boolean formula over Σ, f an objective function over Σ and α and β assignments of F. A dominance relation $\preceq$ is a preorder (a reflexive and transitive binary relation) on the set of Σ-assignments such that whenever $\alpha \preceq \beta$ it holds that

- $f(\alpha) \leq f(\beta)$
- if $\beta \models F$, then also $\alpha \models F$.

I.e., the dominance relation should respect the problem specified by the couple (F, f) in the sense that non-models of F cannot dominate models of F and whenever one assignment dominates another, it should be at least as good as the other assignment in terms of the objective function. As usual, our preorder $\preceq$ induces a partial order $\prec$ (an irreflexive, transitive and asymmetric binary relation) where $\alpha \prec \beta$ if and only if $\alpha \preceq \beta \wedge \beta \not\preceq \alpha$.

The simplest example of a dominance relation one could imagine is:

$$\{(\alpha, \beta) \mid \alpha \models F \wedge \beta \models F \wedge f(\alpha) \leq f(\beta)\}.$$

This relation fulfils all requirements of the definition; α and β both model F and α's objective function value is smaller than β's objective function value.

Using symmetries to define new dominance relations is also possible as long as we take the objective function into account. We define *strong symmetries* as the permutations $\sigma \in \mathcal{S}$, where $\mathcal{S}$ is a group of symmetries of both the formula F and objective function f. Using these symmetries we can define a new dominance relation:

$$\{(\sigma(\alpha), \alpha) \mid \sigma \in \mathcal{S} \wedge \sigma(\alpha) \leq_{lex} \alpha\}. \tag{4.1}$$

Similar to the previously defined relations, this relation also meets all requirements made by the definition of a dominance relation. First of all when $\alpha \models F$, $\sigma(\alpha) \models F$ as well according to the definition of symmetries. Next, beacause σ is a strong symmetry $f(\sigma(\alpha))$ and $f(\alpha)$ are equal. So, to refine this relation, a tie-breaker has been added in the form of a lexicographic order $\leq_{lex}$.

Both relations described above could be used to statically generate breaking formulas, or to dynamically infer new information during the search process, all while trying to minimize (or maximize) the objective function value.

4.2 Dominance Breaking Formulas

First, we describe how to use dominance relations to generate *dominance breaking formulas*. A sound dominance breaking formula for dominance relation $\preceq$ is a pseudo-Boolean formula D with the property that for each assignment α that does not satisfy D there exists some assignment β with $\beta \prec \alpha$. In short, when α does not satisfy the dominance breaking formula D, this means that there exists another assignment that is *strictly* better. As a result, D only prunes assignments that are strictly dominated, and it is safe to add D to the pseudo-Boolean formula F in the sense that we are guaranteed not to prune all optimal solutions. A dominance breaking formula D is also called complete when it has the property that if there exists some assignment β with $\beta \prec \alpha$ then assignment α does not satisfy D. This entails that when an assignment is strictly dominated it will not satisfy the dominance breaking formula. We focus on sound formulas.

Next, we need to define sound dominance breaking formulas, these definitions however depend on the dominance relation that we are trying to break. We take dominance relation 4.1 as an example. The dominance breaking formulas for this relation happen to be the static symmetry breaking constraints described in the previous chapter. When an assignment α does not satisfy these constraints, this means that a permutation σ exists whose symmetric image of α is smaller according to the lexicographic relation $\leq_{lex}$ and thus $\sigma(\alpha) \prec \alpha$. As a result, these symmetry breaking constraints are dominance breaking formulas.

This kind of dominance relation is already statically broken for pseudo-Boolean optimization problems by `Shatter` (F. A. Aloul et al. 2003).

We could wonder whether combining different dominance relations and their breaking formulas might be an option. This is not trivial. The transitive closure of two dominance relations is a dominance relation once again but the dominance breaking formulas of these relations are no longer valid since these depend on the induced partial order of the original relations.

Chapter 5

Weak Symmetry

So far, in the literature, only symmetries of pseudo-Boolean optimization problems that preserve the formula F and objective function f have been considered. However, we know that in many problems, symmetries of F show up that do not preserve f. Take for instance a nurse scheduling problem where large sets of nurses are interchangeable (given that some basic features, e.g. their degrees, are equal), but their preferences are not (the soft constraints, e.g. whether they prefer working during the weekends or nightshifts). The central goal of this thesis is to develop tools that can exploit these so-called *weak* symmetries of pseudo-Boolean optimization problems. In this section, we show how weak symmetries also fit in the dominance framework, how to detect these symmetries and how the symmetry breaking constraints of `BreakID` can be adapted to handle these symmetries. Subsequently, we discuss some implementation details.

5.1 Weak Symmetry Handling

From now on, we call every symmetry ω of F a *weak symmetry* of the optimization problem (F, f). Notice that this definition does not require the symmetry ω to be a symmetry of the objective function f as well. As a result, optimal solutions are not necessarily mapped to optimal solutions, so the previously described symmetry breaking methods cannot be used. If we are able to define a dominance relation for the group of weak symmetries we can however generate dominance breaking formulas.

Provided some adjustments we can indeed define a dominance relation for a group of weak symmetries. We can use the values of the objective functions as an order, but when they are equal a tie-breaker is needed. As a result we can define the following dominance relation:

$$\{(\omega(\alpha), \alpha) \mid \omega \in \mathcal{W} \wedge f(\omega(\alpha)) \leq f(\alpha) \wedge (f(\omega(\alpha)) = f(\alpha) \Rightarrow \omega(\alpha) \leq_{lex} \alpha)\}.$$

Equivalently, this set can be defined as:

$$\{(\omega(\alpha), \alpha) \mid \omega \in \mathcal{W} \wedge f(\omega(\alpha)) < f(\alpha) \vee (f(\omega(\alpha)) = f(\alpha) \wedge \omega(\alpha) \leq_{lex} \alpha)\}.$$

Now we construct a dominance breaking formula for this dominance relation using a static symmetry breaking constraint S as described in Section 3.2. A dominance breaking formula D for this new relation has the property that for for every assignment α that does not satisfy D there exists a weak symmetry ω for which $f(\omega(\alpha)) \leq f(\alpha)$ and if $f(\omega(\alpha)) = f(\alpha)$ this implies that α does not satisfy the symmetry breaking constraint S. In other words, such a dominance breaking formula will discard candidate solutions whose symmetric variant has a strictly better

solution (according to the objective function), as well as candidate solutions whose symmetric variants have an equally good objective function value, but that are lexicographically larger than their symmetric counterpart.

