Applied Presolve Reductions in Pseudo-Boolean Solving

Asger Kjeldsen

Fall 2020/Spring 2021

Abstract

Pseudo-Boolean solving is a powerful approach to solving 0-1 integer linear programs. Building on the well known Conflict Driven Clause Learning algorithm, the gap between satisfiability solving in propositional logic and Pseudo-Boolean optimization is not very wide. This paper reviews the similarities between satisfiability solving in propositional logic and Pseudo-Boolean optimization, as well as the algorithms used in conflict driven solving. Presolve is a key step for mixed integer programs (MIP) in relation to solve speed. Therefore, a program to enable MIP presolve techniques to be applied to Pseudo-Boolean problem instances was developed. In this paper, the implementation of the presolve program which utilizes the PaPILO MIP presolver is described. Executing the presolve step before solving an instance with the solver RoundingSat, reveals that presolving impacts the speed of solving significantly, though not exclusively positively.

1 Introduction

In this paper, we investigate the inner workings of conflict driven solving of logical formulas and how this approach is applied in Pseudo-Boolean (PB) solving. A Pseudo-Boolean instance is a series of linear constraints containing variables that can take the values 0 and 1. In other words, an *Integer Linear Program* with all 0-1 variables. This paper focuses on understanding how well known algorithms from propositional logic such as the Davis-Putnam-Logemann-Loveland (DPLL) [4] and Conflict Driven Clause Learning (CDCL) [9][3] can be used to solve such systems. This paper bridges the

gap from propositional logic to Pseudo-Boolean solving by connecting the fundamental pieces needed to construct a conflict driven Pseudo-Boolean solver. Building on these pieces, we take a look at presolve reductions in Mixed Integer Programming (MIP). Applied to MIP instances, these presolve steps are known to be very effective at decreasing solve time, and can in some cases turn an intractable problem solvable [1]. A PB problem is a MIP sub-problem. The idea of applying such presolve methods to PB instances motivated this paper, with the goal of creating a tool to presolve PB problem instances. Utilizing RoundingSat [15], an implementation of a PB solver, and the presolving tool PaPILO [14], a presolver for MIP, this paper introduces a new presolver for PB problems *PresolveOpb*. PresolveOpb allows for presolving any PB instance in the `.opb` file format as defined in *Input/Output Format and Solver Requirements for the Competitions of Pseudo-Boolean Solvers* [12]. The PB presolver uses the presolvers provided by PaPILO but restricts reductions that would break the structure of a 0-1 integer program. The paper is organized as follows. Section 2 reviews the elements of propositional logic needed to understand DPLL and CDCL, as well as defining the foundation of terminology used in the paper. Then CDCL is reviewed. Section 3 introduces conflict driven PB solving and MIP presolving. Section 4 and 5 cover the implementation of the PresolveOpb program, and cover testing on the PB16 competition instances. The conclusions are summarized in section 6.

2 Preliminaries

2.1 Propositional logic review

Propositional logic [8] concerns propositions that can be either true or false. These propositions occur in natural language for example in *"The sun is shining"*. Formalized these are represented as *variables* which then can be assigned with a *truth symbol*.

A truth symbol is $\texttt{T} = 1$ or $\texttt{F} = 0$.

A Variable $(q, r, s, t...)$ can be assigned T or F.

An atom is a variable or a truth symbol.

A literal is an atom or its negation.

Connectives relate atoms and includes $and(\wedge)$, $not(\neg)$, $or(\vee)$ and $implication(\rightarrow)$. This thesis will use these connectives but let it be noted that any connective can be defined and used and many combinations and subsets are sufficient to describe all problems in propositional logic. Let's consider the atoms of this example sentence:

$$\underset{p}{\textit{The sun}}\ \textit{shines or it}\ \underset{q}{\textit{rains}},\ \textit{but not both.}$$

In this example we can define $p :=$ *The sun is shining* and $q :=$ *It is raining*. Then we can construct the proposition, or formula F, from the sentence:

$$F = (p \vee q) \wedge \neg(p \wedge q)$$

This can also be described with the connective `XOR` (exclusive or) which supports the point that connectives can be translated to combinations of other connectives.

