Working Title

Karthik V. Nukala*, Soumyaditya Choudhuri*, Randal E. Bryant*, Marijn J. H. Heule*
* Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Email: {kvn, soumyadc, rb3l, mheule}@andrew.cmu.edu

Abstract—Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

I. Introduction

A. Related Work

II. Preliminaries

A. Pseudo-Boolean Formulas

B. (Trusted) Binary Decision Diagrams

III. PBIP: Pseudo-Boolean Implication Proof

IV. Implementation

A. IPBIP: Hinting and Trimming Cutting-Planes Proofs

B. PBIP Translation and CNF Generation

C. BDD-Based PBIP Checking and LRAT Generation

V. Results

A. Benchmarks

B. Discussion

VI. Conclusion

Acknowledgments

- Andy Oertel - Lund University
- Ciaran McCreesh - University of Glasgow
- Yong Kiam Tan - A*STAR Singapore
- Ruben Martins/Joseph Reeves - CMU

VII. Notes on Pseudo-Boolean Implication Proofs

A. Pseudo-Boolean Formulas

A pseudo-Boolean constraint is a linear expression, viewing Boolean variables as ranging over integer values 0 and 1. That is, a relational constraint c has the form

$$a_1 l_1 + a_2 l_2 + \ldots + a_n l_n \quad \# \quad b$$

where: 1) the relational operator $\#$ is $<, \leq, \geq, >$, and 2) the coefficients a_i, as well as the constant b, are integers. We can also represent an equational constraint, having relational operator $=$, as the conjunction of two ordering constraints having the same coefficients but one with operator $\leq$ and the other with operator $\geq$.

Constraint c denotes a Boolean function, written $\llbracket c \rrbracket$, mapping assignments to the set of variables X to 1 (true) or 0 (false). Constraints c_1 and c_2 are said to be equivalent when $\llbracket c_1 \rrbracket = \llbracket c_2 \rrbracket$. Constraint c is said to be infeasible when $\llbracket c \rrbracket = \bot$, i.e., it always evaluates to 0. c is said to be trivial when $\llbracket c \rrbracket = \top$, i.e., it always evaluates to 1.

As described in [1], the following are some properties of pseudo-Boolean constraints:

- Constraints with relational operators $<, \leq$, and $>$ can be converted to equivalent constraints with relational operator $\geq$.
- The logical negation of relational constraint c, written $\overline{c}$, can be expressed as a relational constraint. That is, assume that c has been converted to a form where it has relational operator $>$. Then, replacing $\geq$ by $<$ yields the negation of c.

We will generally assume that constraints have relational operator $\geq$, since other forms can be translated to it.

We consider two normalized forms for ordering constraints: a coefficient-normalized constraint has only nonnegative coefficients. By convention, we require with this form that literal $l_i = x_i$ for any i such that $a_i = 0$. A variable-normalized constraint has only positive literals. Converting between the two forms is straightforward using the identity $\overline{x_i} = 1 - x_i$. In reasoning about PB constraints, the two forms can be used interchangeably. Typically, the coefficient-normalized form is more convenient when viewing a PB constraint as a logical expression, while the variable-normalized form is more convenient when viewing

https://doi.org/

a constraint as an arithmetic expression. We focus on the logical aspects, giving the general form of constraint c as

$$a_1 l_1 + a_2 l_2 + \ldots + a_n l_n \geq b \qquad (1)$$

with each $a_i \geq 0$ and with $l_i = x_i$ whenever $a_i = 0$.

Ordering constraint c in coefficient-normalized form is trivial if and only if $b \leq 0$. Similarly, c is infeasible if and only if $b > \sum_{1 \leq i \leq n} a_i$. By contrast, testing feasibility or triviality of an equational constraint is not straightforward, in that an instance of the subset sum problem [2] can be directly encoded as an equational constraint.

