Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs

Stephan Gocht,[1,2] **Jakob Nordström,**[2,1]
[1] Lund University, Lund, Sweden
[2] University of Copenhagen, Copenhagen, Denmark
stephan.gocht@cs.lth.se, jn@di.ku.dk

Abstract

The dramatic improvements in combinatorial optimization algorithms over the last decades have had a major impact in artificial intelligence, operations research, and beyond, but the output of current state-of-the-art solvers is often hard to verify and is sometimes wrong. For Boolean satisfiability (SAT) solvers proof logging has been introduced as a way to certify correctness, but the methods used seem hard to generalize to stronger paradigms. What is more, even for enhanced SAT techniques such as parity (XOR) reasoning, cardinality detection, and symmetry handling, it has remained beyond reach to design practically efficient proofs in the standard *DRAT* format. In this work, we show how to instead use pseudo-Boolean inequalities with extension variables to concisely justify XOR reasoning. Our experimental evaluation of a SAT solver integration shows a dramatic decrease in proof logging and verification time compared to existing *DRAT* methods. Since our method is a strict generalization of *DRAT*, and readily lends itself to expressing also 0-1 programming and even constraint programming problems, we hope this work points the way towards a unified approach for efficient machine-verifiable proofs for a rich class of combinatorial optimization paradigms.

1 Introduction

Since around the turn of the millennium, combinatorial optimization has been successfully applied to solve an ever increasing range of problems in e.g., resource allocation, scheduling, logistics, and disaster management (Pardalos, Du, and Graham 2013), and more recent applications in biology, chemistry, and medicine (Archibald et al. 2019) include, e.g., protein analysis and design (Allouche et al. 2014; Mann, Will, and Backofen 2008) and kidney transplants (Manlove and O'Malley 2012). Yet other examples are government auctions generating billions of dollars in revenue (Leyton-Brown, Milgrom, and Segal 2017), as well as allocation of education and work opportunities (Manlove 2016; Manlove, McBride, and Trimble 2017) and matching of adoptive families with children (Delorme et al. 2019).

As more and more such problems are dealt with using combinatorial optimization solvers, an urgent question is whether we can trust that the solutions computed by such algorithms are correct and complete. The answer, unfortunately, is currently a clear "no": State-of-the-art solvers sometimes return "solutions" that do not satisfy the constraints or erroneously claim optimality (Cook et al. 2013; Akgün et al. 2018; Gillard, Schaus, and Deville 2019). This can be fatal for applications such as, e.g., chip design, compiler optimization, and combinatorial auctions, where correctness is absolutely crucial, not to speak about when human lives depend on finding the best solutions.

Conventional software testing has made little progress in addressing this problem, and formal verification techniques cannot handle the level of complexity of modern solvers. Instead, the most successful approach to date has been that of *proof logging* in the Boolean satisfiability (SAT) community, where solvers are required to *certify* (McConnell et al. 2011) their answer by outputting also a simple, machine-verifiable proof that this answer is correct. This does not certify the correctness of the solver itself, but it does mean that if it ever produces an incorrect answer (even if due to hardware errors), then this can be detected. A number of different proof logging formats such as *RUP* (Goldberg and Novikov 2003), *TraceCheck* (Biere 2006), *DRAT* (Heule, Hunt Jr., and Wetzler 2013a,b; Wetzler, Heule, and Hunt Jr. 2014), *GRIT* (Cruz-Filipe, Marques-Silva, and Schneider-Kamp 2017), and *LRAT* (Cruz-Filipe et al. 2017) have been developed, with *DRAT* now established as the standard in the SAT competitions (`www.satcompetition.org`).

A quite natural, and highly desirable, goal would be to extend these proof logging techniques to stronger paradigms such as pseudo-Boolean (PB) optimization, MaxSAT solving, mixed integer programming, and constraint programming, but such attempts have had limited success. Either the proofs require trusting in powerful and complicated rules (as in, e.g., (Veksler and Strichman 2010)), defeating simplicity and verifiability, or they have to justify such rules by long explanations, leading to an exponential slow-down (see (Gange and Stuckey 2019)). In fact, even for SAT solvers a long-standing problem is that more advanced techniques reasoning with parity constraints (XORs), cardinality constraints, and symmetries have remained out of reach for efficient proof logging. Although in theory there should be no problems—*DRAT* is extremely powerful, and can in principle justify such reasoning and much more with at most a polynomial amount of work (Sinz and Biere 2006;

Heule, Hunt Jr., and Wetzler 2015; Philipp and Rebola-Pardo 2016)—in practice the overhead seems completely prohibitive. Thus, a key challenge on the road to efficient proof logging for more general combinatorial optimization solvers would seem to be to design a method that can capture the full range of techniques used in modern SAT solvers.

Our Contribution

In this work, we present a new, efficient proof logging method for parity reasoning that is—perhaps somewhat surprisingly—based on pseudo-Boolean reasoning with 0-1 linear inequalities. Though such inequalities might seem ill-suited to representing XOR constraints, this can be done elegantly by introducing auxiliary so-called *extension variables* (Dixon, Ginsberg, and Parkes 2004). Using this observation, we strengthen the *VeriPB* tool recently introduced in (Elffers et al. 2020), which can be viewed as a generalization to pseudo-Boolean proofs of *RUP* (Goldberg and Novikov 2003). Borrowing inspiration from (Heule, Kiesl, and Biere 2017; Buss and Thapen 2019), we develop stronger, but still efficient, rules that can handle also extension variables, making *VeriPB*, in effect, into a strict generalization of *DRAT*.

We have implemented our method for representing XOR constraints and performing Gaussian elimination in a library with a simple, clean interface for SAT solvers. As a proof of concept, we have also integrated it in *MiniSat* (Eén and Sörensson 2004), which still serves as the foundation of most state-of-the-art SAT solvers. Our library also provides *DRAT* proof logging for XORs as described in (Philipp and Rebola-Pardo 2016), but with some optimizations, to allow for a comparative evaluation. Our experiments show that the overhead for proof logging, the size of the produced proofs, and the time for verification all go down by orders of magnitude for our method compared to *DRAT*. Furthermore, the fact that PB reasoning forms the basis for solvers like *Sat4j* (Le Berre and Parrain 2010) and *RoundingSat* (Elffers and Nordström 2018) means that our library can also empower such pseudo-Boolean solvers to reason with parities.

Since cardinality constraints are just a special case of PB constraints, it is clear that our method should suffice to justify the cardinality reasoning used in SAT solvers. Symmetry reasoning remains a challenge, but at least our method can subsume anything done by *DRAT*. More excitingly, the original *VeriPB* tool has already been shown to be capable of efficiently justifying a number of constraint programming techniques (Elffers et al. 2020; Gocht, McCreesh, and Nordström 2020; Gocht et al. 2020). Our optimistic interpretation is that pseudo-Boolean reasoning with extension variables shows great potential as a unified method of proof logging for SAT solving, pseudo-Boolean optimization, constraint programming, and maybe even mixed integer programming.