Let ω be a weak symmetry of the pseudo-Boolean optimization problem with formula F and objective function f, $\omega(f)$ denotes the symmetric image of the objective function of an assignment and LL_ω the set of Lex-Leader constraints for ω. We define the dominance breaking formula for $(\omega(\alpha), \alpha)$ by the following constraints:

$$f \leq \omega(f)$$
$$(f = \omega(f)) \Rightarrow LL_\omega$$

The first constraint discards candidate solutions whose symmetric variant has a strictly better solution, the second constraint discards candidate solutions whose symmetric variants have an equally good objective function value, but that are lexicographically larger than their symmetric counterpart. In Section 6.2 we discuss in detail how these constraints are normalised to the pseudo-Boolean formulas used in `BreakID`.

5.2 Weak Symmetry Detection

To detect strong symmetries, i.e. symmetries of the objective function and the constraints, it is necessary to add the objective function to the graph representing the syntax tree of the formula. This ensures that symmetries of the objective function of and formula $pbform$ are detected. In order to detect weak symmetries, there is no need to add the objective function to the graph. This ensures that all symmetries of the formula are found, disregarding whether they are also symmetries of the objective function. The graph generated to detect weak symmetries for a given pseudo-Boolean optimization problem is smaller than the graph generated to detect strong symmetries since it does not feature the objective function.

Chapter 6

Pseudo-Boolean Symmetry Breaking in BreakID

In this section we discuss the details of the implementation of pseudo-Boolean symmetry breaking for `BreakID`. The existing version of `BreakID` was able to break symmetries for pseudo-Boolean problems without an optimization function. This version of `BreakID` was extended to break symmetries for pseudo-Boolean problems *with* an optimization function, using either strong or weak symmetry breaking. Both options are compatible with the existing optimizations that `BreakID` introduced in comparison to `Shatter`. For now, `BreakID` can only guarantee correct results for problems with coefficients whose sum does not surpass 2^{20}. This constraint is valid for the constraints and the objective function of the given problem.

First, we discuss how symmetry detection is done in `BreakID`, next we explain how `BreakID` breaks weak and strong symmetries for pseudo-Boolean optimization functions and lastly we discuss the compatibility of these extensions with the existing optimizations.

6.1 Symmetry Detection

Similar to `Shatter`, symmetry detection in `BreakID` is done by making a graph that represents a simplified version of the syntax tree of the problem. The automorphisms of this graph are then detected by `Saucy 3.0` and represent the symmetries of the problem. The graph generated by `BreakID` differs slightly from the graph described in Section 3.1. `BreakID` does not make the distinction between CNF-clauses and pseudo-Boolean constraints and the terms of a constraint are added differently:

- Terms in a pseudo-Boolean constraint P_i with constraint node Y_i are organized as follows:
  - If the coefficient equals 1, connect the constraint node Y_i to the literal node.
  - Else:
    - Add a node $X_{c,l}$ representing a term with coefficient c and literal l if it does not exist yet. Connect the node to the literal node and the constraint nodeY_i.
    - A different colour is used for each distinct coefficient value encountered.

If the graph needs to represent the objective function as well, the objective function f can be added as follows:

- The objective function f is represented by a vertex Z with a unique colour.
- Terms in the objective function f are organized as follows:
  - If the coefficient equals 1, connect the objective function node Z to the literal node.
  - Else:
    - Add a node $X_{c,l}$ representing a term with coefficient c and literal l if it does not exist yet. Connect the node to the literal node and the objective function node Z.
    - A different colour is used for each distinct coefficient value encountered.

This way of generating the graph ensures a one-to-one correspondence between the automorphism groups of the graph and the syntactic symmetries of the given problem. This graph is slightly smaller than the graph generated by the method described in Section 3.1.

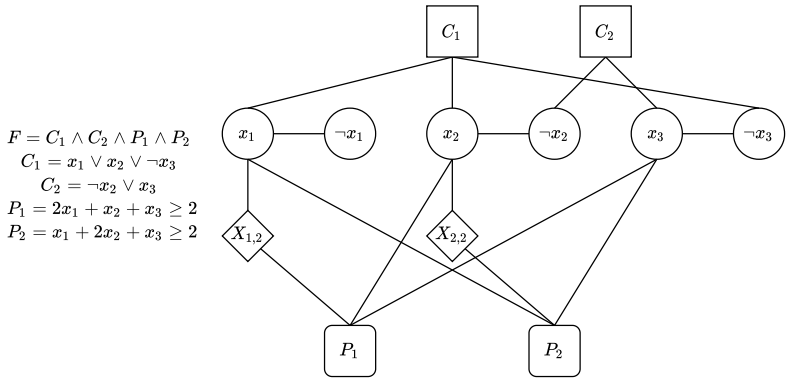
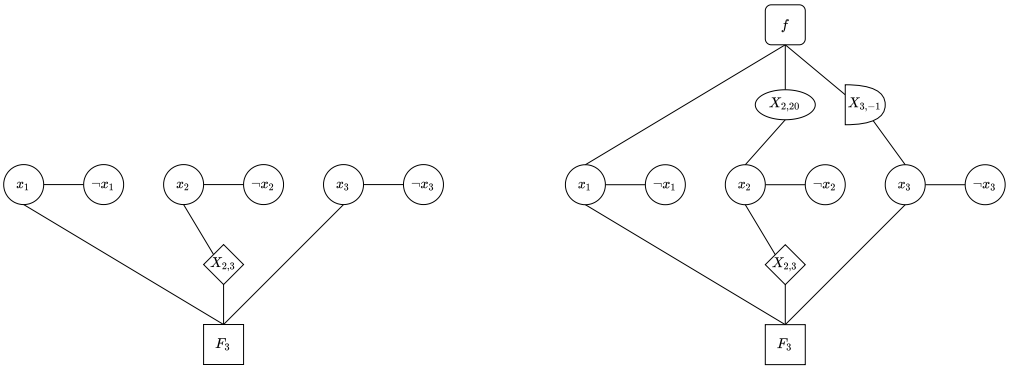


Figure 6.1: Example of a formula F and the constructed graph. The different vertex shapes represent different colours.



(a) Weak symmetry detection

(b) Strong symmetry detection

Figure 6.2: The generated graphs for a pseudo-Boolean optimization problem with a single constraint $F = 1x_1 + 3x_2 + 1x_3 \geq 3$ and optimization function $f = 1x_1 + 20x_2 - 3x_3$. The different vertex shapes represent different colours.