An assignment is a mapping $\rho : X' \rightarrow \{0,1\}$, for some $X' \subseteq X$. The assignment is total when $X' = X$ and partial when $X' \subset X$. Assignment ρ can also be viewed as a set of literals, where $x_i \in \rho$ when $\rho(x_i) = 1$ and $\overline{x_i} \in \rho$ when $\rho(x_i) = 0$. Assignment ρ is said to be consistent with assignment ρ' when $\rho \subseteq \rho'$.

Some nomenclature regarding constraints of the form of 1 will prove useful. The constraint literals are those literals l_i such that $a_i \neq 0$. A cardinality constraint has $a_i \in \{0,1\}$ for $1 \leq i \leq n$. A cardinality constraint with $b = 1$ is referred to as a clausal constraint: at least one of the constraint literals must be assigned 1 to satisfy a constraint. It is logically equivalent to a clause in a conjunctive normal form (CNF) formula. A cardinality constraint with $\sum_{1 \leq i \leq n} a_i = b$ is referred to as a conjunction: all of the constraint literals must be assigned 1 to satisfy the constraint. A conjunction for which $a_i = 1$ for just a single value of i is referred to as a unit constraint: it is satisfied if and only if literal l_i is assigned 1.

We let $c|_\rho$ denote the constraint resulting when c is simplified according to assignment ρ. Assume c has the form of 1 and partition the indices i for $1 \leq i \leq n$ into three sets: I^+, consisting of those indices i such that neither l_i nor $\overline{l_i}$ is in ρ. With this, $c|_\rho$ can be written as $\sum_{1 \leq i \leq n} a'_i \geq b'$ with a'_i equal to a_i for $i \in I^X$ and equal to 0 otherwise, and with $b' = b - \sum_{i \in I^+} a_i$.

A pseudo-Boolean formula F is a set of pseudo-Boolean constraints. We say that F is satisfiable when there is some assignment ρ that satisfies all of the constraints in F, and unsatisfiable otherwise.

B. (Reverse) Unit Propagation

Consider constraint c in coefficient-normalized form. Literal l_i is unit propagated by c when the assignment $\rho = \{\overline{l_i}\}$ causes the constraint $c|_\rho$ to become infeasible. As the name implies, a unit-propagated literal l_i then becomes a unit constraint. Observe that a single constraint can unit propagate multiple literals. For example, $4x_1 + 3\overline{x_2} + x_3 \geq 6$ unit propagates both x_1 and $\overline{x_2}$. For constraint c in coefficient-normalized form 1, detecting which literals unit propagate is straightforward. Let $A = \sum_{1 \leq i \leq n} a_i$. Then, literal l_i unit propagates if and only if $A - a_i < b$ i.e., $a_i > A - b$. For example, the constraint $4x_1 + 3\overline{x_2} + x_3 \geq 6$ has $A = 7$ and $b = 6$, yielding $A - b = 1$. This justifies the unit propagation of both x_1 and $\overline{x_2}$.

For constraint c, we let $Unit(c)$ denote the set of literals it unit propagates. Often, by simplifying a constraint c according to a partial assignment ρ, the simplified constraint $c|_\rho$ will unit propagate new literals, given by $Unit(c|_\rho)$. These literals can then be added to the partial assignment. Formally, define the operation $Uprop$ as $Uprop(\rho, c) = \rho \cup Unit(c|_\rho)$. Unit propagation is then the process of repeatedly applying this operation to a set of constraints to expand the set of literals in a partial assignment.

Consider a formula F consisting a set of constraints $c_1, c_2, \ldots, c_m$. The reverse unit propagation (RUP) proof rule [1] uses unit propagation to prove that target constraint c can be added to a formula while preserving its set of satisfying assignments. That is, any assignment that satisfies F also satisfies $F \wedge c$. A RUP addition justifies c by assuming $\overline{c}$ holds and showing, via a sequence of RUP steps, that this leads to a contradiction. It accumulates a partial assignment ρ based on unit propagations starting with the empty set. Each RUP step accumulates more assigned literals by performing a unit propagation of the form $\rho \leftarrow Uprop(\rho, d)$, where d is either c_j, a prior constraint, or $\overline{c}$, the negation of the target constraint. The final step causes a contradiction, where $d|_\rho$ is infeasible.