Organization of This Paper After some brief background in Section 2, we introduce the key technical notions needed in Section 3 and show how they can be used to justify parity reasoning in Section 4 with a worked out example in Section 5. We present an experimental evaluation in Section 6 and provide some concluding remarks in Section 7.

2 Preliminaries

Let us start by quickly reviewing the required material on pseudo-Boolean reasoning, referring the reader to, e.g., (Buss and Nordström 2021) for more context. 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). The set of all literals is denoted *Lits*. For notational convenience, we define $\overline{\overline{x}} = x$. A *pseudo-Boolean (PB) constraint* C is a 0-1 linear inequality

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

which without loss of generality we always assume to be in *normalized form*; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative integers. We will use equality

$$\textstyle\sum_i a_i \ell_i = A \tag{2a}$$

as syntactic sugar for the pair of inequalities

$$\textstyle\sum_i a_i \ell_i \geq A \tag{2b}$$

$$\textstyle\sum_i - a_i \ell_i \geq -A \tag{2c}$$

(but rewritten in normalized form) and the *negation* $\neg C$ of (1) is (the normalized form of)

$$\textstyle\sum_i - a_i \ell_i \geq -A + 1 \ . \tag{3}$$

A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints. Note that a *clause* $\ell_1 \vee \cdots \vee \ell_k$ is equivalent to the 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 $\{0,1\}$ and a substitution is a (partial) function from variables to *Lits*$\cup\{0,1\}$. For an assignment or substitution ρ we will use the convention $\rho(x) = x$ for x not in the domain of ρ, denoted $x \notin \mathit{dom}(\rho)$, and define $\rho(\overline{x}) = 1 - \rho(x)$. We also write $x \mapsto b$ instead of $\rho(x) = b$ for $b \in \mathit{Lits} \cup \{0,1\}$ when ρ is clear from context or is immaterial. Applying ρ to a constraint C as in (1), denoted $C{\restriction_\rho}$, yields the constraint obtained by substituting values for all assigned variables, shifting constants to the right-hand side, and adjusting the degree appropriately, i.e.,

$$C{\restriction_\rho} = \textstyle\sum_i a_i \rho(\ell_i) \geq A \tag{4}$$

with appropriate normalization, and for a formula F we define $F{\restriction_\rho} = \bigwedge_j C_j{\restriction_\rho}$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$ (or, equivalently, if the restricted constraint (4) has a non-positive degree and is thus trivial). A PB formula is satisfied by ρ if all constraints in it are, in which case it is *satisfiable*. If there is no satisfying assignment, the formula is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

Cutting planes as defined in (Cook, Coullard, and Turán 1987) is a method for iteratively deriving new constraints C implied by a PB formula F. If C and D are previously derived constraints, or are *axiom constraints* in F, then any positive integer *linear combination* of these constraints can be added. We can also add *literal axioms* $\ell_i \geq 0$ at any time. Finally, from a constraint in normalized form

$\sum_i a_i \cdot \ell_i \geq A$ 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 PB formulas F, F' and constraints C, C', we say that F *implies* or *models* C, denoted $F \models C$, if any assignment satisfying F must also satisfy C, and we write $F \models F'$ if $F \models C'$ for all $C' \in F'$. It is not hard to see that any collection of constraints F' derived (iteratively) from F by cutting planes are implied in this sense, and so it holds that F and $F \wedge F'$ are equisatisfiable. A piece of terminology that we will use is that C' is *implied syntactically* by C if C' can be derived from C using only addition of literal axioms.

A constraint C is said to *unit propagate* the literal ℓ under ρ if $C\restriction_\rho$ cannot be satisfied unless $\ell \mapsto 1$. During *unit propagation* on F under ρ, we extend ρ iteratively by any propagated literals $\ell \mapsto 1$ until an assignment ρ' is reached under which no constraint $C \in F$ is propagating, or under which some constraint C propagates a literal that has already been assigned to the opposite value. The latter scenario is referred to as a *conflict*, since ρ' *violates* the constraint C in this case, and ρ' is called a *conflicting* assignment.

Using the generalization of (Goldberg and Novikov 2003) in (Elffers et al. 2020), we say that F implies C by *reverse unit propagation (RUP)*, and write $RUP(F, C)$, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. It is not hard to see that $RUP(F, C)$ implies $F \models C$, but the opposite direction is not necessarily true.

3 Substitution Redundancy

In order to provide proof logging for parity reasoning, we need the ability not only to perform cutting planes reasoning, but also to introduce *fresh variables* not occurring in the formula F under consideration. In particular, we want to be able to use a fresh variable y to encode the *reification* of a constraint $\sum_i a_i \ell_i \geq A$, i.e., that y is true if and only if the constraint is satisfied. We will use the shorthand

$$y \leftrightarrow \textstyle\sum_i a_i \ell_i \geq A \tag{5}$$

for the two constraints

$$A\overline{y} + \textstyle\sum_i a_i \ell_i \geq A \tag{6a}$$

$$\left(-A+1+\textstyle\sum_i a_i\right) \cdot y + \textstyle\sum_i a_i \overline{\ell}_i \geq -A + 1 + \textstyle\sum_i a_i \tag{6b}$$

enforcing this condition. By way of a concrete example, the reification of the constraint

$$x_1 + x_2 + x_3 \geq 2 \tag{7}$$

using y is encoded as

$$2\overline{y} + x_1 + x_2 + x_3 \geq 2 \tag{8a}$$

$$2y + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 2 \tag{8b}$$

in pseudo-Boolean form. Note that introducing such constraints maintains equisatisfiability provided that y does not appear in any other constraint, since depending on whether (7) is satisfied or not we can assign y freely to satisfy (8a) or (8b) as needed.

More generally, it would be convenient to allow the "derivation" of any constraint C from F such that F and $F \wedge C$ are equisatisfiable—in which case we say that C is *redundant with respect to* F—regardless of whether $F \models C$ holds or not. A moment of thought reveals that such a completely generic rule would be too good to be true—for any unsatisfiable formula F we would then be able to "derive" contradiction (say, $0 \geq 1$) in just one step, and this clearly would not be efficiently verifiable. What we need, therefore, is a sufficient criterion for redundancy of pseudo-Boolean constraints that is simple to verify. To this end, we generalize the characterization of redundancy in (Heule, Kiesl, and Biere 2017; Buss and Thapen 2019) from CNF formulas to PB formulas as follows.

