Certified CNF Translations for Pseudo-Boolean Solving

Stephan Gocht  
Lund University, Lund, Sweden  
University of Copenhagen, Copenhagen, Denmark

Ruben Martins  
Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Jakob Nordström  
University of Copenhagen, Copenhagen, Denmark  
Lund University, Lund, Sweden

Andy Oertel  
Lund University, Lund, Sweden  
University of Copenhagen, Copenhagen, Denmark

Abstract

The dramatic improvements in Boolean satisfiability (SAT) solving since the turn of the millennium have made it possible to leverage state-of-the-art conflict-driven clause learning (CDCL) solvers for many combinatorial problems in academia and industry, and the use of proof logging has played a crucial role in increasing the confidence that the results these solvers produce are correct. However, the conjunctive normal form (CNF) format used for SAT proof logging means that it has not been possible to extend guarantees of correctness to the use of SAT solvers for more expressive combinatorial paradigms, where the first step is to translate the input to CNF.

In this work, we show how cutting-planes-based reasoning can provide proof logging for solvers that translate pseudo-Boolean (a.k.a. 0-1 integer linear) decision problems to CNF and then run CDCL. To support a wide range of encodings, we provide a uniform and easily extensible framework for proof logging of CNF translations. We are hopeful that this is just a first step towards providing a unified proof logging approach that will also extend to maximum satisfiability (MaxSAT) solving and pseudo-Boolean optimization in general.

2012 ACM Subject Classification Theory of computation → Program verification; Hardware → Theorem proving and SAT solving; Theory of computation → Logic and verification

Keywords and phrases pseudo-Boolean solving, 0-1 integer linear program, proof logging, certified translation, CNF encoding, cutting planes

Digital Object Identifier 10.4230/LIPIcs...

Funding Stephan Gocht: Swedish Research Council grant 2016-00782.
Ruben Martins: National Science Foundation award CCF-1762363 and Amazon Research Award.
Jakob Nordström: Swedish Research Council grant 2016-00782 and Independent Research Fund Denmark grant 9040-00389B.
Andy Oertel: Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

1 Introduction

Boolean satisfiability (SAT) has witnessed striking improvements over the last couple of decades, starting with the introduction of conflict-driven clause learning (CDCL) SAT solvers [36, 39], and this has lead to a wide range of applications including large-scale problems in both academia and industry [8]. The conflict-driven paradigm has also been
Certified CNF Translations for Pseudo-Boolean Solving

Figure 1 Proof logging workflow for pseudo-Boolean solving (our contribution in boldface).

successfully exported to other areas such as maximum satisfiability (MaxSAT), pseudo-Boolean (PB) solving, constraint programming (CP), and mixed integer linear programming (MIP). As modern combinatorial solvers are used to attack ever more challenging problems, and employ ever more sophisticated optimizations and heuristics to do so, the question arises whether we can trust the results they produce. Sadly, it is well documented that state-of-the-art CP and MIP solvers can return incorrect solutions [1, 14, 24]. For SAT solvers, however, analogous problems [9] have been successfully addressed by the introduction of proof logging, requiring that solvers should be certifying [37] in the sense that they output machine-verifiable proofs of their claims that can be verified by a stand-alone proof checker.

A number of different proof logging formats have been developed for SAT, including RUP [28], TraceCheck [7], DRAT [29, 30, 50], GRIT [16], and LRAT [15], and since 2013 the SAT competitions [45] require solvers to be certifying, with DRAT established as the standard format. It would be highly desirable to have such proof logging also for stronger combinatorial solving paradigms, but while methods such as DRAT are extremely powerful in theory, the fact that they are limited to a clausal format makes it hard to capture more advanced forms of reasoning in a succinct way, and it is not even clear how to deal with input that is not in conjunctive normal form (CNF). One way to address this problem could be to allow extensions to the DRAT format [2], but another approach pursued in recent years is to develop stronger proof logging methods based on binary decision diagrams [4], algebraic reasoning [33, 44], pseudo-Boolean reasoning [21, 25, 26], or integer linear programming [12, 19].

Our Contribution In this work, we consider the use of CDCL for pseudo-Boolean solving, where the pseudo-Boolean input (i.e., a 0-1 integer linear program) is translated to CNF and passed to a SAT solver, as pioneered in MiniSat+ [18]. The two solvers Open-WBO [41] and NaPS [40] using this approach were among the top performers in the latest pseudo-Boolean evaluation [43]. While DRAT proof logging can be used to certify unsatisfiability of the translated formula, it cannot prove the correctness of the translation, not only since there is no known method of carrying out PB reasoning efficiently in DRAT (except for constraints with small coefficients [10]), but also, and more fundamentally, because the input is not in CNF.

We demonstrate how to instead use the cutting planes method [13], enhanced with a rule allowing to introduce extension variables [27], to certify the correctness of translations of pseudo-Boolean constraints into CNF. Since this method is a strict extension of DRAT, we can combine the proof of the translation with the SAT solver DRAT proof log (with appropriate syntactic modifications) to achieve end-to-end verification of the pseudo-Boolean solving process using the proof checker VeriPB [48], as shown in Figure 1.

One challenge when certifying PB-to-CNF translations is that there are many different ways of encoding pseudo-Boolean constraints into CNF (as catalogued in, e.g., [42]), and it is time-consuming (and error-prone) to code up proof logging for every single encoding. However, many of the encodings can be understood as first designing a circuit to evaluate whether the PB constraint is satisfied, and then writing down a CNF encoding of the
computation of this circuit. An important part of our contribution is that we develop a general framework to provide proof logging for a wide class of such circuits in a uniform way. The pseudo-Boolean format used for proof logging makes it easy to derive 0-1 linear inequalities describing the computations in the circuit, and once this has been done the desired clauses in the CNF translation can simply be obtained by so-called reverse unit propagation (RUP) [28, 47]. We have applied this method to the sequential counter [46], totalizer [3], generalized totalizer [32] and adder network [18, 49] encodings, and report results from an empirical evaluation.

**Outline of This Paper** After discussing preliminaries in Section 2, we illustrate our method for the sequential counter encoding in Section 3. Section 4 presents the general framework, and we briefly discuss how to apply it to adder networks in Section 5. (Due to space constraints, details for the totalizer and generalized totalizer encodings are omitted.) We report experimental data for proof logging and verification in Section 6 and conclude with a discussion of some possible directions for future research in Section 7.

2 Preliminaries

Let us start with a review of some standard material that can also be found in, e.g., [27] or in more detail in [11]. A literal $\ell$ over a Boolean variable $x$ is $x$ itself or its negation $\overline{x} = 1 - x$, where variables can be assigned values 0 (false) or 1 (true). For notational convenience, we define $\overline{x} \equiv x$ (where we use $\equiv$ to denote syntactic equality).

We sometimes write $\overline{x} = \{x_1, \ldots, x_m\}$ to denote a set of variables. A pseudo-Boolean (PB) constraint is a 0-1 linear inequality

$$C \equiv \sum_i a_i \ell_i \geq A,$$  

(1)

which without loss of generality we always assume to be in normalized form [5]; i.e., all literals $\ell_i$ are over distinct variables and the coefficients $a_i$ and the degree (of falsity) $A$ are non-negative integers. The normalized form of the negation of $C$ in (1) is

$$\neg C \equiv \sum_i a_i \overline{\ell_i} \geq \sum_i a_i - A + 1.$$  

(2)

An equality constraint $C \equiv \sum_i a_i \ell_i = A$ is just syntactic sugar for the pair of inequalities $C^{\text{eq}} \equiv \sum_i a_i \ell_i \geq A$ and $C^{\text{eq}} \equiv \sum_i -a_i \overline{\ell_i} \geq -A$ (rewritten in normalized form). Summing two equality constraints $C + D$ means taking the two sums $C^{\text{eq}} + D^{\text{eq}}$ and $C^{\text{eq}} + D^{\text{eq}}$.

We write $\sum_i a_i \ell_i \geq A$ for $A \in \{\geq, \leq, =\}$ for constraints that are either inequalities or equalities. A pseudo-Boolean formula is a conjunction $F \equiv \bigwedge_j C_j$ of PB constraints.

Note that a clause $\ell_1 \lor \cdots \lor \ell_k$ is equivalent to the constraint $\ell_1 + \cdots + \ell_k \geq 1$, so CNF formulas are just special cases of PB formulas. A cardinality constraint is a PB constraint with all coefficients equal to 1.

A (partial) assignment $\rho$ is a (partial) function from variables to $\{0, 1\}$. Applying $\rho$ to a constraint $C$ as in (1), denoted $C|_{\rho}$, yields the constraint obtained by substituting values for all assigned variables, shifting constants to the right-hand side, and adjusting the degree appropriately, and for a formula $F$ we define $F|_{\rho} = \bigwedge_j C_j|_{\rho}$. The constraint $C$ is satisfied by $\rho$ if $\sum_{\rho(\ell_i)} a_i \geq A$ (or, equivalently, if the restricted constraint has a non-positive degree and is thus trivial). An assignment $\rho$ satisfies $F \equiv \bigwedge_j C_j$ if it satisfies all $C_j$, in which case $F$ is satisfiable. A formula without satisfying assignments is unsatisfiable. Two formulas are equisatisfiable if they are both satisfiable or both unsatisfiable.
Cutting planes as defined in [13] 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 $\sum_i [a_i/d] \cdot \ell_i \geq \lfloor A/d \rfloor$, 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 clear that any collection of constraints $F'$ derived (iteratively) from $F$ by cutting planes are implied in this sense.