VIII. Pseudo-Boolean Implication Proofs

A Pseudo-Boolean Implication Proof (PBIP) provides a systematic way to prove that a PB formula F is unsatisfiable. It is given by a sequence of constraints, referred to as the proof sequence:

$$c_1, c_2, \ldots, c_m, c_{m+1}, \ldots, c_t$$

such that the first m constraints are those of formula F, while each added constraint c_i for $i > m$ follows by implication from the preceding constraints. That is,

$$\bigwedge_{1 \leq j < i} [\![c_j]\!] \Rightarrow [\![c_i]\!] \qquad (2)$$

The proof completes with the addition of an infeasible constraint for c_t. By the transitivity of implication, we have therefore proved that F is not satisfiable.

Constraints c_i with $i > m$, can be added in two different ways, corresponding to two different reasoning modes.

1) In implication mode, constraint c_i follows by implication from at most two prior constraints in the proof sequence. That is, for some $H_i \subseteq \{c_1, c_2, \ldots, c_{i-1}\}$ with $|H_i| \leq 2$ such that:

$$\bigwedge_{c_j \in H_i} [\![c_j]\!] \Rightarrow [\![c_i]\!] \qquad (3)$$

Set H_i is referred to as the hint for proof step i.

2) In RUP mode, the new constraint is justified by a reverse unit propagation addition. A sequence of hints is provided defining the RUP steps. Each hint is

of the form $[d_1, m_1], [d_2, m_2], \ldots, [d_{k-1}, m_{k-1}], [d_k]$, where each m_j is a unit-propagated literal, and d_j is either a previous constraint $c_{i'}$ for $i' < i$, or it is the negated target constraint $\overline{c_i}$. The final hint $[d_k]$ should have a conflict with literals $\{m_1, m_2, \ldots, m_{k-1}\}$. If a single constraint unit propagates multiple literals, these are listed as separate steps.

Unless $P = NP$, we cannot guarantee that a proof checker can validate even a single implication step of a PBIP proof in polynomial time. In particular, consider an equational constraint c encoding an instance of the subset sum problem, and let $c_{\leq}$ and $c_{\geq}$ denote its conversion into a pair of ordering constraints such that $[\![c]\!] = [\![c_{\leq}]\!] \wedge [\![c_{\geq}]\!]$. Consider a PBIP proof to add the constraint $\overline{c_{\leq}}$ having the $c_{\geq}$ as the only hint. Proving that $[\![c_{\geq}]\!] \Rightarrow [\![\overline{c_{\leq}}]\!]$, requires proving that $[\![c_{\leq}]\!] \wedge [\![\geq]\!] = \bot$, i.e., that c is unsatisfiable.

On the other hand, checking the correctness of a PBIP proof can be performed in pseudo-polynomial time, meaning that the complexity will be bounded by a polynomially sized formula over the numeric values of the integer parameters. This can be done using binary decision diagrams [3]. In particular, an ordering constraint over n variables in coefficient-normalized form with constant b will have a BDD representation with at most $b \cdot n$ nodes. For an implication proof step where the added constraints and the hints all have constants less than or equal to b, the number of BDD operations to validate the step will be $O(b^2 \cdot n)$ when there is a single hint and $O(b^3 \cdot n)$ when there are two hints. This complexity is polynomial in b, but it would be exponential in the size of a binary representation of b. The number of BDD operations for each unit propagation step in a RUP proof will be linear in the size of the BDD and therefore $O(b \cdot n)$.

IX. PBIP (Pseudo-Boolean Implication Proof) Format (pbip.txt)

The PBIP (Pseudo-Boolean Implication Proof) format describes a sequence of transformations on a set of pseudo-Boolean constraints leading to an infeasible constraint. The file therefore describes an unsatisfiability proof for a PB constraint problem.