Proposition 1 (Substitution redundancy)**.** A PB constraint C is redundant with respect to the formula F if and only if there is a substitution ω, called a *witness*, for which it holds that

$$F \wedge \neg C \models (F \wedge C)\restriction_\omega \ .$$

Proof. ($\Rightarrow$) Suppose C is redundant. If F is unsatisfiable, then for any constraint C' it vacuously holds that $F \models C'$. Hence, any substitution ω fulfils the condition. If F is satisfiable, then $F \wedge C$ must also be satisfiable as C is redundant by assumption. If we choose ω to be a satisfying assignment for $F \wedge C$, the implication in the proposition again vacuously holds since $(F \wedge C)\restriction_\omega$ is fixed to true.

($\Leftarrow$) Suppose now that ω is such that $F \wedge \neg C \models (F \wedge C)\restriction_\omega$. If F is unsatisfiable, then every constraint is redundant and there is nothing to check. Otherwise, let α be a (total) satisfying assignment for F. If α also satisfies C, then clearly the constraint is redundant. Now consider the case that α does not satisfy C. If so, α must satisfy $\neg C$ and hence, by the assumed implication, also $(F \wedge C)\restriction_\omega$. But then the assignment β defined by

$$\beta(x) = \begin{cases} \alpha(x) & \text{if } x \notin \mathit{dom}(\omega), \\ \omega(x) & \text{otherwise}, \end{cases} \tag{9}$$

satisfies both C and F (since $(F \wedge C)\restriction_\beta = ((F \wedge C)\restriction_\omega)\restriction_\alpha$ by construction), so $F \wedge C$ is satisfiable. $\square$

We remark that this proof does not make use of that we are operating with a pseudo-Boolean constraint C—we only need that the negation $\neg C$ is easy to represent in the same formalism. Thus, the argument generalizes to other types of constraints with this property.

Let us return to our example reification of the constraint in (7) and show how this can be derived using substitution redundancy. Let us write C_{8a} for the constraint in (8a) and C_{8b} for (8b), where y is fresh with respect to the current formula F. To show that C_{8b} is substitution redundant with respect to F we choose the witness $\omega = \{ y \mapsto 1 \}$, which clearly satisfies C_{8b}. Since y does not appear in F we have $F\restriction_\omega = F$, and so the implication $F \wedge \neg C_{8b} \models (F \wedge C)\restriction_\omega$ vacuously holds. Showing that C_{8a} is substitution redundant with respect to $F \wedge C_{8b}$ is a bit more interesting. For this we choose $\omega = \{ y \mapsto 0 \}$, which satisfies C_{8a} and again leaves F unchanged. Thus, the only implication for which we need to do some work is $F \wedge C_{8b} \wedge \neg C_{8a} \models C_{8b}\restriction_\omega$. The negation of C_{8a} is

$$-2\overline{y} - x_1 - x_2 - x_3 \geq -1 \ , \tag{10}$$

Algorithm 1 Checking substitution redundancy

```
1: procedure RedundancyCheck($F, C, \omega$)
2:         ▷ $C, \omega$ are given in the proof log to be verified
3:     if $RUP(F, C)$ then return pass
4:     for $D \in (F \wedge C)\restriction_\omega$ do
5:         if (not ($D \in F$ or $\neg C \models D$ syntactically)
6:             and not $RUP(F \wedge \neg C, D)$) then
7:             return fail
8:     return pass
```

or, converted to normalized form,

$$2y + \bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 4 \tag{11}$$

using the rewriting rule $\ell = 1 - \bar{\ell}$. Adding the literal axiom $\bar{y} \geq 0$ twice to $\neg C_{8a}$, and using rewriting again to cancel $y + \bar{y} = 1$, we obtain

$$\bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 2, \tag{12}$$

which is $C_{8b}\restriction_\omega$. Hence, $\neg C_{8a}$ syntactically implies $C_{8b}\restriction_\omega$, and so $F \wedge C_{8b} \wedge \neg C_{8a} \models C_{8b}\restriction_\omega$, completing the proof that C_{8a} is redundant with respect to $F \wedge C_{8b}$.

So far, we have not discussed how the implications are verified. Arbitrary implication checks are as hard as determining satisfiability, and hence a certificate that the implication is correct is necessary for efficient verification. One way of providing a certificate could be to exhibit a cutting planes derivation establishing the validity of the implication, as in the example just presented. A more convenient alternative from a proof logging point of view is to follow the lead of *DRAT* and only allow constraints for which the implication can be verified using unit propagation. We describe our pseudo-Boolean version of this method in Algorithm 1. This algorithm is very similar to what is used for checking *DRAT*, except that our unit propagation is on PB constraints rather than clauses and that we need an extra syntactic check on line 5. To see why this check is necessary, note that only unit propagation would fail to certify the correctness of our example above. Assuming for simplicity that $F = \emptyset$, if we try to verify $C_{8b} \wedge \neg C_{8a} \models C_{8b}\restriction_\omega$ by reverse unit propagation we get the constraints

$$2y + \bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 2 \qquad [C_{8b} \text{ in (8b)}] \tag{13a}$$
$$2y + \bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 4 \qquad [\neg C_{8a} \text{ in (11)}] \tag{13b}$$
$$x_1 + x_2 + x_3 \geq 2 \qquad [\neg(C_{8b}\restriction_\omega)] \tag{13c}$$

and although visual inspection shows that this collection of constraints is inconsistent, since it requires a majority of the variables $\{ x_1, x_2, x_3 \}$ to be true and false at the same time, unit propagation is too myopic to see this contradiction and only yields $y \mapsto 1$. Using Algorithm 1, however, allows us to introduce extension variables encoding reifications $y \leftrightarrow C$, which is straightforward to prove by carrying out the same argument as in our example above but for (6a) and (6b) instead of (8a) and (8b).

Proposition 2. Let F be a PB formula and C be a PB constraint, and y is a fresh variable that does not appear in F or C. Then the constraints (6a) and (6b) encoding $y \leftrightarrow C$ can be added to F and checked as redundant by Algorithm 1.

4 Proof Logging for XOR Constraints

An *XOR* or *parity constraint*, i.e., an equality modulo 2, over k variables is written as

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = b \tag{14}$$

where $b \in \{ 0, 1 \}$. Note that we can assume that there is no parity constraint with a negated variable $\bar{x}$, because we can always substitute $\bar{x} = x \oplus 1$.

Systems of XOR constraints can arise in a solver during Gaussian elimination (Soos, Nohl, and Castelluccia 2009; Han and Jiang 2012; Laitinen, Junttila, and Niemelä 2012) or conflict analysis (Laitinen, Junttila, and Niemelä 2012). To provide proof logging for these approaches we need four ingredients:

1. An efficient encoding of XORs to pseudo-Boolean constraints.
2. A way to derive that encoding for a new XOR constraint from the encodings of existing XORs.
3. A method to translate to this efficient encoding from CNF (which is where we will need to go beyond cutting planes by using extension variables).
4. The ability to provide so-called *reason clauses* from the PB encoding that can be used by a SAT solver during conflict analysis.