A constraint $C$ is said to unit propagate the literal $\ell$ under $\rho$ if $C|_{\rho}$ cannot be satisfied unless $\ell$ is satisfied. During unit propagation on $F$ under $\rho$, we extend $\rho$ iteratively by assignments to any propagated literals until an assignment $\rho'$ 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 called a conflict, since $\rho'$ violates the constraint $C$ in this case. We say that $F$ implies $C$ by reverse unit propagation (RUP), and that $C$ is a RUP constraint with respect to $F$, if $F$ and the negation of $C$ unit propagate to conflict under the empty assignment. It is not hard to see that $F \models C$ holds if $C$ is a RUP constraint.

In addition to deriving constraints $C$ that are implied by $F$, we will also need a rule for adding so-called redundant constraints $D$ having the property that $F$ and $F \land D$ are equisatisfiable. For this purpose we will use the reification rules described below, which are shown in [27] to be special cases of the redundancy rule in that paper. Provided that $z$ is a fresh variable that is not in the formula and has not appeared previously in the derivation, we can introduce the reified constraints

$$z \Rightarrow \sum_i a_i \ell_i \geq A \equiv A \varpi + \sum_i a_i \ell_i \geq A$$  \hspace{1cm} (3a)

and

$$z \Leftarrow \sum_i a_i \ell_i \geq A \equiv (\sum_i a_i - A + 1) \cdot z + \sum_i a_i \ell_i \geq \sum_i a_i - A + 1.$$  \hspace{1cm} (3b)

A moment of thought reveals that the constraint (3a) says that if $z$ is true, then $\sum_i a_i \ell_i \geq A$ has to hold, and this explains the notation $z \Rightarrow \sum_i a_i \ell_i \geq A$ introduced for this constraint. In an analogous fashion, the constraint (3b) says that if $\sum_i a_i \ell_i \geq A$ holds, then $z$ has to be true. We will write $z \Leftarrow \sum_i a_i \ell_i \geq A$ for the conjunction of (3a) and (3b). It is easy to see that adding such reification constraints to a formula $F$ preserves equisatisfiability, since any satisfying assignment to $F$ can be extended by setting $z$ as required to satisfy the implications.

3 Certified Translation for the Sequential Counter Encoding

To encode a cardinality constraint of the form $\sum_{i=1}^n \ell_i \geq k$ we can use the sequential counter encoding [46]. This encoding is designed after a circuit accumulating the sum of input bits using the intermediate fresh variables $s_{i,j}$ for $i \in [n], j \in [i]$, where $s_{i,j}$ is true if and only if the first $i$ literals sum up to $j$. The variable $s_{i,j}$ is computed as in Figure 2a, i.e.,

$$s_{i,j} \Leftarrow ((\ell_i \land s_{i-1,j-1}) \lor s_{i-1,j})$$  \hspace{1cm} (4)
that is either the first \( i - 1 \) variables add up to \( j - 1 \) and the \( i \)-th literal is true, or the
first \( i - 1 \) variables already add up to \( j \). The resulting circuit is shown in Figure 2b and
can be divided into multiple blocks, where the \( i \)-th block accumulates the \( i \)-th input
literal and the variables \( s_{i-1,j} \) for \( j \in [i-1] \). We will use this block structure later as
an abstract way to represent the encoding. The clausal encoding is given by translating
the circuit into clausal form, i.e., via the clauses

\[
\ell_i + s_{i-1,j-1} + s_{i,j} \geq 1 \tag{5a}
\]

\[
s_{i-1,j} + s_{i,j} \geq 1 \tag{5b}
\]

\[
\ell_i + s_{i-1,j} + \overline{s_{i,j}} \geq 1 \tag{5c}
\]

\[
s_{i-1,j-1} + \overline{s_{i,j}} \geq 1 \tag{5d}
\]

where \( i \in [n] \) and \( j \in [i] \). To cover corner cases we always replace \( s_{i,j} \) for \( j > i \) with 0
and \( s_{i,j} \) for \( j \leq 0 \) with 1 and simplify the constraints accordingly. For example, for
\( i = j = 1 \) we only get the clauses \( \ell_1 + s_{1,1} \geq 1 \) and \( \ell_1 + \overline{s_{1,1}} \geq 1 \), since \( s_{0,0} \) is replaced by
1 and hence the variable disappears from (5a) while (5d) is satisfied, and \( s_{0,1} \) is replaced
by 0 and thus disappears from (5c) and satisfies (5b). To enforce a greater-or-equal-\( k \)
constraint it is only necessary to add the clause \( s_{n,k} \geq 1 \). Analogously, a less-or-equal-\( k \)
constraint is enforced using the clause \( s_{n,k+1} \geq 1 \). A common optimization, known as
\( k \)-simplification, is to not add the clauses for variable \( s_{i,j} \) if \( j > k + 1 \), as these variables
have no influence on the satisfiability of the clausal encoding.

Before discussing the proof logging, let us study the encoding in more detail, ignoring
\( k \)-simplification for now. Remember that the variable \( s_{i,j} \) should be true if and only if
the first \( i \) literals sum up to \( j \) and hence can be understood as a unary representation,
where we want that \( \sum_{j=1}^{i} \ell_j = \sum_{j=1}^{i} s_{i,j} \) for \( i \in [n] \). However, the circuit is only using
the variables from the previous block \( s_{i-1,j} \) and the literal \( \ell_i \) as input to compute the
\( s_{i,j} \) variables and hence it will instead be more convenient to consider the equality

\[
\ell_i + \sum_{j=1}^{i-1} s_{i-1,j} = \sum_{j=1}^{i} s_{i,j} \quad i \in [n] \tag{6}
\]

We can use this insight to get a more abstract representation of the circuit in Figure 2b,
by thinking of blocks as nodes with two input edges labelled \( \ell_i \) and \( \sum_{j=1}^{i-1} s_{i-1,j} \) and an
output edge labelled \( \sum_{j=1}^{i} s_{i,j} \) as shown in Figure 3a. Additionally, for each inner node
the sum of all input labels should be equal to the sum of all output labels as enforced by (6), which we will call a preserving equality. This graph representation will be helpful to generalize the presented proof logging approach for other encodings.

Note that the sum of input variables coming from the source equals the sum of output variables on the edges going to the sink because each node preserves equality between incoming and outgoing values. That is we have \( \sum_{j=1}^n \ell_i = \sum_{j=1}^n s_{n,j} \), which can also be obtained mathematically by summing all equalities of the form (6). Based on this equality, it is clear that a bound on the input variables \( k \triangleright \sum_{j=1}^n \ell_i \) also implies a bound on the output variables, which can be seen by summing \( k \triangleright \sum_{j=1}^n s_{n,j} \) to get

\[
    k \triangleright \sum_{j=1}^n s_{n,j} . \tag{7}
\]

Another important observation is that the variables \( s_{i,j} \) should not just take any value satisfying (6), but they should also be ordered, that is if \( s_{i,j+1} \) is true, the sum should be at least \( j + 1 \) and hence also at least \( j \) and \( s_{i,j} \) should be true as well (and also \( s_{i,j-1} = 1, s_{i,j-2} = 1 \) etc.). This can be enforced with ordering constraints

\[
    s_{i,j} \geq s_{i,j+1} \quad i \in [n], j \in [i-1] . \tag{8}
\]

With this improved understanding of the encoding, we can now tackle the task of proof logging, which becomes surprisingly simple. The constraints (6), (7), (8) are all pseudo-Boolean constraints and if we are able to derive them, then the clauses of the sequential counter encoding ((5) and \( s_{n,k+1} \geq 1 \) and/or \( s_{n,k} \geq 1 \)) can all be derived via reverse unit propagation: The propagations due to (8) will cause enough variables to propagate, such that (6) is falsified. The derivation of (7) from (6) was already discussed when introducing (7), where we summed all constraints (6) and the constraint to be encoded. This summation can be expressed directly in cutting planes. For deriving the other constraints, remember that for proof logging we want to demonstrate that adding constraints does not change satisfiability. However, it is easy to see that the preserving equality (6) and ordering constraints (8) can always be satisfied by choosing a suitable value for the \( s_{i,j} \) variables. If the constraints are added in ascending order of \( i \), then the \( s_{i,j} \) are fresh and can indeed be chosen freely. In the proof format this reasoning is
Example 1. Let us consider how to derive the preserving equality

\[ \ell_3 + s_{2,1} + s_{2,2} = s_{1,1} + s_{3,2} + s_{3,3} \]  
(9)

for Block 3 in Figure 3a. To satisfy (9) we want that \( s_{3,1} \) is true if \( \ell_3 + s_{2,1} + s_{2,2} \) is greater equal 1, \( s_{3,2} \) is true if it is greater equal 2 and \( s_{3,3} \) is true if it is greater equal 3. We can enforce these conditions by introducing the fresh variables \( s_{3,1}, s_{3,2}, s_{3,3} \) via reification, i.e., \( s_{3,1} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 1 \), \( s_{3,2} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 2 \) and \( s_{3,3} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 3 \), which results in the pseudo-Boolean constraints

\[ \begin{align*}
\bar{s}_{3,1} + \ell_3 + s_{2,1} + s_{2,2} &\geq 1 \\
2s_{3,2} + \ell_3 + s_{2,1} + s_{2,2} &\geq 2 \\
3s_{3,3} + \ell_3 + s_{2,1} + s_{2,2} &\geq 3 \\
3s_{3,1} + \bar{\ell}_3 + \bar{s}_{2,1} + \bar{s}_{2,2} &\geq 3 \\
2s_{3,2} + \bar{\ell}_3 + \bar{s}_{2,1} + \bar{s}_{2,2} &\geq 2 \\
3s_{3,3} + \bar{\ell}_3 + \bar{s}_{2,1} + \bar{s}_{2,2} &\geq 1 \\
\end{align*} \]  
(10a)-(10f)