Two reasoning modes are supported:

- Implication mode, where each new constraint follows by implication from one or two previous constraints
- Reverse unit propagation (RUP) mode, where the validity of a constraint is proved by contradiction. That is, a series of unit propagations are performed based on the complement of the constraint and other constraints until a contradiction is reached.

The format assumes that each input constraint is encoded as a set of clauses in conjunctive normal form (CNF). The clauses for all of the constraints are provided as a file in the standard DIMACS format.

When in implication mode, each derived constraint must follow by implication from either one or two preceding constraints, referred to as the "antecedents". That is, consider PB constraints P_1, P_2, and P. Each of these encodes a Boolean function. For P to follow from P_1, the constraints must satisfy $P_1 \implies P$. An antecedent constraint can be any of the following: an input constraint, one proved by an implication step, or the target constraint of a completed RUP reasoning sequence.

When in RUP mode, the constraint to be derived is set as a target, and a (initially empty) set of unit constraints is accumulated. Each RUP step then derives one or more additional unit constraints based on the set of previously derived unit constraints, as well as either the complement of the target constraint or some other preceding constraint. The final step should then cause a contradiction - the set of accumulated unit constraints falsifies the final constraint.

PBIP files build on the OPB format for describing PB constraints, as documented in [4].

We allow an extension to the OPB format for writing a PB formula: a variable name of the form xN can be preceded by (e.g. x2) to indicate its logical negation. This extension is also allowed by many other tools.

There are five types of lines:

1) Comment lines begin with "*" and contain arbitrary text.
2) Input lines begin with "i". This is followed by a constraint, expressed in OPB format, and terminated by ";". Then, a set of clause numbers is listed, separated by spaces and terminated with end-of-line. Forming the conjunction of these clauses an existentially quantifying any variables that are not listed in the PB formula should yield a Boolean function implying that of the PB constraint.
3) Implication-mode assertion lines begin with "a". This is followed by a constraint, expressed in OPB format and terminated by ";". Then, either one or two constraint IDs is listed, separated by spaces and terminated with end-of-line. Constraints are numbered from 1, starting with the input constraints.
4) RUP lines begin with "u". This is followed by a constraint, expressed in OPB format and terminated by ";". Then, a sequence of lists is given, where each list is of the form $[C\ L1\ \ldots\ L_k]$, indicating that constraint C will propagate additional units $L_1, \ldots, L_k$. C can either be the ID of a previous constraint, or it can be that of the current constraint. The latter case is known as a "self reference", and its unit propagations should be based on the negation of the target constraint. The final list is of the form $[C]$, and it must be falsified by the accumulated set of literals.
5) Summation implication lines begin with "s". This is followed by a constraint C, expressed in OPB format and terminated by ";". Then, a set of constraint IDs is listed, separataed by spaces and terminated with end-of-line. These IDs identify a set of prior

constraints $C_1, C_2, \ldots, C_k$ satisfying:

$$\sum_{i=1}^{k} C_i \Longrightarrow C$$

This line avoids the need to expand a summation of k constraints into $k-1$ implication lines. Instead, the checker performs the summations and tests the final implication, using heuristics to optimize the order in which the k constraints are summed.

For an unsatisfiability proof, the final constraint should be infeasible, e.g. $0 \geq 1$.

X. Converting PBIP Proof into Clausal Proof

We convert PBIP proofs into clausal proofs in the LRAT format using trusted Binary Decision Diagrams, or TBDDs. TBDDs extend conventional BDDs by having their standard operations also generate proof steps. We denote BDDs by their root nodes, using bold letters, e.g. $\mathbf{u}$. A TBDD $\dot{\mathbf{u}}$ consists of the following:

- A BDD having root node $\mathbf{u}$
- A Boolean extension variable u along with an associated proof clause defining the semantic relation between $\mathbf{u}$, the node variable x, and child nodes $\mathbf{u_1}$ and $\mathbf{u_0}$
- A proof of the unit clause $[u]$ indicating that the BDD will evaluate to 1 for any assignment that satisfies the input formula.