In the example in Section 5, we will see how these ingredients come together for propagations from parity constraints.

It was observed by (Dixon, Ginsberg, and Parkes 2004) that XOR constraints can be encoded and refuted efficiently in pseudo-Boolean form by rewriting (14) as

$$\textstyle\sum_{i\in[k]} x_i = b + \sum_{i\in[\lfloor k/2 \rfloor]} 2y_i \tag{15}$$

for fresh variables y_i (where we recall that equality (2a) is a shorthand for (2b) and (2c)). Since the variables y_i are otherwise unconstrained, the right-hand side can take any even (odd) value for $b = 0$ $(b = 1)$ in the range from 0 to k, which are exactly the values that we want to allow for $\sum_{i\in[k]} x_i$.

In fact, we can generalize this by observing that if we let $\mathcal{B}$ denote any integer linear combination of variables, possibly also with a constant term, then the two inequalities

$$\textstyle\sum_{i\in[k]} x_i \geq b + 2\mathcal{B} \tag{16a}$$
$$\textstyle\sum_{i\in[k]} - x_i \geq -b - 2\mathcal{B} \ , \tag{16b}$$

forming the equality $\sum_{i\in[k]} x_i = b + 2\mathcal{B}$, imply the parity

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = b \ . \tag{16c}$$

We will make repeated use of this observation below.

XOR Reasoning

Whenever we want to combine two XOR constraints to derive a new XOR constraint, as is done during Gaussian elimination, we only need to add the equalities that imply it. For example, suppose that we want to do the derivation

$$\frac{x_1 \oplus x_2 \oplus x_3 = 1 \qquad x_2 \oplus x_3 \oplus x_4 = 1}{x_1 \oplus x_4 = 0} \tag{17}$$

and assume the XORs are represented in pseudo-Boolean form as $x_1 + x_2 + x_3 = 2y_1 + 1$ and $x_2 + x_3 + x_4 = 2y_2 + 1$. Then adding both equalities together we obtain $x_1 + 2x_2 + 2x_3 + x_4 = 2y_1 + 2y_2 + 2$, which implies the desired XOR by the observation we just made above for (16c).

Reason Generation

Modern SAT solvers built on *conflict-driven clause learning (CDCL)* (Bayardo Jr. and Schrag 1997; Marques-Silva and Sakallah 1999; Moskewicz et al. 2001) operate with clauses. If we want to use XOR constraints to propagate forced variable assignments or derive contradiction, then we need to provide *reason clauses* that justify such derivation steps. We next show how to derive such reason clauses from pseudo-Boolean encodings of XOR constraints.

Suppose we have a parity constraint encoded by inequalities of the form (16a) and (16b), and let ρ be an assignment to the k variables x_i that is inconsistent with (16a) and (16b), because it falsifies the implied XOR (16c). We want to derive a clause that is falsified under ρ.

Let $\mathcal{T}$ be the variables that are set to true under ρ, and $\mathcal{F}$ the variables set to false. Using literal axioms we can derive (the normalized form of) the trivially true constraint

$$\textstyle\sum_{x\in\mathcal{F}} x + \sum_{x\in\mathcal{T}} -x \geq -|\mathcal{T}| \ , \tag{18}$$

which when added to (16a), yields

$$\textstyle\sum_{x\in\mathcal{F}} 2x \geq b - |\mathcal{T}| + 2\mathcal{B} \ . \tag{19}$$

Observe that $b - |\mathcal{T}|$ is always odd, as otherwise ρ would not falsify the XOR implied by (16a) and (16b), while everything else is divisible by 2. Hence we can divide by 2, and rounding up will increase the degree. If we now multiply by 2 again, then we get

$$\textstyle\sum_{x\in\mathcal{F}} 2x \geq b - |\mathcal{T}| + 1 + 2\mathcal{B} \ . \tag{20}$$

We continue by adding (16b) to get

$$\textstyle\sum_{x\in\mathcal{F}} x - \sum_{x\in\mathcal{T}} x \geq 1 - |\mathcal{T}| \ , \tag{21}$$

which is equivalent to the normalized constraint

$$\textstyle\sum_{x\in\mathcal{F}} x + \sum_{x\in\mathcal{T}} \overline{x} \geq 1 \ . \tag{22}$$

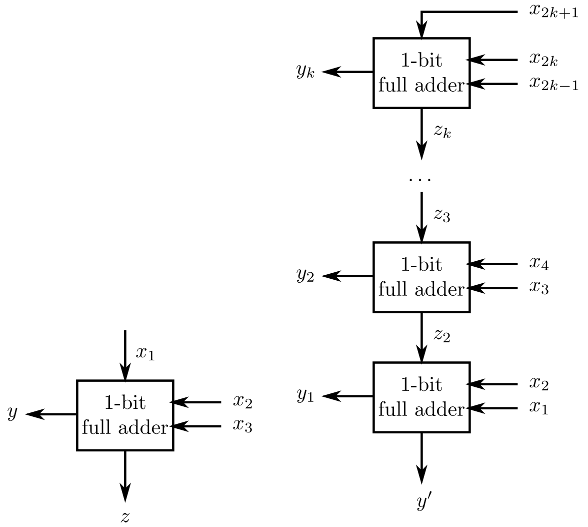
This constraint, which is a disjunctive clause, is falsified under ρ as desired, and is the reason clause that we need to give to the solver to show why the assignment ρ is inconsistent.

Translating to the Pseudo-Boolean XOR Encoding

An XOR as in (14) can be encoded into CNF by having a clause for each of the 2^{k-1} assignments that falsify the constraint. For example, for $k = 3$ and $b = 1$ we get the clauses

$$x_1 + x_2 + x_3 \geq 1 \tag{23a}$$
$$\overline{x}_1 + \overline{x}_2 + x_3 \geq 1 \tag{23b}$$
$$\overline{x}_1 + x_2 + \overline{x}_3 \geq 1 \tag{23c}$$
$$x_1 + \overline{x}_2 + \overline{x}_3 \geq 1 \tag{23d}$$

(in pseudo-Boolean form). Since the number of clauses in this canonical CNF encoding of an XOR constraint scales exponentially with the number of variables, it is only feasible to encode short XORs into CNF in this way. However, it is possible to split up a long XOR into multiple constant-size XORs using auxiliary variables z_i. For example, (14) can be represented as $x_1 \oplus x_2 \oplus z_2 = 0$, $z_2 \oplus x_3 \oplus z_3 = 0$, $\ldots$, $z_{k-2} \oplus x_{k-1} \oplus x_k = b$. If a parity constraint is split up in this way, we only need to re-encode these small parity constraints from CNF into the pseudo-Boolean encoding. Deriving the original long XOR constraint can then be done by XOR reasoning as described earlier.



(a) A 1-bit full adder. (b) Chain of 1-bit full adders.