By design, (10) implies (9) and hence (9) can be derived via cutting planes. To do so in practice, we accumulate the constraints (10a)-(10c) while maintaining the invariant

\[ \sum_{j=1}^{i} s_{3,j} + \ell_3 + s_{2,1} + s_{2,2} \geq i \]

where \( i = 1, 2, 3 \) is the number of accumulated constraints. When starting with (10a) the invariant holds. Next we add (10b) and divide by 2 to obtain \( s_{3,1} + s_{3,2} + \ell_3 + s_{2,1} + s_{2,2} \geq 2 \) and continue by multiplying with 2, adding (10c) and dividing by 3, which results in \( s_{3,1} + s_{3,2} + s_{3,3} + s_{2,1} + s_{2,2} \geq 3 \), which is equivalent to \( \ell_3 + s_{2,1} + s_{2,2} \geq s_{3,1} + s_{3,2} + s_{3,3} \), as desired. Analogously, we can accumulate (10d)-(10f) in reverse order to obtain \( \ell_3 + s_{2,1} + s_{2,2} \leq s_{3,1} + s_{3,2} + s_{3,3} \). The ordering constraints \( s_{3,1} \geq s_{3,2} \) can be obtained by adding (10d) and (10b), which yields \( 3s_{3,1} + 2s_{3,2} \geq 1 \) and can be divided by 3 to obtain \( s_{3,1} + s_{3,2} \geq 1 \), which is equivalent to \( s_{3,1} \geq s_{3,2} \), as desired. Analogously, we can obtain \( s_{3,2} \geq s_{3,3} \) by using (10e) and (10c).

To perform \( k \)-simplification, we could simply omit deriving the unneeded clauses, however this potentially introduces a large overhead for proof logging if \( k \) is small, as we would always introduce \( O(n^2) \) intermediate variables instead of the \( O(kn) \) variables that are needed. To avoid this overhead, as demonstrated in Figure 3b, we want that the edge going to the next block is labelled with \( \sum_{j=1}^{i} s_{i,j} \) instead of \( \sum_{j=1}^{k+1} s_{i,j} \). However, this means we need to introduce an additional edge going directly to the sink with the label \( s_{i,k+2} \) to preserve the equality of in- and output, i.e.,

\[ \ell_i + \sum_{j=1}^{k+1} s_{i-1,j} = \sum_{j=1}^{k+2} s_{i,j} \quad i \in [n] \]  
(11)

Note that without the additional variable \( s_{i,k+2} \) we could not guarantee equality, as we would have \( k+2 \) literals on the left hand side and only \( k+1 \) fresh variable on the right hand side.

Example 2. To demonstrate \( k \)-simplification, consider Block 3 in Figure 3b, which has input edges with labels \( s_{2,1} + s_{2,2} \) and \( \ell_3 \) and let us perform 1-simplification. The output of Block 3 to Block 4 should only contain the 2 variables \( s_{3,1} + s_{3,2} \). To preserve equality of in- and output, we add an edge from Block 3 to the sink labelled \( s_{3,3} \).
As before, we can obtain the constraint that in- and output of the graph are equal by summing the preserving constraint (11) of each node, which yields
\[ \sum_{i=1}^{n} (\ell_i + \sum_{j=1}^{k+1} s_{i,j}) = \sum_{i=1}^{n} (\sum_{j=1}^{k+1} s_{i,j}) \] and can be simplified to
\[ \sum_{i=1}^{n} s_{i,k+2} + \sum_{j=1}^{k+1} s_{n,j}. \]

### 4 General Framework for Certifying CNF Translations

A major challenge of providing proof logging for translations of pseudo-Boolean constraints to CNF is that there are so many different encodings of pseudo-Boolean constraints. To support a wide range of encodings, we can generalize the idea of the graph representation used in the previous section to obtain a general framework. The main ingredient of the framework is a graph representing the connection between the variables of the encoded constraint and auxiliary variables used in the encoding. This graph has the property that we can derive a preserving equality of in- and output for each node and that the CNF encoding follows from these equalities. To derive the preserving equality, we provide proof logging for general purpose operations for different ways to represent natural numbers. Let us start with a formal definition of the graph representation.

**Definition 3 (Arithmetic Graph).** An arithmetic graph with input \( \sum_i a_i x_i \) and output \( \sum_i c_i o_i \) is a directed graph \( G = (V, E) \) with a source node \( s \), a sink node \( t \), and edge labels of the form \( \sum b_{i,v} y^e_i \) for each edge \( e \in E \). For convenience, we allow to have multiple edges between two nodes. Additionally, we require that:
- the source \( s \) has only outgoing edges and the input is split among edges of \( s \), i.e., \( \sum_i a_i x_i \equiv \sum_{(s,v) = e \in E} \sum b_{i,v} y^e_i \);
- the sink \( t \) has only incoming edges and the output is split among edges of \( t \), i.e., \( \sum_i c_i o_i \equiv \sum_{(v,t) = e \in E} \sum b_{i,v} y^e_i \); and
- for every inner node \( v \) the input is equal to the output, which can be derived via proof logging, i.e., we can derive the preserving equality
\[ \sum_{(u,v) = e \in E} \sum b_{i,v} y^e_i = \sum_{(v,u) = e \in E} \sum b_{i,v} y^e_i. \] (12)

The general strategy for providing proof logging will be to formulate the used encoding in terms of an arithmetic graph, where the preserving equality (12) will depend on the representation of natural numbers used in the encoding and will be derived using one of the operations described later in this section. For each encoding, we will make sure that the clauses in the encoding directly correspond to a node in the graph and will follow by reverse unit propagation from the preserving equality (12). However, each encoding has also clauses to restrict the output variables \( o_i \), which can only be derived after translating the bound known on the input variables to a bound on the output variables.

**Proposition 4.** Given an arithmetic graph with input \( \sum_i a_i x_i \) and output \( \sum_i c_i o_i \) and a pseudo-Boolean constraint \( \sum_i a_i x_i \triangleq k \), where \( \triangleq \in \{ \geq, \leq, = \} \), we can derive \( \sum_i c_i o_i \triangleq k \) using cutting planes.

**Proof.** As we have an arithmetic graph, we know that we can derive (12) for every inner node in the graph. By adding all these constraints together, we obtain the constraint \( \sum_i a_i x_i = \sum c_i o_i \), which can be combined with \( \sum c_i o_i \triangleq k \) to obtain \( \sum_i c_i o_i \triangleq k. \)
Once the bound on the input variables is translated to a bound on the output variables, all clauses of the CNF encoding will follow by reverse unit propagation. This results in the general algorithm for proof logging encodings shown in Algorithm 1. Note that the nodes of the graph need to be traversed in a topological order when deriving the preserving equality. Otherwise we can not use that the output variables of a node are fresh, which will be crucial for the presented derivations.

Let us now discuss three common ways to represent natural numbers, as well as some general purpose operations on these representations that are used to derive the preserving equality for inner nodes. The easiest way to encode a natural number \( j \) with domain \( A = \{0, 1, \ldots, m\} \) using Boolean variables is to use a unary number, where the number of variables \( z_i \) set to true is equal to \( j \), i.e., \( j = \sum_{i \in [m]} z_i \). For better propagation behaviour, it is usually required that the \( z_i \) variables are ordered via constraints \( z_i \geq z_{i+1} \), which enforces that \( z_i \) is true if and only if \( j \geq i \). This representation is used in the sequential counter \cite{46} and totalizer encoding \cite{3} and is known as order encoding.

\begin{algorithm}
\begin{algorithmic}
\caption{Algorithm 1 General algorithm for proof logging arithmetic encodings.}
\Procedure{proof_log_encoding}{\( C, f, G, F \)}
\State \textbf{▷ input: } \( C \) is of the form \( \sum_{i=1}^{n} a_i \ell_i \models k \), with \( k, n \in \mathbb{N} \) and \( \models \in \{ \geq, \leq, =\} \).
\State \textbf{▷ input: } an arithmetic graph \( G = (V, E) \) with input \( \sum_{i} a_i x_i \) and output \( \sum_{i} c_i o_i \)
\State \textbf{▷ input: } a function \( f \) that takes a node and derives its preserving equality
\State \textbf{▷ input: } the CNF encoding \( F \) to be derived
\State sum the constraints \( f(v) \) for \( v \in V \) in topological order to obtain \( \sum_{i} a_i x_i = \sum_{i} c_i o_i \)
\State combine \( \sum_{i} a_i x_i = \sum_{i} c_i o_i \) and \( C \) to obtain \( \sum_{i} c_i o_i \models k \)
\State derive each clause in the CNF encoding \( F \) via RUP
\EndProcedure
\end{algorithmic}
\end{algorithm}

\textbf{Proposition 5 (Unary Sum).} For any literals \( \ell_1, \ldots, \ell_n \) we can derive the constraints

\[ \sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i \]
\[ z_i \geq z_{i+1}, \quad i \in [n-1] \]

using \( O(n) \) steps, where \( z_1, \ldots, z_n \) are fresh variables.