2.2 Satisfiability

The primary objective of the algorithms used in this project is to determine whether a formula F or Pseudo-Boolean model is satisfiable.

Definition 2.1. *If a formula F is always true it is valid and we denote that with $\models F$.*

A *truth assignment* ρ is the assignment of T or F to some or all of the literals in a formula. For a variable x let $\sigma \in \{0, 1\}$ and then denote x with x^σ such that $x^0 = \neg x$ and $x^1 = x$. If $x = \mathtt{F}$ we write $x^1 = 0$ and $x^0 = 1$. If $x^{1-\sigma} \in \rho$ then $\rho(x^\sigma) = 0$ and if $\{x^0, x^1\} \notin \rho$ then $\rho(x^\sigma) = *$ [3]. For propositional problems we use T and F interchangeably with 1 and 0 to bridge the gap between Pseudo Boolean Solving and propositional logic. We also use the notation $F|_\rho$ when an assignment ρ is applied to a formula F and the notation $\models F|_\rho$ if ρ satisfies F.

Definition 2.2. *A formula F is satisfiable if $\not\models \neg F$. In other words, F is not false in all cases and a ρ exists such that $\models F|_\rho$*

By example:

$$F ::= (x_1 \vee x_3) \wedge (x_2 \vee x_1) \wedge (\neg x_2 \vee x_4) \tag{2.1}$$
$$\not\models F \tag{2.2}$$
$$\{x_1^1, x_2^0\} \in \rho \tag{2.3}$$
$$\models F|_\rho \tag{2.4}$$

In 2.1 we see the CNF formula F introduced. Referring to definition 2.1 F by itself is not valid, since it does not evaluate T in all cases, proven by for example the assignment $\{x_1^0, x_2^0\}$ which falsifies the second clause. ρ satisfies F as seen in 2.4, so F is indeed satisfiable.
Another way to represent a problem in propositional logic is by constructing a truth table. Consider table 2.5 of an unknown formula F.

x_1	x_2	x_3	F
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

(2.5)

It is apparent that the formula F is satisfiable since at least one of the evaluations result in T. Conversely, an unsatisfiable formula would not have any truth table line evaluating T. Exploring all truth assignments to see if a formula is satisfiable is for many problems not feasible in a timely manner.

2.3 CNF

When taking an algorithmic approach to propositional logic, conventional formulation of problems is beneficial. Conjunctive Normal Form (CNF) allows for validity checking in linear time which for an arbitrary formula can take exponential time in the number of atoms [8]. CNF formulas are of the form:

$$\bigwedge_{i=1}^{n} \phi(i) \text{ where } \phi(i) = \bigvee_{j=1}^{k_i} L_{i,j} \text{ and } L_{i,j} \text{ is the } j\text{'th of } k_i \text{ literals in clause } i$$

In other words, a conjunction of disjunctive clauses.
To see that any formula can be written in CNF consider truth table 2.5. Recall, that this is the table for an unknown formula F. Even though the formula is unknown, *CNF*(F) can be found. For a line in the truth table l we write $l \models F$ iff F is `T` in the given line. To get an F equivalent CNF formula, the truth table has to be equal for both formulas. For one of the lines resulting in `F` to not hold in our CNF formula, a variable in the line has to be assigned with a falsifying truth value. Therefore, we can construct three clauses - one for each line computing `F`:

$$\psi_1 = \neg x_1 \vee x_2 \vee x_3 \qquad \psi_2 = x_1 \vee \neg x_2 \vee x_3 \qquad \psi_3 = x_1 \vee x_2 \vee x_3$$

If all these clauses are satisfied, F will also be satisfied since for any $l \models \psi_1 \wedge \psi_2 \wedge \psi_3$ it holds that $l \models F$. So for this example *CNF*(F) $= \psi_1 \wedge \psi_2 \wedge \psi_3$. All logical formulas have a truth table and this argument holds for all truth tables. Thus, all formulas in propositional logic can be converted to CNF.

