Journal of Artificial Intelligence Research 77 (2023) 1539–1589 Submitted 10/2022; published 08/2023

Certified Dominance and Symmetry Breaking for Combinatorial Optimisation

Bart Bogaerts BART.BOGAERTS@VUB.BE
Vrije Universiteit Brussel, Brussels, Belgium

Stephan Gocht STEPHAN.GOCHT@CS.LTH.SE
Lund University, Lund, Sweden
University of Copenhagen, Copenhagen, Denmark

Ciaran McCreesh CIARAN.MCCREESH@GLASGOW.AC.UK
University of Glasgow, Glasgow, UK

Jakob Nordström JN@DI.KU.DK
University of Copenhagen, Copenhagen, Denmark
Lund University, Lund, Sweden

Abstract

Symmetry and dominance breaking can be crucial for solving hard combinatorial search and optimisation problems, but the correctness of these techniques sometimes relies on subtle arguments. For this reason, it is desirable to produce efficient, machine-verifiable certificates that solutions have been computed correctly. Building on the cutting planes proof system, we develop a certification method for optimisation problems in which symmetry and dominance breaking is easily expressible. Our experimental evaluation demonstrates that we can efficiently verify fully general symmetry breaking in Boolean satisfiability (SAT) solving, thus providing, for the first time, a unified method to certify a range of advanced SAT techniques that also includes cardinality and parity (XOR) reasoning. In addition, we apply our method to maximum clique solving and constraint programming as a proof of concept that the approach applies to a wider range of combinatorial problems.

1. Introduction

Symmetries pose a challenge when solving hard combinatorial problems. As an illustration of this, consider the Crystal Maze puzzle[1] shown in Figure 1, which is often used in introductory constraint modelling courses. A human modeller might notice that the puzzle is the same after flipping vertically, and could introduce the constraint $A < G$ to eliminate this symmetry. Or, they may notice that flipping horizontally induces a symmetry, which could be broken with $A < B$. Alternatively, they might spot that the *values* are symmetrical, and that we can interchange 1 and 8, 2 and 7, and so on; this can be eliminated by saying that $A \leq 4$. In each case a constraint is being added that preserves satisfiability overall, but that restricts a solver to finding (ideally) just one witness from each equivalence class of solutions—the hope is that this will improve solver performance. However, although we may be reasonably sure that any of these three constraints is correct individually, are combinations of these constraints valid simultaneously? What if we had said $F < C$ instead

1. https://theconversation.com/what-problems-will-ai-solve-in-future-an-old-british-gameshow-can-help-explain-49080

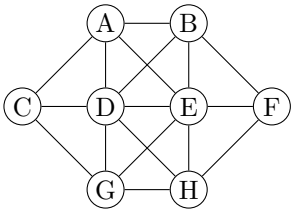

Figure 1: The Crystal Maze puzzle. Place numbers 1 to 8 in the circles, with every circle getting a different number, so that adjadcent circles do not have consecutive numbers.

of $A < B$? And what if we could use numbers more than once? Getting symmetry elimination constraints right can be error-prone even for experienced modellers. And when dealing with larger problems with many constraints and interacting symmetries it can be hard to tell whether a problem instance is genuinely unsatisfiable, or was made so by an incorrectly added symmetry breaking constraint.

Despite these difficulties, symmetry elimination using both manual and automatic techniques has been key to many successes across modern combinatorial optimisation paradigms such as constraint programming (CP) (Garcia de la Banda, Stuckey, Van Hentenryck, & Wallace, 2014), Boolean satisfiability (SAT) solving (Sakallah, 2021), and mixed-integer programming (MIP) (Achterberg & Wunderling, 2013). As these optimisation technologies are increasingly being used for high-value and life-affecting decision-making processes, it becomes vital that we can trust their outputs—and unfortunately, current solvers do not always produce correct answers (Brummayer, Lonsing, & Biere, 2010; Cook, Koch, Steffy, & Wolter, 2013; Akgün, Gent, Jefferson, Miguel, & Nightingale, 2018; Gillard, Schaus, & Deville, 2019; Bogaerts, McCreesh, & Nordström, 2022).

The most promising way to address this problem of correctness appears to be to use *certification*, or *proof logging*, where a solver must produce an efficiently machine-verifiable certificate that the answer to the problem was computed correctly (Alkassar, Böhme, Mehlhorn, Rizkallah, & Schweitzer, 2011; McConnell, Mehlhorn, Näher, & Schweitzer, 2011). This approach has been successfully used in the SAT community, which has developed numerous proof logging formats such as *RUP* (Goldberg & Novikov, 2003), *TraceCheck* (Biere, 2006), *DRAT* (Heule, Hunt Jr., & Wetzler, 2013a, 2013b; Wetzler, Heule, & Hunt Jr., 2014), *GRIT* (Cruz-Filipe, Marques-Silva, & Schneider-Kamp, 2017b), and *LRAT* (Cruz-Filipe, Heule, Hunt Jr., Kaufmann, & Schneider-Kamp, 2017a). However, currently used methods work only for decision problems, and do not support the full range of SAT solving techniques, let alone CP and MIP solving. As a case in point, there is no efficient proof logging for symmetry breaking, except for limited cases with small symmetries which can interact only in simple ways (Heule, Hunt Jr., & Wetzler, 2015). Tchinda and Djamégni (2020) recently proposed a proof logging method *DSRUP* for symmetric learning of variants of derived clauses, but this format does not support symmetry breaking (in the sense just discussed) and is also inherently unable to support pre- and inprocessing techniques, which are crucial in state-of-the-art SAT solvers.

1.1 Our Contribution

In this work, we develop a proof logging method for *optimisation* problems—i.e., problems where we are given both a formula F and an objective function f to be optimised by some assignment satisfying F—that can deal with *dominance*, a generalization of symmetry. Dominance breaking starts from the observation that we can strengthen F by imposing an additional constraint C if every solution of F that does not satisfy C is dominated by another solution of F. This technique is used in many fields of combinatorial optimisation (Walsh, 2006, 2012; Gent, Petrie, & Puget, 2006; McCreesh & Prosser, 2016; Jouglet & Carlier, 2011; Gebser, Kaminski, & Schaub, 2011; Bulhões, Sadykov, & Uchoa, 2018; Hoogeboom, Dullaert, Lai, & Vigo, 2020; Baptiste & Pape, 1997; Demeulemeester & Herroelen, 2002). And even if we are given just a decision problem for a formula F, we can still use this framework by inventing an objective function minimizing the lexicographic order of assignments, and then do symmetry breaking by adding dominance constraints with respect to this order.

The core idea to make our method produce efficiently verifiable proofs is to have it present an *explicit construction* of a dominating solution, so that a verifier can check that this construction strictly improves the objective value and preserves satisfaction of F. This constructed solution might itself be dominated, and hence not satisfy C, but since the objective value decreases with every application of the construction, we can be sure, without performing the repeated process, that it would be guaranteed to eventually terminate. Importantly, verifying the correctness of adding such a constraint C does not require the construction of an actual assignment satisfying C, and can be performed efficiently even when multiple constraints are to be added. This resolves a practical issue with earlier approaches like that of Heule et al. (2015). Such approaches struggle with large or overlapping symmetries, because adding a new symmetry breaking constraint might make it necessary to re-break previously added constraints, and to understand precisely how the different symmetries interact in order to be able to do so. With our method, we never need to revisit previously broken symmetries.

In addition, although we do not develop this direction in the current paper, it should also be noted that a quite intriguing feature of our method is that we could break symmetries of a problem with respect to a higher-order representation in pseudo-Boolean form in a provably correct way, even if the encoding in the conjunctive normal form (CNF) used by SAT solvers is not symmetric. If the solver is given a problem in pseudo-Boolean representation, then it could add symmetry breaking constraints based on this pseudo-Boolean representation, then translate the PB representation to CNF in a certified way (Gocht, Martins, Nordström, & Oertel, 2022), and finally produce a proof that the combination of all of these constraints is a valid encoding of the original problem. Following preliminaries in Section 2, we describe in full detail in Section 3 our dominance-based method of reasoning and prove that it is sound.

We have developed a proof format and verifier for this proof logging method by extending the tool *VeriPB* (Elffers, Gocht, McCreesh, & Nordström, 2020; Gocht & Nordström, 2021; Gocht, McCreesh, & Nordström, 2020b; Gocht, McBride, McCreesh, Nordström, Prosser, & Trimble, 2020a) with dominance reasoning. The pseudo-Boolean constraints and cutting planes proof system (Cook, Coullard, & Turán, 1987) used by *VeriPB* turn out to be quite

convenient to express and reason with dominance constraints, and moreover also make it possible to certify cardinality and parity (XOR) reasoning (Gocht & Nordström, 2021), two other advanced SAT solving techniques which previous proof logging methods have not been able to support efficiently.

After introducing the proof system that we have developed as the foundation of our proof logging method, we exhibit three applications that have not previously admitted efficient certification, and demonstrate that our new method can support simple, practical proof logging in each case.[2] First, in Section 4, we demonstrate that, by enhancing the *BreakID* tool for SAT solving (Devriendt, Bogaerts, Bruynooghe, & Denecker, 2016) with *VeriPB* proof logging, we can cover the entire solving toolchain when symmetries are involved. We show in full generality, and for the first time, that proof logging is practical by running experiments on SAT competition benchmarks. Second, in Section 5, we revisit the Crystal Maze example and describe a tool that provides proof logging for the kind of symmetry breaking constraints discussed above. Third, in Section 6, we discuss how adding a rule for deriving dominance breaking constraints can be used to support vertex domination reasoning in a maximum clique solver. We conclude the paper proper with a brief discussion of future research directions in Section 7. Appendix A contains a worked-out example of proof logging for symmetry breaking.

1.2 Publication History

An extended abstract of this paper was presented at the *36th AAAI Conference on Artificial Intelligence* (Bogaerts, Gocht, McCreesh, & Nordström, 2022a). The current manuscript extends the previous work with full proofs of all formal claims, and includes a more detailed exposition of the proof system and proof logging applications.

2. Preliminaries

Let us start with a brief review of some standard material, referring the reader to, e.g., Buss and Nordström (2021) for more details. A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x} = 1 - x$, where variables take values 0 (false) or 1 (true). A *pseudo-Boolean (PB) constraint* is a 0–1 linear inequality

$$C \doteq \sum_i a_i \ell_i \geq A\,, \tag{1}$$

where a_i and A are integers (and $\doteq$ denotes syntactic equality). We can assume without loss of generality that pseudo-Boolean constraints are *normalized*; i.e., that all literals ℓ_i are over distinct variables and that the *coefficients* a_i and the *degree (of falsity)* A are non-negative, but most of the time we will not need this. Instead, we will write PB constraints in more relaxed form as $\sum_i a_i \ell_i \geq A + \sum_j b_j \ell_j$ or $\sum_i a_i \ell_i \leq A + \sum_j b_j \ell_j$ when convenient, or even use equality $\sum_i a_i \ell_i = A$ as syntactic sugar for the pair of inequalities $\sum_i a_i \ell_i \geq A$ and $\sum_i -a_i \ell_i \geq -A$, assuming that all constraints are implicitly normalized if needed.

The *negation* $\neg C$ of the constraint C in (1) is (the normalized form of)

$$\neg C \doteq \sum_i -a_i \ell_i \geq -A + 1\,. \tag{2}$$

2. All code for our implementations and experiments, as well as data and scripts for all plots, can be found in the repository (Bogaerts, Gocht, McCreesh, & Nordström, 2022b).

A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of PB constraints, which we can also think of as the set $\bigcup_j \{C_j\}$ of constraints in the formula, choosing whichever viewpoint seems most convenient. Note that a *(disjunctive) clause* $\ell_1 \vee \cdots \vee \ell_k$ is equivalent to the pseudo-Boolean constraint $\ell_1 + \cdots + \ell_k \geq 1$, so formulas in *conjunctive normal form (CNF)* are special cases of PB formulas.

A *(partial) assignment* is a (partial) function from variables to $\{0, 1\}$. A partial assignment is also referred to as a *restriction*. A *substitution* (or *affine restriction*) can also map variables to literals. We extend an assignment or substitution ρ from variables to literals in the natural way by respecting the meaning of negation, and for literals ℓ over variables x not in the domain of ρ, denoted $x \notin \mathit{dom}(\rho)$, we use the convention $\rho(\ell) = \ell$. (That is, we can consider all assignments and substitution to be total, but to be the identity outside of their specified domains. Strictly speaking, we also require that all substitutions be defined on the truth constants $\{0, 1\}$ and be the identity on these constants.) We sometimes write $x \mapsto b$ when $\rho(x) = b$, for b a literal or truth value, to specify parts of ρ or when ρ is clear from context.

We write $\rho \circ \omega$ to denote the composed substitution resulting from applying first ω and then ρ, i.e., $\rho \circ \omega(x) = \rho(\omega(x))$. As an example, for $\omega = \{x_1 \mapsto 0, x_3 \mapsto \overline{x}_4, x_4 \mapsto x_3\}$ and $\rho = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto 0\}$ we have $\rho \circ \omega = \{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 0\}$. Applying ω to a constraint C as in (1) yields the *restricted* constraint

$$C\restriction_\omega \doteq \textstyle\sum_i a_i \omega(\ell_i) \geq A\,, \tag{3}$$

substituting literals or values as specified by ω. For a formula F we define $F\restriction_\omega \doteq \bigwedge_j C_j\restriction_\omega$.

Since we will sometimes have to make fairly elaborate use of substitutions, let us discuss some further notational conventions. If F is a formula over variables $\vec{x} = \{x_1, \ldots, x_m\}$, we can write $F(\vec{x})$ when we want to stress the set of variables over which F is defined. For a substitution ω with domain (contained in) $\vec{x}$, the notation $F\big(\vec{x}\restriction_\omega\big)$ is understood to be a synonym of $F\restriction_\omega$. For the same formula F and $\vec{y} = \{y_1, \ldots, y_m\}$, the notation $F(\vec{y})$ is syntactic sugar for $F\restriction_\omega$ with ω denoting the substitution (implicitly) defined by $\omega(x_i) = y_i$ for $i = 1, \ldots, n$. Finally, for a formula $G = G(\vec{x}, \vec{y})$ over $\vec{x} \cup \vec{y}$ and substitutions α and β defined on $\vec{z} = \{z_1, \ldots, z_n\}$ (either of which could be the identity), the notation $G(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ should be understood as $G\restriction_\omega$ for ω defined by $\omega(x_i) = \alpha(z_i)$ and $\omega(y_i) = \beta(z_i)$ for $i = 1, \ldots, n$.

The (normalized) constraint C in (1) is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$. A pseudo-Boolean formula F is satisfied by ρ if all constraints in it are, in which case it is *satisfiable*. If there is no satisfying assignment, F is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable. In this paper, we also consider optimisation problems, where in addition to F we are given an integer linear objective function $f \doteq \sum_i w_i \ell_i$ and the task is to find an assignment that satisfies F and minimizes f. (To deal with maximization problems we can just negate the objective function.)

Cutting planes (Cook et al., 1987) is a method for iteratively deriving constraints C from a pseudo-Boolean formula F. We write $F \vdash C$ for any constraint C derivable as follows. Any *axiom constraint* $C \in F$ is trivially derivable, as is any *literal axiom* $\ell \geq 0$. If $F \vdash C$ and $F \vdash D$, then any positive integer *linear combination* of C and D is derivable. Finally, from a constraint in normalized form $\sum_i a_i \ell_i \geq A$ we can use *division* by a positive integer

d to derive $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$, dividing and rounding up the degree and coefficients. For a set of pseudo-Boolean constraints F' we write $F \vdash F'$ if $F \vdash C$ for all $C \in F'$.

For pseudo-Boolean formulas F, F' and constraints C, C', we say that F *implies* or *models* C, denoted $F \models C$, if any assignment satisfying F also satisfies C, and write $F \models F'$ if $F \models C'$ for all $C' \in F'$. It is easy to see that if $F \vdash F'$ then $F \models F'$, and so F and $F \wedge F'$ are equisatisfiable. A piece of non-standard terminology that will be convenient for us is that we will say that a constraint C *literal-axiom-implies* another constraint C' if C' can be derived from C using only addition of literal axioms $\ell \geq 0$.

An assignment ρ *falsifies* or *violates* the constraint C if the restricted constraint $C\restriction_\rho$ can not be satisfied. For the normalized constraint C in (1), this is the case if $\sum_{\rho(\ell_i)\neq 0} a_i < A$. A constraint C *unit propagates* the literal ℓ under ρ if if $C\restriction_\rho$ cannot be satisfied unless $\ell \mapsto 1$. During *unit propagation* on F under ρ, the assignment ρ is extended iteratively by any propagated literals until an assignment ρ' is reached under which no constraint in F is propagating, or until ρ' violates some constraint $C \in F$. The latter scenario is referred to as a *conflict*. Using the generalization of *reverse unit propagation clauses* (Goldberg & Novikov, 2003) to pseudo-Boolean constraints by Elffers et al. (2020), we say that F *implies* C *by reverse unit propagation (RUP)*, and that C is a *RUP constraint* with respect to F, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. If C is a RUP constraint with respect to F, then it can be proven that there is also a derivation $F \vdash C$. More generally, it can be shown that $F \vdash C$ if and only if $F \wedge \neg C \vdash \bot$, where $\bot$ is a shorthand for the *trivially false constraint* $0 \geq 1$. Therefore, we will extend the notation and write $F \vdash C$ also when C is derivable from F by RUP or by contradiction. It is worth noting here again that, as shown in (2), the negation of any pseudo-Boolean constraint can also be expressed syntactically as a pseudo-Boolean constraint—this fact will be convenient in what follows.

3. A Proof System for Dominance Breaking

We proceed to develop our formal proof system for verifying dominance breaking, which we have implemented on top of the tool *VeriPB* as developed in the sequence of papers (Elffers et al., 2020; Gocht et al., 2020b, 2020a; Gocht & Nordström, 2021). We remark that for applications it is absolutely crucial not only that the proof system be sound, but that all proofs be efficiently machine-verifiable. There are significant challenges involved in making proof logging and verification efficient, but in this section we mostly ignore these more applied aspects of our work and focus on the theoretical underpinnings.

Our foundation is the cutting planes proof system described in Section 2. However, in a proof in our system for (F, f), where f is a linear objective function to be minimized under the pseudo-Boolean formula F (or where $f \doteq 0$ for decision problems), we also allow strengthening F by adding constraints C that are not implied by the formula. Pragmatically, adding such non-implied constraints C should be in order as long as we keep some optimal solution, i.e., a satisfying assignment to F that minimizes f, which we will refer to as an *f-minimal solution* for F. We will formalize this idea by allowing the use of an additional pseudo-Boolean formula $\mathscr{O}_\preceq(\vec{u}, \vec{v})$ that, together with an ordered set of variables $\vec{z}$, defines a relation $\alpha \preceq \beta$ to hold between assignments α and β if $\mathscr{O}_\preceq(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ evaluates to true. We require (a cutting planes proof) that $\mathscr{O}_\preceq$ is such that this defines a preorder, i.e., a reflexive

and transitive relation. Adding new constraints C will be valid as long as we guarantee to preserve some f-minimal solution that is also minimal with respect to $\preceq$. In other words, the preorder $\preceq$ can be combined with the objective function f to define a preorder $\preceq_f$ on assignments by

$$\alpha \preceq_f \beta \quad \text{if} \quad \alpha \preceq \beta \text{ and } f\restriction_\alpha \leq f\restriction_\beta , \tag{4}$$

and we require that all derivation steps in the proof should preserve some solution that is minimal with respect to $\preceq_f$. The preorder defined by $\mathscr{O}_\preceq(\vec{u}, \vec{v})$ will only become important once we introduce our new *dominance-based strengthening rule* later in this section. For simplicity, up until that point the reader can assume that the pseudo-Boolean formula is $\mathscr{O}_\top \doteq \emptyset$ inducing the trivial preorder relating all assignments, though all proofs presented below work in full generality for the orders that will be introduced later.