Figure 1: The output of 1-bit full adders is used to encode new auxiliary Boolean variables, which represent the sum of all Boolean input variables.

The translation to the pseudo-Boolean XOR encoding from CNF is done in two steps. The first step is to derive the constraint

$$\textstyle\sum_{i\in[k]} x_i = \sum_{i\in[\lfloor k/2 \rfloor]} 2y_i + y', \tag{24}$$

where y_i and y' are all fresh variables. Note that adding (24) to any formula does not change satisfiability because we can always assign the fresh variables so that the equality holds.

Although the constraint is redundant, we cannot use Proposition 1 directly, because we can not construct a witness assignment ω that is independent of the existing x_i variables, and if ω contains any existing variables then the redundancy check in Algorithm 1 may not be strong enough in general. Instead we introduce each fresh variable individually, analogously to what was done in Section 3.

The second step is to brute-force all possible assignments for the variables x_i, which together with the CNF encoding of the XOR allows us to derive $y' = b$, meaning that we have a constraint of the desired form (24). We remark that this brute-force step is polynomial in the number of clauses of the CNF encoding. We now describe this process in detail.

Step 1a. To derive (24) we will construct a chain of 1-bit full adders (see Figure 1b). But let us start first by showing how the encoding of a single 1-bit full adder can be derived. A 1-bit full adder (Figure 1a) computes the sum of three variables x_1 to x_3 and returns the result as a binary number. This can be encoded using the pseudo-Boolean equality

$$2y + z = x_1 + x_2 + x_3. \tag{25}$$

To obtain (25) we start by deriving the reifications

$$y \leftrightarrow x_1 + x_2 + x_3 \geq 2 \tag{26a}$$
$$z \leftrightarrow x_1 + x_2 + x_3 - 2y \geq 1, \tag{26b}$$

for fresh variables y and z using substitution redundancy as described in Proposition 2. Writing the constraints in normalized form yields

$$x_1 + x_2 + x_3 + 2\bar{y} \geq 2 \tag{27a}$$
$$\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + 2y \geq 2 \tag{27b}$$
$$x_1 + x_2 + x_3 + 2\bar{y} + 3\bar{z} \geq 3 \tag{27c}$$
$$\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + 2y + 3z \geq 3. \tag{27d}$$

To derive the less-than-or-equal part of (25), which in normalized form is

$$x_1 + x_2 + x_3 + 2\bar{y} + \bar{z} \geq 3 \ , \tag{28}$$

we add Equation (27c) and 2 times Equation (27a) followed by division by 3. In a similar fashion, to derive the greater-than-or-equal part of (25), which in normalized form is

$$\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + 2y + z \geq 3 \ , \tag{29}$$

we add Equation (27d) and 2 times Equation (27b) followed by division by 3.

Step 1b. To derive Equation (24) we use a chain of 1-bit full adders (Figure 1b). The x_i variables are used as input and x_{2k+1}, which is used in the topmost adder, will only occur if the number of variables is odd, otherwise the topmost adder will only have x_{2k} and x_{2k-1} as input. The variables y_i, y' encode the sum of the input variables as required for Equation (24). The z_i variables are intermediate parity bits. For the topmost adder in Figure 1b we thus have

$$2y_k + z_k = x_{2k+1} + x_{2k} + x_{2k-1} \ , \tag{30}$$

for the intermediate adders we have, for $i \in \{2, \ldots, k-1\}$,

$$2y_i + z_i = z_{i+1} + x_{2i} + x_{2i-1} \ , \tag{31}$$

and for the bottom adder we have

$$2y_1 + y' = z_2 + x_2 + x_1 \ . \tag{32}$$

By adding the encoding of all 1-bit adders, i.e., Equations (30) to (32), we obtain

$$\sum_{i=1}^{k} 2y_i + y' + \sum_{i=2}^{k} z_i = \sum_{i=2}^{k} z_i + \sum_{i=1}^{2k+1} x_i \ . \tag{33}$$

Note that $\sum_{i=2}^{k} z_i$ appears on both sides of the equation and hence Equation (33) is equivalent to Equation (24). Indeed, we do not need to remove $\sum_{i=2}^{k} z_i$ explicitly during proof logging because it will disappear automatically due to constraint normalization.

Step 2. The final step is to obtain the value of y' by brute-forcing all values of x_i. Consider an assignment ρ with $\rho(y') = 1 - b$ that additionally assigns all variables x_i and no other variables. Note that there are 2^k such assignments. Either ρ falsifies one of the clauses that encode the XOR, or else $\sum_{i \in [k]} \rho(x_i) \bmod 2 = b \neq \rho(y')$, meaning that Equation (33) is falsified so that we can derive a clause falsified under ρ as described above in our discussion of reason generation. Combining these 2^k clauses together we can obtain a clause that forces y' to take value b, which we can add to Equation (33) to replace y' with a constant value. This concludes the derivation of the pseudo-Boolean encoding (15) of the XOR constraint from a CNF encoding.

5 A Worked-Out Proof Logging Example

Consider the two parity constraints $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_2 \oplus x_3 \oplus x_4 = 1$, which are encoded as clauses by writing

```
* #variable= 4 #constraint= 8
+1 ~x1 +1  x2 +1  x3 >= 1 ;
+1  x1 +1 ~x2 +1  x3 >= 1 ;
+1  x1 +1  x2 +1 ~x3 >= 1 ;
+1 ~x1 +1 ~x2 +1 ~x3 >= 1 ;
+1  x2 +1  x3 +1  x4 >= 1 ;
+1  x2 +1 ~x3 +1 ~x4 >= 1 ;
+1 ~x2 +1  x3 +1 ~x4 >= 1 ;
+1 ~x2 +1 ~x3 +1  x4 >= 1 ;
```

using the standard OPB file format[1]. The solver reads this formula and runs an algorithm to detect clausal encodings of parities. Once a parity is detected, a proof is generated that translates the parity from the clausal encoding into the PB encoding. For this translation it is necessary to introduce fresh variables via substitution redundancy. The proof log contains a line of the form

```
red [constraint C] ; [assignment omega]
```

where `red` identifies the line as a substitution redundancy step, followed by the constraint C to be added and the witness substitution ω, where each variable to be substituted is listed followed by its substitution. A variable and its substitution can optionally be separated by '->'.