6.2 Symmetry Breaking

To implement symmetry breaking for pseudo-Boolean problems with an optimization function a distinction needs to be made between weak and strong symmetry breaking. In Section 4.2 we described that strong symmetries can be broken by generating symmetry breaking constraints. `BreakID` was already able to generate these constraints. The generated symmetry breaking constraints are a pseudo-Boolean encoding of the lex-leader constraints. To break weak symmetries `BreakID` needs to generate the dominance breaking formulas described in Section 5.1, but these need to be normalized to pseudo-Boolean formulas first.

First, we introduce an auxiliary y_0 that is true if and only if the objective function value of the formula and its symmetric counterpart are equal. This new variable then initiates the symmetry breaking done by LL_ω.

$$\begin{aligned} &f \leq \omega(f) \\ &y_0 \Leftrightarrow \omega(f) = f \\ &y_0 \Rightarrow LL_\omega \end{aligned}$$

The first dominance breaking constraint, $f \leq \omega(f)$, can then be normalized to the following constraint: $\omega(f) - f \geq 0$. Since the first constraint already requires f to be smaller than $\omega(f)$ we can weaken the equality in the second constraint, $y_0 \Leftrightarrow \omega(f) = f$, to $y_0 \Leftrightarrow f \geq \omega(f)$. The equivalence is then split in two implications: $f - \omega(f) \geq 0 \Rightarrow y_0$ and $y_0 \Rightarrow f - \omega(f) \geq 0$.

To normalize the first implication, $f - \omega(f) \geq 0 \Rightarrow y_0$, its contraposition $\overline{y_0} \Rightarrow \omega(f) - f \geq 1$ is transformed into the pseudo-Boolean constraint $cy_0 + \omega(f) - f \geq 1, c \in \mathbb{Z}$. The coefficient c of literal y_0 needs to be sufficiently large to ensure that the constraint is always true when y_0 is true and that when y_0 is false, the constraint amounts to $\omega(f) - f \geq 1$. To find the right coefficient for y_0 it suffices to take the sum of the absolute values of all coefficients of $\omega(f) - f$ and divide this sum by two.

The second implication $y_0 \Rightarrow f - \omega(f) \geq 0$ is transformed to the pseudo-Boolean constraint $c\overline{y_0} + f - \omega(f) \geq 0, c \in \mathbb{Z}$. When y_0 is false and the coefficient is sufficiently large this constraint is always true, when y_0 is true this constraint is equivalent to solving $f - \omega(f) \geq 0$. The coefficient for y_0 in this constraint equals the coefficient of y_0 in the former constraint. This constraint can be normalized by replacing $\overline{y_0}$ by $(1 - y_0)$ yielding $-cy_0 + f - \omega(f) \geq -c$. This last constraint is only added when the user chooses to add this constraint, it is not necessary for correctness but it can speed up the search. By relaxing the equation of the equivalence to an inenquality the search space grows and this is the opposite of what we are trying to achieve by adding breaking formulas. This effect can be counteracted by adding this last constraint which makes the search space shrink again, we call this option *full translation*.

After normalizing, the dominance breaking formulas generated by `BreakID` are:

$$\begin{aligned} &\omega(f) - f \geq 0 \\ &cy_0 + \omega(f) - f \geq 1 \quad c \in \mathbb{Z} \\ &y_0 \Rightarrow LL_\omega \end{aligned}$$

If a full translation is used the generated dominance breaking formulas are:

$$\begin{aligned} &\omega(f) - f \geq 0 \\ &cy_0 + \omega(f) - f \geq 1 \qquad c \in \mathbb{Z} \\ &-cy_0 + f - \omega(f) \geq -c \quad c \in \mathbb{Z} \\ &y_0 \Rightarrow LL_\omega \end{aligned}$$

6.3 Compatibility With Previous Optimizations

Compared to `Shatter`, `BreakID` (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016) introduced three optimizations for static symmetry breaking for SAT problems. The three introduced optimizations are a compact encoding of the Lex-Leader constraints, the exploitation of row interchangeability and the generation of binary symmetry breaking clauses.

The first optimization is a compact encoding of Lex-Leader constraints. A more compact encoding of Lex-Leader constraints reduces the overhead introduced by adding the constraints to the problem. This more compact encoding is exactly the encoding used for LL_ω in the dominance breaking formulas generated by `BreakID`. The encoding is defined as follows: let ω be a weak symmetry of the pseudo-Boolean optimization problem. Let $Supp(\omega) = \{x_1, \ldots, x_n\}$ be ordered such that $x_i \leq_{lex} x_j$ if and only if $i \leq j$ and let $\{y_0, \ldots, y_{n-1}\}$ be a set of auxiliary variables distinct from $Supp(\omega)$. Using `BreakID`'s compact encoding of Lex-Leader constraints (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016) we define LL_ω by the following constraints:

$$
\begin{array}{ll}
y_0 & \\
\neg y_{i-1} \vee \neg x_i \vee \omega(x_i) & \text{for } 1 \leq i \leq n \\
y_j \vee \neg y_{j-1} \vee \neg x_j & \text{for } 1 \leq j < n \\
y_j \vee \neg y_{j-1} \vee \omega(x_j) & \text{for } 1 \leq j < n
\end{array}
$$

The size of the encoding can be further reduced by limiting the number of auxiliary variables y_j that are introduced. The Lex-Leader constraints can also be encoded in a *clausal* encoding. This is an encoding of the constraints with coefficients of an absolute value of maximum one.

The second optimization is the exploitation of row interchangeability. This is a type of symmetry present when a subset of variables can be structured as a two-dimensional matrix where each permutation of the rows induces a symmetry. If such a structure of symmetries can be found, the group of symmetries that forms this structure can be broken as a whole. This can also be applied to pseudo-Boolean optimization problems since the implementation detects this kind of symmetries based on symmetries detected in the generated graph. The detected row-symmetries are then broken using the implemented symmetry breaking constraints. For problems where there are weak symmetries present in the structure, dominance breaking formulas are used instead of symmetry breaking constraints to break those symmetries.

Lastly, it is known that posting lex-leader breaking constraints for all detected symmetries in a symmetry group can be infeasible. Hence, often lex-leader constraints are only added for the generators of the symmetry group. `BreakID` introduced an alternative; the generation of binary symmetry breaking clauses. These are short symmetry breaking clauses, equivalent to posting symmetry breaking constraints in the compact encoding with $j = 1$. These very short clauses are then posted for a large set of symmetries in the symmetry group. Since we know that the effect of the first constraints added is higher, this is a cost-effective way to break symmetries. For pseudo-Boolean optimization problems, this option is available as well. The decision was made however to only generate binary breaking clauses for strong symmetries of pseudo-Boolean problems. Since generating binary breaking clauses is done to prevent the overhead of posting many long clauses, adding the longer dominance breaking constraints for each weak symmetry used defeats the purpose.