A proof for (F, f) in our proof system consists of a sequence of *proof configurations* $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$, where

- $\mathscr{C}$ is a set of pseudo-Boolean *core constraints*;
- $\mathscr{D}$ is another set of pseudo-Boolean *derived constraints*;
- $\mathscr{O}_\preceq$ is a pseudo-Boolean formula encoding a preorder and $\vec{z}$ a set of literals on which this preorder will be applied; and
- v is the best value found so far for f.

The initial configuration is $(F, \emptyset, \mathscr{O}_\top, \emptyset, \infty)$. The distinction between $\mathscr{C}$ and $\mathscr{D}$ is only relevant when a nontrivial preorder is used; we will elaborate on this when discussing the dominance-based strengthening rule. The intended relation between f and v is that if $v < \infty$, then there exists a solution α satisfying F such that $f\restriction_\alpha \leq v$, and in this case the proof can make use of the constraint $f \leq v - 1$ in the search for better solutions. As long as the optimal solution has not been found, it should hold that f-minimal solutions for $\mathscr{C} \cup \mathscr{D}$ have the same objective value as f-minimal solutions for F. The precise relation is formalized in the notion of (F, f)-*valid configurations*, which we will define next. In some cases, it will be convenient to also have a less stringent notion of *weak* (F, f)-*validity*, which holds even if constraints have been removed from the original formula. Jumping ahead a bit, what this means is that proofs preserving weak (F, f)-validity can only be used to show that no solutions better than a given value exist, while proofs preserving (F, f)-validity can establish that the optimal value *equals* a certain value.

Definition 1. *A configuration* $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ *is* weakly (F, f)-valid *if the following conditions hold:*

1. *For every* $v' < v$*, it holds that if* $F \cup \{f \leq v'\}$ *is satisfiable, then* $\mathscr{C} \cup \{f \leq v'\}$ *is satisfiable.*
2. *For every total assignment* ρ *satisfying the constraints* $\mathscr{C} \cup \{f \leq v - 1\}$*, there exists a total assignment* $\rho' \preceq_f \rho$ *satisfying* $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v - 1\}$*, where* $\preceq_f$ *is the relation defined in* (4)*.*

The configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ *is* (F, f)-valid *if in addition the following conditions hold:*

3. *If $v < \infty$, then $F \cup \{f \leq v\}$ is satisfiable.*
4. *For every $v' < v$, it holds that if $\mathscr{C} \cup \{f \leq v'\}$ is satisfiable, then $F \cup \{f \leq v'\}$ is satisfiable.*

When we present the derivation rules in our proof system below, we will show that (F, f)-validity is an invariant of the proof system, i.e., that it is preserved by all derivation rules. For the *deletion* rule, which can remove constraints in the input formula, we will present a version of the rule that only preserves weak (F, f)-validity. This alternative rule is introduced for pragmatic reasons to support proof logging for SAT solvers. In such a setting, deletions can be treated in a more relaxed fashion, since the generated proofs are only used to establish unsatisfiability.

To see why Definition 1 is relevant, note that items 1, 2, and 4 together imply that if the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is such that v is not yet the value of an optimal solution, then f-minimal solutions for F and $\mathscr{C} \cup \mathscr{D}$ have the same objective value, just as desired. A proof in our proof system ends when the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v^*)$ is such that $\mathscr{C} \cup \mathscr{D}$ contains contradiction $\bot \doteq 0 \geq 1$. If the resulting state is (F, f)-valid, either $v^* = \infty$ and F is unsatisfiable, or v^* is the optimal value (or $v^* = 0$ for a satisfiable decision problem). If the resulting state is only weakly (F, f)-valid, we get slightly weaker conclusions. We collect the precise statements in two formal theorems.

Theorem 2. *Let F be a pseudo-Boolean formula and f an objective function. If the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v^*)$ is weakly (F, f)-valid and is such that $\mathscr{C} \cup \mathscr{D}$ contains the contradictory constraint $0 \geq 1$, then it holds that*

- *for any solution α for F we have $f\restriction_\alpha \geq v^*$;*
- *in particular, if $v^* = \infty$, then F is unsatisfiable.*

Proof. If F is satisfiable, then let α be a satisfying assignment for F. If $f\restriction_\alpha < v^*$, then α satisfies $F \cup \{f \leq v^* - 1\}$. Hence, item 1 in Definition 1 says that there exists an α' that satisfies $\mathscr{C} \cup \{f \leq v^* - 1\}$ and item 2 then implies the existence of an α'' that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v^* - 1\}$. But this contradicts the assumption that the unsatisfiable constraint $0 \geq 1$ is in $\mathscr{C} \cup \mathscr{D}$. It follows that $f\restriction_\alpha \geq v^*$, and that no satisfying assignments for F can exist if $v^* = \infty$. □

Theorem 3. *Let F be a pseudo-Boolean formula and f an objective function. If the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v^*)$ is (F, f)-valid and is such that $\mathscr{C} \cup \mathscr{D}$ contains $0 \geq 1$, then*

- *F is unsatisfiable if and only if $v^* = \infty$; and*
- *if F is satisfiable, then there is an f-minimal solution α for F with objective value $f\restriction_\alpha = v^*$.*

Proof. By appealing to Theorem 2, we can conclude that if $v^* = \infty$, then F is unsatisfiable. If F is unsatisfiable, then we must have $v^* = \infty$ due to item 3 in Definition 1. This establishes the first part of Theorem 3.

For the second part, suppose that F is satisfiable. Then $v^* < \infty$ by the preceding paragraph, and so by item 3 of Definition 1 there is a solution α for F such that $f\restriction_\alpha \leq v^*$.

But Theorem 2 says that for any solution α for F it holds that $f\restriction_\alpha \geq v^*$. Hence, α is an f-minimal solution for F with objective value $f\restriction_\alpha = v^*$ as claimed. □

We are now ready to give a formal description of the rules in our proof system and argue that these rules preserve (F, f)-validity (or, in the case of the alternative deletion rule, weak (F, f)-validity). The attentive reader might have noted that we did not use item 4 in Definition 1 in our proofs of Theorems 2 and 3, but this condition will be critical to argue that (F, f)-validity is an invariant of our proof system.

3.1 Implicational Derivation Rule

If we can exhibit a derivation of the pseudo-Boolean constraint C from $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\}$ in our (slightly extended) version of the cutting planes proof system as described in Section 2 (i.e., in formal notation, if $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \vdash C$), then we should be allowed to add the implied constraint C to our collection of derived constraints. Let us write this down as our first formal derivation rule.

Definition 4 (Implicational derivation rule)**.** *If $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \vdash C$, then we can transition from the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to the configuration $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ by the* implicational derivation rule*. In this case, we will also say that C is derivable from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ by the implicational derivation rule.*

The implicational derivation rule preserves (F, f)-validity (and weak (F, f)-validity) since $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \models C$ holds by soundness of the cutting planes proof system, but, more importantly, the cutting planes derivation provides a simple and efficient way for an algorithm to *verify* that this implication holds. This is a key feature of all rules in our proof system—not only are they sound, but the soundness of every rule application can be efficiently verified by checking a simple, syntactic object.

When doing proof logging, the solver would need to specify by which sequence of cutting planes derivation rules C was obtained. For practical purposes, though, it greatly simplifies matters that in many cases the verifier can figure out the required proof details automatically, meaning that the proof logger can just state the desired constraint without any further information. One important example of this is when C is a reverse unit propagation (RUP) constraint with respect to $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\}$. Another case is when C is literal-axiom-implied by some other constraint.

3.2 Objective Bound Update Rule

The *objective bound update rule* allows improving the estimate of what value can be achieved for the objective function f.

Definition 5 (Objective bound update rule)**.** *We say that we can transition from the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v')$ by the* objective bound update rule *if there is an assignment α satisfying $\mathscr{C}$ such that $f\restriction_\alpha = v' < v$.*

When actually doing proof logging, the solver would specify such an assignment α, which would then be checked by the proof verifier (in our case *VeriPB*).

To argue that this rule preserves (F, f)-validity (and weak (F, f)-validity), note that items 1, 2, and 4 are trivially satisfied. (For item 2, observe that whenever $\rho' \preceq_f \rho$ and ρ satisfies $\{f \leq v' - 1\}$, so does ρ'.) Item 3 is satisfied after updating the objective bound since item 4 guarantees the existence of an α' satisfying F with an objective value that is at least as good as v'.

Note that we have no guarantee that α itself will be a solution for F. However, although we will not emphasize this point here, it follows from our formal treatment below that the proof system guarantees that such a solution α' for the original formula F can be efficiently reconstructed from the proof (where efficiency is measured in the size of the proof).

3.3 Redundance-Based Strengthening Rule

The redundance-based strengthening rule allows deriving a constraint C from $\mathscr{C} \cup \mathscr{D}$ even if C is not implied, provided that it can be shown that any assignment α that satisfies $\mathscr{C} \cup \mathscr{D}$ can be transformed into another assignment $\alpha' \preceq_f \alpha$ that satisfies both $\mathscr{C} \cup \mathscr{D}$ and C (in case $\mathscr{O}_\preceq = \mathscr{O}_\top$, the condition $\alpha' \preceq_f \alpha$ just means that $f\restriction_{\alpha'} \leq f\restriction_\alpha$). This rule is borrowed from Gocht and Nordström (2021), who in turn rely heavily on Heule, Kiesl, and Biere (2017) and Buss and Thapen (2019). We extend this rule here from decision problems to optimization problems in the natural way.

Definition 6 (Redundance-based strengthening rule)**.** *If for a pseudo-Boolean constraint C there is a substitution ω such that*

$$\mathscr{C} \cup \mathscr{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \vdash (\mathscr{C} \cup \mathscr{D} \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})\,, \tag{5}$$

then we can transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ by redundance-based strengthening*, or just* redundance *for brevity. We refer to the substitution ω as the* witness*, and also say that C can be derived from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ by redundance.*

Intuitively, (5) says that if some assignment α satisfies $\mathscr{C} \cup \mathscr{D}$ but falsifies C, then the assignment $\alpha' = \alpha \circ \omega$ still satisfies $\mathscr{C} \cup \mathscr{D}$ and also satisfies C. In addition, the condition $f\restriction_\omega \leq f$ ensures that $\alpha \circ \omega$ achieves an objective function value that is at least as good as that for α. This together with the constraints $\mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})$ guarantees that $\alpha' \preceq_f \alpha$. For proof logging purposes, the witness ω as well as any non-immediate cutting planes derivations of constraints on the right-hand side of (5) would have to be specified, but, e.g., all RUP constraints or literal-axiom-implied constraints can be left to the verifier to check.

Proposition 7. *If we can transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ by the redundance-based strengthening rule and $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is [weakly] (F, f)-valid, then $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ is also [weakly] (F, f)-valid.*

Proof. First assume $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is weakly (F, f)-valid. Item 1 in Definition 1 remains satisfied for $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ since F, v, and $\mathscr{C}$ are unchanged.

For item 2 we can recycle proofs of similar properties for decision problems (Heule et al., 2017; Buss & Thapen, 2019; Gocht & Nordström, 2021). Consider a total assignment ρ satisfying $\mathscr{C} \cup \{f \leq v - 1\}$. Without loss of generality we can assume that ρ also satisfies $\mathscr{D}$ (since $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ satisfies item 2 in Definition 1). We wish to construct an assignment ρ'

that satisfies the constraints $\mathscr{C} \cup \mathscr{D} \cup \{C\}$ and also the condition $\rho' \preceq_f \rho$, which is to say that $f\restriction_{\rho'} \leq f\restriction_\rho$ and $\mathscr{O}_\preceq(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_\rho)$ should hold.

If ρ satisfies C, then we use $\rho' = \rho$ and all conditions are satisfied (recall that $\mathscr{O}_\preceq$ induces a preorder, and hence a reflexive relation: for any ρ, $\mathscr{O}_\preceq(\vec{z}\restriction_\rho, \vec{z}\restriction_\rho)$ holds). Otherwise, choose $\rho' = \rho \circ \omega$. Since ρ is total and does not satisfy C, it satisfies $\neg C$. This means that ρ satisfies $\mathscr{C} \cup \mathscr{D} \cup \neg C$, and hence by (5) also

$$(\mathscr{C} \cup \mathscr{D} \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})\,. \tag{6}$$

Clearly, for any constraint D, it holds that $(D\restriction_\omega)\restriction_\rho = D\restriction_{\rho\circ\omega}$ and thus if ρ satisfies $D\restriction_\omega$, then $\rho' = \rho \circ \omega$ satisfies D. Therefore, ρ' satisfies $\mathscr{C} \cup \mathscr{D} \cup C$. By the same reasoning, since ρ satisfies $f\restriction_\omega \leq f$ we have that $f\restriction_{\rho\circ\omega} = f\restriction_{\rho'} \leq f\restriction_\rho$ holds. Finally, the fact that ρ satisfies $\mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})$ means that $\mathscr{O}_\preceq(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_\rho)$ holds. This shows that item 2 holds for $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$, which concludes our proof that weak (F, f)-validity is preserved.

To see that also (F, f)-validity is preserved, it suffices to note that items 3 and 4 do not depend on $\mathscr{D}$ and hence are satisfied in $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ whenever they are satisfied in $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$. □

3.4 Deletion Rules

An interesting property of the redundance rule introduced in Section 3.3 is that when the set of constraints in $\mathscr{C} \cup \mathscr{D}$ grows, it can get harder to derive new constraints, since every constraint D already in $\mathscr{C} \cup \mathscr{D}$ adds a new constraint $D\restriction_\omega$ needing to be derived on the right-hand side of (5). For this reason, and also due to efficiency concerns during verification of proofs, we need to be able to delete previously derived constraints.

Definition 8 (Deletion rule)**.** *We can transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ by the* deletion rule *if*

1. *$\mathscr{D}' \subseteq \mathscr{D}$ and*
2. *$\mathscr{C}' = \mathscr{C}$ or $\mathscr{C}' = \mathscr{C} \setminus \{C\}$ for some constraint C derivable via the redundance rule from $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$.*

The last condition above perhaps seems slightly odd, but it is there since deleting arbitrary constraints could violate (F, f)-validity in two different ways. Firstly, it could allow finding better-than-optimal solutions. Secondly, and perhaps surprisingly, in combination with the dominance-based strengthening rule, which we will discuss below, arbitrary deletion is unsound, as it can turn satisfiable instances into unsatisfiable ones. We will discuss this further in Example 15 once we have introduced the dominance rule.

Proposition 9. *If we can transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ by the deletion rule and $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is [weakly] (F, f)-valid, then $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ is also [weakly] (F, f)-valid.*

Proof. First assume that $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is weakly (F, f)-valid. Item 1 in Definition 1 clearly remains satisfied for $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ since $\mathscr{C}' \subseteq \mathscr{C}$. To prove that item 2 is satisfied after applying the deletion rule, let α be any assignment that satisfies $\mathscr{C}' \cup \{f \leq v-1\}$. If $\mathscr{C}' = \mathscr{C}$,

we find an $\alpha' \preceq_f \alpha$ that satisfies $\mathscr{C}' \cup \mathscr{D}' \cup \{f \leq v-1\}$ using weak (F,f)-validity of the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$. Hence, we can assume that $\mathscr{C}' = \mathscr{C} \setminus \{C\}$. If α satisfies C, this assignment satisfies all of $\mathscr{C}$, and again the claim follows from weak (F,f)-validity of the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ before deletion. Assume therefore that α does not satisfy C. Since C is derivable via redundance from $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$, it holds that

$$\mathscr{C}' \cup \{f \leq v-1\} \cup \{\neg C\} \vdash (\mathscr{C}' \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}) \tag{7}$$

for some witness ω. We can use this witness to obtain an assignment $\alpha'' = \alpha \circ \omega$ satisfying $\mathscr{C} = \mathscr{C}' \cup \{C\}$ such that $f\restriction_{\alpha''} \leq f\restriction_\alpha \leq v-1$, which shows that $\mathscr{C} \cup \{f \leq v-1\}$ is satisfiable. Appealing to the weak (F,f)-validity of the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ before deletion, we then find an α' with $f\restriction_{\alpha'} \leq v-1$ that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\}$. In this way, we establish that item 2 in Definition 1 holds for the configuration $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ after deletion.

Now assume that $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is (F,f)-valid. Item 3 clearly remains satisfied for $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ since v is unchanged. We show that item 4 holds for $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$; the proof is very similar to the proof of item 2 above. Starting from an assignment α satisfying $\mathscr{C}' \cup \{f \leq v'\}$ for some $v' < v$, we use α to construct a satisfying assignment α' for $F \cup \{f \leq v'\}$. If $\mathscr{C}' = \mathscr{C}$, we get α' from the (F,f)-validity of $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$, so assume $\mathscr{C}' = \mathscr{C} \setminus \{C\}$. If α satisfies C, the same assignment satisfies $\mathscr{C}$, and again the claim follows from (F,f)-validity of the configuration before deletion. Assume therefore that α does not satisfy C. Since C is derivable via redundance from $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$, it holds that

$$\mathscr{C}' \cup \{f \leq v-1\} \cup \{\neg C\} \vdash (\mathscr{C}' \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})\,. \tag{8}$$

From this we can construct an assignment $\alpha'' = \alpha \circ \omega$ that satisfies $\mathscr{C} = \mathscr{C}' \cup \{C\}$ and is such that $f\restriction_{\alpha''} \leq f\restriction_\alpha \leq v'$, showing that $\mathscr{C} \cup \{f \leq v'\}$ is satisfiable. Appealing to the (F,f)-validity of the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$, we then find an α' with $f\restriction_{\alpha'} \leq v'$ that satisfies F. This proves that item 4 holds for the configuration $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ after deletion. □

The deletion rule in Definition 8 is more cumbersome than that in *DRAT*-style proof systems, in that deletions of constraints in the core set must be accompanied by proofs so that the validity of the deletions can be checked. When we want to highlight this property of the rule, we will refer to it as *checked deletion*. This property can make it more difficult to implement proof logging in a solver, and can also have negative effects on the time required for proof generation and proof verification. If we are only interested in certifying unsatisfiability of decision problem instances—which has traditionally been the case in Boolean satisfiability solving—then an alternative is to use a more liberal deletion rule that allows unrestricted deletion of constraints from the core set $\mathscr{C}$ provided that the derived set $\mathscr{D}$ is empty. When such an *unchecked deletion rule* is used, it is easy to see that unsatisfiable sets of constraints can turn satisfiable, and for optimisation problems spurious solutions can appear that yield better objective function values than are possible for the original input constraints. But proofs of unsatisfiability are still valid, as are lower bounds on the value of the objective function to be minimized. Phrased in the language of Definition 1, we are guaranteed to preserve weak (F,f)-validity, but not necessarily (F,f)-validity.