The translation from clausal to PB encoding starts with the reification $y_1 \leftrightarrow x_1 + x_2 + x_3 \geq 2$ for the fresh variable y_1. In the proof format this is done by the two lines

```
red 1  x1 1  x2 1  x3 2 ~y1 >= 2 ; y1 -> 0
red 1 ~x1 1 ~x2 1 ~x3 2  y1 >= 2 ; y1 -> 1
```

which can be checked using Algorithm 1 as discussed in Section 3. After the check passes, the verifier adds these two new constraints to the database and assigns them ids 9 and 10 (the ids 1 to 8 are used for the constraints from the input formula). Similarly, we can do another reification $y_2 \leftrightarrow x_1 + x_2 + x_3 - 2y_1 \geq 1$ using the proof lines

```
red 1  x1 1 x2 1 x3 2 ~y1 3 ~y2 >= 3 ; y2 0
red 1 ~x1 1 ~x2 1 ~x3 2 y1 3 y2 >= 3 ; y2 1
```

The variables y_1, y_2 now correspond to the output bits of a single full adder. Because the parity is only over three variables, we do not need to derive a chain of multiple full adders. Note that the constraints arising from reification do not need to be added to the database of the solver—they are only used for the proof log—but the solver should stay clear of using the variables y_1, y_2 for other purposes.

The next step is to combine the constraints we just derived (ids 9 to 12) via a sequence of cutting planes steps, which are

[1]http://www.cril.univ-artois.fr/PB16/format.pdf

written down in reverse polish notation (using the p-rule in *VeriPB*), also known as postfix notation

```
p 11 9 2 * + 3 d
p 12 10 2 * + 3 d
```

The first line starts with the constraint with id 11 and adds two times the constraint with id 9 and then divides by 3 and rounds up. The same operations are done in the second line but with the constraints with ids 12 and 10. The two lines derive the constraints

$$\text{(ID: 13)} \qquad x_1 + x_2 + x_3 + 2\bar{y}_1 + \bar{y}_2 \geq 3 \tag{34a}$$

$$\text{(ID: 14)} \qquad \bar{x}_1 + \bar{x}_2 + \bar{x}_3 + 2y_1 + y_2 \geq 3 \ , \tag{34b}$$

which correspond to (24). The constraints in (34a) and (34b) do not correspond to a parity constraint yet, as they can always be satisfied by setting the fresh variables y_1, y_2 to the right value. To get a proper PB encoding of the first parity we need to fix the value of y_2, which can be done by generating a brute-force proof via p-rules that derives $\bar{y}_2 \geq 1$, which gets id 15 and can be used to remove y_2 from (34b). To remove y_2 from (34a) we can simply use the literal axiom $y_2 \geq 0$ which is obtained by writing the literal `y2` in the p-rule. Both steps together can be written as

```
p 13 y2 +
p 14 15 +
```

deriving the inequalities encoding the first parity, namely

$$\text{(ID: 16)} \qquad x_1 + x_2 + x_3 + 2\bar{y}_1 \geq 2 \tag{35a}$$

$$\text{(ID: 17)} \qquad \bar{x}_1 + \bar{x}_2 + \bar{x}_3 + 2y_1 \geq 3 \ , \tag{35b}$$

which correspond to (15). Analogously, we can derive the pseudo-Boolean encoding for the second parity

$$\text{(ID: 25)} \qquad x_2 + x_3 + x_4 + 2\bar{y}_3 \geq 3 \tag{36a}$$

$$\text{(ID: 26)} \qquad \bar{x}_2 + \bar{x}_3 + \bar{x}_4 + 2y_3 \geq 4 \ . \tag{36b}$$

This concludes the proof logging done after detecting parities. Note that the detection of parities and the proof generation for translating from clausal to PB encoding is only done once in our implementation at the start of the solver.

Let us now assume that the solver decides $x_1 = 0$. Note that adding the two parities of the formula yields $x_1 \oplus x_4 = 1$, and hence x_4 should propagate to 1, which is detected by the XOR propagator via Gaussian elimination. The proof for this step is to perform the same addition of the parities, but on their PB representations

```
p 16 25 +
p 17 26 +
```

which yields new constraints

$$\text{(ID: 27)} \quad x_1 + 2x_2 + 2x_3 + x_4 + 2\bar{y}_1 + 2\bar{y}_3 \geq 5 \tag{37a}$$

$$\text{(ID: 28)} \quad \bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 + \bar{x}_4 + 2y_1 + 2y_3 \geq 7. \tag{37b}$$

These two constraints imply the parity $x_1 \oplus x_4 = 1$ by the observation made in connection with (16a)–(16c).

The reason clause $x_1 + x_4 \geq 1$ provided by the XOR propagator must be derived in the proof. The assignment falsifying the reason clause is $\rho = \{ x_1 \mapsto 0, x_4 \mapsto 0 \}$. Following the approach in (18)–(22), we derive $x_1 + x_4 \geq 0$ and combine it with the constraints representing the parity

Instance	*MiniSat* + XOR (*PBP*)	(*DRAT*)	*PR2DRAT*
Urquhart-s5-b1	76.8	3033.1	3878.4
Urquhart-s5-b2	79.8	2844.4	3575.2
Urquhart-s5-b3	116.9	7584.0	7521.0
Urquhart-s5-b4	94.7	5058.6	5271.5

Table 1: Proof sizes (KiB) for previous Tseitin formulas.

```
p 27 x1 x4 + + 2 d 2 * 28 +
```

which derives the constraint $x_1 + x_4 \geq 1$ as desired. The solver can continue using this clause in the same way as any other clause in its database. These steps for XOR reasoning and reason generation are repeated for every propagation.

6 Implementation and Evaluation

As just illustrated in Section 5, we have added a rule to *VeriPB*[2] and its pseudo-Boolean proof format (*PBP*) to support redundancy checks as described in Algorithm 1, and have implemented our proof logging approach for XOR reasoning in a library together with an XOR engine using Gaussian elimination $\bmod 2$ to detect XOR propagations.[3] We integrated this library into *MiniSat* to call the XOR propagation method every time propagation reached fix point. If the library detects a propagation or conflict a callback is used to notify *MiniSat*, but the reason clause is only generated when needed in conflict analysis. This *lazy reason generation* technique (Soos, Gocht, and Meel 2020) is crucial, since it avoids generating proofs for reasons that are not used. For comparison, our library also provides *DRAT* proof logging for XORs as described in (Philipp and Rebola-Pardo 2016).

Importantly, our goal was not to investigate whether XOR reasoning is useful—this is already known—but to provide efficient proof logging for such reasoning. Therefore, we focused on SAT competition benchmarks from the last 5 years that could be solved by *MiniSat* with our XOR propagator but not by *Kissat*, the winner of the 2020 SAT competition. There were 39 such instances, and they could be solved in 0.03 seconds on average by *MiniSat* with the XOR propagator. With our new proof logging the average running time increased to 0.05 seconds and unsatisfiability could be verified in 1.71 seconds on average. For *DRAT* proof logging, on the other hand, the average solving time jumped to 7.72 seconds and verification took 3291 seconds on average.