Chapter 7

Research Questions and Experiments

Our experiments aim to answer three research questions. First, we take a closer look at the impact of symmetry breaking on pseudo-Boolean problems. Next, we find out whether the three optimizations that `BreakID` introduced compared to `Shatter` are also relevant for pseudo-Boolean symmetry breaking. Lastly, we study the effects of exploiting weak symmetries, which is the central goal of this thesis.

7.1 Questions

The Relevance of Static Symmetry Breaking for Modern Pseudo-Boolean Solvers

First, we research the relevance of symmetry breaking for modern pseudo-Boolean solvers such as `RoundingSAT` (Elffers and Nordström 2018). These solvers often use a different search method than SAT solvers since their search method can be based on the so-called *cutting-plane proof-system* from *0–1 integer linear programming*. It is known that for certain problems the *resolution proof-system* used by SAT-solvers generates proofs with exponential lower bounds. For the *pigeon hole problem* such an exponential lower bound was proven by Haken (1985). Cook et al. (1987) have shown that the cutting-plane proof-system can be exponentially stronger and thus is able to generate short proofs for problems such as the pigeon hole problem. As a result, for the prototypical problem on which symmetry breaking is indispensable for SAT, we know that in theory, cutting-planes–based solvers could find short refutations. This does not mean of course that they will find them in practice, and it does not tell us whether or not symmetry breaking is crucial for pseudo-Boolean solvers for other types of problems. This raises the question whether symmetry breaking is as relevant for pseudo-Boolean solving as it is for SAT.

The Relevance of `BreakID`'s Optimizations for Pseudo-Boolean Symmetry Breaking

Section 6.3 gives an overview of the optimizations for static symmetry breaking for SAT that `BreakID` introduced compared to `Shatter`. These optimizations are all extended to pseudo-

Boolean symmetry breaking as well. We would like to find out whether these optimizations also enhance the performance for pseudo-Boolean symmetry breaking.

The Effect of Weak Symmetry Breaking

As described in Section 4.2, the impact of breaking strong symmetries has been researched already. We would like to study the impact of breaking weak symmetries as well by running experiments using `BreakID`'s implementation of weak symmetry breaking as described in Section 6.

7.2 Experiments

We will perform several experiments to help us answer the research questions. We have chosen to use the benchmark sets of the *PB 16 competition* (**pbcompetition pbcompetition**) for our experiments. In this competition, a distinction is made to allow solvers to participate without having to implement arbitrary precision. The Pseudo-Boolean Competition classifies problems as fixed–precision problems when the constraints are guaranteed to have a sum of coefficients lower than 2^{20}. We only implemented the techniques described in this paper for fixed–precision problems, but the techniques apply in general as well. As a result, we only selected the fixed–precision benchmarking sets from the PB 16 Competition. The Pseudo-Boolean Competition consists of decision and optimization problems. Because of timing issues, this set of experiments only consists of the optimization problems. We decided to only use the optimization problems because weak symmetry breaking does not apply to decision problems. The experiments will be repeated with sets of decision problems to verify whether the results concerning the first two research questions apply to such problems as well. The Pseudo-Boolean Competition also contains non-linear problems, i.e., problems where the constraints consist of the sum of an integer and a product of literals and weighted pseudo-Boolean problems, i.e., problems where soft constraints may be violated. BreakID is not equipped to break symmetries of such problems so the benchmarking sets containing these problems were not used for the experiments.

As a result we ended up using seven benchmark sets for the experiments:

- **Opt06**: The linear fixed–precision pseudo-boolean optimization problems of the year 2006. (954 instances)
- **Opt07**: The linear fixed–precision pseudo-boolean optimization problems of the year 2007. (85 instances)
- **Opt09**: The linear fixed–precision pseudo-boolean optimization problems of the year 2009. (29 instances)
- **Opt10**: The linear fixed–precision pseudo-boolean optimization problems of the year 2010. (204 instances)
- **Opt11**: The linear fixed–precision pseudo-boolean optimization problems of the year 2011. (52 instances)
- **Opt12**: The linear fixed–precision pseudo-boolean optimization problems of the year 2012. (80 instances)
- **Opt15**: The linear fixed–precision pseudo-boolean optimization problems of the year 2015. (196 instances)

These sets are all the linear fixed–precision optimization problems used in the *PB 16 Competition* (**pbcompetition pbcompetition**), this is a total of 1600 problems .

Solving the instances is done by `RoundingSAT` (Elffers and Nordström 2018) with the default configuration. The resources available to solve an instance were 16GB of memory and 3600s on an Intel(R) Xeon(R) Gold 6148 CPU. The symmetry breaking is done by the extended version of `BreakID` which is bundled with `Saucy 3.0` (Katebi et al. 2010) for symmetry detection. The resources available to break symmetries for an instance were 16GB of memory and 1800s on an Intel(R) Xeon(R) Gold 6148 CPU. The operating system was CentOS 7 with Linux kernel 3.10.

The Relevance of Static Symmetry Breaking for Modern Pseudo-Boolean Solvers

To measure the relevance of static symmetry breaking for modern pseudo-Boolean solvers we solved the benchmark sets in two different configurations. First, we solved the instances without any preprocessing, this configuration is called *NoSymm*. Next, we solved the instances after adding strong symmetry breaking clauses generated by `BreakID`, we call this configuration *StrongSymm-NoOpt*. All breaking constraints are clausally encoded (Section 6.3) and fully translated (Section 6.2) unless stated otherwise. An instance is called *solved* if a correct solution is found within the timelimit set for solving an instance (i.e. 3600s).

Table A.1 shows us the number of solved instances per benchmarking set. It first contains the results for all instances per benchmarking set. Next, it repeats the results for the subset of problems for which `BreakID` detected strong symmetries. The total amount of instances per set is added between brackets and in bold the configuration that solved most instances is marked, this number gives us a way to compare performance. The preprocessing done by `BreakID` seems to have had almost no impact on the total number of instances solved.

Figures A.1 and A.2 show us the total runtime in function of the number of instances solved. The total runtime is the sum of the runtimes of `BreakID` (if any preprocessing was done) and `RoundingSAT`. The horizontal red line marks the time limit of the Pseudo-Boolean Competition, this limit is set at 2600s. We can see that for all instances the preprocessing had no impact and that the total runtimes for solving the instances are comparable.

We can conclude that static symmetry breaking is not as relevant for pseudo-Boolean solving as it is for SAT. We need still need to repeat the experiments on benchmark sets of decision problems to verify these results.

