

Notes on Pseudo-Boolean Implication Proofs Version of October 6, 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 . For $x \in X$, *literal* ℓ can denote either x or its negation, written \bar{x} . We write $\bar{\ell}$ to denote \bar{x} when $\ell = x$ and x_i when $\ell = \bar{x}$.

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

$$\sum_{1 \leq i \leq n} a_i \ell_i \# b \tag{1}$$

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

¹We 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 $\bar{x}_i = 1 - x_i$. 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

$$\sum_{1 \leq i \leq n} a_i \ell_i \geq b \quad (2)$$

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 $\bar{x}_i \in \rho$ 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 $\bar{\ell}_i \in \rho$, and I^X consisting of those indices i such that neither ℓ_i nor $\bar{\ell}_i$ 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 is *unit propagated* by c when the assignment $\rho = \{\bar{\ell}\}$ causes the constraint $c|_\rho$ to become infeasible. As the name implies, a unit-propagated literal ℓ then becomes a unit constraint. Observe that a single constraint can unit propagate multiple literals. For example, $4x_1 + 3\bar{x}_2 + x_3 \geq 6$ unit propagates both x_1 and \bar{x}_2 . For constraint c in coefficient-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 $4x_1 + 3\bar{x}_2 + x_3 \geq 6$ has $A = 7$ and $b = 6$, yielding $A - b = 1$. This justifies the unit propagations of both x_1 and \bar{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 of a set of constraints c_1, c_2, \dots, 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 $F \wedge c$. RUP justifies c by assuming \bar{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 constraint d is either c_j , a prior constraint, or \bar{c} , the negation of the target constraint. The final step causes a contradiction, where $d|_\rho$ is infeasible.

2.1 RUP Example

As an example, consider the following three constraints:

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

Our goal is to add the constraint $c = 2x_1 + x_2 + x_3 \geq 2$. The RUP steps proceed as follows:

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

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

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,

$$\bigwedge_{1 \leq j < i} \llbracket c_j \rrbracket \Rightarrow \llbracket c_i \rrbracket \quad (3)$$

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 $H_i \subseteq \{c_1, c_2, \dots, c_{i-1}\}$ with $|H_i| \leq 2$ such that:

$$\bigwedge_{c_j \in H_i} \llbracket c_j \rrbracket \Rightarrow \llbracket c_i \rrbracket \quad (4)$$

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

2. In *RUP mode*, the new constraint is justified by reverse unit propagation. A sequence of hints is provided of the form $[d_1, m_1], [d_2, m_2], \dots, [d_{k-1}, m_{k-1}], [d_k]$, 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 \bar{c}_i . The final hint $[d_k]$ should have a conflict with literals $\{m_1, m_2, \dots, m_{k-1}\}$. 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 $\llbracket c \rrbracket = \llbracket c_{\leq} \rrbracket \wedge \llbracket c_{\geq} \rrbracket$. Consider a PBIP proof step to add the constraint \bar{c}_{\leq} having the c_{\geq} as the only hint. Proving that $\llbracket c_{\geq} \rrbracket \Rightarrow \llbracket \bar{c}_{\leq} \rrbracket$, requires proving that $\llbracket c_{\leq} \rrbracket \wedge \llbracket c_{\geq} \rrbracket = \perp$, 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 \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 \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

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 $[\bar{u} \vee \bar{v} \vee w]$ proving the $(u \wedge v) \Rightarrow w$.

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

BDD_FALSIFIES(\mathbf{u}, ρ): For (possibly partial) assignment $\rho = \{m_1, m_2, \dots, m_k\}$ generate proof steps ending with the addition of the clause $[\bar{m}_1 \vee \bar{m}_2 \vee \dots \vee \bar{m}_k \vee \bar{u}]$. This clause indicates that any total assignment consistent with ρ will define a path in the BDD leading from root node \mathbf{u} to the leaf node representing false.

BDD_SATISFIES(\mathbf{u}, ρ): For (possibly partial) assignment $\rho = \{m_1, m_2, \dots, m_k\}$ generate proof steps ending with the addition of the clause $[\bar{m}_1 \vee \bar{m}_2 \vee \dots \vee \bar{m}_k \vee u]$. This clause indicates that any total assignment consistent with ρ will define a path in the BDD leading from root node \mathbf{u} to the leaf node representing true.

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

TBDD_MAKE_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 to 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 $\text{BDD}(c_i)$. 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 $\text{BDD_IMPLY}(\dot{\mathbf{u}}_{i'}, \mathbf{u}_i)$, generating the clause $[\bar{u}_{i'} \vee u_i]$. 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 $\mathbf{w} = \text{BDD_AND}(\dot{\mathbf{u}}_{i'}, \dot{\mathbf{u}}_{i''})$, generating the clause $[\bar{u}_{i'} \vee \bar{u}_{i''} \vee w]$, and then perform the implication test $\text{BDD_IMPLY}(w, \mathbf{u}_i)$, generating the clause $[\bar{w} \vee u_i]$. 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 sequence of RUP steps requires performing a series of *clause generation* steps followed by a single step that combines these clauses into a single clausal RUP step. 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 each RUP hint of the form $[d_j, m_j]$, we justify the unit propagation of literal m_j based on the assignment $\rho_{j-1} \cup \{\bar{m}_j\}$. If $d_j = c_{i'}$ for some $i' < j$, calling $\text{BDD_FALSIFIES}(\dot{\mathbf{u}}_{i'}, \rho_j)$ will generate the clause $[\bar{m}_1 \vee \bar{m}_2 \vee \dots \vee \bar{m}_{j-1} \vee m_j \vee \bar{u}_{i'}]$, and we can resolve this with unit clause $[u_{i'}]$ to get $[\bar{m}_1 \vee \bar{m}_2 \vee \dots \vee \bar{m}_{j-1} \vee m_j]$. If $d_j = \bar{c}_i$, calling $\text{BDD_SATISFIES}(\mathbf{u}_i, \rho_j)$ will generate the clause $[\bar{m}_1 \vee \bar{m}_2 \vee \dots \vee \bar{m}_{k-1} \vee m_k \vee u_i]$. The final RUP step has unit clause $[u_i]$ as its target, and therefore it will start with unit literal \bar{u}_i and accumulate successive literals from the hints. The hints will consist of the clauses generated for each RUP step. These will propagate the literals m_1, m_2, \dots, m_{k-1} in succession, and the final clause will cause a contradiction.

5 Optimizing for Clausal Constraints

In practice, many of the constraints encountered in PBIP proofs are of the form $c = m_1 + m_2 + \dots + m_s \geq 1$. This is logically equivalent to the clause $C = [m_1 \vee m_2 \vee \dots \vee m_s]$. For proof steps based on RUP, we can use the clausal representations of such constraints directly, rather than converting them to BDDs. To do so, we must include clause C in the proof sequence. If C was derived by our (modified) RUP derivation, then it will already be part of the proof. Otherwise, we may can invoke `TBDD_MAKE_CLAUSE` to add C to the proof from its TBDD representation. Similarly, we can generate the TBDD representation of proof clause C by invoking `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 of the form $[c_{i'}, m]$ has $c_{i'}$ represented as clause $C_{i'}$. Then we can list it directly in the list of hints for the derivation and have the RUP checker use it for unit propagation. Second, suppose the target constraint c_i is itself clause C_i . Then we can start the RUP derivation with the 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.