In order to get systematic measurements for the performance of our new proof logging technique, we ran experiments on so-called Tseitin formulas, including some that have been studied before in the context of proof logging. Tseitin formulas consist of a large inconsistent set of parity constraints, and can thus be viewed as a worst case for XOR reasoning. To the best of our knowledge, the shortest DRAT proofs[4] for these formulas obtained so far are based on hand-crafted so-called *propagation redundancy*

[2]https://gitlab.com/miao%5Fresearch/veripb

[3]https://gitlab.com/miao%5Fresearch/xorengine

[4]The proofs and instances can be found at https://github.com/marijnheule/drat2er-proofs

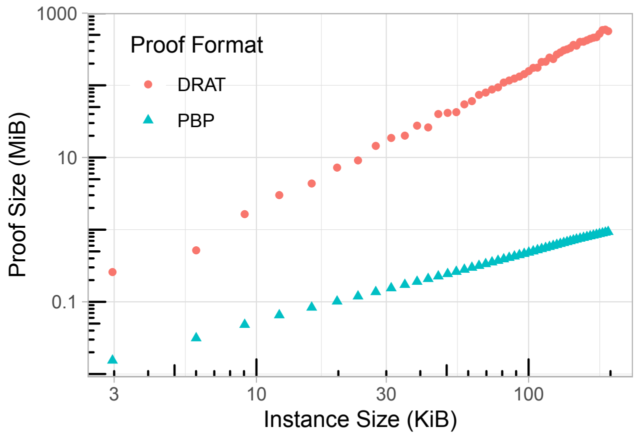


Figure 2: Proof sizes for Tseitin formulas.

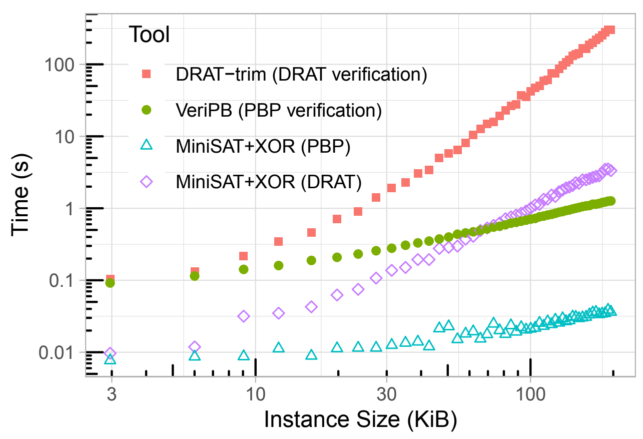


Figure 3: Solving and verification time for Tseitin formulas.

(PR) proofs which have been translated to *DRAT* using the tool *PR2DRAT* (Kiesl, Rebola-Pardo, and Heule 2018). Table 1 shows the disk space required for the proofs of Tseitin formulas in (Kiesl, Rebola-Pardo, and Heule 2018). The pseudo-Boolean proofs obtained by *MiniSat* with the XOR propagator are dramatically smaller than the DRAT proofs it produces, and the size of our DRAT proofs are similar to that of the best previously known DRAT proofs.

To get a sense of the asymptotic behaviour of the proof logging we generated 50 new, larger Tseitin formulas with up to 500 XORs and up to 1250 variables. In Figure 2 we compare the proof size of the proofs as generated. Notice that both proof logging approaches result in a straight line in the log-log plot, which is a strong indication that both approaches are polynomial. Studying the slopes of the lines yields the estimates that *DRAT* produces quadratic-size proofs while the proof size of the pseudo-Boolean proof is linear in the size of the formula. In Figure 3 we compare the running time (system time + user time) of solving and producing the proof, as well as time spend for verification. It is clear that the larger proof size required for *DRAT* proofs does not only increase verification time, but also causes a clearly increased time overhead during solving. All running times were measured on an Intel® Core™ i3-7100U CPU @ 2.40GHz$\times$2 with a memory limit of 8GB, disk write speed of 154 MB/s and read speed of 518 MB/s. The used tools, benchmarks, data and evaluation scripts are available at https://doi.org/10.5281/zenodo.4569840.

7 Conclusion

In this work, we present an efficient method for proof logging parity reasoning in conflict-driven clause learning (CDCL) solvers, which has been a long-standing challenge in SAT solving. Our method circumvents the prohibitive overhead of current *DRAT*-based methods by instead using pseudo-Boolean inequalities with extension variables. An experimental evaluation shows that this makes the proof logging overhead, the size of the proof, and the time required for verification all go down by an order of magnitude or more compared to *DRAT*. While there is certainly ample room for further improvements, our first proof-of-concept implementation already shows the power of this approach.

It is clear that the same method can also be used to solve another task that has remained very challenging for *DRAT*, namely efficient proof logging for cardinality detection and reasoning. We have not investigated this in this paper, since this is mostly an engineering issue rather than a research problem, given the methods that have already been developed in (Biere et al. 2014; Elffers and Nordström 2020).

Dealing with symmetries appears to be much more challenging, but our method can do at least as well as (Heule, Hunt Jr., and Wetzler 2015) since it is a strict generalization of *DRAT*. We would therefore propose that the current extension of the *VeriPB* method in (Elffers et al. 2020) should be an allowed proof logging format in the SAT competitions. This would make it possible for solvers making use of these advanced techniques to take part in the main track of the SAT competitions, where proof logging is mandatory.

However, we believe that the potential benefit of PB proof logging with extension variables goes well beyond the SAT competitions. The original *VeriPB* method is capable of efficient justification of important constraint programming techniques (Elffers et al. 2020), and can also provide proof logging for a wide range of graph problem solvers (Gocht, McCreesh, and Nordström 2020; Gocht et al. 2020). The pseudo-Boolean rules for reasoning with 0-1 linear constraints provide a simple yet very expressive formalism, and it does not seem out of the question to hope that they could be extended to deal with mixed integer programming (MIP). Thus, we believe that the ultimate goal of this line of research should be to design a unified proof logging approach for as wide as possible a range of combinatorial optimization paradigms. In addition to furnishing efficient machine-verifiable proofs of correctness, proof logging could also serve as a valuable tool for debugging and empirical performance analysis during solver development. Furthermore, the proofs produced could in principle provide auditability by third parties using independently developed software, and/or be a stepping stone towards explainability by showing, e.g., why certain solutions are optimal.

Acknowledgments

We want to thank Mate Soos and Kuldeep Meel for helpful discussions on how to implement Gaussian elimination modulo 2. The authors 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.

References

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

Allouche, D.; André, I.; Barbe, S.; Davies, J.; Givry, S. d.; Katsirelos, G.; O'Sullivan, B.; Prestwich, S.; Schiex, T.; and Traoré, S. 2014. Computational Protein Design as an Optimization Problem. *Artificial Intelligence* 212(1): 59–79.

Archibald, B.; Dunlop, F.; Hoffmann, R.; McCreesh, C.; Prosser, P.; and Trimble, J. 2019. Sequential and Parallel Solution-Biased Search for Subgraph Algorithms. In *Proceedings of the 16th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '19)*, volume 11494 of *Lecture Notes in Computer Science*, 20–38. Springer.