Definition 10 (Unchecked deletion rule)**.** *If $\mathscr{C}' \subseteq \mathscr{C}$, then we can transition from the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to the configuration $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$ using the* unchecked deletion rule*.*

Proposition 11. *If $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is a weakly (F, f)-valid configuration and we can transition from it to $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$ using unchecked deletion, then $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$ is also weakly (F, f)-valid.*

Proof. Item 1 in Definition 1 is clearly maintained when transitioning to $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$ since the set of core constraints shrinks. As to item 2, it is trivially satisfied when the set of derived constraints is empty. □

3.5 Transfer Rule

Constraints can always be moved from the derived set $\mathscr{D}$ to the core set $\mathscr{C}$ using the *transfer rule*.

Definition 12 (Transfer rule)**.** *We can transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}', \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ by the* transfer rule *if $\mathscr{C} \subseteq \mathscr{C}' \subseteq \mathscr{C} \cup \mathscr{D}$.*

This transfer rule clearly preserves (F, f)-validity (and weak (F, f)-validity).

The transfer rule together with deletion allows replacing constraints in the original formula with stronger constraints. For example, assume that $x + y \geq 1$ is in $\mathscr{C}$ and that we derive $x \geq 1$. Then we can move $x \geq 1$ from $\mathscr{D}$ to $\mathscr{C}$ and then delete $x + y \geq 1$. The required redundance check $\{x \geq 1, \neg(x + y \geq 1)\} \vdash \bot$ is immediate.

The rules discussed so far in this section do not change $\mathscr{O}_\preceq$, and so any derivation using these rules only will operate with the trivial preorder $\mathscr{O}_\top$ imposing no conditions. The proof system defined in terms of these rules is a straightforward extension of *VeriPB* as developed by Elffers et al. (2020), Gocht et al. (2020a, 2020b), Gocht and Nordström (2021) to an optimisation setting. We next discuss the main contribution of this paper, namely the new dominance rule making use of the preorder encoding $\mathscr{O}_\preceq$.

3.6 Dominance-Based Strengthening Rule

Any preorder $\preceq$ induces a strict order $\prec$ defined by $\alpha \prec \beta$ if $\alpha \preceq \beta$ and $\beta \not\preceq \alpha$. (Note that if $\alpha \preceq \beta$ and $\beta \preceq \alpha$ both hold, this means that neither $\alpha \prec \beta$ nor $\beta \prec \alpha$ holds.) The relation $\prec_f$ obtained in this way from the preorder (4) coincides with what Chu and Stuckey (2015) call a *dominance relation* in the context of constraint optimisation. Our dominance rule allows deriving a constraint C from $\mathscr{C} \cup \mathscr{D}$ even if C is not implied, similar to the redundance rule. However, for the dominance rule an assignment α satisfying $\mathscr{C} \cup \mathscr{D}$ but falsifying C only needs to be mapped to an assignment α' that satisfies $\mathscr{C}$, but not necessarily $\mathscr{D}$ or C. On the other hand, the new assignment α' should satisfy the strict inequality $\alpha' \prec_f \alpha$ and not just $\alpha' \preceq_f \alpha$ as in the redundance rule. To show that this new dominance rule preserves (F, f)-validity, we will prove that it is possible to construct an assignment that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$ by iteratively applying the witness of the dominance rule, in combination with (F, f)-validity of the configuration before application of the dominance rule. As our base case, if α' satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$, we are done. Otherwise, since α' satisfies $\mathscr{C}$, by (F, f)-validity we are guaranteed the existence of an assignment α''

satisfying $\mathscr{C} \cup \mathscr{D}$ for which $\alpha'' \prec_f \alpha' \prec_f \alpha$ holds. If α'' still does not satisfy C, we can repeat the argument. In this way, we get a strictly decreasing sequence (with respect to $\prec_f$) of assignments. Since the set of possible assignments is finite, this sequence will eventually terminate.

Formally, we can derive C by dominance-based strengthening given a substitution ω such that

$$\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \cup \{\neg C\} \vdash \mathscr{C}\restriction_\omega \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \neg\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega) \cup \{f\restriction_\omega \leq f\}\,, \tag{9}$$

where $\mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z})$ and $\neg\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ together state that $\alpha \circ \omega \prec \alpha$ for any assignment α. A minor technical problem is that the pseudo-Boolean formula $\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ may contain multiple constraints, so that the negation of it is no longer a PB formula. To get around this, we split (9) into two separate conditions and shift $\neg\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ to the premise of the implication, which eliminates the negation.

After this adjustment, the formal version of our *dominance-based strengthening rule*, or just *dominance rule* for brevity, can be stated as follows.

Definition 13 (Dominance-based strengthening rule)**.** *If for a pseudo-Boolean constraint C there is a* witness *substitution ω such that the conditions*

$$\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \cup \{\neg C\} \vdash \mathscr{C}\restriction_\omega \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \leq f\} \tag{10a}$$

$$\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \cup \{\neg C\} \cup \mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot \tag{10b}$$

are satisfied, then we can transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$ using dominance-based strengthening*, or just* dominance *for brevity. In this case, we also say that C is derivable from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ by dominance.*

Just as for the redundance rule, the witness ω as well as any non-immediate derivations would have to be specified in the proof log.

Proposition 14. *If we can transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$ by the dominance-based strengthening rule and $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is [weakly] (F, f)-valid, then $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is also [weakly] (F, f)-valid.*

Proof. Since F, $\mathscr{C}$, and v are not affected in a transition using the dominance rule, items 1, 3, and 4 in Definition 1 hold for $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ if and only if they hold for $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$. Hence, we only need to prove that item 2 holds for $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$. Assume towards contradiction that it *does not* hold. Let S denote the set of total assignments α that

(1) satisfy $\mathscr{C} \cup \{f \leq v-1\}$ and

(2) admit no $\alpha' \preceq_f \alpha$ satisfying $\mathscr{C} \cup \mathscr{D} \cup \{C\}$.

By our assumption, S is non-empty.

Let α be some $\prec_f$-minimal assignment in S (since $\prec_f$ is a strict order and S is finite, minimal elements of S exist). Since $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is weakly (F, f)-valid, there exists some $\alpha_1 \preceq_f \alpha$ that satisfies $\mathscr{C} \cup \mathscr{D}$. We know that α_1 cannot satisfy C since $\alpha \in S$. Hence, α_1 satisfies $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\}$. From (10a) it follows that α_1 satisfies $\mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \leq f\}$ and

thus that $\mathscr{O}_{\preceq}(\vec{z}\restriction_{\alpha_1\circ\omega}, \vec{z}\restriction_{\alpha_1})$ and $f\restriction_{\alpha_1\circ\omega} \leq f\restriction_{\alpha_1}$ hold. In other words, $\alpha_1 \circ \omega \preceq_f \alpha_1$. By (10b), it follows that α_1 does not satisfy $\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega})$, i.e., $\mathscr{O}_{\preceq}(\vec{z}\restriction_{\alpha_1}, \vec{z}\restriction_{\alpha_1\circ\omega})$ does not hold, and thus $\alpha_1 \not\preceq_f \alpha_1 \circ \omega$. Now let α_2 be $\alpha_1 \circ \omega$. We showed that $\alpha_2 \prec_f \alpha_1 \preceq_f \alpha$. Furthermore, since α_1 satisfies $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\}$, (10a) yields that α_2 satisfies $\mathscr{C}$. Thus α_2 satisfies $\mathscr{C} \cup \{f \leq v - 1\}$. Since $\alpha_2 \prec_f \alpha$, and α is a minimal element of S, it cannot be that $\alpha_2 \in S$. Thus, there must exist a $\alpha' \preceq_f \alpha_2$ that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$. However, it also holds that $\alpha' \preceq_f \alpha$, and since $\alpha \in S$ this means that α' cannot satisfy $\mathscr{C} \cup \mathscr{D} \cup \{C\}$. This yields a contradiction, thereby finishing our proof. □

When introducing the deletion rule, we already mentioned that deleting arbitrary constraints can be unsound in combination with dominance-based strengthening. We now illustrate this phenomenon.

Example 15. *Consider the formula* $F = \{p \geq 1\}$ *with objective* $f \doteq 0$ *and the configuration*

$$(\mathscr{C}_1 = \{p \geq 1\}, \mathscr{D}_1 = \{p \geq 1\}, \mathscr{O}_{\preceq}, \{p\}, \infty)\,, \tag{11}$$

where $\mathscr{O}_{\preceq}(u, v)$ *is defined as* $\{v + \overline{u} \geq 1\}$. *This configuration is* (F, f)*-valid and* $\mathscr{C} \cup \mathscr{D}$ *is satisfiable.* If *we were allowed to delete constraints arbitrarily from* $\mathscr{C}$*, we could derive a configuration with* $\mathscr{C}_2 = \emptyset$ *and* $\mathscr{D}_2 = \{p \geq 1\}$. *However, now the dominance rule can derive* $C \doteq \overline{p} \geq 1$*, using the witness* $\omega = \{p \mapsto 0\}$. *To see that all conditions for applying dominance-based strengthening are indeed satisfied, we notice that* (10a)–(10b) *simplify to*

$$\emptyset \cup \{p \geq 1\} \cup \{p \geq 1\} \vdash \emptyset \cup \{p + 1 \geq 1\} \cup \emptyset \tag{12a}$$

$$\emptyset \cup \{p \geq 1\} \cup \{p \geq 1\} \cup \{0 + \overline{p} \geq 1\} \vdash \bot \tag{12b}$$

and both of these derivations are immediate (e.g., by reverse unit propagation). This means that we can derive a third configuration with $\mathscr{C}_3 = \emptyset$ *and* $\mathscr{D}_3 = \{p \geq 1, \overline{p} \geq 1\}$*, from which we immediately get contradiction* $0 \geq 1$ *by adding the two constraints in* $\mathscr{D}_3$*, although the formula* F *that we started with is satisfiable.*

Remark 16. *The unchecked deletion rule introduced in Definition 10 requires that the derived set* $\mathscr{D}$ *be empty. Example 15 gives the motivation for this, highlighting the complex interplay between dominance-based strengthening and deletion of constraints. However, in the special case where the pre-order is the trivial order* $\mathscr{O}_{\top}$*, the dominance rule can only be used to derive implied constraints. In this case, we could in principle weaken the condition for unchecked deletion to also allow transitioning from* $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\top}, \vec{z}, v)$ *to any configuration* $(\mathscr{C}', \mathscr{D}, \mathscr{O}_{\top}, \vec{z}, v)$ *whenever* $\mathscr{C}' \subseteq \mathscr{C}$.

Allowing such a further relaxation of the deletion rule might be especially useful if one does not want to make a distinction between core and derived constraints in proof logging applications such as SAT solving. However, if we would want to extend the unchecked deletion rule to cover also this case, then such unchecked deletion transitions would no longer preserve weak validity. In addition, our invariant would need to be extended with a special case stating that item 2 in Definition 1 should hold only if $\mathscr{O}_{\preceq}$ *is nontrivial. For reasons of mathematical elegance, we chose not to adopt such a specially tailored version of the unchecked deletion rule in the current paper. We would like to point out, however, that such a version of unchecked deletion has been implemented in the formally verified pseudo-Boolean proof checker used in the SAT Competition 2023 (Bogaerts, McCreesh, Myreen, Nordström, Oertel, & Tan, 2023).*

3.7 Preorder Encodings

As mentioned before, $\mathscr{O}_{\preceq}$ is shorthand for a pseudo-Boolean formula $\mathscr{O}_{\preceq}(\vec{u},\vec{v})$ over two sets of placeholder variables $\vec{u} = \{u_1,\dots,u_n\}$ and $\vec{v} = \{v_1,\dots,v_n\}$ of equal size, which should also match the size of $\vec{z}$ in the configuration. To use $\mathscr{O}_{\preceq}$ in a proof, it is required to show that this formula encodes a preorder. This is done by providing (in a proof preamble) cutting planes derivations establishing

$$\emptyset \vdash \mathscr{O}_{\preceq}(\vec{u},\vec{u}) \tag{13a}$$

$$\mathscr{O}_{\preceq}(\vec{u},\vec{v}) \cup \mathscr{O}_{\preceq}(\vec{v},\vec{w}) \vdash \mathscr{O}_{\preceq}(\vec{u},\vec{w}) \tag{13b}$$

where (13a) formalizes reflexivity and (13b) transitivity (and where notation like $\mathscr{O}_{\preceq}(\vec{v},\vec{w})$ is shorthand for applying to $\mathscr{O}_{\preceq}(\vec{u},\vec{v})$ the substitution ω that maps u_i to v_i and v_i to w_i, as discussed in Section 2). These two conditions guarantee that the relation $\preceq$ defined by $\alpha \preceq \beta$ if $\mathscr{O}_{\preceq}(\vec{z}{\restriction_\alpha},\vec{z}{\restriction_\beta})$ forms a preorder on the set of assignments.

By way of example, to encode the lexicographic order $u_1u_2\dots u_n \preceq_{\text{lex}} v_1v_2\dots v_n$, we can use a single constraint

$$\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u},\vec{v}) \doteq \textstyle\sum_{i=1}^{n} 2^{n-i}\cdot(v_i-u_i) \geq 0\,. \tag{14}$$

Reflexivity is vacuously true since $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u},\vec{u}) \doteq 0 \geq 0$, and transitivity also follows easily since adding $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u},\vec{v})$ and $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{v},\vec{w})$ yields $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u},\vec{w})$ (where we tacitly assume that the constraint resulting from this addition is implicitly simplified by collecting like terms, performing any cancellations, and shifting any constants to the right-hand side of the inequality, as mentioned in Section 2).

A potential concern with encodings such as (14) is that coefficients can become very large as the number of variables in the order grows. It is perfectly possible to address this by allowing order encodings using auxiliary variables in addition to $\vec{u}$ and $\vec{v}$. We have chosen not to develop the theory for this in the current paper, however, since we feel that it would make the exposition significantly more complicated without providing a commensurate gain in terms of scientific contributions.

3.8 Order Change Rule

The final proof rule that we need is a rule for introducing a nontrivial order, and it turns out that it can also be convenient to be able to use different orders at different points in the proof. Switching orders is possible, but to maintain soundness it is important to first clear the set $\mathscr{D}$ (after transferring the constraints we want to keep to $\mathscr{C}$ using the transfer rule in Section 3.5). The reason for this is simple: if we allow arbitrary order changes, then item 2 of weak (F,f)-validity would no longer hold, and we could potentially derive contradiction even from satisfiable formulas. However, when $\mathscr{D} = \emptyset$ the condition in item 2 is trivially true.

Definition 17 (Order change rule)**.** *Provided that $\mathscr{O}_{\preceq^2}$ has been established to be a preorder over $2{\cdot}n$ variables (by proofs of* (13a) *and* (13b) *with explicit cutting planes derivations) and provided that $\vec{z}_2$ is a list of n variables, we say that we can transition from the configuration $(\mathscr{C},\emptyset,\mathscr{O}_{\preceq^1},\vec{z}_1,v)$ to the configuration $(\mathscr{C},\emptyset,\mathscr{O}_{\preceq^2},\vec{z}_2,v)$ by the* order change rule*.*

As explained above, it is clear that this rule preserves (F,f)-validity (and weak (F,f)-validity).

3.9 Concluding Remarks on the Proof System for Dominance Breaking

This concludes the presentation of our proof system, which we have defined in two slightly different flavours with different conditions for the deletion rule. In case we use the version with unchecked deletion, each rule in the proof system has been shown to preserve weak (F, f)-validity, where as in the version with (standard) deletion each derivation rule has been shown to preserve (F, f)-validity. The initial configuration is clearly (F, f)-valid. Therefore, by Theorem 2 our proof system is sound: whenever we can derive a configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ such that $\mathscr{C} \cup \mathscr{D}$ contains $0 \geq 1$, it holds that v is at least the value of f in any f-minimal solution for F (or, for a decision problem, we have $v = \infty$ only when F is unsatisfiable). If there are no applications of the unchecked deletion rule, then the final configuration is also (F, f)-valid, and hence by Theorem 3 it holds that v is *exactly* the value of f in any f-minimal solution for F (or, for a decision problem, we have $v = \infty$ if and only if F is unsatisfiable). As mentioned above, in this latter case the full sequence of proof configurations $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ together with annotations about the derivation steps—including, in particular, any witnesses ω—contains all information needed to efficiently reconstruct such an f-minimal solution for F. It is also straightforward to show that our proof system is complete: after using the bound update rule to log an optimal solution v^*, it follows from the implicational completeness of cutting planes that contradiction can be derived from $F \cup \{f \leq v^* - 1\}$.

Building on the ideas developed in this section, other variants of the proof system could be designed. For instance, we opted to define the relation $\preceq_f$ as

$$\alpha \preceq_f \beta \quad \text{if} \quad \alpha \preceq \beta \text{ and } f\restriction_\alpha \leq f\restriction_\beta\,, \tag{15}$$

but an alternative definition could be

$$\alpha \preceq'_f \beta \quad \text{if} \quad f\restriction_\alpha \leq f\restriction_\beta \text{ and if } f\restriction_\alpha = f\restriction_\beta \text{ then also } \alpha \preceq \beta\,. \tag{16}$$

What this says, essentially, is that we only compare assignments with respect to the order $\preceq$ in case they have the same objective value. Combining this with the intuition that a witness ω for the redundance rule should map each assignment α to an assignment that is at least as good (in terms of $\preceq'_f$) as α would here then yield the conditions

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash (\mathscr{C} \cup \mathscr{D} \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \text{ and} \tag{17a}$$
$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \{f\restriction_\omega = f\} \vdash \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \tag{17b}$$

for the redundance-based strengthening rule instead of (5). It can be observed that this actually results in slightly weaker proof obligations on the right-hand side and thus a more generally applicable rule. Similarly, for the dominance rule, the witness should map each assignment α to an assignment that is *strictly better* than α (again, in terms of $\preceq'_f$). This requirement would be encoded by the conditions

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \mathscr{C}\restriction_\omega \cup \{f\restriction_\omega \leq f\}\,, \tag{18a}$$
$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \{f\restriction_\omega = f\} \vdash \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z})\,, \text{ and} \tag{18b}$$
$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \{f\restriction_\omega = f\} \cup \mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot \tag{18c}$$

instead of (10a) and (10b). This again results in a slightly more generally applicable rule. All proofs of correctness go through with this slight modification of the proof system, and in fact would go through for any alternative to the order $\preceq_f$, as long as the redundance and dominance rules are adapted accordingly. In this paper we opted for using $\preceq_f$, prioritizing simplicity of exposition.

4. Symmetry Breaking in SAT Solvers

In this section we discuss the use of symmetry breaking in the context of Boolean satisfiability (SAT) solving and how our proof system can be used to provide efficiently verifiable proofs of correctness for symmetry breaking constraints. We also present results from a thorough empirical evaluation.

4.1 Certified Symmetry Breaking with Dominance-Based Strengthening

Symmetry handling has a long and successful history in SAT solving, with a wide variety of techniques considered by, e.g., Aloul, Sakallah, and Markov (2006), Benhamou and Saïs (1994), Benhamou, Nabhani, Ostrowski, and Saïdi (2010), Devriendt, Bogaerts, De Cat, Denecker, and Mears (2012), Devriendt, Bogaerts, and Bruynooghe (2017), Metin, Baarir, and Kordon (2019), and by Sabharwal (2009). These techniques were used to great effect in, e.g., the 2013 and 2016 editions of the *SAT competition*,[3] where the *SAT+UNSAT hard combinatorial track* and the *no-limit* track, respectively, were won by solvers employing symmetry breaking techniques. However, it later turned out that the victory in 2013 was partly explained by a small parser bug in the symmetry breaking tool, which could result in the solver taking a shortcut and declaring a formula unsatisfiable before even starting to solve it. For reasons such as this, proof logging is now obligatory in the main track of the SAT competition. While it is hard to overemphasize the importance of this development, and the value proof logging has brought to the SAT community, it has unfortunately also meant that it has not been possible to use symmetry breaking in the SAT competition, since there has been no way of efficiently certifying the correctness of such reasoning in *DRAT*. We will now explain how pseudo-Boolean reasoning with the dominance rule can provide proof logging for the *static symmetry breaking* techniques of Devriendt et al. (2016).