2.4 DPLL

A direct way of solving the satisfiability problem is using the Davis-Putnam-Logemann–Loveland (DPLL) procedure [4]. The intuition for the algorithm in it's simplest form is to check *truth assignments* until either the algorithm finds a satisfying assignment or all possible satisfying assignments are exhausted.
Let's look at ways to simplify a CNF formula $F = \phi_1 \wedge \phi_2 \wedge ... \wedge \phi_n$ with $0 < k \leq n$ constraints under the assignment ρ.

- If a variable x_i^0 and x_i^1 both occur in ϕ_k, we can remove ϕ_k from the formula, since $\models x_i^0 \vee x_i^1$ in all cases (algorithm 1).

- If a variable $x_i^\sigma \in \rho$ is in ϕ_k we can remove ϕ_k from F since ϕ_k then evaluates 1 in all cases (algorithm 2).
- If a variable $x_k^{1-\sigma} \in \rho$ remove all x_k^σ from F. In other words, remove x_k^σ where it can no longer satisfy ϕ (algorithm 3).

Algorithm 1: Remove Clauses With Complements

```
RemoveComplements (F)
    foreach clause φ ∈ F do
        if φ contains a pair of x_k^1 and x_k^0 then
            F ← F \ φ
    return F
```

Algorithm 2: Remove Satisfied Clauses

```
RemoveSatClauses (F, ρ)
    foreach clause φ ∈ F do
        if φ contains a x_k^σ ∈ ρ then
            F ← F \ φ
    return (F, ρ)
```

Algorithm 3: Remove Non-satisfying Variables

```
RemoveNonSatVars (F, ρ)
    foreach clause φ ∈ F do
        foreach var x^σ ∈ φ do
            if x^{1−σ} ∈ ρ then
                φ ← φ \ x^σ
    return (F, ρ)
```

Utilizing these rules we will encounter cases where some clause ϕ_l only contains one literal. Since the goal of the DPLL algorithm is to determine whether a formula is satisfiable, that literal has to satisfy ϕ_l so we use *unit propagation* to set that literal in ρ [4]. Let's call the single variable x_k^σ then it follows that $\rho \leftarrow \rho \cup x_k^\sigma$ to satisfy clause ϕ_l.

Algorithm 4: Unit Propagation

```
UnitPropagate (F, ρ)
    while F has a unit clause φ_k with var x_l^σ do
        ρ ← ρ ∪ x_l^σ
        F ← RemoveSatClauses(F, ρ)
        F ← RemoveNonSatVars(F, ρ)
    return (F, ρ)
```

When unit propagation is not possible due to no unit clauses being present, DPLL selects a random variable and explores the satisfiability of that assignment. If at any point the algorithm reaches the decision stage and there are no unassigned variables, the algorithm returns immediately with the satisfying assignment ρ. If **RemoveNonSatVars** finds that some clause can't be satisfied, the algorithm returns. So if for some variable x^σ both decisions x^0 and x^1 are unsatisfiable and there are no more decisions to be made, the algorithm returns false since no ρ such that $\models F|_\rho$ exist. Tying it all together we get algorithm 5: DPLL.