Conceptually, adding these constraints does not change satisfiability, because they can always be satisfied using the fresh variables. We already discussed deriving these constraints in the context of the sequential counter encoding. The general idea is to introduce the fresh variables via reification \( z_i \leftrightarrow \sum_{i=1}^{n} \ell_i \geq i \), after which we can obtain the greater-than part of the equality by maintaining the invariant \( \sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j-1} z_i \geq j \) and analogously for the less-than part. A detailed description of the algorithm for deriving a unary sum is provided in Appendix A.1.

If we want to encode a natural number \( j \), for which we know that it can only take values in a small domain \( A \), then introducing variables for all values in the range introduces a lot of redundant variables. For example if \( j \in \{0, 50, 75\} \), then the first 50 variables in a full unary representation are either all true or all false, but will never take different values. For a more concise encoding we can use a sparse representation, i.e., we represent \( j \in \{0, 50, 75\} \) as \( 50 \cdot z_{50} + 25 \cdot z_{75} \) and enforce that \( z_{50} \geq z_{75} \). In general, we use

\[ \text{sparse}(z, A) = \sum_{i \in A \setminus \{0\}} (i - \text{pred}(i, A)) z_i, \]

where \( \text{pred}(i, A) = \max(\{j \in A \mid j < i\}) \). Additionally, we enforce that the \( z_i \) variables are ordered, i.e., \( z_i \geq z_{\text{succ}(i, A)} \), where \( \text{succ}(i, A) = \min(\{j \in A \cup \{\infty\} \mid j > i\}) \). This
representation is used in the sequential weight counter [31] and generalized totalizer encoding [32].

**Proposition 6 (Sparse Unary Sum).** Given \( A, B \subseteq \mathbb{N} \), \( E = \{ i + j \mid i \in A, j \in B \} \), ordering constraints on variables \( \vec{y} \) and \( \vec{y}' \), as well as fresh variables \( \vec{z} \), we can derive

\[
sparse(\vec{y}, A) + sparse(\vec{y}', B) = sparse(\vec{z}, E) , \quad \text{and} \quad z_i \geq z_{\text{succ}(i, E)} \quad i \in E \setminus \{ \max (E) \} ,
\]

using \( O(|A| \cdot |B|) \) steps.

As in the case of the unary sum, these constraints can be added without changing satisfiability, because we can always set the fresh \( z_i \) variables such that the constraints are satisfied. The general idea is to introduce the fresh variables via reification \( z_i \Leftrightarrow \sum_{i=1}^{n} \ell_i \geq i \). Then we simulate a brute-force search on the possible combinations of values for \( A \) and \( B \), showing that the equality holds in all cases. A detailed description can be found in Appendix A.2.

Finally, if we want to represent a natural number that is large and has a large domain with maximal value \( m \), then we can encode it using a binary representation, i.e., \( j = \sum_{i=0}^{\lfloor \log_2(m) \rfloor} 2^i z_i \). To build a binary number (as is discussed in Section 5) it sufficient to compose multiple full adders, which compute the sum of up to three input bits, using a binary adder circuit [18].

**Proposition 7.** For literals \( \ell_1, \ell_2, \ell_3 \) and fresh variables \( z_1, z_0 \) we can derive the constraints

\[
\ell_1 + \ell_2 + \ell_3 = 2z_1 + z_0
\]

using \( O(1) \) steps.

Again, it should be clear that this equality can be added without changing satisfiability because it can be satisfied using the fresh variables. To derive it, we reify

\[
c \Leftrightarrow x + y + z \geq 2 \quad (18a)
\]
\[
s \Leftrightarrow x + y + z + 2s \geq 3 \quad . \quad (18b)
\]

The equality can be derived by multiplying (18a) by 2, adding (18b) and dividing the result by 3 as discussed in detail in [27].

In Section 5 and Appendix B, it is demonstrated how to apply this framework for the binary adder and the (generalized) totalizer encoding, respectively.

5 Binary Adder Encoding

The **binary adder encoding** [18] is used to encode general pseudo-Boolean constraints of the form \( \sum_i a_i \ell_i \gg k \). The idea is to use an adder network to obtain the value of \( \sum_i a_i \ell_i \) as a binary number \( \sum_{i=0}^{\text{bits}} o_i \), where \( o_i \) are the output literals and \( \text{bits} = \lceil \log_2(\sum_i a_i) \rceil \) is the required bit width. To enforce the constraint, the output bits \( o_i \) are constrained by clauses that perform a bitwise comparison with \( k \) in binary representation.

To recapitulate the algorithm for the construction of the adder network in [18], we need some more notation. A \( 2^m \)-bit is a literal that represents the numerical value \( 2^m \). A \( 2^m \)-bucket is a queue of bits where each bit has the value \( 2^m \) and that supports
operations to insert and extract bits. We use $[m]_2$ to denote the binary representation of a natural number $m$.

The construction of the network starts by initializing each $2^m$-bucket with all literals $\ell_i$ such that the $2^m$-bit of $[a_i]_2$ is 1. Then we repeat the following steps until there is at most one element left in each bucket. Consider the $2^m$-bucket with the smallest value that has at least 2 elements in it. If there are only 2 elements in the $2^m$-bucket, take $x$ and $y$ from the bucket and set $z = 0$. Otherwise, let $x, y$ and $z$ be 3 elements from the $2^m$-bucket and remove them from the $2^m$-bucket. The bits $x, y$ and $z$ are used as input for a new full adder with fresh variables $c$ and $s$ as output, where $c$ is a $2^{m+1}$-bit and $s$ is a $2^m$-bit. The bits $c$ and $s$ are then inserted in their respective buckets, possibly creating a new bucket. An algorithm for constructing the network is given in Appendix A.3.

The arithmetic graph is constructed directly from the adder network such that each full adder is represented by a node. Each inner node constructed from the $2^m$-bucket, i.e., which has $2^m$-bits as input, has input edges with labels $2^m x, 2^m y$ and $2^m z$ and output edges with labels $2^m s$ and $2^{m+1} c$. An example of the resulting graph is shown in Figure 4. The preserving equality can be derived using Proposition 7 and multiplying the resulting equality $x + y + z = 2c + s$ by $2^m$ to obtain $2^m x + 2^m y + 2^m z = 2^{m+1} c + 2^m s$.

After construction of the adder network, each $2^m$-bucket has at most one $2^m$-bit left and we connect the corresponding edges to the sink, resulting in an output of the form $\sum_{i=0}^{\ell} 2^i c_i$. If the $2^m$-bucket is empty, $o_m$ is set to 0.

Each full adder of the network is encoded to CNF via the clauses

$$x + y + z + s \geq 1 \quad x + y + z + \bar{s} \geq 1$$

$$y + z + c \geq 1 \quad x + y + z + \bar{c} \geq 1$$

$$x + z + c \geq 1 \quad x + y + z + \bar{s} \geq 1$$

$$x + y + z + \bar{c} \geq 1 \quad x + y + z + \bar{s} \geq 1$$

Note that all the clauses in (19) are RUP with respect to the preserving equality $x + y + z = 2c + s$.

To encode $k$ with the output of the circuit, the encoding performs the comparison $\bar{x} \geq \bar{y}$ for bit vectors $\bar{x}$ and $\bar{y}$, where either $\bar{x} = o_{bits} \ldots o_1 o_0$ and $\bar{y} = [k]_2$ or vice versa, depending on whether we want to encode $\sum_{i=1}^{n} a_i \ell_i \geq k$ or $\sum_{i=1}^{n} a_i \ell_i \leq k$, respectively.

If we want to encode $\sum_{i=1}^{n} a_i \ell_i = k$, then the comparison for both directions is performed. If the size of these vectors is different, the shorter vector is padded with 0. Then, for $i = 0, \ldots, bits$, the constraint

$$\bar{x}_i + y_i + \sum_{j=1}^{bits} x_j \bar{y}_j + \bar{x}_j y_j \geq 1$$

(20)
is added to the CNF encoding. Note that either $\vec{x}$ or $\vec{y}$ is constant and hence the constraint is always a clause. This clause guarantees that the $2^i$-bit on the variable side is equal to the $2^i$-bit in $[k]_2$ or there was already a $2^j$-bit for $j > i$ that is different to the $2^i$-bit in $[k]_2$.