Let σ be a permutation of the set of literals in a given CNF formula F (i.e., a bijection on the set of literals), extended to (sets of) clauses in the obvious way. We say that σ is a *symmetry* of F if it commutes with negation, i.e., $\sigma(\bar{\ell}) = \overline{\sigma(\ell)}$, and preserves satisfaction of F, i.e., $\alpha \circ \sigma$ satisfies F if and only if α does. A *syntactic symmetry* in addition satisfies that $\sigma(F) \doteq F\restriction_\sigma \doteq F$. As is standard in symmetry breaking, we will only consider syntactic symmetries.

The most common way of breaking symmetries is by adding *lex-leader constraints* (Crawford, Ginsberg, Luks, & Roy, 1996). We will write $\preceq_{\text{lex}}$ to denote the lexicographic order on assignments induced by the sequence of variables $x_1, \ldots, x_m$, i.e., $\alpha \preceq_{\text{lex}} \beta$ if $\alpha = \beta$ or if there is an $i \leq m$ such that $\alpha(x_j) = \beta(x_j)$ for all $j < i$ and $\alpha(x_i) < \beta(x_i)$. Given a set G of symmetries of F, a lex-leader constraint is a formula ψ_{LL} such that α satisfies ψ_{LL} if and only if $\alpha \preceq_{\text{lex}} \alpha \circ \sigma$ for each $\sigma \in G$. The actual choice of a set G of symmetries to

3. `www.satcompetition.org`

break, as well as the choice of variables on which to define the lexicographic order, has a significant effect on the quality of the breaking constraints (Devriendt et al., 2016), but this is an orthogonal concern to the goals of the current paper, which is to show how to certify an encoding of lex-leader constraints using the dominance rule.

Let $\{x_{i_1}, \dots, x_{i_n}\}$ be the *support* of σ (i.e., all variables x such that $\sigma(x) \neq x$), ordered so that $i_j \leq i_k$ if and only if $j \leq k$. Then the constraints

$$y_0 \geq 1 \tag{19a}$$

$$\overline{y}_{j-1} + \overline{x}_{i_j} + \sigma(x_{i_j}) \geq 1 \qquad 1 \leq j \leq n \tag{19b}$$

$$\overline{y}_j + y_{j-1} \geq 1 \qquad 1 \leq j < n \tag{19c}$$

$$\overline{y}_j + \overline{\sigma(x_{i_j})} + x_{i_j} \geq 1 \qquad 1 \leq j < n \tag{19d}$$

$$y_j + \overline{y}_{j-1} + \overline{x}_{i_j} \geq 1 \qquad 1 \leq j < n \tag{19e}$$

$$y_j + \overline{y}_{j-1} + \sigma(x_{i_j}) \geq 1 \qquad 1 \leq j < n \tag{19f}$$

form a lex-leader constraint for σ (for fresh variables y_j). The intuition behind this encoding is as follows: the variable y_j is true precisely when α and $\alpha \circ \sigma$ are equal up to x_{i_j} (so y_0 is trivially true as claimed in (19a)). The constraint (19b) does the actual symmetry breaking: it states that if α and $\alpha \circ \sigma$ are equal up to $x_{i_{j-1}}$, then $x_{i_j} \leq \sigma(x_{i_j})$ must hold. The constraints (19c)–(19f) encode the definition of y_j as $y_{j-1} \wedge (x_{i_j} \leftrightarrow \sigma(x_{i_j}))$.

To derive the clausal constraints (19a)–(19f) in our proof system, assume that we have a configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{x}, v)$ where assignments are compared lexicographically on the subset of variables $\vec{x} = \{x_1, \dots, x_m\}$ according to $\mathcal{O}_{\preceq}$ as defined in (14). Let σ be a syntactic symmetry of $\mathscr{C}$ (i.e., such that $\mathscr{C}\restriction_\sigma \doteq \mathscr{C}$) with support contained in $\vec{x}$. As a first step, we use the dominance-based strengthening rule in Section 3.6 to derive the pseudo-Boolean constraint

$$C_{LL} \doteq \textstyle\sum_{i=1}^{m} 2^{m-i} \cdot (\sigma(x_i) - x_i) \geq 0 \tag{20}$$

expressing that $\sigma(\vec{x})$ is greater than or equal to $\vec{x}$. We emphasize that the constraint C_{LL} only exists in the proof and is nothing that the SAT solver will see—since it only understands clauses—but this constraint will help us to construct efficient derivations of the clausal constraints that will be used by the solver to enforce symmetry breaking.

To see how the constraint C_{LL} in (20) can be derived by the dominance rule, note first that in SAT solving we are dealing with decision problems, and so there is no need to worry about the trivial objective function $f \doteq 0$. Second, observe that we have $\mathcal{O}_{\preceq}(\vec{x}, \vec{x}\restriction_\sigma) \doteq \{C_{LL}\}$, According to Definition 13, we need to show that derivations as in (10a)–(10b) exist. To argue that (10a) holds, we note that $\neg C_{LL}$ expresses that $\vec{x}$ is strictly larger than $\sigma(\vec{x})$, and hence this implies $\mathscr{O}_{\preceq}(\vec{x}\restriction_\sigma, \vec{x})$. Since these are single pseudo-Boolean constraints, this is easily verifiable by literal-axiom implication. It is also easy to verify that (10b) is true, since we have both C_{LL} and its negation among the premises on the left-hand side, and for any pseudo-Boolean constraint C it holds that adding C and $\neg C$ together yields $0 \geq 1$.

Once we have C_{LL} as in (20), we can use this to obtain the constraints (19a)–(19f). Since all y_j-variables are fresh, it is straightforward to derive all constraints (19a) and (19c)–(19f) using redundance-based strengthening as explained by Gocht and Nordström (2021). It is important to note that in these derivations the witness substitutions only operate on the

new, fresh y_j-variables. Hence, we have $\vec{x}\restriction_\omega = \vec{x}$, and so $\mathscr{O}_\preceq(\vec{x}\restriction_\omega, \vec{x})$ holds by the reflexivity of $\mathscr{O}_\preceq$. A more interesting challenge is to derive the constraints (19b), and this is where we need C_{LL}.

Recall that our assumption is that the support of σ is $\{x_{i_1}, \ldots, x_{i_n}\}$ with $i_j \leq i_k$ if and only if $j \leq k$. Note first that for all variables x_i that are not in the support of σ, the difference $\sigma(x_i) - x_i$ disappears since $\sigma(x_i) = x_i$. This means that the constraint C_{LL} simplifies to

$$\textstyle\sum_{j=1}^{n} 2^{m-i_j} \cdot (\sigma(x_{i_j}) - x_{i_j}) \geq 0\,, \tag{21}$$

and this inequality can only hold if the contributions of the variables with the largest coefficient 2^{m-i_1} is non-negative. In other words, the constraint C_{LL} implies that $\sigma(x_{i_1}) - x_{i_1} \geq 0$, and this implication can be verified by reverse unit propagation (RUP). From this, in turn, we can obtain the constraint (19b) for $j = 1$ by literal-axiom implication.

To derive constraints (19b) for $j > 1$, let us introduce the notation

$$C_{LL}(0) \doteq C_{LL} \tag{22a}$$

$$C_{LL}(k) \doteq C_{LL}(k-1) + 2^{m-i_k} \cdot (19d\,[j = k]) \tag{22b}$$

where $(19d\,[j = k])$ denotes substitution of j by k in (19d). Simplifying $C_{LL}(k)$ yields

$$\textstyle\sum_{j=1}^{k} 2^{m-i_j} \cdot \overline{y}_j + \sum_{j=k+1}^{n} 2^{m-i_j} \cdot (\sigma(x_{i_j}) - x_{i_j}) \geq 0\,, \tag{23}$$

which, when combined with all constraints (19c), directly entails the constraint (19b) with $j = k$. To see this, note that if y_k is false, then (19b) is trivially true for $j = k+1$. On the other hand, if y_k is true, then so are all the preceding y_j-variables, and the dominant contribution in $C_{LL}(k)$ is from $\sigma(x_{i_k}) - x_{i_k}$, which implies (19b) for $j = k$ analogously to the case for $j = 1$.

It is important to note here that the order used for the dominance-based strengthening is fixed at the beginning and remains the same for all symmetries $\sigma \in G$ to be broken. Since constraints are added only to the derived set $\mathscr{D}$, dominance rule applications for different symmetries will not interfere with each other. Furthermore, in contrast to the approach of Heule et al. (2015), handling a symmetry once is enough to guarantee complete breaking. In Appendix A we include a complete worked-out example of symmetry breaking in *VeriPB* syntax together with explanations of how the proof logging syntax matches rules in our proof system.

4.2 Relation to *DRAT*-Style Symmetry Breaking

As discussed in the introduction, it is currently not known whether general symmetry breaking can be certified efficiently using *DRAT* proof logging. Previous work on symmetry breaking with *DRAT* (Heule et al., 2015) is limited to special cases of simple symmetries. To compare and contrast this with our work, let us describe the method of Heule et al. (2015) in our language (which is easily done, since the *RAT* rule is a special case of our redundance-based strengthening rule), and explain the difficulties in extending this to a general symmetry breaking method.

Redundance-based strengthening can easily deal with a single simple symmetry σ that swaps two variables, say x_i and x_j (with $j > i$). In this case, the constraint C_{LL} in (20)

simplifies to

$$x_j - x_i \geq 0\,, \tag{24}$$

which can be derived with a single application of the redundance rule with the symmetry σ serving as the witness. However, this approach no longer works when several symmetries need to be broken at the same time, or when we need to deal with more complex symmetries.

When multiple symmetries that swap two variables are involved, the second symmetry cannot necessarily be broken in the way described above, since the witness for the second symmetry might invalidate the symmetry breaking constraint for the first symmetry (which has been added to the set of core constraints, and so will appear as one of the proof obligations on the right-hand side in (5)). If there are two symmetries σ_1 and σ_2 that share a variable, then these symmetries might need to be applied three times in order to obtain a lexicographically minimal assignment. While this makes the proof logging more complicated, it is still possible to deal with this scenario in *DRAT* proof logging by making use of a sorting network encoding as explained by Heule et al. (2015).

A more serious problem is when the symmetries to be broken are more complex than just swaps of a single pair of variables. For the case of *involutions* σ (i.e., where σ is its own inverse), some steps towards a solution have been taken (Heule et al., 2015, Conjecture 1). But already a symmetry σ that is a cyclic shift of three variables (i.e., such that $\sigma(x) = y$, $\sigma(y) = z$, and $\sigma(z) = x$) brings us beyond what *DRAT*-based proof logging symmetry breaking is currently able to handle. The obstacle that arises here is that we do not know in advance whether the permutation σ should be applied once or twice to an assignment α to get the lexicographically smallest assignment in the orbit of α under σ.

The beauty of the dominance-based strengthening rule is that it completely eliminates these problems. There is no requirement that our witness substitutions should generate minimal assignments—all that is needed is that the witnesses yield smaller assignments. And since the symmetry breaking constraints are all added to the derived set $\mathscr{D}$, we do not need to worry about what previous symmetry breaking constraints might have been added when we are breaking the next symmetry. Instead, symmetry breaking constraints for different symmetries can be added independently of one another (for as long as the order remains unchanged).

4.3 Extensions of Basic Symmetry Breaking

So far we have discussed the core ideas that underlie most modern symmetry breaking tools for SAT solving. The tool *BreakID* (Devriendt et al., 2016) extends these ideas further in a couple of ways. We now briefly discuss these extensions and how they are dealt with in our proof system.

The most important contribution of Devriendt et al. (2016) is detecting so-called *row interchangeability*. The goal of this optimization is to not just take an arbitrary set of generators of the symmetry group and an arbitrary lexicographic order, but to choose "the right" set of generators and "the right" variable order (with respect to which to define the lexicographic order). Devriendt et al. (2016) showed that for groups that exhibit a certain structure, breaking symmetries of a good set of generators using an appropriate order can guarantee that the entire symmetry group is broken completely. Since our proof logging techniques simply use the same lexicographic order as the symmetry breaking tool, and

work for an arbitrary generator set, this automatically works with the techniques described above.

Another (optional) modification to the basic symmetry breaking techniques implemented in *BreakID* is the use of a more compact encoding, where the clauses (19c) and (19d) are omitted. Since our definition of $C_{LL}(k)$ uses these clauses, we cannot simply omit them in our proof. However, there are no restrictions on deletions from the derived set $\mathscr{D}$. Therefore, we can first derive all the symmetry breaking constraints as described above, and then remove the superfluous clauses from $\mathscr{D}$ as soon as they are no longer needed for the proof logging derivations.

Next, *BreakID* has an optimization based on *stabilizer subgroups* to detect a large number of binary clauses. Since these binary clauses are all clauses of the form (19b) with $j = 1$, the proof logging techniques described above could support also this optimization provided that we keep track of which symmetry is used for each such binary clause. However, *BreakID* currently does no such bookkeeping. While it is in principle possible to add this feature, we have not done so in this work.

Finally, *BreakID* supports *partial symmetry breaking*. That is, instead of adding the constraints (19b)–(19f) for every j, this is only done for $j < L$ with L a limit that can be chosen by the user. The reasoning behind this is that the larger j gets, the weaker the added breaking constraint is. By setting $L = 100$ and adding symmetry breaking clauses only for the L first variables, we add significantly fewer constraints while retaining most of the symmetry breaking power.

Since we only need to do proof logging for the clauses that are actually added by *BreakID*, this optimization works out-of-the-box. However, there is an important caveat here: for benchmark formulas where there are large symmetries, e.g., symmetries permuting all the variables in the problem, a naive implementation of our proof logging technique will suffer from serious performance problems even when this optimisation is used. The reason is that in principle the order $\mathscr{O}_{\preceq}$ is defined on all variables that are permuted by the symmetries. If there are many such variables, this order in itself can get huge (with the largest coefficient being of exponential magnitude measured in terms of the number of variables). Luckily, there is a simple solution to this problem: instead of defining the order on all variables that are permuted, we can use the set of variables on which we will actually do the symmetry breaking (which for each symmetry are the first L variables in its support). This is the approach implemented in our experimental evaluation.

4.4 Experimental Evaluation of SAT Symmetry Breaking

To validate our approach, we implemented pseudo-Boolean proof logging in the *VeriPB* proof format for the symmetry breaking method in *BreakID*, and modified *Kissat*[4] to output *VeriPB*-proofs (since the redundance rule is a generalization of the RAT rule, this required only purely syntactic changes). We employed the unchecked deletion rule in Definition 10 since the generated proofs are only used to certify unsatisfiability of decision problems and never to prove satisfiability.

Among the benchmark instances from all the SAT competitions between the years 2016 and 2020, we selected all instances in which at least one symmetry was detected; there were

4. `http://fmv.jku.at/kissat/`

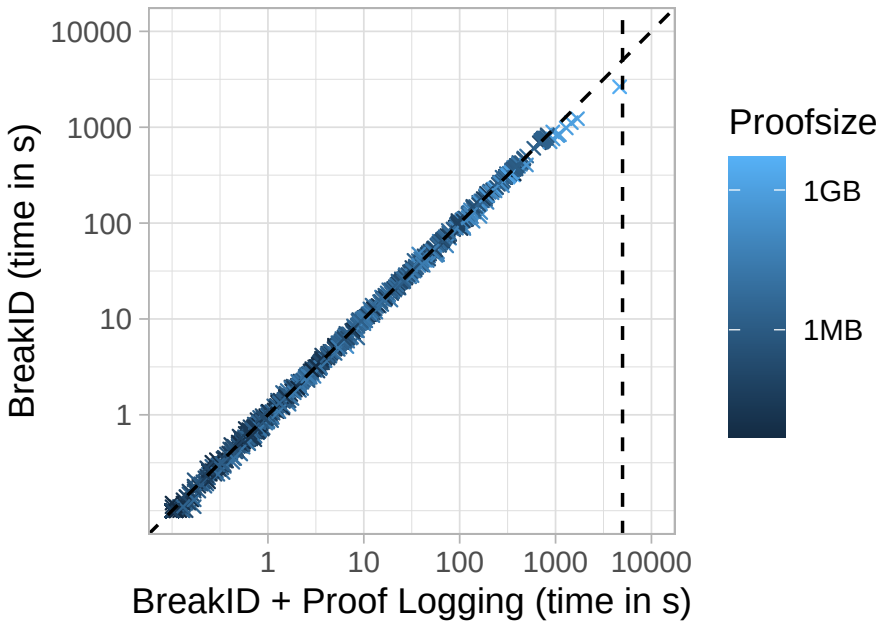

Figure 2: Time required for symmetry breaking with and without proof logging.

1089 such instances in total. We performed our computational experiments on machines with dual Intel Xeon E5-2697A v4 processors with 512GBytes RAM and solid-state drives (SSD) running on the Ubuntu 20.04 operating system. We ran twenty instances in parallel on each machine, where we limited each instance to 16GBytes RAM. We used a timeout of 5,000 seconds for solving and 100,000 seconds for verification of the proofs produced during the solving process.

Figure 2 shows the performance overhead for symmetry breaking, comparing for each instance the running time with and without proof logging. For most instances, the overhead is negligible (99% of instances are at most 32% slower). Figures 3 and 4 display the relationship between the time needed to generate a proof (both for SAT and UNSAT instances) and to verify the correctness of this proof. When considering symmetry breaking in isolation (i.e., not solving the instance completely, but only breaking the symmetries), as plotted in Figure 3, 1058 instances out of 1089 could be verified, 2 timed out, and 29 terminated due to running out of memory. 75% of the instances could be verified within 3.2 times the time required for symmetry breaking and 95% within a factor 20. The time needed for verification is thus considerably longer than the time required to generate the proofs, but still practical in the majority of cases. After symmetry breaking, 721 instances could be solved with the SAT solver (Figure 4) and we could verify 671 instances, while for 33 instances verification timed out and for 17 instances the verifier ran out of memory. Notably, 84 instances could only be solved by the SAT solver when symmetry breaking clauses were added, and out of these instances we could verify correctness for 81 instances.