Bayardo Jr., R. J.; and Schrag, R. 1997. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, 203–208.

Biere, A. 2006. TraceCheck. http://fmv.jku.at/tracecheck/.

Biere, A.; Le Berre, D.; Lonca, E.; and Manthey, N. 2014. Detecting Cardinality Constraints in CNF. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, 285–301. Springer.

Buss, S. R.; and Nordström, J. 2021. Proof Complexity and SAT Solving. In Biere, A.; Heule, M. J. H.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, 233–350. IOS Press, 2nd edition.

Buss, S. R.; and 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)*, volume 11628 of *Lecture Notes in Computer Science*, 71–89. Springer.

Cook, W.; Coullard, C. R.; and Turán, G. 1987. On the Complexity of Cutting-Plane Proofs. *Discrete Applied Mathematics* 18(1): 25–38.

Cook, W.; Koch, T.; Steffy, D. E.; and Wolter, K. 2013. A Hybrid Branch-and-Bound Approach for Exact Rational Mixed-Integer Programming. *Mathematical Programming Computation* 5(3): 305–344.

Cruz-Filipe, L.; Heule, M. J. H.; Hunt, W. A.; Kaufmann, M.; and Schneider-Kamp, P. 2017. Efficient Certified RAT Verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, 220–236. Springer.

Cruz-Filipe, L.; Marques-Silva, J.; and Schneider-Kamp, P. 2017. 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)*, volume 10205 of *Lecture Notes in Computer Science*, 118–135. Springer.

Delorme, M.; García, S.; Gondzioa, J.; Kalcsics, J.; Manlove, D.; and Pettersson, W. 2019. Mathematical Models for Stable Matching Problems with Ties and Incomplete Lists. *European Journal of Operational Research* 277(2): 426–441.

Dixon, H. E.; Ginsberg, M. L.; and Parkes, A. J. 2004. Generalizing Boolean Satisfiability I: Background and Survey of Existing Work. *Journal of Artificial Intelligence Research* 21: 193–243.

Eén, N.; and Sörensson, N. 2004. An Extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Elffers, J.; Gocht, S.; McCreesh, C.; and Nordström, J. 2020. Justifying All Differences Using Pseudo-Boolean Reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, 1486–1494.

Elffers, J.; and Nordström, J. 2018. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, 1291–1299.

Elffers, J.; and Nordström, J. 2020. A Cardinal Improvement to Pseudo-Boolean Solving. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, 1495–1503.

Gange, G.; and Stuckey, P. 2019. Certifying Optimality in Constraint Programming. Presentation at KTH Royal Institute of Technology. Slides available at https://www.kth.se/polopoly_fs/1.879851.1550484700!/CertifiedCP.pdf.

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

Gocht, S.; McBride, R.; McCreesh, C.; Nordström, J.; Prosser, P.; and Trimble, J. 2020. 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)*, volume 12333 of *Lecture Notes in Computer Science*, 338–357. Springer.

Gocht, S.; McCreesh, C.; and Nordström, J. 2020. Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, 1134–1140.

Goldberg, E.; and 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)*, 886–891.

Han, C.; and Jiang, J. R. 2012. When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way. In *Proceedings of the 24th International Conference on Computer Aided Verification, (CAV '12)*, volume 7358 of *Lecture Notes in Computer Science*, 410–426. Springer.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2013a. Trimming While Checking Clausal Proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, 181–188.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2013b. Verifying Refutations with Extended Resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, 345–359. Springer.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2015. Expressing Symmetry Breaking in DRAT Proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, 591–606. Springer.

Heule, M. J. H.; Kiesl, B.; and Biere, A. 2017. Short Proofs Without New Variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, 130–147. Springer.

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

Laitinen, T.; Junttila, T.; and Niemelä, I. 2012. Conflict-Driven XOR-Clause Learning. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT '12)*, volume 7317 of *Lecture Notes in Computer Science*, 383–396. Springer.

Laitinen, T.; Junttila, T.; and Niemelä, I. 2012. Extending Clause Learning SAT Solvers with Complete Parity Reasoning. In *Proceedings of the IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI '12)*, 65–72.

Le Berre, D.; and Parrain, A. 2010. The Sat4j Library, Release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7: 59–64.

Leyton-Brown, K.; Milgrom, P.; and Segal, I. 2017. Economics and Computer Science of a Radio Spectrum Reallocation. *Proceedings of the National Academy of Sciences* 114(28): 7202–7209.

Manlove, D. F. 2016. Hospitals/Residents Problem. In Kao, M.-Y., ed., *Encyclopedia of Algorithms*, 926–930. Springer New York.

Manlove, D. F.; McBride, I.; and Trimble, J. 2017. "Almost-stable" Matchings in the Hospitals / Residents Problem with Couples. *Constraints* 22(1): 50–72.

Manlove, D. F.; and O'Malley, G. 2012. Paired and Altruistic Kidney Donation in the UK: Algorithms and Experimentation. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA '12)*, volume 7276 of *Lecture Notes in Computer Science*, 271–282. Springer.

Mann, M.; Will, S.; and Backofen, R. 2008. CPSP-tools – Exact and Complete Algorithms for High-Throughput 3D Lattice Protein Studies. *BMC Bioinformatics* 9: 230:1–230:8.

Marques-Silva, J. P.; and Sakallah, K. A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers* 48(5): 506–521. Preliminary version in *ICCAD '96*.

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

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, 530–535.

Pardalos, P. M.; Du, D.-Z.; and Graham, R. L., eds. 2013. *Handbook of Combinatorial Optimization*. Springer, 2nd edition.

Philipp, T.; and Rebola-Pardo, A. 2016. DRAT Proofs for XOR Reasoning. In *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA '16)*, volume 10021 of *Lecture Notes in Computer Science*, 415–429. Springer.

Sinz, C.; and Biere, A. 2006. Extended Resolution Proofs for Conjoining BDDs. In *Proceedings of the 1st International Computer Science Symposium in Russia (CSR '06)*, volume 3967 of *Lecture Notes in Computer Science*, 600–611. Springer.

Soos, M.; Gocht, S.; and Meel, K. S. 2020. Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling. In *Proceedings of the 32nd International Conference on Computer Aided Verification, (CAV '20)*, volume 12224 of *Lecture Notes in Computer Science*, 463–484. Springer.

Soos, M.; Nohl, K.; and Castelluccia, C. 2009. Extending SAT Solvers to Cryptographic Problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, 244–257. Springer.

Veksler, M.; and Strichman, O. 2010. A Proof-Producing CSP Solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, 204–209.

Wetzler, N.; Heule, M. J. H.; and 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)*, volume 8561 of *Lecture Notes in Computer Science*, 422–429. Springer.