We assume our trusted BDD package implements the following operations:

- $\mathsf{BDD}(c)$: Generate a BDD representation of pseudo-Boolean constraint c
- $\mathsf{BDD\_AND}(\mathbf{u}, \mathbf{v})$: Compute BDD $\mathbf{w}$ as the conjunction of BDDs $\mathbf{u}$ and $\mathbf{v}$. Also, generate proof steps ending with the addition of the clause $[\overline{u} \vee \overline{v} \vee w]$ proving the clause $(u \wedge v) \Rightarrow w$.
- $\mathsf{BDD\_IMPLY}(\mathbf{u}, \mathbf{v})$: Generate proof steps ending with the addition of the clause $[\overline{u} \vee v]$ indicating that $[u \Rightarrow v]$.
- $\mathsf{BDD\_AND\_LITERALS}(L)$: For a set of literals $L = \{m_1, m_2, \ldots, m_k\}$ generate a BDD representation $\mathbf{u}$ of $\bigwedge_{1 \leq j \leq k} m_j$ and add clause $[\overline{m_1} \vee \overline{m_2} \vee \ldots \vee \overline{m_k} \vee u]$ to the proof.
- $\mathsf{BDD\_OR\_LITERALS}(L)$: For a set of literals $L = \{m_1, m_2, \ldots, m_k\}$ generate a BDD representation $\mathbf{u}$ of $\bigvee_{1 \leq j \leq k} m_j$ and add clause $[m_1 \vee m_2 \vee \ldots \vee m_k \vee \overline{u}]$ to the proof.
- $\mathsf{TBDD\_FROM\_CLAUSE}(C)$: Generate a TBDD representation $\dot{\mathbf{u}}$ of the proof clause C.

Our goal is to create a TBDD representation $\dot{\mathbf{u}}_i$ for each constraint $\mathbf{c_i}$ in the proof sequence. The final step will generate a trusted BDD for the BDD leaf node representing false. This will cause the empty clause to be added to the proof. When adding constraint c_i, its BDD representation $\mathbf{u}_i$ can be generated as $\mathsf{BDD}(c_i)$. To upgrade this to the trusted BDD $\dot{\mathbf{u}}_i$ requires generating the unit clause $[u_i]$. We assume that every prior proof constraint $c_{i'}$, with $i' < i$, has a TBDD representation $\dot{\mathbf{u}}_i$ with an associated unit clause $[u_{i'}]$.

When c_i is added by implication mode, generating its unit clause is based on the constraints given as the hint. If the hint consists of the single constraint $c_{i'}$, we can make use of its TBDD representation $\dot{\mathbf{u}}_{i'}$ by performing the implication test $\mathsf{BDD\_IMPLY}(\mathbf{u}_{i'}, \mathbf{u}_i)$, generating the clause $[\overline{u}_{i'}, u_i]$. Resolving this with the unit clause $[u_{i'}]$ then gives the unit clause $[u_i]$. When the hint consists of two constraints $c_{i'}$ and $c_{i''}$, we can make use of their TBDD representations $\cdot\mathbf{u}_{i'}$ and $\cdot\mathbf{u}_{i''}$. That is, let $\mathbf{w} = \mathsf{BDD\_AND}(\cdot\mathbf{u}_{i'}, \cdot\mathbf{u}_{i''})$, generating the clause $[\overline{u}_{i'} \vee \overline{u}_{i''} \vee w]$, and then perform the implication test $\mathsf{BDD\_IMPLY}(w, \mathbf{u}_i)$, generating the clause $[\overline{w} \vee u_i]$. Resolving these clauses with the unit clauses for TBDDs $\dot{\mathbf{u}}_{i'}$ and $\dot{\mathbf{u}}_{i''}$ yields the unit clause $[u_i]$.