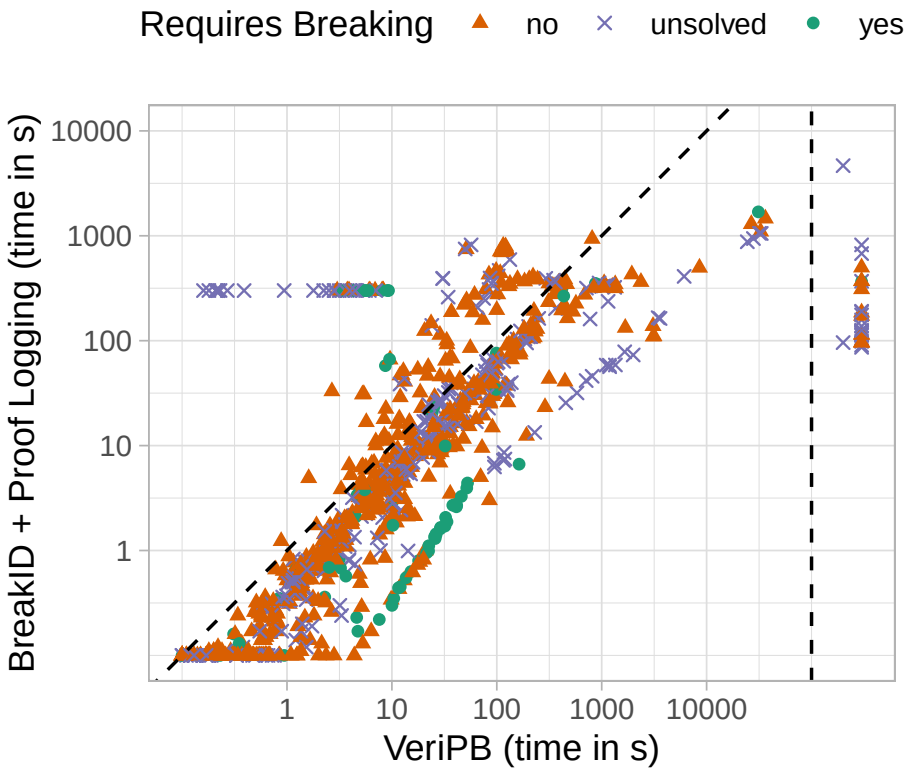

Figure 3: Time for symmetry breaking with proof logging compared to verification time for the generated symmetry breaking constraints. Points to the right of the vertical dashed line indicate timeouts (left) and out of memory (right).

5. Symmetries in Constraint Programming

In the general setting considered in constraint programming, we must deal with variables with larger (non-Boolean) domains and with rich constraints supported by propagation algorithms. One might think that a proof system based upon Boolean variables and linear inequalities would not be suitable for this larger class of problem. However, Elffers et al. (2020) showed how to use *VeriPB* for constraint satisfaction problems by first encoding variables and constraints in pseudo-Boolean form, and then constructing cutting planes proofs to certify the behaviour of the *all-different* propagator, and Gocht, McCreesh, and Nordström (2022) later extended this to a wider range of CP constraints and propagators. Similarly, the work we present here can also be applied to constraint satisfaction and optimisation problems.

Recall the list of symmetry breaking constraints proposed for the Crystal Maze puzzle in the introductory section. Given the difficulties in knowing which combinations of symmetry breaking constraints are valid, it would be desirable if these constraints could be *introduced as part of a proof*, rather than taken as part of the encoding of the original problem. This

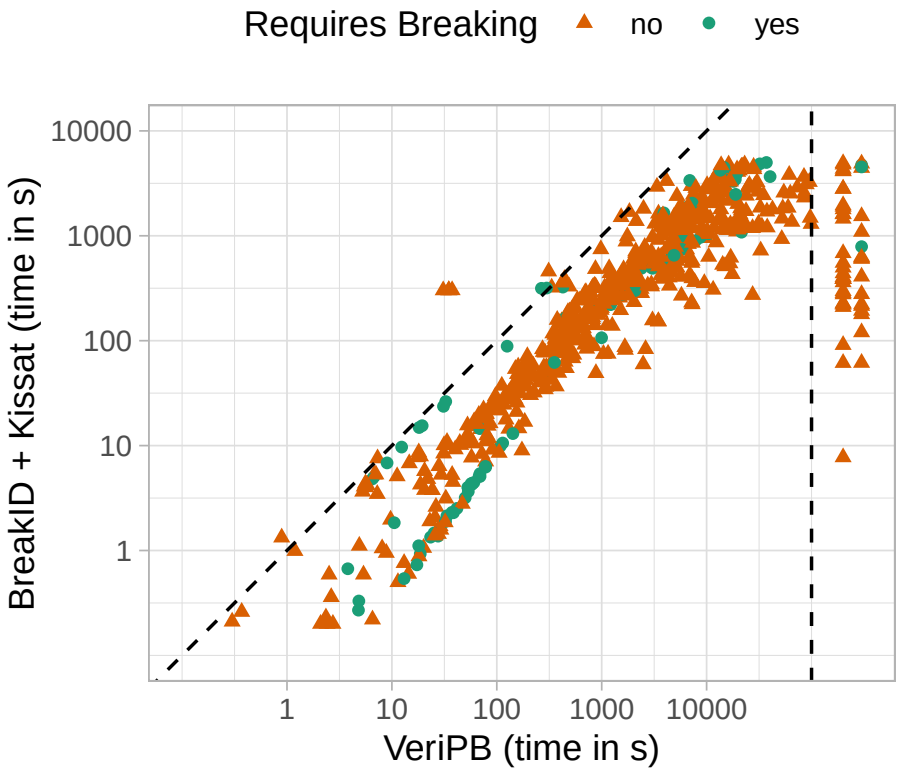

Figure 4: Time for symmetry breaking plus SAT solving (with proof logging) compared to verification time for the whole solving process. Points to the right of the vertical dashed line indicate timeouts (left) and out of memory (right).

would give a modeller immediate feedback as to whether the constraints have been chosen correctly. Our cutting planes proof system enhanced with redundance-based strengthening and dominance-based strengthening rules is indeed powerful enough to express all three of the examples we presented in the introduction, and we have implemented a small tool which can write out the appropriate proof fragments certifying that this is so; this allows the entire Crystal Maze example to be verified with *VeriPB*.

The source code to run our tool for the Crystal Maze puzzle is located in the `tools/crystal-maze-solver` directory of the code and data repository (Bogaerts et al., 2022b) associated to this paper. Full instructions for how to use the tool are given in the `tools/crystal-maze-solver/README.md` file. Below we provide a summary of our work on this problem.

We modelled the Crystal Maze puzzle as a constraint satisfaction problem in the natural way: there is a decision variable for each circle, whose values are the possible numbers that can be taken, and an all-different constraint over all decision variables. We use a table constraint for each edge for simplicity. We also included symmetry elimination constraints.

We implemented this model inside a small proof-of-concept CP solver (which can be found in the file `src/crystal_maze.cc`) that we created for this paper. (Full proof logging for CP is an entire research program in its own right, which we do not at all claim to have carried out within the framework of this project—what we do claim, though, is that our contribution shows that symmetries do not stand in the way of this work.) When executed, the solver compiles this high level CP model to a pseudo-Boolean model, which it will output as `crystal_maze.opb`. This is done following the framework introduced by Elffers et al. (2020), but as well as using a one-hot (direct) encoding of CP decision variables to pseudo-Boolean variables, it additionally creates channelled greater-or-equal PB variables for each CP variable-value. We remark that the encoding of the table constraints also introduces additional auxiliary variables.

As the solver works on the problem, it produces a file `crystal_maze.veripb` that provides a proof that it has found all non-symmetric solutions. The proof log will contain reverse unit propagation (RUP) clauses that justify backtracking, as well as cutting planes derivations that prove the correctness of propagations by the all-different and table constraints. Once the solver has finished, the correctness of the solver output can be checked by feeding the two files `crystal_maze.opb` and `crystal_maze.veripb` to *VeriPB*.

To verify that the symmetry constraints introduced in the high-level model are actually valid, we can remove them from the pseudo-Boolean model in `crystal_maze.opb` and introduce them as part of the proof instead. We describe how to do this editing in `README.md`. We also include a script `make-symmetries.py` that will output the necessary proof fragment to reintroduce the symmetry constraints. The output of this script can be verified on top of the reduced pseudo-Boolean model using our modified version of *VeriPB* supporting the dominance-based strengthening rule, and this verification can be performed with or without the remainder of the proof—that is, we can verify both that the constraints introduced are valid (in that they do not alter the satisfiability of the model), and that they line up with the actual execution of the solver.

Interestingly, although symmetries can be broken in different ways in high-level constraint programming models (including through lexicographic and value precedence constraints), when we encode the problem in pseudo-Boolean form these differences largely disappear, and after creating a suitable order we can easily re-use the SAT proof logging techniques for symmetry breaking that we discussed above. So, although a full proof-logging constraint solver does not yet exist, we can confidently claim that symmetry breaking should not block the road towards this goal.

6. Lazy Global Domination in Maximum Clique Solving

Gocht et al. (2020a) showed how *VeriPB* can be used to implement proof logging for a wide range of maximum clique algorithms, observing that the cutting planes proof system is rich enough to certify a rich variety of bound and inference functions used by various solvers (despite cutting planes not knowing what a graph or clique is). However, there is one clique-solving technique in the literature that is *not* amenable to cutting planes reasoning. In order to solve problem instances that arise from a distance-relaxed clique-finding problem, McCreesh and Prosser (2016) enhanced their maximum clique algorithm with a *lazy global domination* rule that works as follows. Suppose that the solver has constructed a candidate

clique C and is considering to extend C by two vertices v and w, where the neighbourhood of v excluding w is a (non-strict) superset of the neighbourhood of w excluding v. Then if the solver first tries v and rejects it, there is no need to branch on w as well (since any clique including w could be exchanged for at least as good a clique including v).

In principle, it should be possible to introduce additional constraints certifying this kind of reasoning in advance using redundance-based strengthening, without the need for the full dominance breaking framework in Section 3 (with some technicalities involving consistent orderings for tiebreaking). However, due to the prohibitive cost of computing the full vertex dominance relation in advance, McCreesh and Prosser instead implement a form of *lazy* dominance detection, which only triggers following a backtrack. To provide proof logging for this, we cannot derive the constraints justifying global domination steps in the proof log in advance, but must instead be able to introduce vertex dominance constraints on the fly precisely when they are used. It is hard to see how to achieve this with the redundance rule, but it is possible using dominance-based strengthening. In what follows, we first present a pseudo-Boolean encoding of the maximum clique problem and then a high-level description of the max clique solver used. Afterwards, we discuss how to add certification to it with the redundance rule and the dominance rule.

6.1 The Maximum Clique Problem

Throughout this section we let $G = (V, E)$ denote an undirected, unweighted graph without self-loops with vertices V and edges E. We write $N(u)$ to denote the *neighbours* of a vertex $u \in V$, i.e., the set of vertices $N(u) = \{w \mid (u, w) \in E\}$ that are adjacent to u in the graph, and define neighbours of sets of vertices in the natural way by taking unions $N(U) = \bigcup_{u \in U} N(u)$. We say that u *dominates* v if

$$N(u) \setminus \{v\} \supseteq N(v) \setminus \{u\} \tag{25}$$

holds, which intuitively says that the neighbourhood of u is at least as large as that of v. It is straightforward to verify that this domination relation is transitive.

When representing the maximum clique problem in pseudo-Boolean form, we overload notation and identify every vertex $v \in V$ with a Boolean variable, where $v = 1$ means that the vertex v is in the clique. The task is to maximize $\sum_{v \in V} v$ under the constraints that all chosen vertices should be neighbours, but since, syntactically speaking, we require an objective function to be minimized, we obtain

$$\min \textstyle\sum_{v \in V} \overline{v} \tag{26a}$$

$$\overline{v} + \overline{w} \geq 1 \qquad [\text{for all } (v, w) \in V^2 \setminus E \text{ with } v \neq w] \tag{26b}$$

as the formal pseudo-Boolean specification of the problem.

6.2 High-Level Description of the Max Clique Solver

At a high level, the maximum clique solver of McCreesh and Prosser (2016) *before* addition of vertex dominance breaking, works as described in Algorithm 1. The first call to the MaxCliqueSearch algorithm is with parameters G, $V_{\text{rem}} = V$, $C_{\text{curr}} = \emptyset$, and $C_{\text{best}} = \emptyset$.

Algorithm 1: Max clique algorithm without dominance.

```
1  MaxCliqueSearch(G, V_rem, C_curr, C_best) :
2  E_rem ← E(G) ∩ (V_rem × V_rem);
3  G_rem ← (V_rem, E_rem);
4  if |C_curr| > |C_best| then
5      C_best ← C_curr;
6  (S_1, ..., S_m) ← colour classes in colouring of G_rem ;
7  j ← m;
8  while j ≥ 1 and |C_curr| + j > |C_best| do
9      for v ∈ S_j do
10         C_best ← MaxCliqueSearch(G, V_rem ∩ N(v), C_curr ∪ {v}, C_best);
11     V_rem ← V_rem \ S_j;
12     j ← j − 1;
13 return C_best;
```

When MaxCliqueSearch is called with a candidate clique C_{curr}, the best solution so far C_{best}, and a subset of vertices V_{rem}, it considers the residual graph $G_{\text{rem}} = (V_{\text{rem}}, E_{\text{rem}})$ assumed to be defined on all vertices in $V \setminus C_{\text{curr}}$ that are neighbours of all $c \in C_{\text{curr}}$. Thus, the set V_{rem} contains all vertices to which C_{curr} could possibly be extended. The algorithm produces a colouring of G_{rem} (where as usual adjacent vertices are assigned a different colour), which we assume results in m disjoint colour classes $(S_1, \ldots, S_m)$ such that $V_{\text{rem}} = \bigcup_{i=1}^{m} S_i$. It is clear that any clique extending C_{curr} can contain at most one vertex from each colour class S_i, since vertices of the same colour class are non-adjacent. The clique search algorithm now iterates over all colour classes in the order $S_m, S_{m-1}, \ldots, S_1$. Whenever the clique is extended with a new vertex, a new recursive call to MaxCliqueSearch is made. Therefore, when we reach S_j in the loop, we are considering the case when all vertices in $S_m, S_{m-1}, \ldots, S_{j+1}$ have been rejected. For this reason, if the condition $|C_{\text{curr}}| + j > |C_{\text{best}}|$ fails to hold, we know that the current clique candidate cannot possibly be extended to a clique that is larger than what we have already found in C_{best}. At the end of the first call MaxCliqueSearch$(G, V, C_{\text{curr}} = \emptyset, C_{\text{best}} = \emptyset)$, after completion of all recursive subcalls, the vertex set C_{best} will be a clique of maximum size in G. A certifying version of essentially this algorithm with *VeriPB* proof logging was presented by Gocht et al. (2020a). It might be worth noting in this context that one quite interesting challenge is to certify the backtracking performed when the condition $|C_{\text{curr}}| + j > |C_{\text{best}}|$ fails, and this is one place where the strength of the pseudo-Boolean reasoning in the cutting planes proof system is very helpful (as opposed to the clausal reasoning in, e.g., *DRAT*, where certifying this type of counting arguments is quite challenging).

The vertex dominance breaking of McCreesh and Prosser (2016) is based on the following observation: If the algorithm is about to consider $v \in S_j$ in the innermost for loop on line 9 in Algorithm 1, but has previously considered a vertex $u \in \bigcup_{i=j}^{m} S_i$ that dominates v in the sense of (25), then it is safe to ignore v. This is so since if the algorithm would find a solution that includes v but not u, then we could swap u for v and obtain a solution that is at least as good.

In pseudo-Boolean notation, this type of reasoning could be enforced by adding the constraint $u + \overline{v} \geq 1$ to the formula, but there is no way this can be semantically derived from the constraints (26a) or the requirement to minimize (26b). Therefore, the proof logging method in (Gocht et al., 2020a) is inherently unable to deal with such constraints.

In general, the vertex dominance breaking as described above does not need to break ties consistently. By this we mean that if u and v dominate each other, in principle it might happen that in a given branch of the search tree, u is chosen to dominate v, while in another one, v is chosen to dominate u, simply because of the order in which nodes are considered. While in principle, it should be possible to adapt our proof logging methods to work in this case, the argument is subtle. Luckily, it turns out that in practical implementations, tie breaking only happens in a consistent manner.

Fact 18. *In the vertex dominance breaking of McCreesh and Prosser (2016), there exists a total order $\succ_G$ on the set V of vertices such that whenever v is ignored because u has previously been considered, $u \succ_G v$.*

Moreover, this order $\succ_G$ is known before the algorithm starts: $u \succ_G v$ holds if u has a larger degree than v, or in case they have the same degree but the identifier used to represent u internally is larger than that of v. To see that this order indeed guarantees consistent tie breaking, we provide some properties of the actual implementation of the algorithm.[5]

1. If u and v dominate each other and are not adjacent, then u and v are guaranteed to be in the same colouring class. If furthermore $u \succ_G v$, u is considered before v in the loop in Line 8 (due to the order in which this for loop iterates over the nodes).

2. If u and v dominate each other, *are* adjacent, and satisfy $u \succ_G v$, then u is assigned a *larger* colouring class than v (due to the order in which the greedy colouring algorithm in Line 6 iterates over the nodes). Hence, also in this case u will be considered before v.

In what follows below, we will explain:

- first, how the redundance rule introduced to *VeriPB* by Gocht and Nordström (2021) could in principle be used to provide proof logging for vertex dominance breaking, although with potentially impractical overhead; and

- then, how the dominance rule introduced in this paper can be used to resolve the practical problems in a very simple way.

An implementation for both techniques can be found in the code and data repository (Bogaerts et al., 2022b).

5. When inspecting the two conditions below, we can see that they rely on a very subtle argument, namely that when constructing a greedy colouring, we should iterate over all nodes with the same degree in the *opposite* order of the way we iterate over nodes in line 8. This (strange) condition is satsisfied by many max clique algorithms due to a happy coincidence of an efficient data structure and colouring algorithm introduced by Tomita and Kameda (2007).

6.3 Vertex Dominance with the Redundance-Based Strengthening Rule

In order to provide proof logging for vertex dominance breaking using the redundance rule, we could in theory proceed as follows. First, we let the solver check the vertex dominance condition (25) for all pairs of vertices u, v in V. Before starting the solver, we add all pseudo-Boolean constraints for vertex dominance breaking using the redundance rule. For all u, v such that u dominates v and $u \succ_G v$, we derive the *vertex dominance breaking constraint*

$$u + \overline{v} \geq 1\,, \tag{27}$$

doing so *in decreasing order* for u with respect to $\succ_G$. Our witness for the redundance rule derivation of (27) will be $\omega = \{u \mapsto v, v \mapsto u\}$, i.e., ω will simply swap the dominating and dominated vertices. Hence, the objective function (26a) is syntactically unchanged after substitution by ω, and so the condition in (5) that the objective should not increase is always vacuously satisfied.

We need to argue that deriving the vertex dominance breaking constraints (27) is valid in our proof system. Towards this end, suppose we are in the middle of the process of adding such constraints and are currently considering $u + \overline{v} \geq 1$ for u dominating v and $u \succ_G v$. Let $\mathscr{C} \cup \mathscr{D}$ be the set of constraints in the current configuration. In order to add $u + \overline{v} \geq 1$, we need to show that

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg(u + \overline{v} \geq 1)\} \tag{28a}$$

can be used to derive all constraints in

$$(\mathscr{C} \cup \mathscr{D} \cup \{u + \overline{v} \geq 1\})\restriction_\omega \tag{28b}$$

by the cutting planes method (i.e., without any extension rules).

Starting with the vertex dominance constraint being added, note that from the negated constraint $\neg(u + \overline{v} \geq 1) \doteq \overline{u} + v \geq 2$ in (28a) we immediately obtain

$$\overline{u} \geq 1 \tag{29a}$$

$$v \geq 1 \tag{29b}$$

as reverse unit propagation (RUP) constraints, meaning that the weaker pseudo-Boolean constraint $(u + \overline{v} \geq 1)\restriction_\omega \doteq v + \overline{u} \geq 1$ is also RUP with respect to the constraints in (28a).

