Notes on Pseudo-Boolean Implication Proofs Version of October 7, 2023

Randal E. Bryant

Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Randy.Bryant@cs.cmu.edu

This notes describes some ideas on converting unsatisfiability proofs for pseudo-Boolean (PB) constraints into clausal proofs based on the DRAT proof system.

1 Background

This section gives brief definitions of pseudo-Boolean constraints and their properties. A more extensive introduction is provided by Gocht in his PhD thesis [Gocht-Phd-2022].

Consider a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. For $x_i \in X$, *literal* ℓ_i can denote either x_i or its negation, written $\overline{x}_i$. We write $\overline{\ell}_i$ to denote $\overline{x}_i$ when $\ell_i = x_i$ and x_i when $\ell_i = \overline{x}_i$.

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[1] has the form

$$a_1\ell_1 + a_2\ell_2 + \cdots a_n\ell_n \quad \# \quad b \tag{1}$$

where: 1) the relational operator $\#$ is $<$, $\leq$, $\geq$, or $>$, 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 $[\![c]\!]$, mapping assignments to the set of variables X to 1 (true) or 0 (false). Constraints c_1 and c_2 are said to *equivalent* when $[\![c_1]\!] = [\![c_2]\!]$. Constraint c is said to be *infeasible* when $[\![c]\!] = \bot$, i.e., it always evaluates 0. c is said to be *trivial* when $[\![c]\!] = \top$, i.e., it always evaluates to 1.

As described in [Gocht-Phd-2022], 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 also be expressed as a relational constraint. That is, assume that c has been converted to a form where it has relational operator $\geq$. 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

[1] We use lower case c to denote a constraint, reserving upper case C to denote a clause.

$\ell_i = x_i$ 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 a constraint as an arithmetic expression. In this document, we focus on the logical aspects, giving the general form of constraint c as

$$a_1\ell_1 + a_2\ell_2 + \cdots a_n\ell_n \quad \geq \quad b \tag{2}$$

with each $a_i \geq 0$, and with $\ell_i = x_i$ 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 [Garey] 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 (2) 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 the 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 assignment ρ. Assume c has the form of (2) and partition the indices i for $1 \leq i \leq n$ into three sets: I^+, consisting of those indices i such that $\ell_i \in \rho$, I^-, consisting of those indices i such that $\overline{\ell}_i \in \rho$, and I^X consisting of those indices i such that neither ℓ_i nor $\overline{\ell}_i$ 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.

2 (Reverse) Unit Propagation

Consider constraint c in coefficient-normalized form. Literal ℓ_i is *unit propagated* by c when the assignment $\rho = \{\overline{\ell}_i\}$ 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 coefficent-normalized form (2), 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 propagations 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 $\mathit{Unit}(c|_\rho)$. These literals can then be added to the partial assignment. Formally, define the operation *Uprop* as $\mathit{Uprop}(\rho, c) = \rho \cup \mathit{Unit}(c|_\rho)$. *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 of a set of constraints $c_1, c_2, \ldots, c_m$. The *reverse unit propagation* (RUP) proof rule [Gocht-Phd-2022] 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 $\rho \leftarrow \mathit{Uprop}(\rho, d)$, where constraint 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 infeasbile.

2.1 RUP Example

As an example, consider the following three constraints:

ID	Constraint					
c_1	x_1	$+$	$2x_2$	$+$	$\overline{x}_3$	≥ 2
c_2	$\overline{x}_1$	$+$	$\overline{x}_2$	$+$	$2x_3$	≥ 2
c_3	x_1	$+$	$2\overline{x}_2$	$+$	$3\overline{x}_3$	≥ 3

Our goal is to add the constraint $c = 2x_1 + x_2 + x_3 \geq 2$. RUP addition proceeds by the following four RUP steps:

1. We can see that $\overline{c} = 2\overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 3$, and this unit propagates assignment $\rho_1 = \{\overline{x}_1\}$.
2. With this, constraint c_1 simplifies to $2x_2 + \overline{x}_3 \geq 2$, and this unit propagates x_2, giving $\rho_2 = \{\overline{x}_1, x_2\}$.
3. Constraint c_2 simplifies to $2x_3 \geq 1$, which unit propagates x_3, giving $\rho_3 = \{\overline{x}_1, x_2, x_3\}$.
4. Constraint c_3 simplifies to $0 \geq 3$, which is infeasible.