The clauses (20) are RUP with respect to $\sum_{i=0}^{\text{bits}} 2^i o_i \sim k$, which we obtain from the arithmetic graph using Proposition 4. The clauses are RUP because the RUP step will set all $2^j$-bits, where $j > i$, to the same value as in $[k]_2$ and the $2^i$-bit to the opposite value of the $2^i$-bit in $[m]_2$, which falsifies $\sum_{i=0}^{\text{bits}} 2^i o_i \sim k$.

6 Experimental Results

To show the generality of our approach for proof logging arithmetic encodings, we implemented the sequential counter encoding [46], binary adder encoding [18], totalizer [3] and generalized totalizer encodings [32], in a certified encoding framework called VeritasPBLib. This framework inputs a pseudo-Boolean formula in OPB format and returns a CNF translation with the corresponding proof logging certificate. We used the verifier VeriPB [48] to verify the proof logging certificate returned by VeritasPBLib. The CNF formula is then solved by a modified version of the SAT solver kissat [34] that generates proof logging compatible with the VeriPB verifier. Finally, we conjoin the proof logging from the CNF translation with the proof logging from SAT solving and verify the end-to-end pipeline with VeriPB.

The experiments were conducted on Amazon EC2 r5.large instances (2 vCPU) with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz CPUs, 16 GB of memory, and gp2 volumes. We ran one process on each instance with a memory limit of 15 GB and a time limit of 7,200 seconds for verifying the proof with VeriPB, and a time limit of 1,800 seconds for CNF translation with VeritasPBLib and SAT solving with kissat. We gave additional time for verification, since verification is slower than solving the problem.

To evaluate VeritasPBLib, we collected 1,803 pseudo-Boolean formulas from the PB 2016 Evaluation. We can split these instances into four categories: (1) formulas with only clauses (279 instances), (2) formulas with clauses and cardinality constraints (772 instances), (3) formulas with clauses and general PB constraints (444 instances), and (4) formulas with clauses, cardinality and general PB constraints (308 instances). Since this work targets the verification of formulas with cardinality or general PB constraints, we excluded the 279 pure CNF formula instances, as those can already be certified with existing techniques. More details about the instances can be found in Appendix C.

The goal of our evaluation is to answer the following questions:

1. Can we use the end-to-end framework to verify the results of SAT-based approaches to solve pseudo-Boolean formulas and how efficient is verification?
2. How long does verification of the proof logging take when compared to translating the pseudo-Boolean formula to CNF?

End-to-End Solving and Verification Table 1 shows how VeritasPBLib can be used to generate a CNF formula that can be solved by kissat and verified by VeriPB. For instances with cardinality constraints (Card), we use the sequential and totalizer encoding to translate those constraints to CNF. For instances with general PB constraints

---

1 Available at https://gitlab.com/MIAOresearch/kissat_fork
2 Available at http://www.cril.univ-artois.fr/PB16/
Table 1 Number of translated, solved and verified instances for each encoding

<table>
<thead>
<tr>
<th>Category</th>
<th>#Inst</th>
<th>Encoding</th>
<th>#CNF</th>
<th>#Veri</th>
<th>#Solved</th>
<th>#Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SAT/UNSAT</td>
<td>SAT/UNSAT</td>
<td>SAT/UNSAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Card</td>
<td>772</td>
<td>Sequential Totalizer</td>
<td>772</td>
<td>772</td>
<td>139</td>
<td>480</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cardinality constraints</td>
<td></td>
<td></td>
<td>80</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generalized totalizer constraints</td>
<td></td>
<td></td>
<td>80</td>
<td>133</td>
</tr>
<tr>
<td>PB</td>
<td>444</td>
<td>Adder</td>
<td>444</td>
<td>444</td>
<td>179</td>
<td>167</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>178</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GTE</td>
<td>425</td>
<td>414</td>
<td>164</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>151</td>
</tr>
<tr>
<td>Card+PB</td>
<td>308</td>
<td>Seq+Adder</td>
<td>306</td>
<td>296</td>
<td>134</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128</td>
<td>151</td>
</tr>
</tbody>
</table>

(PB), we use the adder and generalized totalizer encoding (GTE) to translate general PB constraints to CNF. Finally, for instances with both cardinality and general PB constraints (Card+PB), we use the sequential encoding for cardinality constraints and the encoding for PB constraints, henceforth denoted by Seq+Adder. Even though other combinations of cardinality and PB encodings could be explored, the goal of this work is not to find the best performing encodings but to show that we can verify the final result with a variety of encodings.

The column #CNF shows for how many instances VERITASPBLIB successfully generated the CNF translation. For most of the formulas, we can translate the PB formula to CNF. The exceptions are 19 instances using the generalized totalizer (GTE) encoding and 2 instances using the Seq+Adder encoding. In those cases, the number of clauses generated is too large and exceeds the resource limits used in our evaluation.

The column #Veri under translation shows how many instances VeriPB can verify the proof logging certificate generated by VERITASPBLIB. Except for a few instances for the GTE and Seq+Adder where the proof is large, VeriPB can verify the CNF translation. Note that if verification of the translation is successful, then this guarantees that the CNF encoding does not remove any solutions of the PB formula.

The columns #Solved and #Verified under solving show how many instances can be solved by the SAT solver kissat and from those how many can be verified by VeriPB. If a satisfiable formula is verified, then it means that all clauses derived by kissat are due to correct derivations and the satisfying assignment returned by the SAT solver is a satisfying assignment of the original PB formula. If an unsatisfiable formula is verified, then it means that the reason of unsatisfiability is due to correct derivations.

We can verify 99% of the solved instances for unsatisfiable instances, which shows that the current approach can be used in practice to verify unsatisfiable results of SAT solvers when solving PB formulas. For satisfiable instances, we can verify 95% of the solved instances. However, for instances that VeriPB does not verify the result within the time limit, we can still certify that the satisfying assignment of the SAT solver satisfies the original PB formula. Even though VeriPB is already able to verify the majority of the proof logging, improvements to the verifier are orthogonal to our approach and can further increase the number of verified instances.

Translation and Verification Let us now focus on the CNF translation without solving. Our experiments show that the average overhead for proof logging ranges from 2× to 3× slower for all encodings with the exception of GTE which is around 5× slower. However, since translation is fast for the majority of instances (see Figure 6), the additional overhead of proof logging is not an issue when translating the PB formulas to CNF. A more detailed comparison of running times between CNF translation with and without
proof logging can be found in Appendix C.2.

The overhead for translation can be explained with the increased proof size compared to the size of the CNF encoding as shown in Figure 5. For most instances the proof size seems to be within a constant factor of the CNF file size. However, there is a series of benchmarks for which the sequential counter encoding requires super linear (but still polynomial) proofs. It turns out that these instances are all crafted instances encoding a vertex cover [20]. These instances contain a constraint enforcing a constant fraction of the literals in the formula to be true, which is the worst case scenario for the sequential counter. At first glance, this super linear relationship seems to contradict the expected linear relationship between the number of clauses in the CNF and the number of steps in the proof. However, this can be explained as each reification step for deriving the unary sum introduces a constraint of linear size, so even though the number of steps for deriving a unary sum is linear, the proof size will be quadratic. It would be desirable to find a derivation of the unary sum that only requires linear proof size.

Figure 6 shows the relationship between the time to generate the CNF translation using VeriPPLB and the time to verify the translation using VeriPB. The time to verify the translation compared to the translation itself is not negligible. Over all encodings, for 75% of benchmarks verification takes at-most 49 times longer than translation and for 98% of benchmarks take at-most 100 times longer. To some degree, such an overhead in verification time of the translation is expected, as the translation does not need to reason about its steps and the verification needs to perform some reasoning to justify the correctness of the proof steps. However, this also indicates that there is still room for improvement, both in terms of improving the performance of the verifier but potentially also by finding easier to verify derivation steps.

## 7 Concluding Remarks

In this work, we develop a general framework for certified translations of pseudo-Boolean constraints into CNF using cutting-planes-based proof logging. Since our method is a strict extension of DRAT, the proof for the translation can be combined with a SAT solver DRAT proof log to provide, for the first time, end-to-end verification for CDCL-based pseudo-Boolean solvers. Our use of the cutting planes method is not only
crucial to deal with the pseudo-Boolean format of the input, but the expressivity of the 0-1 linear constraints also allows us to certify the correctness of the translation to CNF in a concise and elegant way. While there is still room for performance improvements in proof logging and verification, our experimental evaluation shows that this approach is feasible in practice.

We want to point out that the tools we develop can also be used for the more general task of proving equivalence of reformulated problems. For the decision problem for a PB formula \( F \), we only need to show that the CNF translation \( \text{Tr}(F) \) can be derived from \( F \), since a proof of unsatisfiability of \( \text{Tr}(F) \) then shows that \( F \) is also unsatisfiable. However, our method can be adapted to show that if the PB formula \( F \) over variables \( X \) is translated to a CNF formula \( \text{Tr}(F) \) over variables \( X \cup Y \), then the two formulas are equivalent in the sense that (i) any satisfying assignment \( \alpha \) to \( F \) propagates an assignment \( \beta \) to \( Y \) such that \( \alpha \cup \beta \) satisfies \( \text{Tr}(F) \), and (ii) for any satisfying assignment \( \alpha \cup \beta \) to \( \text{Tr}(F) \) it holds that \( \alpha \) satisfies \( F \). We believe that such certified problem reformulation should be useful also in, e.g., constraint programming.