The Relevance of `BreakID`'s Optimizations for Pseudo-Boolean Symmetry Breaking

To find out whether `BreakID`'s optimizations are relevant for pseudo-Boolean symmetry breaking, we compare the performance of *StrongSymm-NoOpt* to solving instances with added strong symmetry breaking constraints enhanced with the three optimizations described in Section 6.3, we call this configuration *StrongSymm-Opt*.

Table A.2 contains the number of solved instances per benchmarking set for the *StrongSymm-NoOpt* and *StrongSymm-Opt* configurations. We see that the configuration without optimizations performs better (i.e. it solves more instances). This trend is visible for almost all benchmarking sets as well as for the subsets of symmetric instances.

When comparing the runtimes in figures A.3 and A.4 we observe that the *StrongSymm-Opt* configuration's total runtime peaks higher. Table A.3 contains the mean runtime in seconds for `RoundingSAT`, `BreakID` and the sum of these runtimes for the solved instances of each individual benchmark set and for all benchmark sets combined. In bold the smallest mean runtime for each

set is marked. We also added the same information for the subsets problems for which `BreakID` detected symmetries. We notice that the optimizations have a positive impact on the mean runtime of `RoundingSAT` but that `BreakID` has a higher a mean runtime. The overall impact of the optmizations on the mean total runtime of the solved instances seems to vary.

Even though `RoundingSAT` is able to solve problems faster there are fewer problems for which it finds solutions in a timely manner. As a result, the relevance of `BreakID`'s optimizations for pseudo-Boolean solving remains to be shown. Extra experiments are needed to verify whether finetuning the optimizations (i.e. finding out which optimizations have a positive impact and which do not) influences these results. We also need to verify these results for decision problems.

The Relevance of Weak Symmetry Breaking

To verify whether weak symmetry breaking is relevant, we solved instances with added weak symmetry breaking constraints. These constraints are not in a clausal encoding but they use a limited number of auxiliary variables (i.e. $j = 28$, as explained in Section 6.3). This configuration is called *WeakSymm-Long-NoOpt*. We then compare this configuration to the *StrongSymm-Opt* and *StrongSymm-Noopt* configurations.

Table A.4 contains an overview of the number of solved instances per configuration per benchmarking set of optimization problems. We also added the results for the subsets of instances for which `BreakID` detected strong symmetries and for the instances for which `BreakID` detected a different number of strong and weak symmetries. We see that *WeakSymm-Long-NoOpt* cannot outperform the non-optimized strong symmetry breaking configuration.

Table A.5 contains the same data as Table A.3 but for the *StrongSymm-NoOpt* and *WeakSymm-Long-NoOpt* configurations. We can see that *StrongSymm-NoOpt* outperforms *WeakSymm-Long-NoOpt* most of the time. As a result, this implementation of weak symmetry does not seem to be very relevant. We notice however that a big speed-up in solving time is caused by weak symmetry breaking for the weakly symmetric instances of *Opt11* and *Opt15*. It is worth studying these exact instances (46 instances) to see why such a high speedup is caused for these exact instances and not for the other problems. More experiments need to be done to confirm these results. If these results are correct weak symmetry breaking might still have potential.

Weak Symmetry Breaking and `BreakID`'s Optimizations

We also ran the experiments for two extra configurations; *WeakSymm-Short-Opt* and *WeakSymm-Long-Opt*. These respectively are the optimized weak symmetry breaking configuration in clausal and non-clausal encoding (i.e. using a limited number of auxiliary variables, $j = 28$, as explained in Section 6.3). The experiments showed that *WeakSymm-Short-Opt* returned one wrong solution (while solving 954 instances in total) and that *WeakSymm-Long-Opt* returned one wrong solution (while solving 940 instances in total). In both cases the returned solution was an optimum that was too high. Since the optimizations and weak symmetry breaking are theoretically compatible, this implies that there is a bug in the implementation. We have decided to still share the results since all other solutions were correct, these results are however not reliable. We first filtered out the two erroneously solved instances since these were not solved correctly.

Table A.6 shows the number of solved instances for the *StrongSymm-Opt*, *WeakSymm-Short-Opt* and *WeakSymm-Long-Opt* configurations. Upon examination, we see that the flawed configuration *WeakSymm-Short-Opt* was able to solve more instances than *StrongSymm-Opt*. When comparing to Table A.1 we see that *NoSymm* still solves more instances in total. The optimized configurations also do not outperform *WeakSymm-Long-NoOpt* upon comparing these results

with the data in table A.4. As a result, the faulty implementation of optimized weak symmetry breaking does not seem relevant for pseudo-Boolean solving.

7.3 Summary

The results are quite unexpected and dissapointing. Although static symmetry breaking has been known to improve performance for SAT solving, its relevance for pseudo-Boolean solving remains to be confirmed. We can see that the preprocessing done by `BreakID` influences `RoundingSAT`'s solving time, but not always in a positive way. The optimizations that `BreakID` introduced for SAT do not enhance performance for static symmetry breaking for pseudo-Boolean solving. This effect is the same for weak and strong symmetry breaking. More experiments need to be done to see whether finetuning the optimizations has any effect on the results. We also need to repeat the experiments on benchmarking sets of decision problems to verify whether these results only apply to optimization problems or whether they apply to decision problems as well. It is also worth noting that we only tested `BreakID` with one particular configuration of one solver, it should be verified whether the results also apply to the other configurations of `RoundingSAT` as well as to other solvers since these might use a different search method.

Chapter 8

Conclusion

In this paper, we have studied static symmetry breaking for pseudo-Boolean optimization problems. To be able to statically break symmetries for such problems we have defined a new notion of symmetry: *weak symmetry*. In short, a weak symmetry is a symmetry of the formula of an optimization problem that does not necessarily respect the objective function of that problem. We have also developed the theory for weak symmetry handling as well as a tool that performs static symmetry breaking for strong and weak symmetries. When experimentally validating this implementation on the latest Pseudo-Boolean Competition; the results were unexpected and disappointing. Even though many of the benchmarks exhibit symmetry, in general, the used solver seemed to slow down in case symmetry is broken. At this point, we have no clear hypothesis as to why this is the case. There are several directions for future work:

- The experiments uncovered a bug in the implementation of optimized weak symmetry breaking. This should be investigated, for instance by making use of recent proof logging techniques for pseudo-Boolean optimization problems (**bogaerts2022certified bogaerts2022certified**).
- While we tested the effect of symmetry breaking on the performance of the default configuration of `RoundingSAT`, we did not test it on other configurations (some of which use fundamentally different search techniques), or other solvers. The results should be verified for other solvers and other configurations of `RoundingSAT` as well.
- Additionally, we would like to run experiments on a larger set of benchmarks, for instance, those used in recent papers on pseudo-Boolean optimization (**CuttingToTheCore-DevriendtNordstrom CuttingToTheCore-DevriendtNordstrom**).