Consider next any non-edge constraints $\overline{x} + \overline{y} \geq 1$ in (26b) in the original formula. Clearly, such constraints are only affected by ω if $\{u, v\} \cap \{x, y\} \neq \emptyset$; otherwise they are present in both (28a) and (28b) and there is nothing to prove. Any non-edge constraint $\overline{v} + \overline{y} \geq 1$ containing $\overline{v}$ will after application of ω contain $\overline{u}$, and will hence be RUP with respect to (29a) and hence also with respect to (28a). For non-edge constraints $\overline{u} + \overline{y} \geq 1$ with $y \neq v$, substitution by ω yields $(\overline{u} + \overline{y} \geq 1)\restriction_\omega \doteq \overline{v} + \overline{y} \geq 1$. Since by assumption u dominates v and $y \neq v$ is not a neighbour of u, it follows from (25) that y is not a neighbour of v either. Hence, the input formula in (28a) already contains the desired non-edge constraint $\overline{v} + \overline{y} \geq 1$.

It remains to analyse what happens to vertex dominance breaking constraints

$$x + \overline{y} \geq 1 \tag{30}$$

that have already been added to $\mathscr{D}$ before the dominance breaking constraint $u + \overline{v} \geq 1$ that we are considering now. Again, such a constraint is only affected by ω if $\{u, v\} \cap \{x, y\} \neq \emptyset$; otherwise it is present in both (28a) and (28b). We obtain the following case analysis.

1. $x = u$: In this case, $(x + \overline{y} \geq 1)\restriction_\omega \doteq v + \overline{\omega(y)} \geq 1$, which is RUP with respect to $v \geq 1$ in (29b) and hence also with respect to (28a).

2. $x = v$: This is impossible, since $u \succ_G v$ and any dominance breaking constraints with $v = x$ as the dominating vertex will be added only once we are done with u as per the description right below (27).

3. $y = u$: In this case x dominates u. Since $x \succ_G u$, $u \succ_G v$, and u dominates v, by transitivity we have $x \succ_G v$ and also that x dominates v. Hence, the breaking constraint $x + \overline{v} \geq 1$ has already been added to $\mathscr{D}$. But since $u \neq x \neq v$, we see that our desired constraint is $(x + \overline{u} \geq 1)\restriction_\omega \doteq x + \overline{v} \geq 1$, which is precisely this previously added constraint.

4. $y = v$: Here we see that the desired constraint $(x + \overline{y} \geq 1)\restriction_\omega \doteq \omega(x) + \overline{u} \geq 1$ is again RUP with respect to (28a).

This concludes our proof that all vertex dominance breaking constraints that are consistent with our constructed linear order $\succ_G$ can be added and certified by the redundance rule before the solvers starts searching for cliques.

So all of this works perfectly fine in theory. The problem that rules out this approach in practice, however, is that the solver will not have the time to compute the dominance relation between vertices in advance, since this is far too costly and does not pay off in general. Instead the solver designed by McCreesh and Prosser (2016) will detect and apply vertex dominance relations on the fly during search. And from a proof logging perspective this is too late—during search, when $\mathscr{C} \cup \mathscr{D}$ will also contain constraints certifying any backtracking made, our proof logging approach above no longer works. The constraints added to the proof log to certify backtracking are no longer possible to derive when substituted by ω as in (28b), for the simple reason that they are not semantically implied by (28a). One possible way around this would be to run the solver twice—the first time to collect all information about what vertex dominance breaking will be applied, and then the second time to do the actual proof logging—but this seems like quite a cumbersome approach.

We deliberately discuss this problem in some detail here, because this is an example of an important and nontrivial challenge that shows up also in other settings when designing proof logging for other algorithms. It is not sufficient to just come up with a proof logging system that is strong enough in principle to certify the solver reasoning (which the redundance rule is for the clique solver with vertex dominance breaking, as shown above). It is also crucial that the solver have enough information available at the right time and can extract this information efficiently enough to actually be able to emit the required proof logging commands with low enough overhead. For constraint programming solvers, it is not seldom the case that the solver knows for sure that some variable should propagate to a value, because the domain has shrunk to a singleton, or that the search should backtrack because some variable domain is empty, but that the solver cannot reconstruct the detailed derivation steps required to certify this without incurring a massive overhead in running time (e.g.,

since the reasoning has been performed with bit-parallel logical operations). It is precisely for this reason that it is important that our proof system allow adding RUP constraints. This makes it possible for the solver to claim facts that it knows to be true, and that it knows can be easily verified, while leaving the work of actually producing a detailed proof to the proof checker.

6.4 Vertex Dominance with the Dominance-Based Strengthening Rule

Let us now discuss how the dominance rule can be used to provide proof logging for lazy global domination. As was the case for the redundance rule, we will make use of Fact 18. Before starting the proof logging, we use the order change rule to activate the lexicographic order on the the assignments to the vertices/variables induced by $\succ_G$.

Suppose now that the solver is running and that the current candidate clique is C_{curr}. The solver has an ordered list of unassigned candidate vertices that it is iterating over when considering how to enlarge this clique, and this list is defined by the colour classes $(S_m, S_{m-1}, \dots, S_1)$. (We note that this ordered list depends on C_{curr}, and would be different for a different clique C'_{curr}.) Suppose the next vertex in that list is v. Then when it is time to make the next decision on line 9 in Algorithm 1 about enlarging the clique, the solver enhanced with proof logging does the following:

- If there exists a vertex u that has already been considered in the current iteration and that dominates v (and for which it hence holds that $u \succ_G v$), then the solver discards v by vertex dominance and adds the constraint $u + \overline{v} \geq 1$ by the dominance rule with witness $\omega = \{u \mapsto v, v \mapsto u\}$. We will explain in detail below why this is possible.
- Otherwise, the solver enlarges C_{curr} with v and makes a recursive call.

When the solver has explored all ways of enlarging C_{curr} and is about to backtrack, here is what will happen on the proof logging side (where we refer to (Gocht et al., 2020a) for a more detailed description of how proof logging for backtracking CP solvers works in general):

1. For every u that was explored in an enlarged clique $C_{\text{curr}} \cup \{u\}$, when backtracking the solver will already have added $\overline{u} + \sum_{w \in C_{\text{curr}}} \overline{w} \geq 1$ as a RUP constraint.
2. The solver now inserts the explicit cutting planes derivation required to show that the inequality $|C_{\text{curr}}| + j > |C_{\text{best}}|$ must hold.
3. After this, the solver adds the claim that $\sum_{w \in C_{\text{curr}}} \overline{w} \geq 1$ is a RUP constraint.

We need to argue why $\sum_{w \in C_{\text{curr}}} \overline{w} \geq 1$ will be accepted as a RUP constraint, allowing the solver to backtrack. The RUP check for $\sum_{w \in C_{\text{curr}}} \overline{w} \geq 1$ propagates $w = 1$ for all $w \in C_{\text{curr}}$. This in turn propagates $u = 0$ for all explored vertices u by the backtracking constraints for $C_{\text{curr}} \cup \{u\}$ added in step 1. The vertex dominance breaking constraints then propagate $v = 0$ for all vertices v discarded because of vertex domination. At this point, the proof checker has the same information that the solver had when it detected that the colouring constraint forced backtracking. This means that the proof checker will unit propagate to

contradiction, and so the backtracking constraint $\sum_{w \in C_{\text{curr}}} \overline{w} \geq 1$ is accepted as a RUP constraint.

We still need to explain how and why the pseudo-Boolean dominance rule applications allow deriving the constraint $u + \overline{v} \geq 1$ in case u dominates v (and hence $u \succ_G v$). Recall that the order used in our proof is the lexicographic order induced by $\succ_G$. This means that if vertices/variables u and v are assigned by α in such a way as to violate a dominance breaking constraint $u + \overline{v} \geq 1$, then $\alpha \circ \omega$ will flip u to 1 and v to 0 to produce a lexicographically smaller assignment (since v is considered before u in the lexicographic order).

The conditions for the dominance rule are that we have to exhibit proofs of (10a) and (10b). In this discussion, let us focus on (10a) which says that starting with the constraints

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg(u + \overline{v} \geq 1)\} \tag{31a}$$

and using only cutting planes rules, we should be able to derive

$$\mathscr{C}\restriction_\omega \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \leq f\}\,. \tag{31b}$$

Note first that our lexicographic order in fact does *not* in itself respect the objective function (26b). However, since ω just swaps two variables it leaves the objective syntactically unchanged, meaning that the inequality $f\restriction_\omega \leq f$ in (10a) is seen to be trivially true.

As in our analysis of the redundance rule, from (31a) we obtain $\overline{u} \geq 1$ and $v \geq 1$ as in (29a)–(29b), and $\mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z})$ is easily verified to be RUP with respect to these constraints, since what the formula says after cancellation is precisely that $v \geq u$.

It remains to consider the pseudo-Boolean constraints in the solver constraint database $\mathscr{C} \cup \mathscr{D}$. The crucial difference from the redundance rule is that we no longer have to worry about proving $\mathscr{D}\restriction_\omega$ in (31b)—we only need to show how to derive $\mathscr{C}\restriction_\omega$. But this means that all we need to consider are the non-edge constraints in (26b). We already explained in our analysis for the redundance rule derivation that the fact that u dominates v means that for any non-edge constraints affected by ω their substituted versions are already there as input constraints or are easily seen to be RUP constraints. In addition to these non-edge constraints there might also be all kinds of interesting derived constraints in $\mathscr{D}$, but the dominance rule says that we can ignore those constraints.

Finally, although we skip the details here, it is not hard to argue analogously to what has been done above to show that $\neg C \doteq \neg(u + \overline{v} \geq 1)$ and $\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ in (10b) together unit propagate to contradiction. This concludes our discussion of how to certify vertex dominance breaking in the maximum clique solver by McCreesh and Prosser (2016) using the pseudo-Boolean dominance rule introduced in this paper.

7. Conclusion

In this paper, we introduce a method for showing the validity of constraints obtained by symmetry or dominance breaking by adding simple, machine-verifiable certificates of correctness. Using as our foundation the cutting planes method (Cook et al., 1987) for reasoning about pseudo-Boolean constraints (also known as 0–1 linear inequalities), and building on and extending the version of the *VeriPB* tool developed by Gocht and Nordström (2021), we present a proof logging method in which symmetry and dominance breaking is easily

expressible. Our method is a strict extension of *DRAT* (Heule et al., 2013a, 2013b; Wetzler et al., 2014) and other similar methods earlier used for solver proof logging, which means that we can produce efficient proofs of validity for SAT solving with fully general symmetry breaking, and we provide a thorough evaluation showing that this approach is feasible in practice. Since *VeriPB* can also certify cardinality and parity (XOR) reasoning, we now have for the first time a unified proof logging method for SAT solvers using all of these enhanced solving methods. To demonstrate that our proof logging approach is not limited to Boolean satisfiability, in this work we also present applications to symmetry breaking in constraint programming and vertex domination in maximum clique solving.

From a theoretical point of view, it would be interesting to understand better the power of the dominance-based strengthening rule. *DRAT* viewed as a proof system is closely related to *extended resolution* (Tseitin, 1968) in that these two proof systems have the same proof power up to polynomial factors (Kiesl, Rebola-Pardo, & Heule, 2018), and extended resolution, in turn, is polynomially equivalent to the *extended Frege* proof system (Cook & Reckhow, 1979), which is one of the strongest proof systems studied in proof complexity. However, there are indications that the cutting planes proof system equipped with the dominance-based strengthening rule might be strictly stronger than extended Frege (Kołodziejczyk & Thapen, 2023).

Another question is whether such a strong derivation rule as dominance-based strengthening is necessary to generate efficient proofs of validity for general symmetry breaking constraints, or whether redundance-based strengthening is enough. To phrase this cleanly as a proof complexity problem, we can restrict our attention to decision problems and also fix the order to be lexicographic order over some set of variables $\vec{x}$. Let us define cutting planes with symmetry breaking to be the cutting planes proof system extended with a rule that allows to derive the constraint $\vec{x} \preceq_{\text{lex}} \vec{x}\restriction_\sigma$ for any symmetry σ of the input formula F. Now we can ask whether cutting planes with redundance-based strengthening efficiently simulates this cutting planes proof system with symmetry breaking. To the best of our understanding, this question is wide open.

From a proof logging perspective, a natural next problem to investigate is whether our certification method is strong enough to capture other solving techniques such as those used for SAT-based optimisation in so-called MaxSAT solvers. A crucial component of several MaxSAT solving techniques is the translation of pseudo-Boolean constraints to CNF. This is used in linear SAT-UNSAT (LSU) solvers for adding the objective-improving constraints (e.g., Koshimura, Zhang, Fujita, & Hasegawa, 2012; Paxian & Becker, 2022), in core-guided solvers for reformulation of the objective function (e.g., Ignatiev, Morgado, & Marques-Silva, 2019), and in implicit hitting set solvers that make use of abstract cores (Berg, Bacchus, & Poole, 2020). While developing proof logging methods that can support the full range of modern MaxSAT solving techniques remains a formidable challenge, we want to point out that significant progress has been made of late. The proof system introduced by Gocht and Nordström (2021) has been used to certify correctness of the translations of pseudo-Boolean constraints into CNF for a range of encodings (Gocht et al., 2022), and quite similar ideas have been employed to design proof logging for an objective-improving solver for unweighted MaxSAT (Vandesande, De Wulf, & Bogaerts, 2022). Very recently, this has been extended also to state-of-the-art core-guided MaxSAT solvers (Berg, Bogaerts, Nordström, Oertel, & Vandesande, 2023).

Another intriguing problem is how to design efficient proof logging for *symmetric learning* as explored by Devriendt et al. (2017). In contrast to the symmetry breaking techniques considered in the current paper, when using symmetric learning one can only derive constraints that are semantically implied by the input formula. However, if at some point the solver derives a constraint D, and if σ is a symmetry of the formula, then it is clear that the permuted constraint $\sigma(D)$ is also implied by the formula and so should be sound to derive in a single extra step.

If we wanted to argue formally about such symmetric learning, we could apply σ to the whole derivation leading up to D to get a proof that $\sigma(D)$ can be derived. Taking this observation one step further, if from the solver execution we can extract a subderivation π showing that a constraint D is implied by constraints $C_1, \ldots, C_m$, and if $\sigma(C_1), \ldots, \sigma(C_m)$ have also been derived, then it should be valid to use π as a "lemma" to conclude $\sigma(D)$. The usefulness of such lemmas in the context of proof logging has been discussed by, e.g., Kraiczy and McCreesh (2021). The question, however, is whether such reasoning can be incorporated in the *VeriPB* framework with the current set of derivation rules in the proof system. One way of attempting to do this could of course be to add special rules for symmetric learning or lemmas, but this goes against the goal of keeping the proof logging system as simple as possible, so that derivations are obviously sound. As a case in point, it is worth noting that the naive way of adding such dedicated rules for symmetries or lemmas to our redundance-based strengthening and dominance-based strengthening rules would result in an unsound proof system.

The use of lemmas in proofs has been studied in the so-called *substitution Frege* proof system (Cook & Reckhow, 1979), and it has been shown that extended Frege has the same deductive strength as substitution Frege except possibly for a polynomial overhead in proof size (Krajíček & Pudlák, 1989). However, it is not clear whether such results can be scaled down from Frege systems to cutting planes, so that cutting planes with redundance-based strengthening can be made to simulate the usage of lemmas as described above, and whether such reasoning could be implemented efficiently enough in *VeriPB* to allow a formalization of reasoning with lemmas that would be feasible in practice.

As noted above, we have also shown in this work that our proof logging techniques can be used beyond SAT solving and SAT-based optimization for other combinatorial solving paradigms such as graph solving algorithms and constraint programming (CP), and Elffers et al. (2020) and Gocht et al. (2022) have made important contributions towards providing *VeriPB*-style proof logging for a full-blown CP solver. For mixed integer linear programming (MIP) solvers, Cheung, Gleixner, and Steffy (2017) and Eifler and Gleixner (2021) have also developed limited proof logging support (using another proof format), but this method still seems quite far from being able to support advanced MIP techniques. It would be very exciting if all of these different proof logging techniques could be strengthened to provide full proof logging support for state-of-the-art CP and MIP solvers.

Finally, we want to point out that another important research direction in proof logging is to develop formally verified proof checkers, so that we can be sure when proof verification passes that this is not due to bugs in the proof checker but that the claimed result is guaranteed to be valid. Such verified checkers have been built for *DRAT* and other clausal proof logging systems (Cruz-Filipe et al., 2017b, 2017a; Lammich, 2020; Tan, Heule, & Myreen, 2021), and it would be highly desirable to obtain such tools for pseudo-Boolean

proof logging with *VeriPB* as well. As a first step in this direction, a formally verified pseudo-Boolean proof checker for decision problems was recently submitted to and used in the SAT Competition 2023 (Bogaerts et al., 2023).

Acknowledgements

The authors gratefully acknowledge fruitful and stimulating discussions on proof logging with Jeremias Berg, Armin Biere, Jo Devriendt, Jan Elffers, Ambros Gleixner, Marijn Heule, Daniela Kaufmann, Daniel Le Berre, Matthew McIlree, Magnus Myreen, Yong Kiam Tan, James Trimble, and many other colleagues whom we have probably forgotten and to whom we apologize. We are also grateful to the anonymous *AAAI* and *JAIR* reviewers, whose comments helped us to improve the exposition considerably and also to fix some mistakes in the definitions.

Part of this work was carried out while taking part in the semester program *Satisfiability: Theory, Practice, and Beyond* in the spring of 2021 at the Simons Institute for the Theory of Computing at UC Berkeley, and in the extended reunion of this semester program in the spring of 2023. This work has also benefited greatly from discussions during the Dagstuhl Seminars 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond*.

Stephan Gocht and Jakob Nordström were supported by the Swedish Research Council grant 2016-00782, and Jakob Nordström also received funding from the Independent Research Fund Denmark grant 9040-00389B. Ciaran McCreesh was supported by a Royal Academy of Engineering research fellowship. Bart Bogaerts was supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G0B2221N), by the Flemish Government (AI Research Program), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

Appendix A. A Proof Logging Example for Symmetry Breaking

In this appendix, we present a fully worked-out example of symmetry breaking using the dominance-based strengthening rule for the well-known pigeonhole principle formulas claiming that $n + 1$ pigeons can be mapped to n pigeonholes in a one-to-one fashion. Haken (1985) showed that resolution proofs of unsatisfiability for such formulas requires a number of clauses that scales exponentially with n, and since conflict-driven clause learning SAT solvers can be seen to search for resolution proofs (Beame, Kautz, & Sabharwal, 2004), this means that solvers without enhanced reasoning methods will have to run for exponential time. However, since pigeonhole principle formulas are fully symmetric with respect to both pigeons and holes, it is possible to add symmetry breaking constraints encoding that without loss of generality pigeon 1 resides in hole 1, pigeon 2 resides in hole 2, et cetera, and once such symmetry breaking constraints have been added the problem becomes trivial.

What we show in this appendix is how the *BreakID* tool can break pigeonhole principle symmetries in a fully automated fashion, and produce proofs of correctness for the symmetry breaking clauses that the proof checker will accept. The proofs for these clauses can then be concatenated with the SAT solver proof log (rewritten from *DRAT* to *VeriPB*-format) and fed to *VeriPB* to provide end-to-end verification for the whole solving process.