Adding constraint c_i via a RUP addition rqeuires performing a series of clause generations for each RUP step and then assembling the generated clauses as the hints for a single clausal RUP addition. Suppose we start with the partial assignment $\rho_0 = \emptyset$ and accumulate successive literals through unit propagation, such that $\rho_j = \rho_{j-1} \cup \{m_j\}$ for $1 \leq j < k$. For step j, the RUP hint is of the form $[d_j, m_j]$. We justify the unit propagation of literal m_j based on the assignment $\rho_{j-1} \cup \{\overline{m}_j\}$. That is, when $d_j = c_{i'}$ for some $i' < i$, calling $\mathsf{BDD\_OR\_LITERALS}(\{\overline{l} \mid l \in \rho_{j-1}\} \cup \rho_{j-1} \cup \{\overline{m}_j\})$ will generate BDD node $\mathbf{v}$ as well as proof clause $[\overline{m}_1 \vee \overline{m}_2 \vee \ldots \vee \overline{m}_{j-1} \vee m_j \vee \overline{v}]$, and then calling $\mathsf{BDD\_IMPLY}(\mathbf{u}_{i'}, \mathbf{v})$ will generate the proof clause $[\overline{u}_{i'} \vee v]$. Resolving these two clauses, along with unit clause $[u_{i'}]$ yields proof clause $H_j = [\overline{m}_1 \vee \overline{m}_2 \vee \ldots \overline{m}_{j-1} \vee m_j \vee u_i]$. The final RUP step will not involve any unit propagations. It will generate either proof clause $H_k = [\overline{m}_1 \vee \overline{m}_2 \vee \ldots \vee \overline{m}_{k-1}]$ or proof clause $H_k = [\overline{m}_1 \vee \overline{m}_2 \vee \ldots \vee \overline{m}_{k-1} \vee u_i]$. The final clausal RUP addition has unit clause $[\overline{u}_i]$ and accumulate the literals $m_1, m_2, \ldots, m_{k-1}$ as unit literals from the hints. The final clause will cause a conflict.

XI. Optimizing for Clausal Constraints

In practice, many of the constraints encountered in PBIP proofs are of the form $c = m_1 + m_2 + \ldots + m_k \geq 1$. This is logically equivalent to the clause $C = [m_1 \vee m_2 \vee \ldots m_k]$. We can convert PBIP RUP additions into clausal proofs using such constraints directly, rather than converting them to BDDs. To do so, clause C must occur in the proof. If C was derived by our (modified) RUP derivation, then it will already be part of the proof. Otherwise, for a constraint represented by $\dot{\mathbf{u}}$ invoking $\mathsf{BDD\_OR\_LITERALS}(C)$ should return node $\mathbf{u}$ as the result, and we can resolve the generated proof clause with the unit clause $[u]$ to get C added as a proof clause. On the other hand, if the proof clause C is later needed as part of the hint for an implication-mode addition, we can generate its TBDD representation $\dot{\mathbf{u}}$ with $\mathsf{TBDD\_FROM\_CLAUSE}(C)$.

We consider two refinements to our method of converting the RUP derivations of PBIP into RUP derivations in the clausal proof. First, suppose some RUP step $[d_j, m_j]$ has $d_j = c_{i'}$ and constraint $c_{i'}$ is represented as clause $C_{i'}$. Then, we can let $H_j = c_{i'}$ and have the RUP checker use it for unit propagation. Second, suppose the target constraint c_i is itself clause C_i. Then, final clause RUP addition can have C as its target, and we can build the hint sequence $H_1, H_2, \ldots, H_k$ by starting with assignment ρ_0 consisting of the negated literals of C_i. With these optimizations, we can see that if a PBIP proof is simply a constraint version of a clausal RUP proof, we will generate that clausal proof.

References

[1] S. Gocht, "Certifying correctness for combinatorial algorithms: by using pseudo-boolean reasoning," 2022.

[2] M. R. Garey, "Optimal binary identification procedures," SIAM Journal on Applied Mathematics, vol. 23, no. 2, pp. 173–186, 1972.

[3] R. E. Bryant, A. Biere, and M. J. Heule, "Clausal proofs for pseudo-boolean reasoning," in International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 443–461, Springer, 2022.

[4] O. Roussel and V. Manquinho, "Input/output format and solver requirements for the competitions of pseudo-boolean solvers," 2012.