Acknowledgements

The resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government.

Bibliography

Aloul, F. A. et al. (Oct. 2003). "Symmetry-Breaking for Pseudo-Boolean Formulas". In: *International Workshop on Symmetry on Constraint Satisfaction Problems*.

— (2002). "Solving difficult SAT instances in the presence of symmetry". In: *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pp. 731–736. DOI: 10.1109/DAC.2002.1012719.

Aloul, F.A., K.A. Sakallah, and I.L. Markov (2006). "Efficient symmetry breaking for Boolean satisfiability". In: *IEEE Transactions on Computers* 55.5, pp. 549–558. DOI: 10.1109/TC.2006.75.

Benhamou, B. and T. Nabhani (2010). "Dynamic symmetry breaking in the satisfiability problem". In:

Benhamou, B., T. Nabhani, et al. (Nov. 2010). "Enhancing Clause Learning by Symmetry in SAT Solvers". In: vol. 1, pp. 329–335. DOI: 10.1109/ICTAI.2010.55.

Biere, A. et al. (Oct. 1999). "Symbolic Model Checking without BDDs". In: vol. 4144. ISBN: 978-3-540-65703-3. DOI: 10.1007/3-540-49059-0_14.

Bruynooghe, M. (1981). "Solving Combinatorial Search Problems by Intelligent Backtracking". In: *Inf. Process. Lett.* 12, pp. 36–39.

Chu, G. and P. Stuckey (Apr. 2015). "Dominance breaking constraints". In: *Constraints* 20. DOI: 10.1007/s10601-014-9173-7.

Cook, W., C.R. Coullard, and Gy. Turán (1987). "On the complexity of cutting-plane proofs". In: *Discrete Applied Mathematics* 18.1, pp. 25–38. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(87)90039-4. URL: https://www.sciencedirect.com/science/article/pii/0166218X87900394.

Crawford, J. et al. (Mar. 1997). "Symmetry-Breaking Predicates for Search Problems". In:

Davis, M., G. Logemann, and D. Loveland (July 1962). "A Machine Program for Theorem-Proving". In: *Commun. ACM* 5.7, pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

Devriendt, J., B. Bogaerts, and M. Bruynooghe (Aug. 2017). "Symmetric Explanation Learning: Effective Dynamic Symmetry Handling for SAT". In: ISBN: 978-3-319-66262-6. DOI: 10.1007/978-3-319-66263-3_6.

Devriendt, J., B. Bogaerts, M. Bruynooghe, and M. Denecker (July 2016). "Improved Static Symmetry Breaking for SAT". In: pp. 104–122. ISBN: 978-3-319-40969-6. DOI: 10.1007/978-3-319-40970-2_8.

Devriendt, J., B. Bogaerts, B. De Cat, et al. (2012a). "Symmetry Propagation: Improved Dynamic Symmetry Breaking in SAT". In: *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*. Vol. 1, pp. 49–56. DOI: 10.1109/ICTAI.2012.16.

— (2012b). "Symmetry Propagation: Improved Dynamic Symmetry Breaking in SAT". In: *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*. Vol. 1, pp. 49–56. DOI: 10.1109/ICTAI.2012.16.

Elffers, J. and J. Nordström (July 2018). "Divide and Conquer: Towards Faster Pseudo-Boolean Solving". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, pp. 1291–1299. DOI: 10.24963/ijcai.2018/180. URL: https://doi.org/10.24963/ijcai.2018/180.

Haken, A. (1985). "The intractability of resolution". In: *Theoretical Computer Science* 39. Third Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 297–308. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(85)90144-6. URL: https://www.sciencedirect.com/science/article/pii/0304397585901446.

Heule, M. et al. (Jan. 2005). "CNF Symmetry Breaking Options in Conflict Driven SAT Solving". In:

Junttila, T. and P. Kaski (2007). "Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs". In: *Proceedings of the Meeting on Algorithm Engineering and Expermiments*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, pp. 135–149.

Katebi, H., K.A. Sakallah, and I.L. Markov (June 2010). "Symmetry and Satisfiability: An Update". In: pp. 113–127. ISBN: 978-3-642-14185-0. DOI: 10.1007/978-3-642-14186-7_11.

Kautz, H. and B Selman (Jan. 1992). "Planning as Satisfiability." In: pp. 359–363.

Marques-Silva, J.P. and K.A. Sakallah (1999). "GRASP: a search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5, pp. 506–521. DOI: 10.1109/12.769433.

Massacci, F. and L. Marraro (2000). "Logical cryptanalysis as a SAT-problem: Encoding and analysis". In: *In Journal of Automated Reasoning* 24, pp. 165–203.

McKay, B. D. and A. Piperno (2014). "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60, pp. 94–112. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2013.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0747717113001193.

Metin, H. et al. (Apr. 2018). "CDCLSym: Introducing Effective Symmetry Breaking in SAT Solving". In: *Tools and Algorithms for the Construction and Analysis of Systems – TACAS*. Tessaloniki, Greece. URL: https://hal.sorbonne-universite.fr/hal-01766948.

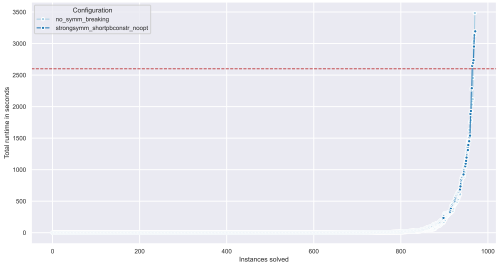
Appendix A

Experiments

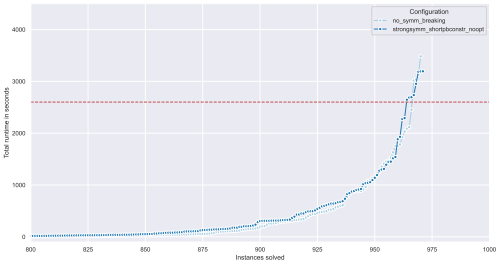
Table A.1: Number of solved instances for the *NoSymm* and *StrongSymm-NoOpt* configurations for the selected benchmark sets (*all instances*) and the subsets of instances for which `BreakID` detected strong symmetries (*symmetric instances*).