In our view, proof logging for pseudo-Boolean decision problems is only a first step. We believe that our method should also be sufficient to support proof logging for MaxSAT solvers. As a concrete example, using the techniques developed in this paper it should be possible to certify the clauses added during core extraction and objective function reformulation in core-guided MaxSAT solving [23, 38]. While supporting MaxSAT solvers using approaches such as implicit hitting set (IHS) [17] and abstract cores [6] seems a bit more challenging, we are still hopeful that our work could lead to a unified proof logging method for both MaxSAT solving and pseudo-Boolean optimization using cutting-planes-based reasoning as in [22, 35].

References


S. Gocht, R. Martins, J. Nordström and A. Oertel


NaPS (Nagoya pseudo-Boolean solver). https://www.trs.cm.is.nagoya-u.ac.jp/projects/NaPS/.

Open-WBO: An open source version of the MaxSAT solver WBO. http://sat.inesc-id.pt/open-wbo/.


### A Derivations for Building Blocks

Before going into detail on the derivations and presenting their respective algorithms, the notation for the proof logging is described. This is similar to the notation of the proof file used by VeriPB.

Lines are added to the proof file using the proof_log(·) command. In this format, every constraint in the proof gets a unique identifier (or just id for brevity). We can
Algorithm 2 Deriving a unary sum over fresh variables $z_i$.

1: procedure derive_unary_sum($C'$)  
2:  ▷ input: $C'$ is of the form $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i$ and describes the constraint to be derived  
3:  ▷ the $z_i$ variables need to be fresh, the left hand side is the sum to be encoded  
4:  for $j$ from 1 to $k$ do  
5:      $D_{j,eq}^D, D_{j,eq}^I \leftarrow \text{Reify}(z_j \Leftrightarrow \sum_{i=1}^{n} 1 \cdot \ell_i \geq j)$  
6:          ▷ Step 1: introduce variables as reification  
7:  end for  
8:  end for  
9: return $C_{eq}, C_{eq}$  

Algorithm 3 Reify $\sum_{i=1}^{n} a_i \ell_i \geq j$ using the fresh variable $z_j$.

1: procedure reify($z_j \Leftrightarrow \sum_{i=1}^{n} a_i \ell_i \geq j$)  
2:  $C_{eq} \leftarrow \sum_{i=1}^{n} a_i \ell_i + j z_j \geq j$  
3:  ▷ $z_j \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq j$ in normalized form  
4:  proof_log(red $C_{eq}$ ; $z_j \rightarrow 0$)  
5:  $C_{\text{eq}} \leftarrow \sum_{i=1}^{n} a_i \ell_i + (\sum_{i=1}^{n} a_i - j + 1) z_j \geq \sum_{i=1}^{n} a_i - j + 1$  
6:          ▷ $z_j \Leftrightarrow \sum_{i=1}^{n} a_i \ell_i \geq j$ in normalized form  
7:  proof_log(red $C_{\text{eq}}$ ; $z_j \rightarrow 1$)  
8: return $C_{eq}, C_{\text{eq}}$

expressing cutting planes derivations in reverse polish notation where constraints are referred to by their ids. For example, given previously derived constraints $C$ and $D$, the line `proof_log(pol C D * 3 * 4 d)` adds $C$ and $D$, multiplies the result by 3, and finally divides by 4 (rounding up). In the concrete format constraints in reverse polish notation are represented by an identifier, but we omit this detail for simplicity and operate on the constraints directly. The proof format also supports the saturation rule, which, given a normalized constraint $\sum a_i \ell_i \geq A$, allows to derive $\sum \min(a_i, A) \ell_i \geq A$. We use `proof_log(pol C s)` to denote saturation in the proof format.

A RUP constraint $C$ can be added using `proof_log(rup C)`. The syntax for adding a constraint as reification is ‘red $z \Rightarrow C ; z 1$’ and ‘red $z \Leftarrow C ; z 0$’, respectively (for more details please refer to [27]).

A.1 Deriving the Unary Sum

Deriving the constraints of a unary sum over fresh variables $z_i$, i.e.,

\[ \sum_{i=1}^{n} \ell_i \geq \sum_{i=1}^{n} z_i \quad , \quad (21a) \]

\[ \sum_{i=1}^{n} \ell_i \leq \sum_{i=1}^{n} z_i \quad , \quad \text{and} \]

\[ z_i \geq z_{i+1} \quad i \in [n-1] \quad , \quad (21b) \]

is described in Algorithm 2, which is split into four steps. Step 1 is to introduce the fresh variables $z_j$ as reifications of the constraints $\sum_{i=1}^{n} \ell_i \geq j$, which is shown in Algorithm 3 for the more general case using arbitrary positive coefficients.

Step 2: Deriving the Lower Bound. To derive (21a) in Algorithm 4 we maintain the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{n} \tau_i \geq j$, which holds by induction. For $j = 1$ the invariant...
Algorithm 4 Derive sum of reification variables.

1: procedure deriveSum(D_1, \ldots, D_n)
2:  \triangleright input: D_j is of the form \( \sum_{i=1}^{n} \ell_i + j \varpi_j \geq j \)
3:  C \leftarrow D_1
4:  for \( j \) from 2 to \( n \) do \triangleright Invariant: \( C \cdot \sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \varpi_i \geq j \)
5:      proof_log(pol \ C \ j - 1 \ * \ D_j + \ j \ d)
6:      C \leftarrow (C + D_j)/j
7:  return C

Algorithm 5 Deriving an ordering constraint \( z_A \geq z_B \) from the reification constraints.

1: procedure DeriveOrdering(C, D)
2:  \triangleright input: C is of form \( z_A \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq A \)
3:  \triangleright input: D is of form \( z_B \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq B \)
4:  divisor \leftarrow \sum_{i=1}^{n} a_i
5:  \triangleright derive \( z_A \geq z_B \) if \( A < B \)
6:  proof_log(pol \ C \ D + \ divisor \ d)

is equivalent to the reification constraint \( z_1 \Rightarrow \sum_{i=1}^{n} \ell_i \geq 1 \), which in normalized form
is \( \sum_{i=1}^{n} \ell_i + \varpi_1 \geq 1 \) and hence the base case is covered. For the inductive step going
from \( j \) to \( j + 1 \), we multiply the invariant by \( j \) and add the reification constraint
\( z_{j+1} \Rightarrow \sum_{i=1}^{n} \ell_i \geq j + 1 \), which is \( \sum_{i=1}^{n} \ell_i + (j + 1) \varpi_{j+1} \geq j + 1 \) in normalized form, to
get \( (j+1)\sum_{i=1}^{n} \ell_i + j \sum_{i=1}^{j} \varpi_i + (j + 1) \varpi_{j+1} \geq j^2 + j + 1 \). Note that \( j^2 + j + 1 = (j+1)^2 - j \)
and hence division by \( j + 1 \) and rounding up yields \( \sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \varpi_i + \varpi_{j+1} \geq j + 1 \),
i.e., the invariant for \( j + 1 \). For \( j = k + 1 \) the invariant is the normalized form of (21a).

Step 3: Deriving the Upper Bound. To derive (21b) we can use Algorithm 4
again but need to provide the constraints in reverse order to fit the required input format.

Step 4: Deriving the Ordering Constraints. The ordering constraint is derived
in Algorithm 5, using the reification constraints: We add the constraints used for
reification, that is \( z_{j+1} \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq j + 1 \) and \( z_j \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq j \). In normalized
form these two constraints are \( (j+1)\varpi_{j+1} + \sum_{i=1}^{n} a_i \ell_i \geq j + 1 \) and \( (m - j + 1)z_j + \sum_{i=1}^{n} a_i \ell_i \geq m - j + 1 \), where \( m = \sum_{i=1}^{n} a_i \). Adding both constraints together yields
\( (m - j + 1)z_j + (j + 1)\varpi_{j+1} \geq 2 \) and we get the desired ordering constraint after division
by a large enough number, e.g., \( m \).

A.2 Deriving the Sparse Unary Sum

In this section we prove Proposition 6 by providing Algorithm 6, which derives the
sparse unary sum of two numbers in sparse unary representation. As for the unary
sum, we start in Step 6.1 by introducing the required fresh variables via reification.
However, we only need to introduce the variables that will be used, i.e., those with index
in \( E \). If \( k \)-simplification is used, then also variables with index bigger than \( k \) need to be
introduced, as without them equality cannot be derived. (The introduction of variables
with index bigger than \( k \) can be avoided by having an arithmetic graph each for the upper
and lower bound and relaxing the preserving equality to inequalities.) After introducing
the variables we can derive the ordering constraints as before.

In Step 6.2 we introduce a variable \( z_{eq} \) which is true if and only if the equality to be
derived is true. Note that we need to represent an equality as two inequalities and hence
need to introduce separate variables \( z_{eq}, z_{leq} \) for each inequality and then combine them into \( z_{eq} \).

In Step 6.3 we derive \( z_{eq} \geq 1 \) by checking all combinations of values in \( A \) and \( B \), which requires \( O(|A| \cdot |B|) \) steps. Note that asymptotically this is the same number of steps as is required to compute which elements are in \( E \) so this step is still linear in the time needed to construct the encoding.