Algorithm 5: DPLL

```
DPLL (F, ρ)
    if F contains an empty clause then
        return false
    UnitPropagate(F, ρ)
    if All variables ∈ ρ then
        Exit and return true and the assignment ρ
    L ← a variable from F not in ρ
    return DPLL(F, ρ ∪ L^1) ∨ DPLL(F, ρ ∪ L^0)
```

It has to be noted that there are several ways to implement DPLL.
Let's consider the formula:

$$F = (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1)$$

We can apply the DPLL algorithm to check if F is satisfiable. We initialize the procedure with F as a set of clauses and $\rho := \emptyset$. DPLL starts by running **UnitPropagate** as we already have a unit clause namely x_1^0. We can follow the values of F and ρ in the following table:

Operation	F	ρ
Initialization	$\{x_2^0 \vee x_3^0 \vee x_4^0,\ x_1^1 \vee x_2^1,\ x_3^0 \vee x_4^1,\ x_1^0\}$	$\emptyset$
UnitPropagate	$\{x_2^0 \vee x_3^0 \vee x_4^0,\ x_2^1,\ x_3^0 \vee x_4^1\}$	$\{x_1^0\}$
UnitPropagate	$\{x_3^0 \vee x_4^0,\ x_3^0 \vee x_4^1\}$	$\{x_1^0,\ x_2^1\}$
Decision x_3^1	$\{x_3^0 \vee x_4^0,\ x_3^0 \vee x_4^1\}$	$\{x_1^0,\ x_2^1,\ x_3^1\}$
UnitPropagate	$\{x_4^0,\ x_4^1\}$	$\{x_1^0,\ x_2^1,\ x_3^1\}$
UnitPropagate	$\{\emptyset\}$	$\{x_1^0,\ x_2^1,\ x_3^1,\ x_4^0\}$
Return false	$\{\emptyset\}$	$\{x_1^0,\ x_2^1,\ x_3^1,\ x_4^0\}$
Decision x_3^0	$\{x_3^0 \vee x_4^0,\ x_3^0 \vee x_4^1\}$	$\{x_1^0,\ x_2^1,\ x_3^0\}$
UnitPropagate	$\{\}$	$\{x_1^0,\ x_2^1,\ x_3^0\}$
Return true	$\{\}$	$\{x_1^0,\ x_2^1,\ x_3^0\}$

Table 1: Example execution of DPLL

Hence, the procedure returns true. We first see the decision x_3^1 leading to unsatisfying assignments. Then the other recursive call x_3^0 leads to a satisfying assignment. The satisfying assignment ρ is also returned as an *argument of satisfiability*. Running **RemoveComplements** on F before executing DPLL might reduce the number of clauses in turn reducing the running time of DPLL.

Proposition 2.1. *Given a formula F if there exists an assignment ρ such that $\models F|_\rho$ DPLL will return true for F.*

Proof. Assume that F is a satisfiable formula of n clauses. By definition 2.2 a satisfying assignment ρ must exist so that $\models F|_\rho$. Since $F = \wedge_{i=1}^n \phi_i$ then for any clause it holds that $\models \phi_i|_\rho$. DPLL will exit iff a satisfying assignment is found or all evaluations results in finding an empty clause in F. The only way for the empty clause to be found is if **RemoveNonSatVars** finds a clause such that $\not\models \phi_i|_\rho$. But when the algorithm is evaluating ρ no such clause exists. So DPLL will either exit when it finds some satisfying assignment ρ' or continue until exactly ρ is evaluated. True is returned in both cases. □

2.5 Conflict Driven Clause Learning

One issue with DPLL is that we might encounter cases where one or several truth assignments are unnecessary if a previous truth assignment is the reason for a conflict. Furthermore, some states might always lead to conflicting

assignments wasting computing time evaluating already known states.

$$F := G \cup \{x_1^1 \vee x_2^1 \vee x_3^1,\ x_1^1 \vee x_2^1 \vee x_3^0\}$$

In this example, assume G is any CNF formula. If $\rho \in \{x_1^0,\ x_2^0\}$ we will always reach conflicting unit clauses namely $\{x_3^1,\ x_3^0\}$. The objective is to tell the solver that $\{x_1^0,\ x_2^0\}$ cannot be assigned at the same time after the conflict is initially discovered. This can be achieved utilizing conflict analysis to search for the reason of a conflict, and in this case adding the clause $x_1^1 \vee x_2^1$ to F. This concept was initially proposed by J.P. Marques-Silva and Karem A. Sakallah [9] introducing the algorithm structure GRASP enabling non-chronological backtracking.