	NoSymm	StrongSymm-NoOpt
All Instances (1600)	971	**972**
Opt06 (954)	**563**	561
Opt07 (85)	66	**69**
Opt09 (29)	24	**25**
Opt10 (204)	**165**	**165**
Opt11 (52)	**22**	**22**
Opt12 (80)	**44**	41
Opt15 (196)	87	**89**
Symmetric Instances (1233)	747	**748**
Opt06 (722)	**438**	436
Opt07 (81)	62	**65**
Opt09 (9)	4	**5**
Opt10 (199)	**160**	**160**
Opt11 (39)	**15**	**15**
Opt12 (80)	**44**	41
Opt15 (103)	24	**26**

Figure A.1: Total runtime for the *NoSymm* and *StrongSymm-NoOpt* configurations in function of the total number of solved optimization problems.

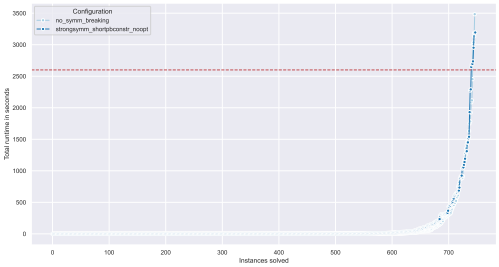


(a) Total runtime in function of the number of solved optimization problems, all instances.

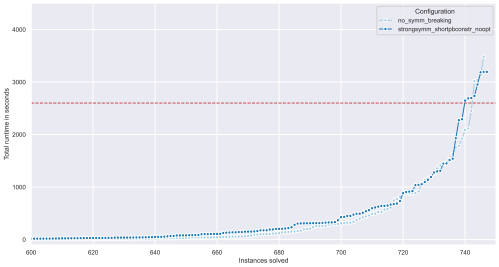


(b) Total runtime in function of the number of solved optimization problems, first 800 solved instances skipped.

Figure A.2: Total runtime for the *NoSymm* and *StrongSymm-NoOpt* configurations in function of the total number of solved symmetric opimization problems.

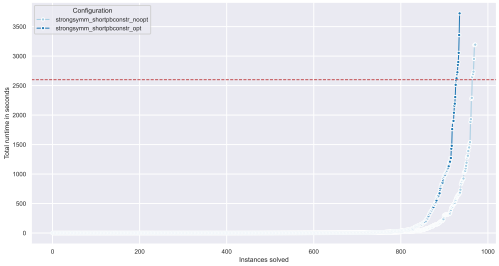


(a) Total runtime in function of the number of solved opimization problems, all instances.

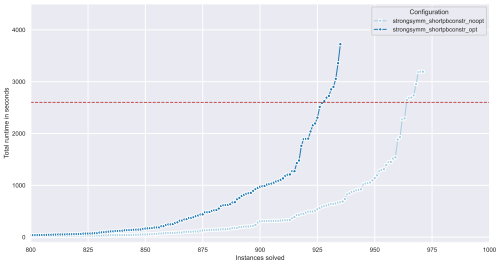


(b) Total runtime in function of the number of solved optimization problems, first 650 solved instances skipped.

Figure A.3: Total runtime for the *StrongSymm-NoOpt* and *StrongSymm-Opt* configurations in function of the number of solved optimization problems.



(a) Total runtime in function of the number of solved optimization problems, all instances.

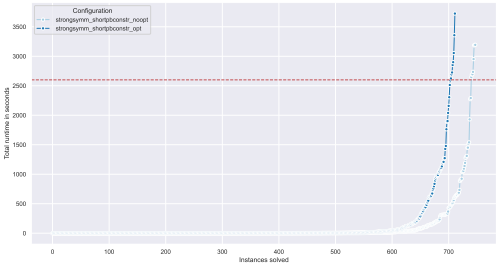


(b) Total runtime in function of the number of solved optimization problems, first 800 solved instances skipped.

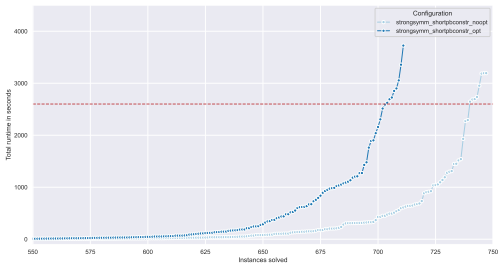
Table A.2: Number of solved instances for the *StrongSymm-NoOpt* and *StrongSymm-Opt* configurations for the selected benchmark sets (*all instances*) and the subsets of instances for which `BreakID` detected strong symmetries (*symmetric instances*).

	StrongSymm-NoOpt	StrongSymm-Opt
All Instances (1600)	**972**	936
Opt06 (954)	**561**	547
Opt07 (85)	**69**	68
Opt09 (29)	25	**26**
Opt10 (204)	**165**	**165**
Opt11 (52)	**22**	20
Opt12 (80)	**41**	20
Opt15 (196)	89	**90**
Symmetric Instances (1233)	**748**	712
Opt06 (722)	**436**	422
Opt07 (81)	**65**	64
Opt09 (9)	5	**6**
Opt10 (199)	**160**	**160**
Opt11 (39)	**15**	13
Opt12 (80)	**41**	20
Opt15 (103)	26	**27**

Figure A.4: Total runtime for the *StrongSymm-NoOpt* and *StrongSymm-Opt* configurations in function of the number of solved symmetric opimization problems.



(a) Total runtime in function of the number of solved opimization problems, all instances.



(b) Total runtime in function of the number of solved optimization problems, first 650 solved instances skipped.

Table A.3: Mean runtimes in seconds for the *StrongSymm-NoOpt* and *StrongSymm-Opt* configurations for the selected benchmark sets (*all instances*) and the subsets of instances for which `BreakID` detected strong symmetries (*symmetric instances*).

	StrongSymm-NoOpt			StrongSymm-Opt		
	Avg. Time BreakID	Avg. Time RoundingSAT	Avg. Total Time	Avg. Time BreakID	Avg. Time RoundingSAT	Avg. Total Time
All Instances (1600)	**10.966**	79.733	**85.748**	57.852	**74.595**	102.117
Opt06 (954)	**12.297**	**53.245**	**55.370**	66.079	59.189	82.400
Opt07 (85)	**0.353**	**123.022**	**123.302**	0.951	150.299	150.789
Opt09 (29)	**0.403**	**142.178**	**142.507**	100.600	177.694	253.859
Opt10 (204)	**1.036**	**12.939**	**13.747**	1.138	13.001	13.938
Opt11 (52)	15.784	262.081	271.431	**12.895**	**237.292**	**238.258**
Opt12 (80)	**55.576**	481.707	**585.804**	357.870	**363.087**	904.449
Opt15 (196)	**1.498**	**89.171**	**89.396**	23.343	93.903	94.449
Symmetric Instances (1233)	**14.582**	90.788	**98.574**	93.380	**83.617**	119.775
Opt06 (722)	**17.829**	65.543	65.543	128.497	**53.733**	**56.456**
Opt07 (81)	**0.406**	**117.077**	**117.077**	1.104	129.764	130.059
Opt09 (9)	**0.651**	**11.353**	**11.353**	323.563	550.303	550.779
Opt10 (199)	**1.058**	17.690	17.690	1.161	**13.338**	**14.165**
Opt11 (39)	19.428	**86.934**	**86.934**	**15.759**	233.896	247.557
Opt12 (80)	**55.576**	**353.138**	**353.138**	357.870	481.707	585.804
Opt15 (103)	**2.625**	**257.256**	**257.256**	46.194	303.953	304.335