In Step 6.4 we use that \( z_{eq} \geq 1 \) and hence \( z_{eq} = z_{leq} = 1 \), which allows us to derive \( \text{sparse}(\bar{y}, A) + \text{sparse}(\bar{y}', B) \geq \text{sparse}(\bar{z}, E) \) and \( \text{sparse}(\bar{y}, A) + \text{sparse}(\bar{y}', B) \leq \text{sparse}(\bar{z}, E) \) respectively by removing \( z_{eq}, z_{leq} \) from the constraints introduced in Step 6.2.

Algorithm 7 describes in detail how to derive \( z_{eq} \geq 1 \) by checking all combinations of values in \( A \) and \( B \). Let us illustrate how the algorithm works with an example. Let \( A = \{0, 2\} \) and \( B = \{0, 2, 4\} \). After the first iteration of the outer loop the algorithm derives the clauses

\[
y_2 + y_2' + z_{eq} \geq 1 , \tag{22a}
y_2 + y_2' + z_{eq} \geq 1 \ , \text{ and} \tag{22b}
y_2 + \bar{y}_2 + z_{eq} \geq 1 . \tag{22c}
\]

Note that deriving (22a) by reverse unit propagation sets \( y_2 = y_2' = z_{eq} = 0 \). This causes the ordering constraints to propagate all variables in \( \bar{y} \) and \( \bar{y}' \). As all \( \bar{y} \) and \( \bar{y}' \) variables are set, the reification constraints introduced in Step 6.1 will cause all \( z \) variables to propagate. As the constraints reified in Step 6.2 are now satisfied we also get the propagation \( z_{eq} = z_{leq} = 1 \) and hence \( z_{eq} \) should be set to 1 as well. However, we already set \( z_{eq} \) to 0 and hence have a contradiction showing that (22a) can be derived. Deriving the other clauses works analogously.

If we add all clauses in (22) together, then \( y_2' \) and \( y_4' \) get canceled out and we are left with \( 3y_2 + 3z_{eq} \geq 1 \) which is saturated to obtain \( y_2 + z_{eq} \geq 1 \). Analogously, the second iteration of the outer loop derives \( \bar{y}_2 + z_{eq} \geq 1 \), which added to the result of the first iteration yields \( 2z_{eq} \geq 1 \) and using saturation we obtain \( z_{eq} \geq 1 \) as desired.

### A.3 Derivation for binary adder encoding

This section provides the algorithm for constructing the adder network in Algorithm 8 and the proof logging and derivation of the preserving equality (17) from Proposition 7 for a single binary full adder in Algorithm 9.

### B Totalizer and Generalized Totalizer Encoding

The totalizer and generalized totalizer encoding accumulate the input in form of a balanced binary tree. The totalizer encoding is designed for encoding cardinality constraints and uses the order encoding to represent values, while the generalized totalizer is designed for general pseudo-Boolean constraints and uses a sparse representation. An example of an arithmetic graph for the generalized totalizer encoding is shown in Figure 7. This graph contains a leaf node for each of the variables in the encoded constraint (to obtain a unique source we simply combine all leaf nodes into one node). The leaf nodes are combined in form of a binary tree, where we ensure that the value is preserved for each inner node, i.e., each possible value of incoming edges is representable as value of the outgoing edges. To perform \( k \)-simplification the arithmetic graph has...
Algorithm 6 Deriving a sparse unary sum over fresh variables $\vec{z}$.

1: procedure derive_sparse_unary_sum($C'$)
2:  \( \triangleright \) input: $C'$ is of the form $\text{sparse}({\vec{y}}, A) + \text{sparse}({\vec{y}'}, B) = \text{sparse}(\vec{z}, E)$ and describes the constraint to be derived such that $A, B \subseteq \mathbb{N}$, $E = \{i + j \mid i \in A, j \in B\}$ and $\vec{z}$ variables are fresh
3:  \( \triangleright \) Step 6.1: introduce variables as reification and derive ordering
4:  for $j \in E \setminus \{0\}$ do
5:  \[ D_{eq}^j, D_{eq}^j \leftarrow \text{reify}(z_j \leftrightarrow \text{sparse}({\vec{y}}, A) + \text{sparse}({\vec{y}'}, B) \geq j) \]
6:  for $i \in E \setminus \{0, \text{max}(E)\}$ do
7:  DeriveOrdering($D_{eq}^i, D_{eq}^{\text{succ}(i, E)}$) \( \triangleright \) derive $z_i \geq z_{\text{succ}(i, E)}$
8:  \( \triangleright \) Step 6.2: reify constraint to be derived
9:  $C_{eq} \leftarrow \text{reify}(z_{eq} \leftrightarrow \text{sparse}({\vec{y}}, A) + \text{sparse}({\vec{y}'}, B) \geq \text{sparse}(\vec{z}, E))$
10: $C_{eq} \leftarrow \text{reify}(z_{eq} \leftrightarrow \text{sparse}({\vec{y}}, A) + \text{sparse}({\vec{y}'}, B) \leq \text{sparse}(\vec{z}, E))$
11: reify($z_{eq} \leftrightarrow z_{eq} + z_{eq} \geq 2$)
12: \( \triangleright \) Step 6.3: derive that $z_{eq} \geq 1$
13: try_all_values($\text{sparse}({\vec{y}}, A), \text{sparse}({\vec{y}'}, B), z_{eq}$)
14: \( \triangleright \) Step 6.4: derive constraint to be derived from its reification
15: $M \leftarrow \text{max}(A) + \text{max}(B)$ \( \triangleright \) Coefficient so that reification variables get eliminated.
16: $D \leftarrow z_{eq} \geq 1$
17: proof_log(rup $D$)
18: proof_log(pol $C_{eq} \cdot D \cdot M \cdot \ast \ast$)
19: $C_{eq} \leftarrow C_{eq} + M \cdot D$
20: $D \leftarrow z_{eq} \geq 1$
21: proof_log(rup $D$)
22: proof_log(pol $C_{eq} \cdot D \cdot M \cdot \ast \ast$)
23: $C_{eq} \leftarrow C_{eq} + M \cdot D$
24: return $C_{eq}, C_{eq}$

Definition 8 (Arithmetic graph for the generalized totalizer encoding). Given a linear sum $\sum a_i x_i$ over $n$ variables, let $G$ be a binary tree with edges directed towards the root $r$, leaves $s_i$ for $i \in [n]$ and an additional sink node $t$ with an edge $(r, t)$. In what follows we will consider $r$ as an inner node. The edge $(s_i, v)$ from the leave $s_i$ is labeled with $a_i x_i$, which can be viewed as a sparse representation for values $\{0, a_i\}$. For an inner node $v$ with two incoming edges with labels sparse($\vec{y}$, $A$) and sparse($\vec{y}'$, $B$), the outgoing edge $e$ is labeled sparse($\vec{z}$, $E$), where $\vec{z}$ are fresh variables and $E = \{i + j \mid i \in A, j \in B\}$. To obtain a graph with a single source we combine all $s_i$ into a single node $s$. To perform $k$-simplification we split $\text{sparse}(\vec{z}, E) = \sum_{i \in E} a_i z_i$ into $\sum_{i \leq \text{succ}(k, E)} a_i z_i$, which is the label of the outgoing edge $e$, and $\sum_{i > \text{succ}(k, E)} a_i z_i$, which is the label for an addition outgoing edge $e' = (v, t)$.

To see that the defined graph is an arithmetic graph, we only need to check that we can derive the preserving equality for each inner node. Each inner node has two incoming edges that are labeled with a sparse unary representation and all outgoing edges together form a sparse unary representation as well, so that we can use Proposition 6 to derive the required preserving equality. Note that Proposition 6 also requires to have ordering...
Algorithm 7 Given a reified sparse unary sum, derive that the reification variable is true.

1: procedure fix(sparse($\bar{y}, A$), $a$)
2: return $\bar{y}_a + y_{\text{succ}(a,A)}$ \hspace{1cm} \triangleright replace $y_0$ by 1 and $y_\infty$ by 0
3: procedure try_all_values(sparse($\bar{y}, A$), sparse($\bar{y}', B$), $z_{eq}$)
4: $C_{\text{outer}} \leftarrow 0 \geq 0$
5: for $i \in A$ do
6: \hspace{1cm} $C_{\text{inner}} \leftarrow 0 \geq 0$
7: for $j \in B$ do
8: \hspace{2cm} \triangleright assuming that $a$ (respectively $b$) is the value encoded by sparse($\bar{y}, A$) (sparse($\bar{y}', B$))
9: \hspace{4cm} \triangleright encode that $(a = i \land b = j) \implies z_{eq}$
10: $D \leftarrow \text{fix(sparse($\bar{y}, A$), } i) + \text{fix(sparse($\bar{y}', B$), } j) + z_{eq} \geq 1$
11: $\text{proof_log(rup } D)$
12: $\text{proof_log(pol } C_{\text{inner}} \text{ D +)}$
13: $C_{\text{inner}} \leftarrow C_{\text{inner}} + D$
14: $\text{proof_log(pol } C_{\text{outer}} \text{ C_{\text{inner}} } s + )$
15: $C_{\text{outer}} \leftarrow C_{\text{outer}} + \text{saturate(C_{\text{inner}})}$
16: $\text{proof_log(pol } C_{\text{outer}} \text{ s)}$
17: $C_{\text{outer}} \leftarrow \text{saturate(C_{\text{outer}})}$
18: return $C_{\text{outer}}$ \hspace{1cm} \triangleright $C_{\text{outer}}$ is now $z_{eq} \geq 1$

Figure 7 Layout of the arithmetic graph for the generalized totalizer encoding of $x_1 + x_2 + x_3 + x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 \leq 2$. Edges introduced for k-simplification are colored cyan.

constraints on the input variables, however, it is easy to see by an inductive argument that the ordering constraints on the variables will be present, when processing the graph in topological order: Edges from the source only contain a single variable and hence the ordering constraints exist trivially. For inner nodes we get the ordering constraints by applying Proposition 6.