3 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,

$$\bigwedge_{1 \leq j < i} [\![c_j]\!] \quad \Rightarrow \quad [\![c_i]\!] \tag{3}$$

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:

$$\bigwedge_{c_j \in H_i} [\![c_j]\!] \quad \Rightarrow \quad [\![c_i]\!] \tag{4}$$

   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,[2] 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 step 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 $[\![c_{\leq}]\!] \wedge [\![c_{\geq}]\!] = \bot$, 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 [BBH-2022]. 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 $O(b \cdot n)$.

4 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 clauses defining the semantic relation between $\mathbf{u}$, the node variable x, and child nodes 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

[2] Note that the indexing of these literals, e.g., m_j is unrelated to the indexing of the variables, e.g., x_j.

BDD(c): Generate a BDD representation of pseudo-Boolean constraint c

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 $(u \wedge v) \Rightarrow w$.

BDD_IMPLY($\mathbf{u}, \mathbf{v}$): Generate proof steps ending with the addition of the clause $[\overline{u} \vee v]$ indicating that $u \Rightarrow v$.

BDD_AND_LITERALS(ρ): For (possibly partial) assignment $\rho = \{m_1, m_2, \ldots, m_k\}$ generate a BDD representation $\mathbf{u}$ of $\bigwedge_{1 \leq j \leq k} m_j$ and add proof steps ending with the addition of the clause $[\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_k \vee u]$.

BDD_NAND_LITERALS(ρ): For (possibly partial) assignment $\rho = \{m_1, m_2, \ldots, m_k\}$ generate a BDD representation $\mathbf{u}$ of $\neg \bigwedge_{1 \leq j \leq k} m_j$ and add proof steps ending with the addition of the clause $[\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_k \vee \overline{u}]$.

TBDD_FROM_CLAUSE(C): Generate a TBDD representation $\dot{\mathbf{u}}$ of proof clause C.

TBDD_JUSTIFY_CLAUSE($\dot{\mathbf{u}}, C$): This is to be called when BDD $\mathbf{u}$ is logically equivalent to clause C. Generate a series of proof steps that culminate with the addition of C to the proof.

Our goal is the create a TBDD representation $\dot{\mathbf{u}}_i$ for each constraint 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 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 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 representation $\dot{\mathbf{u}}_{i'}$ and $\dot{\mathbf{u}}_{i''}$. That is, let $\mathbf{w} =$ BDD_AND($\mathbf{u}_{i'}, \mathbf{u}_{i''}$), generating the clause $[\overline{u}_{i'} \vee \overline{u}_{i''} \vee w]$, and then perform the implication test 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 requires 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 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 BDD_NAND_LITERALS($\rho_{j-1} \cup \{\overline{m}_j\}$) will generate BDD node $\mathbf{v}$ as well as proof clause $[\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{j-1} \vee m_j \vee \overline{v}]$, and then calling 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 $H_j = [\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{j-1} \vee m_j]$. Similarly, when $d_j = \overline{c}_i$, calling BDD_AND_LITERALS($\rho_{j-1} \cup \{\overline{m}_j\}$) will generate BDD node $\mathbf{v}$ as well as proof clause $[\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{j-1} \vee m_j \vee v]$, and then calling BDD_IMPLY($\mathbf{v}, \mathbf{u}_i$) will generate the proof clause $[\overline{v} \vee u_i]$. Resolving these two clauses yields proof clause $H_j = [\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{j-1} \vee m_j \vee u_i]$. The final RUP step will not involve any unit propagations. It will generate either proof clause $H_k = [\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{k-1}]$ or proof clause $H_k = [\overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{k-1} \vee u_i]$. The final clausal RUP addition has unit clause $[u_i]$ as its target and the clauses $H_1, H_2, \ldots, H_k$ as its hints. RUP addition will start with unit literal $\overline{u}_i$ 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.

5 Optimizing for Clausal Constraints

In practice, many of the constraints encountered in PBIP proofs are of the form $c = m_1 + m_2 + \cdots + m_k \geq 1$. This is logically equivalent to the clause $C = [m_1 \vee m_2 \vee \cdots \vee m_k]$. 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, we can invoke TBDD_JUSTIFY_CLAUSE to add C to the proof from its TBDD representation. Similarly, if proof clause C is later needed as part of the hint for an implication-mode addition we can generate its TBDD representation TBDD_FROM_CLAUSE.

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 clausal 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.