Table A.4: Number of solved instances for the *StrongSymm-NoOpt*, *StrongSymm-Opt* and *WeakSymm-Long-NoOpt* configurations for the selected benchmark sets (*all instances*), the subsets of instances for which `BreakID` detected strong symmetries (*symmetric instances*) and the subsets of instances for which `BreakID` detected a different number of strong and weak symmetries (*weakly symmetric instances*).

	StrongSymm-NoOpt	StrongSymm-Opt	WeakSymm-Long-NoOpt
All Instances (1600)	**972**	936	970
Opt06 (954)	**561**	547	559
Opt07 (85)	**69**	68	**69**
Opt09 (29)	25	**26**	25
Opt10 (204)	**165**	**165**	**165**
Opt11 (52)	**22**	20	21
Opt12 (80)	41	20	**43**
Opt15 (196)	89	**90**	88
Symmetric Instances (1233)	**748**	712	746
Opt06 (722)	**436**	422	434
Opt07 (81)	**65**	64	**65**
Opt09 (9)	5	**6**	5
Opt10 (199)	**160**	**160**	**160**
Opt11 (39)	**15**	13	14
Opt12 (80)	41	20	**43**
Opt15 (103)	26	**27**	25
Weakly Symmetric Instances (571)	**378**	355	**378**
Opt06 (390)	**272**	266	**272**
Opt07 (16)	**10**	**10**	**10**
Opt10 (46)	**46**	**46**	**46**
Opt11 (6)	**4**	2	**4**
Opt12 (73)	**36**	17	22
Opt15 (40)	10	**14**	11

Table A.5: Mean runtimes in seconds for the *StrongSymm-NoOpt* and *WeakSymm-Long-NoOpt* configurations for the selected benchmark sets (*all instances*), the subsets of instances for which `BreakID` detected strong symmetries (*symmetric instances*) and the subsets of instances for which `BreakID` detected a different number of strong and weak symmetries (*weakly symmetric instances*).

	StrongSymm-NoOpt			WeakSymm-Long-NoOpt		
	Avg. Time BreakID	Avg. Time RoundingSAT	Avg. Total Time	Avg. Time BreakID	Avg. Time RoundingSAT	Avg. Total Time
All Instances (1600)	**10.966**	**79.733**	**85.748**	23.175	84.130	94.637
Opt06 (954)	**12.297**	53.245	**55.370**	32.082	**47.707**	57.179
Opt07 (85)	**0.353**	**123.022**	**123.302**	0.470	169.530	169.840
Opt09 (29)	**0.403**	**142.178**	**142.507**	0.480	157.390	157.763
Opt10 (204)	**1.036**	**12.939**	**13.747**	1.031	14.471	15.279
Opt11 (52)	**15.784**	262.081	271.431	24.897	**228.694**	**238.338**
Opt12 (80)	**55.576**	**481.707**	**585.804**	59.664	625.142	730.136
Opt15 (196)	**1.498**	89.171	89.396	1.585	**59.478**	**59.655**
Symmetric Instances (1233)	**14.582**	**90.788**	**98.574**	27.473	95.890	109.358
Opt06 (722)	**17.829**	65.543	65.543	38.127	**47.338**	**59.229**
Opt07 (81)	**0.406**	**117.077**	**117.077**	0.492	179.413	179.741
Opt09 (9)	**0.651**	**11.353**	**11.353**	0.966	622.341	623.161
Opt10 (199)	1.058	17.690	17.690	**1.052**	**14.918**	**15.745**
Opt11 (39)	**19.428**	**86.934**	**86.934**	31.445	124.748	139.160
Opt12 (80)	**55.576**	**353.138**	**353.138**	59.664	625.142	730.136
Opt15 (103)	**2.625**	257.256	257.256	2.731	**208.053**	**208.539**
Weakly Symmetric Instances (571)	**11.106**	104.221	**116.572**	43.563	**103.700**	123.719
Opt06 (390)	**4.866**	46.303	**47.863**	51.527	**39.577**	50.882
Opt07 (16)	**0.068**	**139.259**	**139.270**	0.126	197.428	197.444
Opt10 (46)	**0.195**	**0.688**	**0.883**	0.257	0.715	0.972
Opt11 (6)	**0.898**	812.382	812.672	1.121	**79.400**	**79.900**
Opt12 (73)	**60.400**	**502.417**	**619.996**	64.829	665.539	783.387
Opt15 (40)	**0.475**	404.037	404.131	0.654	**99.753**	**99.793**

Table A.6: Number of solved instances for the *StrongSymm-Opt*, *WeakSymm-Short-Opt* and *WeakSymm-Long-Opt* configurations for the selected benchmark sets (*all instances*), the subsets of instances for which `BreakID` detected strong symmetries (*symmetric instances*) and the subsets of instances for which `BreakID` detected a different number of strong and weak symmetries (*weakly symmetric instances*).

	StrongSymm-Opt	WeakSymm-Short-Opt	WeakSymm-Long-Opt
All Instances (1600)	936	**953**	939
Opt06 (954)	547	**555**	547
Opt07 (85)	**68**	66	**68**
Opt09 (29)	**26**	25	25
Opt10 (204)	**165**	**165**	**165**
Opt11 (52)	20	**21**	20
Opt12 (80)	20	**31**	27
Opt15 (196)	**90**	**90**	87
Symmetric Instances (1233)	712	**728**	715
Opt06 (722)	422	**430**	422
Opt07 (81)	**64**	62	**64**
Opt09 (9)	**6**	5	5
Opt10 (199)	**160**	**160**	**160**
Opt11 (39)	13	**14**	13
Opt12 (80)	20	**31**	27
Opt15 (103)	27	**26**	24
Weakly Symmetric Instances (571)	355	**364**	355
Opt06 (390)	266	**268**	263
Opt07 (16)	**10**	8	9
Opt10 (46)	**46**	**46**	**46**
Opt11 (6)	2	**4**	**4**
Opt12 (73)	17	**26**	22
Opt15 (40)	**14**	12	11