If the set of achievable values $E$ is dense for some node, i.e., $E$ contains all values from 0 to max($E$), then we can also use Proposition 5 to derive the required preserving equality, which only requires $O(|E|)$ instead of $O(|A| \cdot |B|)$ steps and hence can reduce the proof logging overhead.

For each inner node in the graph with incoming edge labels sparse($\bar{y}, A$) and sparse($\bar{y}', B$), the (generalized) totalizer encoding contains the clauses

\[
\begin{align*}
\bar{y}_i + \bar{y}'_j + z_{i+j} & \geq 1 \quad \text{for } i \in A, j \in B \\
\sum_{i=1}^{7} z_i & \geq 2\sum_{i=1}^{7} \sum_{i=1}^{3} (2z_{i} + 2z_{i}) \\
2x_1 + 2x_2 + 2x_3 + 2x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 & \leq 2
\end{align*}
\]
Algorithm 8 Construction of the binary adder network [18].

1: procedure adder_network(b)
2:  \[ \triangleright \text{input: vector of buckets } b \]
3:  for \( i \) from 0 to \( b_{\text{size}}() \) do
4:     while \( b_{\text{size}}() \geq 2 \) do
5:         if \( b_{\text{size}}() = 2 \) then
6:             \( x, y \leftarrow b_{i,\text{enqueue}}() \)
7:             \( c, s \leftarrow \text{full_adder}(x, y, 0) \)
8:         else
9:             \( x, y, z \leftarrow b_{i,\text{enqueue}}() \)
10:            \( c, s \leftarrow \text{full_adder}(x, y, z) \)
11:            \( b_{i+1,\text{enqueue}}(s) \)
12:          end
13:       end
14:     end
15:  \end{algorithm}

Algorithm 9 Proof logging the encoding of a single full adder.

1: procedure full_adder\((x, y, z)\)
2:  \( D_{\text{carry}}, D_{\text{equ}} \leftarrow \text{reify}(c \iff x + y + z \geq 2) \)
3:  \( D_{\text{sum}}, D_{\text{sum}} \leftarrow \text{reify}(s \iff x + y + z + 2\sigma \geq 3) \)
4:  \( D_{\text{eq}} \leftarrow (2 \cdot D_{\text{carry}} + D_{\text{sum}}) / 3 \)
5:  \( \text{proof_log}(\text{pol} D_{\text{carry}}^2 \cdot D_{\text{sum}}^3 + 3 \text{ d}) \)
6:  \( D_{\text{eq}} \leftarrow (2 \cdot D_{\text{carry}} + D_{\text{sum}}) / 3 \)
7:  \( \text{proof_log}(\text{pol} D_{\text{carry}}^2 \cdot D_{\text{sum}}^3 + 3 \text{ d}) \)
8:  \text{return} D, c, s \[ \triangleright D \text{ is the preserving equality of the full adder} \]

where \( \text{succ}(i, A) = \min\{j \mid j \in A \cup \{\infty\}, j > i \} \) and we replace \( y_0, y_0' \) with 1, and 
\( y_\infty, y'_\infty, z_\infty \) with 0 and simplify accordingly. Note that, (23) encodes that \( a + b = c \)
(where \( a \) and \( b \) are the incoming values and \( c \) is the output value), because (23a) encodes
that if \( a \geq i \) (expressed by assigning \( y_i \) to 1) and \( b \geq j \) then \( c \geq i + j \) while (23a) encodes
that if \( a \leq i \) (which is the same as saying that \( a < \text{succ}(i, A) \), expressed by assigning
\( y_{\text{succ}(i, A)} \) to 0) and \( b \leq j \) then \( c \leq i + j \).

For proof logging the CNF encoding we can simply add all clauses using RUP: A RUP
check of (23a) will assign \( y_i = y'_i = 1 \) and \( z_{i+j} = 0 \). The ordering constraints on \( \hat{y}, \hat{y}' \)
will cause a propagation setting multiple \( \hat{y}, \hat{y}' \) variables to true such that \( \text{sparse}(y, A) + \text{sparse}(y', B) \) has a value of at least \( i + j \), while the ordering constraints on \( \hat{s} \) will propagate
multiple \( \hat{s} \) to false such that \( \text{sparse}(z, E) \) can only take a value that is strictly less than
\( i + j \) and hence causes a conflict with the preserving equality \( \text{sparse}(z, E) = \text{sparse}(y, A) + \text{sparse}(y', B) \). Similarly, a RUP check of (23b) will assign \( y_{\text{succ}(i, A)} = y'_{\text{succ}(j, B)} = 0 \) and
\( z_{\text{succ}(i+j, E)} = 1 \) causing propagations such that \( \text{sparse}(y, A) + \text{sparse}(y', B) \) takes a value
less than or equal to \( i + j \) and \( \text{sparse}(z, E) \) takes a value strictly greater than \( i + j \)
causing again a conflict with the preserving equality.

To enforce a pseudo-Boolean constraint \( \sum_i a_i x_i \preceq k \), we first derive a bound on the
output of the arithmetic graph \( \sum_x c_{i1} \preceq k \), using Proposition 4. Then we can derive
unit clauses on the output via reverse unit propagation.

To encode \( \sum_i a_i x_i \geq k \) or \( \sum_i a_i x_i \leq k \) the clause \( z_{\text{succ}(k-1, E)} \geq 1 \) or \( z_{\text{succ}(k, E)} \geq 1 \)
is added, respectively. This clause is RUP, as the derived sum \( \sum_i c_{i1} a_i \) has a value of
at most \( k - 1 \) or at least \( k + 1 \) and thus the constraint \( \sum_i c_{i1} a_i \geq k \) or \( \sum_i c_{i1} a_i \leq k \) is
falsified, respectively. To encode \( \sum_i a_i x_i = k \) both clauses are added.
Table 2 Properties of pseudo-Boolean formulas used in the experimental results.

<table>
<thead>
<tr>
<th></th>
<th>Card</th>
<th>PB</th>
<th>Card+PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Inst.</td>
<td>772</td>
<td>442</td>
<td>308</td>
</tr>
<tr>
<td>Avg. #</td>
<td>107.01±252.57</td>
<td>0.00</td>
<td>1,154.43±5,881.78</td>
</tr>
<tr>
<td>Avg. #Lits</td>
<td>36.45±47.43</td>
<td>0.00</td>
<td>16.96±26.57</td>
</tr>
<tr>
<td>Avg. Coeff. Size</td>
<td>1.00±0.00</td>
<td>0.00</td>
<td>1.00±0.00</td>
</tr>
</tbody>
</table>

Figure 8 Comparison of runtimes between CNF translation with and without proof logging.

C Additional Evaluation Data

C.1 Benchmarks

Table 2 shows some properties of the benchmarks used in the experimental results, namely, the average number of cardinality constraints (Card), the average number of literals in each constraint, and the average size of coefficients associated with each literal. (The same is shown for PB constraints.) Since the benchmark set is composed of instances from multiple domains, there is a large dispersion of values between instances. For example, the number of cardinality constraints for instances in the Card benchmark set ranges from 1 to 2,720. Whereas the number of PB constraints for instances in the PB benchmark set ranges from 1 to 18,798. In the Card+PB benchmark set, we have an even larger dispersion with instances that have from 1 to 2,378,901 PB constraints and from 1 to 75,582 cardinality constraints.

C.2 Overhead of Proof Logging

Figure 8 shows the overhead of proof logging when translating the pseudo-Boolean formulas to CNF. For the majority of the instances, the overhead is not too significant, and formulas with just cardinality constraints can still be translated under 10 seconds, while formulas with PB constraints can be translated under 100 seconds. The exception
are the cardinality formulas from vertex cover that require super linear proofs, which lead to a higher overhead when storing the proof. Additionally, there were 6 instances that had memory outs when storing the proof in memory, which could be improved in the future by a more compact representation of the proof logging in VeritasPBLib.

**C.3 Solving and Verification**

Figure 9 shows the relationship between the time to generate the CNF translation and solve it using kissat and the time to verify the translation and solution using VeriPB. It can be seen that even though we can verify most instances, verification is often considerably slower than solving.

A lot of instances are spread in a wide range of different overheads. This wide range only comes from verifying the solution, which is out of the scope of this work. However, it motivates potential improvements to VeriPB which are complementary to the work proposed in this paper and can further increase the number of verified instances.