We present the symmetry breaking proof logging in the syntactic format used by *VeriPB*, so that the reader will be able to see what pseudo-Boolean proof files actually look like. To keep the example manageable, we consider a very small instance of the pigeonhole principle with only 4 pigeons and 3 pigeonholes. We encode the pigeonhole principle formula using variables p_{ij}, with the intended interpretation that p_{ij} is true if pigeon i resides in hole j. The input for the symmetry breaking preprocessor consists of a CNF formula which can be written in pseudo-Boolean form as

$$p_{11} + p_{12} + p_{13} \geq 1 \tag{C1}$$
$$p_{21} + p_{22} + p_{23} \geq 1 \tag{C2}$$
$$p_{31} + p_{32} + p_{33} \geq 1 \tag{C3}$$
$$p_{41} + p_{42} + p_{43} \geq 1 \tag{C4}$$
$$\overline{p}_{11} + \overline{p}_{21} \geq 1 \tag{C5}$$
$$\overline{p}_{11} + \overline{p}_{31} \geq 1 \tag{C6}$$
$$\overline{p}_{11} + \overline{p}_{41} \geq 1 \tag{C7}$$
$$\overline{p}_{21} + \overline{p}_{31} \geq 1 \tag{C8}$$
$$\overline{p}_{21} + \overline{p}_{41} \geq 1 \tag{C9}$$
$$\overline{p}_{31} + \overline{p}_{41} \geq 1 \tag{C10}$$
$$\overline{p}_{12} + \overline{p}_{22} \geq 1 \tag{C11}$$
$$\overline{p}_{12} + \overline{p}_{32} \geq 1 \tag{C12}$$
$$\overline{p}_{12} + \overline{p}_{42} \geq 1 \tag{C13}$$
$$\overline{p}_{22} + \overline{p}_{32} \geq 1 \tag{C14}$$
$$\overline{p}_{22} + \overline{p}_{42} \geq 1 \tag{C15}$$
$$\overline{p}_{32} + \overline{p}_{42} \geq 1 \tag{C16}$$
$$\overline{p}_{13} + \overline{p}_{23} \geq 1 \tag{C17}$$
$$\overline{p}_{13} + \overline{p}_{33} \geq 1 \tag{C18}$$
$$\overline{p}_{13} + \overline{p}_{43} \geq 1 \tag{C19}$$
$$\overline{p}_{23} + \overline{p}_{33} \geq 1 \tag{C20}$$
$$\overline{p}_{23} + \overline{p}_{43} \geq 1 \tag{C21}$$
$$\overline{p}_{33} + \overline{p}_{43} \geq 1 \tag{C22}$$

where constraints (C1)–(C4) represent that each pigeon resides in at least one hole, and constraints (C5)–(C22) enforce that each hole is occupied by at most one pigeon (by specifying for every pigeonhole and every pair of distinct pigeons that it cannot be the case that both of these pigeons reside in the hole). When *VeriPB* is used for SAT proof logging, the proof checker parses CNF formulas in the standard DIMACS format used by SAT solvers, but the CNF formula will be reprented internally in the proof checker as a set of pseudo-Boolean constraints as above.

A.1 Starting the Proof and Introducing the Order

A *VeriPB* proof starts with a proof header (stating which version of the proof system is used) and an instruction to load the input formula

```
L1  pseudo-Boolean proof version 2.0
L2  f 22
```

where the number 22 specifies the number of pseudo-Boolean constraints in the input. All constraints in the proof file will be numbered consecutively starting with the input constraints, and will be referred to in the derivations by these numbers.

To prepare for the symmetry breaking, *BreakID* then introduces a pre-order that compares two binary sequences in lexicographic order. This pre-order is defined by inserting the lines

```
 L3  pre_order exp22
 L4    vars
 L5      left  u1 u2 u3 u4 u5 u6 u7 u8 u9 u10 u11 u12
 L6      right v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12
 L7      aux
 L8    end
 L9
L10    def
L11      -1 u12 1 v12 -2 u11 2 v11 -4 u10 4 v10 -8 u9 8 v9 -16 u8 16 v8 -32
             ↪ u7 32 v7 -64 u6 64 v6 -128 u5 128 v5 -256 u4 256 v4 -512 u3
             ↪ 512 v3 -1024 u2 1024 v2 -2048 u1 2048 v1 >= 0;
L12    end
L13
L14    transitivity
L15      vars
L16        fresh_right w1 w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12
L17      end
L18      proof
L19        proofgoal #1
L20          pol 1 2 + 3 +
L21        qed -1
L22      qed
L23    end
L24  end
```

in the proof file. The pre-order is named `exp22` on Line 3, so that we can refer to it later in the proof. Lines 5 and 6 introduce placeholder names for the 2×12 variables used to define the order. Line 11 then provides the exponential encoding (14) of the claim that the sequence of variables `u1`, ..., `u12` is lexicographically smaller than or equal to `v1`, ..., `v12`.

To prove that the pseudo-Boolean formula consisting of the single constraint on Line 11 defines a transitive relation, a third set of placeholder variables `w1`, ..., `w12` is declared on Line 16, and these variables are used in a formal derivation as specified in (13b) on page 1554 that $\mathscr{O}_{\trianglelefteq}(\vec{u}, \vec{v})$ and $\mathscr{O}_{\trianglelefteq}(\vec{v}, \vec{w})$ together imply $\mathscr{O}_{\trianglelefteq}(\vec{u}, \vec{w})$. The *VeriPB* tool has a certain preference for proofs by contradiction, however, so the way this is established is by showing that $\mathscr{O}_{\trianglelefteq}(\vec{u}, \vec{v})$ and $\mathscr{O}_{\trianglelefteq}(\vec{v}, \vec{w})$ together with the negation $\neg\mathscr{O}_{\trianglelefteq}(\vec{u}, \vec{w})$ is contradictory (and if the order consistent of more than one PB constraint, one negates the constraints

one by one and derives contradiction for every negated constraint. For the order on Line 11 we get the three constraints

$$-u_{12} + v_{12} - 2u_{11} + 2v_{11} - 4u_{10} + 4v_{10} - \ldots \geq 0 \tag{T1}$$
$$-v_{12} + w_{12} - 2v_{11} + 2w_{11} - 4v_{10} + 4w_{10} - \ldots \geq 0 \tag{T2}$$
$$u_{12} - w_{12} + 2u_{11} - 2w_{11} + 4u_{10} - 4w_{10} + \ldots \geq +1, \tag{T3}$$

where we use the labels (T1)–(T3) to emphasize that these are not constraints learned in the proof system, but are temporary constraints, local to the proof of transitivity. Line 20 is an instruction in reverse polish notation to add the constraints (T1), (T2), and (T3) together, resulting (after simplification) in the constraint

$$0 \geq +1\,. \tag{T4}$$

Line 21 then concludes the subproof, which is possible since the last derived constraint (which, using relative indexing, is what the number -1 refers to) is indeed a conflicting constraint. Note that in order to prove that the pseudo-Boolean constraint defines a pre-order we also need to show that the relation it defines is reflexive, but for a simple order like the one in this example everything in the formula trivializes when you substitute the same variables for both the u- and the v-sequence, and so *VeriPB* can figure out that reflexivity holds by itself without the help of any written proof.

The proof continues with the instruction

```
L25  load_order exp22 p21 p22 p23 p11 p12 p13 p31 p32 p33 p41 p42 p43
```

which specifies that the order `exp22` should be used and should be applied to all variables in the formula in the specified order. The reader might find slightly odd that in the order specified above all variables related to pigeon 2 are ordered before those referring to pigeon 1, followed by variables mentioning pigeons 3 and 4. In other words, in the lex-leader order, assignments are sorted with respect to pigeon 2 first. The reason for this seemingly strange choice is that the entire proof presented in this appendix is actually generated fully automatically by *BreakID*. This tool only takes a propositional logic formula as input, and has no information about high-level interpretations of what the different variables mean, or which ordering of these variables might seem more or less natural from the point of view of a human observer.

A.2 Logging the Breaking of a First Symmetry

Now that the order has been defined and has been proven to be an order, we can start adding symmetry breaking constraints to the proof log. The first symmetry that *BreakID* considers is

$$\pi := (p_{11}p_{43})(p_{12}p_{42})(p_{13}p_{41})(p_{21}p_{23})(p_{31}p_{33})\,, \tag{32}$$

which is the symmetry that simultaneously swaps pigeons 1 and 4 and pigeonholes 1 and 3. We remark that the questions of why *BreakID* chooses to break this particular symmetry, and how it finds the symmetry in the first place, are interesting and nontrivial questions, but they are not relevant for our work. This is so since we are not trying to construct new symmetry breaking tools, but to design proof logging methods to certify the correctness of

the symmetry breaking constraints added by existing tools. From the point of view of proof logging, it is mostly irrelevant to dwell on the possible reasons why a specific symmetry was chosen, or how it was found. Instead, we can just take the symmetry as a given and focus on proof logging for the symmetry breaking constraints.

As explained in our discussion of symmetry breaking in Section 4, in order to break a given symmetry we first use the dominance rule to derive a pseudo-Boolean exponential encoding of a lex-leader by the proof lines

```
L26  dom -1 p43 1 p11 -2 p42 2 p12 -4 p41 4 p13 -8 p33 8 p31 -32 p31 32 p33
       ↪ -64 p13 64 p41 -128 p12 128 p42 -256 p11 256 p43 -512 p23 512 p21
       ↪ -2048 p21 2048 p23 >= 0 ;  p11 -> p43 p12 -> p42 p13 -> p41 p21 ->
       ↪  p23 p23 -> p21 p31 -> p33 p33 -> p31 p41 -> p13 p42 -> p12 p43 ->
       ↪  p11 ;  begin
L27    proofgoal #2
L28      pol -1 -2 +
L29    qed -1
```

which we will now discuss in more detail.

Line 26 above says that the dominance rule should be used to derive (and add to the derived set $\mathscr{D}$) the constraint (20), which expresses that the assignment to the variables should not become lexicographically smaller when the symmetry is applied. The variables related to pigeon 2 occur with the highest coefficients, since they were given the highest priority when the order was instantiated. Notice that the pseudo-Boolean constraint specified on Line 26 contains two occurrences of every variable. This is because the constraint has been generated automatically, and *VeriPB* will perform cancellations to simplify the constraint before storing it internally.

An application of the dominance rule needs to provide more information to *VeriPB* than just the constraint to be derived by dominance, however. The proof should also specify

- the witness, which in this case is just the symmetry, written as a substitution at the end of Line 26, and
- explicit derivations (*subproofs*) for all constraints (*proof goals*) where the proof checker cannot automatically figure out such a derivation—in this case, we have a single such proof goal that is taken care of on Lines 27–29.

Let us discuss in more detail what subproofs are required. In order to apply the dominance rule to derive the constraint C using witness ω, we need to establish that derivations

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \mathscr{C}\restriction_\omega \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \preceq f\} \tag{33a}$$

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot \tag{33b}$$

exist. In the concrete case of the first symmetry breaking constraint for the formula consisting of the clauses (C1)–(C22), *VeriPB* will generate the following numbered list of derivations (or *proof obligations*) that it expects to see:

1. $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})$
2. $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot$

3. $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash f\restriction_\omega \leq f$

4–25. $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash D$ for each $D \in \mathscr{C}\restriction_\omega$.

Except for the second proof obligation, all of them can be proved automatically. Perhaps the simplest case is the third obligation, which is trivial since we are dealing with a decision problem for which the objective function is the constant $f = 0$. And since ω is a syntactic symmetry of the set of core constraints $\mathscr{C}$ (which at this point is the CNF formula in the input), the proof obligations 4–25 are also trivial.

Let us describe how the subproof for the second proof obligation on Lines 27–29 is checked by *VeriPB*. First, *VeriPB* makes available in the subproof the constraint $\neg C$, which (after simplification) equals

$$\begin{aligned} &255 \cdot p_{11} + 126 \cdot p_{12} + 60 \cdot p_{13} + 1536 \cdot p_{21} + 1536 \cdot \overline{p}_{23} \\ &\quad +24 \cdot p_{31} + 24 \cdot \overline{p}_{33} + 60 \cdot \overline{p}_{41} + 126 \cdot \overline{p}_{42} + 255 \cdot \overline{p}_{43} \geq 2002\,, \end{aligned} \tag{C23}$$

giving this constraint the next available number 23 as constraint reference ID. Next, *VeriPB* makes available the constraint $\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega)$, which, after simplification, equals

$$\begin{aligned} &255 \cdot \overline{p}_{11} + 126 \cdot \overline{p}_{12} + 60 \cdot \overline{p}_{13} + 1536 \cdot \overline{p}_{21} + 1536 \cdot p_{23} \\ &\quad +24 \cdot \overline{p}_{31} + 24 \cdot p_{33} + 60 \cdot p_{41} + 126 \cdot p_{42} + 255 \cdot p_{43} \geq 2001\,. \end{aligned} \tag{C24}$$

As explained above, all constraints are referred to by numbers, but as we have already seen *VeriPB* also allows relative indexing. Therefore, Line 28 simply states in reverse polish notation that the two most recently generated constraints (i.e., (C24) and (C23)) should be added, which yields the inequality

$$255 + 126 + 60 + 1536 + 1536 + 24 + 24 + 60 + 126 + 255 \geq 2002 + 2001 \tag{34}$$

or, in simplified form,

$$0 \geq 1\,, \tag{C25}$$

which is contradictory. Line 29 ends the proof by giving the (relative) identifier of the contradicting constraint that was just derived. Finally, when this proof is finished and all other proof obligation have been automatically checked, the desired symmetry breaking constraint

$$\begin{aligned} &-p_{43} + 1 \cdot p_{11} - 2 \cdot p_{42} + 2 \cdot p_{12} - 4 \cdot p_{41} + 4 \cdot p_{13} - 8 \cdot p_{33} + 8 \cdot p_{31} \\ &\quad -32 \cdot p_{31} + 32 \cdot p_{33} - 64 \cdot p_{13} + 64 \cdot p_{41} - 128 \cdot p_{12} + 128 \cdot p_{42} \\ &\quad -256 \cdot p_{11} + 256 \cdot p_{43} - 512 \cdot p_{23} + 512 \cdot p_{21} - 2048 \cdot p_{21} + 2048 \cdot p_{23} \geq 0 \end{aligned} \tag{C26}$$

is added to the derived set $\mathscr{D}$. Once all proof obligations have been taken care of, the constraints (C23), (C24), and (C25) are automatically erased by the proof checker and can no longer be referred to without triggering an error, since they are only relevant for the derivations in the subproof.

The inequality (C26) is a lex-leader constraint for the symmetry we are considering, but this constraint is nothing that the SAT solver knows about or can understand, since the solver only operates with disjunctive clauses. What happens next in the proof, therefore, is that the constraint (C26) is converted to a set of clauses on the form (19a)–(19f) that the solver can use for symmetry breaking.

First, (19a) is added with the redundance rule with the instruction

```
L30  red 1 y0 >= 1 ; y0 -> 1
```

which contains both the constraint

$$y_0 \geq 1 \tag{C27}$$

and the witness $y_0 \mapsto 1$ to apply the redundance rule. All proof obligations are checked automatically by *VeriPB*.

In our chosen lexicographic order, the first variable is p_{21}. Therefore, the first clause for symmetry breaking is

$$\overline{p}_{21} \vee \pi(p_{21}) \doteq \overline{p}_{21} \vee p_{23}\,, \tag{35}$$

which is a simplification of (19b), omitting the trivially true variable y_0. This clause is implied by the constraint (C26), which can be seen by weakening away all other variables in it (i.e., adding literal axioms to cancel them). Instead of providing an explicit derivation, we can simply add the clause (35) it with the reverse unit propagation rule and let *VeriPB* figure out the details, which we do by inserting the line

```
L31  rup 1 ~p21 1 p23 >= 1 ;
```

that derives the desired constraint

$$\overline{p}_{21} + p_{23} \geq 1\,. \tag{C28}$$

Next, the fresh variable y_1 is introduced with four redundance rule applications

```
L32  red 1   p23 1 ~y0   1 y1   >= 1 ; y1 -> 1
L33  red 1 ~p21 1 ~y0   1 y1   >= 1 ; y1 -> 1
L34  red 1 ~y1   1   y0        >= 1 ; y1 -> 0
L35  red 1 ~y1   1 ~p23 1 p21 >= 1 ; y1 -> 0
```

with witnesses mapping y_1 to either 0 or to 1, resulting in the constraints

$$p_{23} + \overline{y}_0 + y_1 \geq 1 \tag{C29}$$
$$\overline{p}_{21} + \overline{y}_0 + y_1 \geq 1 \tag{C30}$$
$$\overline{y}_1 + y_0 \geq 1 \tag{C31}$$
$$\overline{y}_1 + \overline{p}_{23} + p_{21} \geq 1 \tag{C32}$$

corresponding to the clauses (19c)–(19f).

Before repeating this procedure for the next variable y_2, we use the recently derived constraints to cancel out the dominant terms in constraint (C26) with the instructions

```
L36  pol 26 32 2048 * +
L37  del id 26
```

The first line above adds 2048 times (C32) to (C26), yielding

$$\begin{aligned} &255 \cdot \overline{p}_{11} + 126 \cdot \overline{p}_{12} + 60 \cdot \overline{p}_{13} + 512 \cdot p_{21} + 512 \cdot \overline{p}_{23} \\ &\quad +24 \cdot \overline{p}_{31} + 24 \cdot p_{33} + 60 \cdot p_{41} + 126 \cdot p_{42} + 255 \cdot p_{43} + 2048 \cdot \overline{y}_1 \geq 977\,. \end{aligned} \tag{C33}$$

The second line deletes (C26) from $\mathscr{D}$ since it will no longer be required.

The next variable in lexicographic order after p_{21} is p_{22}. However, since our symmetry π maps p_{22} to itself, no symmetry breaking clauses are added for it. The variable after that in the ordering is p_{23}, which is mapped to p_{21}, resulting in the (conditional on y_1) symmetry breaking constraint

$$\overline{y}_1 + \overline{p}_{23} + p_{21} \geq 1 \tag{C34}$$

obtained by adding the line

```
L38  rup 1 ~y1 1 ~p23 1 p21 >= 1 ;
```

to the proof. After this, the next fresh variable y_2 is introduced in the same way as y_1 using the redundance rule in the instructions

```
L39  red 1  p21 1 ~y1  1 y2   >= 1 ; y2 -> 1
L40  red 1 ~p23 1 ~y1  1 y2   >= 1 ; y2 -> 1
L41  red 1 ~y2  1  y1         >= 1 ; y2 -> 0
L42  red 1 ~y2  1 ~p21 1 p23  >= 1 ; y2 -> 0
```

yielding the clauses

$$p_{21} + \overline{y}_1 + y_2 \geq 1 \tag{C35}$$
$$\overline{p}_{23} + \overline{y}_1 + y_2 \geq 1 \tag{C36}$$
$$y_1 + \overline{y}_2 \geq 1 \tag{C37}$$
$$\overline{p}_{21} + p_{23} + \overline{y}_2 \geq 1 \tag{C38}$$

As before, our pseudo-Boolean symmetry breaking constraint is simplified with

```
L43  pol 33 38 512 * +
L44  del id 33
```

where the first instruction again cancels out the dominant terms in (C33), replacing them by a y-variable, to express a conditional symmetry breaking constraint, resulting in

$$\begin{aligned} 255 \cdot \overline{p}_{11} + 126 \cdot \overline{p}_{12} + 60 \cdot \overline{p}_{13} + 24 \cdot \overline{p}_{31} + 24 \cdot p_{33} \\ +60 \cdot p_{41} + 126 \cdot p_{42} + 255 \cdot p_{43} + 2048 \cdot \overline{y}_1 + 512 \cdot \overline{y}_2 \geq 465 \end{aligned} \tag{C39}$$

and the second instruction deletes constraint (C33).

The next variable in our chosen order is p_{11}. Since p_{11} is mapped to p_{43}, we want to have the symmetry breaking clause

$$\overline{p}_{11} + p_{43} + \overline{y}_2 \geq 1\,, \tag{C40}$$

which can be derived by the proof line

```
L45  rup 1 ~y2 1 ~p11 1 p43 >= 1 ;
```

To see that the clause (C40) indeed follows by reverse unit propagation, consider what happens if all literals in the clause are set to false. Whenever y_2 is true, so are y_1 and y_0 (by (C37) and (C31)). If furthermore p_{43} is false and p_{11} is true, then (C39) simplifies to

$$126 \cdot \overline{p}_{12} + 60 \cdot \overline{p}_{13} + 24 \cdot \overline{p}_{31} + 24 \cdot p_{33} + 60 \cdot p_{41} + 126 \cdot p_{42} \geq 465\,, \tag{36}$$

which can never be satisifed since the coefficients on the left only add up to 420.

The process of introducing a new variable is the same as before, appending the constraints

$$p_{43} + \overline{y}_2 + y_3 \geq 1 \tag{C41}$$
$$\overline{p}_{11} + \overline{y}_2 + y_3 \geq 1 \tag{C42}$$
$$y_2 + \overline{y}_3 \geq 1 \tag{C43}$$
$$p_{11} + \overline{p}_{43} + \overline{y}_3 \geq 1 \tag{C44}$$

to the derived set $\mathscr{D}$. The last constraint can then again be used to simplify (C39) with the instruction

```
L46  pol 39 44 256 * +
```

yielding the constraint

$$\begin{aligned} p_{11} + 126 \cdot \overline{p}_{12} + 60 \cdot \overline{p}_{13} + 24 \cdot \overline{p}_{31} + 24 \cdot p_{33} + 60 \cdot p_{41} \\ +126 \cdot p_{42} + \overline{p}_{43} + 2048 \cdot \overline{y}_1 + 512 \cdot \overline{y}_2 + 256 \cdot \overline{y}_3 \geq 211 \end{aligned} \tag{C45}$$

This process continues in the same way for all variables in the lexicographic ordering that are not mapped to themselves, or *stabilized*, by π.

A.3 Logging the Breaking of More Symmetries

As the *BreakID* execution continues, more symmetries are detected and broken, and the corresponding symmetry breaking clauses are derived in the proof. The process is completely analogous to what is described above for the first symmetry. For our concrete toy example formula with 4 pigeons and 3 holes, *BreakID* next breaks the symmetries

$$(p_{11}p_{12})(p_{21}p_{32})(p_{22}p_{31})(p_{23}p_{33})(p_{41}p_{42}) \tag{37a}$$
$$(p_{21}p_{11})(p_{22}p_{12})(p_{23}p_{13}) \tag{37b}$$
$$(p_{11}p_{31})(p_{12}p_{32})(p_{13}p_{33}) \tag{37c}$$
$$(p_{31}p_{41})(p_{32}p_{42})(p_{33}p_{43}) \tag{37d}$$
$$(p_{21}p_{22})(p_{11}p_{12})(p_{31}p_{32})(p_{41}p_{42}) \tag{37e}$$
$$(p_{22}p_{23})(p_{12}p_{13})(p_{32}p_{33})(p_{42}p_{43}) \tag{37f}$$

in the order listed (where, just to help decode the notation, the first of these symmetries swaps holes 1 and 2 and simultaneously swaps pigeons 2 and 3). It is crucial to note here that when we break later symmetries in this list, we do not have to worry about previously added symmetry breaking clauses. There is no interaction between the different symmetry breaking derivations, since all the symmetry breaking constraints are added to the derived set $\mathscr{D}$, whereas the core set $\mathscr{C}$ containing the proof obligation for the dominance rule applications only consists of the symmetric input formula.

A.4 Proof for the Preprocessed Formula

The symmetry breaking performed here only serves as a preprocessing step for the solving; it is not a complete proof leading to a contradiction. A revision of the *VeriPB* proof format

used in the SAT Competition 2023 (Bogaerts et al., 2023), and currently under further development, will allow proofs for preprocessing steps, establishing that the formulas before and after preprocessing are equisatisfiable (or, for optimization problems, that the two formulas have the same optimal value for the objective function).

Very briefly, the way a proof for preprocessing will work is that at the end of the proof all constraints to be output should be moved to the core set $\mathscr{C}$, after which the derived set $\mathscr{D}$ is emptied. The constraints in $\mathscr{C}$ then constitute the output formula, which is guaranteed to be equisatisfiable to the input formula if all deletion steps are instances of checked deletion as described in Definition 8 in Section 3.4. We remark that this is quite similar to the concept of *finalization*.

For our toy example, in order to prove equisatisfiability of the formula after having added symmetry breaking clauses one should end the proof with the lines

```
L492   core id 27 28 29 30 31 32 34 35 36 37 38 40 41 42 43 44 46 47 48 49 50
           ↪ 52 57 58 59 60 61 62 64 65 66 67 68 70 71 72 73 74 76 77 78 79 80
           ↪  82 87 88 89 90 91 92 94 95 96 97 98 100 105 106 107 108 109 110
           ↪ 112 113 114 115 116 118 123 124 125 126 127 128 130 131 132 133
           ↪ 134 136 141 142 143 144 145 146 148 149 150 151 152 154 155 156
           ↪ 157 158 160 165 166 167 168 169 170 172 173 174 175 176 178 179
           ↪ 180 181 182 184
L493   output EQUISATISFIABLE PERMUTATION
L494   * #variable= 39 #constraint=136
L495   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 27 28 29 30 31
           ↪  32 34 35 36 37 38 40 41 42 43 44 46 47 48 49 50 52 57 58 59 60
           ↪ 61 62 64 65 66 67 68 70 71 72 73 74 76 77 78 79 80 82 87 88 89 90
           ↪  91 92 94 95 96 97 98 100 105 106 107 108 109 110 112 113 114 115
           ↪    116 118 123 124 125 126 127 128 130 131 132 133 134 136 141
           ↪ 142 143 144 145 146 148 149 150 151 152 154 155 156 157 158 160
           ↪ 165 166 167 168 169 170 172 173 174 175 176 178 179 180 181 182
           ↪ 184
L496   conclusion NONE
L497   end pseudo-Boolean proof
```

The intended semantics here is that Line 492 will move all the symmetry breaking clauses that have been derived to the core set $\mathscr{C}$. Line 493 claims that the input and the set of constraints that is now in the core are equisatisfiable. At the time of writing, the public version of *VeriPB* does not yet support non-trivial output statements, (i.e., does not provide support for checking that the list of constraint identifiers are precisely the set of constraints in the core set at the of the proof), and only `output NONE` is supported in the version of *VeriPB* used in the SAT Competition 2023 (Bogaerts et al., 2023). However, support for outputting formulas at the end of the proof is currently being developed and is expected to be available in a not too distant future. Line 494 states that the output formula has 39 variables and 136 constraints and Line 495 lists the IDs of those 136 constraints. Finally, the conclusion line 496 states that neither satisfiability nor unsatisfiability can be concluded from this proof. More details about the intended format and semantics of the output section in *VeriPB* proofs can be found in the technical documentation for the SAT Competition 2023 (Bogaerts et al., 2023), but it should be emphasized again that these features of the proof checker are currently under development and minor changes could and should be expected on the way from the current tentative specification to the final finished product.

References

Achterberg, T., & Wunderling, R. (2013). Mixed integer programming: Analyzing 12 years of progress. In Jünger, M., & Reinelt, G. (Eds.), *Facets of Combinatorial Optimization*, pp. 449–481. Springer.

Akgün, Ö., Gent, I. P., Jefferson, C., Miguel, I., & Nightingale, P. (2018). Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, Vol. 11008 of *Lecture Notes in Computer Science*, pp. 727–736. Springer.

Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C., & Schweitzer, P. (2011). An introduction to certifying algorithms. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, *53*(6), 287–293.

Aloul, F. A., Sakallah, K. A., & Markov, I. L. (2006). Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, *55*(5), 549–558.

Baptiste, P., & Pape, C. L. (1997). Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP '97)*, Vol. 1330 of *Lecture Notes in Computer Science*, pp. 375–389. Springer.

Beame, P., Kautz, H., & Sabharwal, A. (2004). Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, *22*, 319–351. Preliminary version in *IJCAI '03*.

Benhamou, B., Nabhani, T., Ostrowski, R., & Saïdi, M. R. (2010). Enhancing clause learning by symmetry in SAT solvers. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '10)*, Vol. 1, pp. 329–335.

Benhamou, B., & Saïs, L. (1994). Tractability through symmetries in propositional calculus. *Journal of Automated Reasoning*, *12*(1), 89–102.

Berg, J., Bacchus, F., & Poole, A. (2020). Abstract cores in implicit hitting set MaxSat solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, Vol. 12178 of *Lecture Notes in Computer Science*, pp. 277–294. Springer.

Berg, J., Bogaerts, B., Nordström, J., Oertel, A., & Vandesande, D. (2023). Certified core-guided maxsat solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE)*. Accepted for publication.

Biere, A. (2006). Tracecheck. `http://fmv.jku.at/tracecheck/`.

Biere, A., Heule, M. J. H., van Maaren, H., & Walsh, T. (Eds.). (2021). *Handbook of Satisfiability* (2nd edition)., Vol. 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bogaerts, B., Gocht, S., McCreesh, C., & Nordström, J. (2022a). Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pp. 3698–3707.

Bogaerts, B., Gocht, S., McCreesh, C., & Nordström, J. (2022b). Certified symmetry and dominance breaking for combinatorial optimisation (code and data). `https://doi.org/10.5281/zenodo.6373986`.

Bogaerts, B., McCreesh, C., Myreen, M. O., Nordström, J., Oertel, A., & Tan, Y. K. (2023). Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`.

Bogaerts, B., McCreesh, C., & Nordström, J. (2022). Solving with provably correct results: Beyond satisfiability, and towards constraint programming. Tutorial at the *28th International Conference on Principles and Practice of Constraint Programming*. Slides available at `http://www.jakobnordstrom.se/presentations/`.

Brummayer, R., Lonsing, F., & Biere, A. (2010). Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, Vol. 6175 of *Lecture Notes in Computer Science*, pp. 44–57. Springer.

Bulhões, T., Sadykov, R., & Uchoa, E. (2018). A branch-and-price algorithm for the minimum latency problem. *Computers & Operations Research*, *93*, 66–78.

Buss, S. R., & Nordström, J. (2021). Proof complexity and SAT solving. In Biere et al. (Biere, Heule, van Maaren, & Walsh, 2021), chap. 7, pp. 233–350.

Buss, S. R., & Thapen, N. (2019). DRAT proofs, propagation redundancy, and extended resolution. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 71–89. Springer.

Cheung, K. K. H., Gleixner, A. M., & Steffy, D. E. (2017). Verifying integer programming results. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization (IPCO '17)*, Vol. 10328 of *Lecture Notes in Computer Science*, pp. 148–160. Springer.

Chu, G., & Stuckey, P. J. (2015). Dominance breaking constraints. *Constraints*, *20*(2), 155–182. Preliminary version in *CP '12*.

Cook, S. A., & Reckhow, R. A. (1979). The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, *44*(1), 36–50. Preliminary version in *STOC '74*.

Cook, W., Coullard, C. R., & Turán, G. (1987). On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, *18*(1), 25–38.

Cook, W., Koch, T., Steffy, D. E., & Wolter, K. (2013). A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, *5*(3), 305–344.

Crawford, J. M., Ginsberg, M. L., Luks, E. M., & Roy, A. (1996). Symmetry-breaking predicates for search problems. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR '96)*, pp. 148–159.

Cruz-Filipe, L., Heule, M. J. H., Hunt Jr., W. A., Kaufmann, M., & Schneider-Kamp, P. (2017a). Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, Vol. 10395 of *Lecture Notes in Computer Science*, pp. 220–236. Springer.

Cruz-Filipe, L., Marques-Silva, J. P., & Schneider-Kamp, P. (2017b). Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, Vol. 10205 of *Lecture Notes in Computer Science*, pp. 118–135. Springer.

Demeulemeester, E. L., & Herroelen, W. S. (2002). *Project Scheduling: A Research Handbook*, Vol. 49 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers.

Devriendt, J., Bogaerts, B., & Bruynooghe, M. (2017). Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, Vol. 10491 of *Lecture Notes in Computer Science*, pp. 83–100. Springer.

Devriendt, J., Bogaerts, B., Bruynooghe, M., & Denecker, M. (2016). Improved static symmetry breaking for SAT. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, Vol. 9710 of *Lecture Notes in Computer Science*, pp. 104–122. Springer.

Devriendt, J., Bogaerts, B., De Cat, B., Denecker, M., & Mears, C. (2012). Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *Proceedings of the 24th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '12)*, pp. 49–56.

Eifler, L., & Gleixner, A. (2021). A computational status update for exact rational mixed integer programming. In *Proceedings of the 22nd International Conference on Integer Programming and Combinatorial Optimization (IPCO '21)*, Vol. 12707 of *Lecture Notes in Computer Science*, pp. 163–177. Springer.

Elffers, J., Gocht, S., McCreesh, C., & Nordström, J. (2020). Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pp. 1486–1494.

Garcia de la Banda, M., Stuckey, P. J., Van Hentenryck, P., & Wallace, M. (2014). The future of optimization technology. *Constraints*, *19*(2), 126–138.

Gebser, M., Kaminski, R., & Schaub, T. (2011). Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, *11*(4–5), 821–839.

Gent, I. P., Petrie, K. E., & Puget, J. (2006). Symmetry in constraint programming. In Rossi, F., van Beek, P., & Walsh, T. (Eds.), *Handbook of Constraint Programming*, Vol. 2 of *Foundations of Artificial Intelligence*, pp. 329–376. Elsevier.

Gillard, X., Schaus, P., & Deville, Y. (2019). SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, Vol. 11802 of *Lecture Notes in Computer Science*, pp. 565–582. Springer.

Gocht, S., Martins, R., Nordström, J., & Oertel, A. (2022). Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, Vol. 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 16:1–16:25.

Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., & Trimble, J. (2020a). Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, Vol. 12333 of *Lecture Notes in Computer Science*, pp. 338–357. Springer.

Gocht, S., McCreesh, C., & Nordström, J. (2020b). Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pp. 1134–1140.

Gocht, S., McCreesh, C., & Nordström, J. (2022). An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, Vol. 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 25:1–25:18.

Gocht, S., & Nordström, J. (2021). Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pp. 3768–3777.

Goldberg, E., & Novikov, Y. (2003). Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 886–891.

Haken, A. (1985). The intractability of resolution. *Theoretical Computer Science*, *39*(2-3), 297–308.

Heule, M. J. H., Hunt Jr., W. A., & Wetzler, N. (2013a). Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pp. 181–188.

Heule, M. J. H., Hunt Jr., W. A., & Wetzler, N. (2013b). Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, Vol. 7898 of *Lecture Notes in Computer Science*, pp. 345–359. Springer.

Heule, M. J. H., Hunt Jr., W. A., & Wetzler, N. (2015). Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, Vol. 9195 of *Lecture Notes in Computer Science*, pp. 591–606. Springer.

Heule, M. J. H., Kiesl, B., & Biere, A. (2017). Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, Vol. 10395 of *Lecture Notes in Computer Science*, pp. 130–147. Springer.

Hoogeboom, M., Dullaert, W., Lai, D., & Vigo, D. (2020). Efficient neighborhood evaluations for the vehicle routing problem with multiple time windows. *Transportation Science*, *54*(2), 400–416.

Ignatiev, A., Morgado, A., & Marques-Silva, J. (2019). RC2: an Efficient MaxSAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *11*, 53–64.

Jouglet, A., & Carlier, J. (2011). Dominance rules in combinatorial optimization problems. *European Journal of Operational Research*, *212*(3), 433–444.

Kiesl, B., Rebola-Pardo, A., & Heule, M. J. H. (2018). Extended resolution simulates DRAT. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR '18)*, Vol. 10900 of *Lecture Notes in Computer Science*, pp. 516–531. Springer.

Koshimura, M., Zhang, T., Fujita, H., & Hasegawa, R. (2012). QMaxSAT: A Partial Max-SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *8*(1-2), 95–100.

Kołodziejczyk, L., & Thapen, N. (2023) Personal communication.

Kraiczy, S., & McCreesh, C. (2021). Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI '21)*, pp. 1396–1402.

Krajíček, J., & Pudlák, P. (1989). Propositional proof systems, the consistency of first order theories and the complexity of computations. *Journal of Symbolic Logic*, *54*(3), 1063–1079.

Lammich, P. (2020). Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, *64*(3), 513–532. Extended version of paper in *CADE* 2017.

McConnell, R. M., Mehlhorn, K., Näher, S., & Schweitzer, P. (2011). Certifying algorithms. *Computer Science Review*, *5*(2), 119–161.

McCreesh, C., & Prosser, P. (2016). Finding maximum k-cliques faster using lazy global domination. In *Proceedings of the 9th Annual Symposium on Combinatorial Search (SOCS '16)*, pp. 72–80.

Metin, H., Baarir, S., & Kordon, F. (2019). Composing symmetry propagation and effective symmetry breaking for SAT solving. In *Proceedings of the 11th International NASA Formal Methods Symposium (NFM '19)*, Vol. 11460 of *Lecture Notes in Computer Science*, pp. 316–332. Springer.

Paxian, T., & Becker, B. (2022). Pacose: An iterative SAT-based MaxSAT solver. In *MaxSAT Evaluation 2021 : Solver and Benchmark Descriptions.*

Sabharwal, A. (2009). SymChaff: Exploiting symmetry in a structure-aware satisfiability solver. *Constraints*, *14*(4), 478–505. Preliminary version in *AAAI '05.*

Sakallah, K. A. (2021). Symmetry and satisfiability. In Biere et al. (Biere et al., 2021), chap. 13, pp. 509–570.

Tan, Y. K., Heule, M. J. H., & Myreen, M. O. (2021). cake_lpr: Verified propagation redundancy checking in CakeML. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, Vol. 12652 of *Lecture Notes in Computer Science*, pp. 223–241. Springer.

Tchinda, R. K., & Djamégni, C. T. (2020). On certifying the UNSAT result of dynamic symmetry-handling-based SAT solvers. *Constraints*, *25*(3–4), 251–279.

Tomita, E., & Kameda, T. (2007). An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. Glob. Optim.*, *37*(1), 95–111.

Tseitin, G. (1968). On the complexity of derivation in propositional calculus. In Silenko, A. O. (Ed.), *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pp. 115–125. Consultants Bureau, New York-London.

Vandesande, D., De Wulf, W., & Bogaerts, B. (2022). QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, Vol. 13416 of *Lecture Notes in Computer Science*, pp. 429–442. Springer.

Walsh, T. (2006). General symmetry breaking constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP '06)*, Vol. 4204 of *Lecture Notes in Computer Science*, pp. 650–664. Springer.

Walsh, T. (2012). Symmetry breaking constraints: Recent results. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI '12)*, pp. 2192–2198.

Wetzler, N., Heule, M. J. H., & Hunt Jr., W. A. (2014). DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, Vol. 8561 of *Lecture Notes in Computer Science*, pp. 422–429. Springer.