2.5.1 Resolution

Introducing the resolution proof system is beneficial in regard to conflict analysis [3]. Specifically, the resolution proof system is used to find clauses that prevent exploration of already known states. The resolution proof system works directly on clauses and can be used to construct a refutation, refuting unsatisfiable formulae as well as proving derived clauses. The version of the resolution proof system described is based on the one by Biere, Heule, van Maaren and Walsh [3]. The core rule (the resolution rule) of this system is:

$$\frac{B \vee x^1 \qquad C \vee x^0}{B \vee C}$$

B and C can be any formula in propositional logic. A resolution derivation $\pi = (D_1, D_2, ...D_L = D)$ of clause D consists of clauses from F (axioms) or clauses derived using the resolution rule. When finding a clause D_L through a resolution proof we write $\pi : F \vdash D$, the resolution derivation π of F leads to the clause D. Resolution proofs can be used to find a *resolution refutation* of F in the case where $\pi : F \vdash \bot$. We write $\text{Res}(B \vee x^0,\ C \vee x^1)$ when using the resolution rule and call $B \vee C$ the *resolvent* over x. Reaching $\bot$ can be achieved when resolving two unit clauses x^0 and x^1 since these are equivalent to $\bot \vee x^0$ and $\bot \vee x^1$. In this resolution refutation of the example CNF formula

$$F := (x_1^1 \vee x_2^0) \wedge (x_1^1 \vee x_2^1) \wedge (x_1^0 \vee x_3^1 \vee x_4^1) \wedge (x_1^0 \vee x_3^0) \wedge (x_1^0 \vee x_4^0)$$

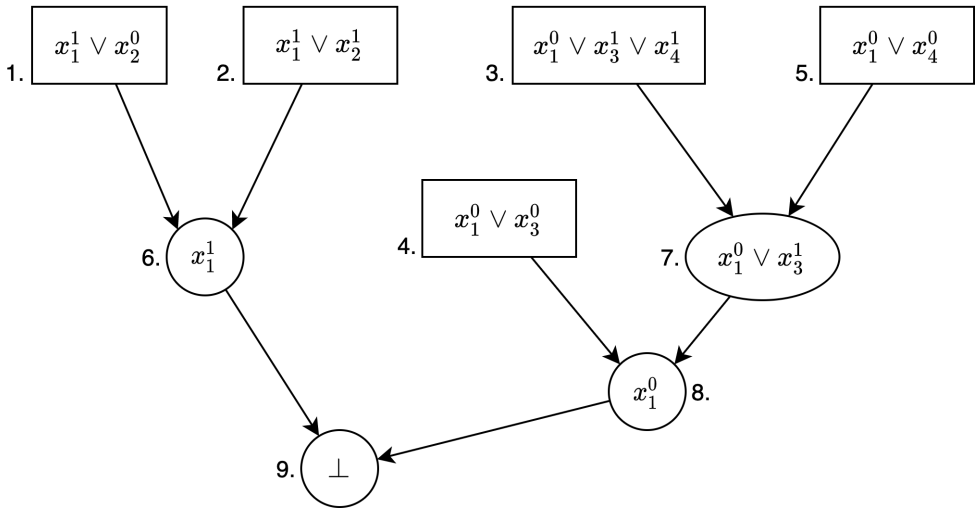

Figure 1: DAG representation of resolution refutation over F

We can represent the steps in a list as follows [3]:

1.	$x_1^1 \vee x_2^0$	Clause from F	
2.	$x_1^1 \vee x_2^1$	Clause from F	
3.	$x_1^0 \vee x_3^1 \vee x_4^1$	Clause from F	
4.	$x_1^0 \vee x_3^0$	Clause from F	
5.	$x_1^0 \vee x_4^0$	Clause from F	(2.6)
6.	x_1^1	Res(2, 1)	
7.	$x_1^0 \vee x_3^1$	Res(3, 5)	
8.	x_1^0	Res(7, 4)	
9.	$\perp$	Res(6, 8)	

For this example π are the clauses 1-9 refuting F. The proof can also be represented as a directed acyclic graph G_π with a vertex for each D_i of $1 \leq i \leq L$ clauses in π and edges showing how each clause is derived (figure 1). Each vertex is labeled with the corresponding line in the list representation of the proof. This visualization will be useful to later understand how a new clause is chosen and added to F when performing conflict analysis on a conflict found by the CDCL algorithm.

2.5.1.1 Trivial Resolution

When performing conflict analysis a restricted form of resolution proof can be utilized namely *trivial resolution*. Trivial resolution consists of two restrictions. First the resolution derivation π has to be *regular*.

Definition 2.3. *For $\pi = (D_1, D_2, ..., D_L = D)$ consider the DAG (proof graph) G_π. For each path from every leaf to the resolvent if no variable is resolved over multiple times the path is regular. If all such paths for G_π are regular, π is regular.*

Looking at figure 1, writing $S(var)$ where S is the label of the node and var is the variable resolved over, it will be clear that the graph is regular. If the node is a clause from F we write *Axiom* since no resolution has been made and we write *Resolvent* when the resolvent is reached.

$$
\begin{aligned}
&1.(Axiom) \rightarrow 6.(x_2) \rightarrow 9.(Resolvent) && Conclusion : Regular \\
&2.(Axiom) \rightarrow 6.(x_2) \rightarrow 9.(Resolvent) && Conclusion : Regular \\
&3.(Axiom) \rightarrow 7.(x_4) \rightarrow 8.(x_3) \rightarrow 9.(Resolvent) && Conclusion : Regular \\
&5.(Axiom) \rightarrow 7.(x_4) \rightarrow 8.(x_3) \rightarrow 9.(Resolvent) && Conclusion : Regular \\
&4.(Axiom) \rightarrow 8.(x_3) \rightarrow 9.(Resolvent) && Conclusion : Regular
\end{aligned}
$$

Since all the paths in π are regular, π is also regular. Secondly, π has to be *input*.

Definition 2.4. *For $\pi = (D_1, D_2, ..., D_L = D)$ if D_1 is a clause from F and for all $1 \leq i \leq \frac{L}{2}$ all D_{2i} are clauses from F and all D_{2i+1} are resolvents over D_{2i} and D_{2i-1} then π is input.*

So a proof of the input form will have the structure of starting by resolving over 2 clauses from F and then resolving over the found resolvent and another clause from F repeatedly until D is reached. The list representation 2.7 of our example shows that π is not of the input form.
Combining these we have the definition of a trivial resolution proof

Definition 2.5. *If π is regular and π is input, π is a trivial resolution proof.*

Let's look at an example of a trivial resolution refutation proof of:

$$F := (x_1^1 \vee x_2^1) \wedge (x_2^0 \vee x_3^1) \wedge x_3^0 \wedge x_1^0$$

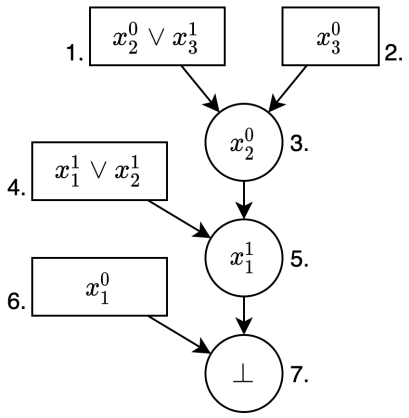

Figure 2: DAG representation of trivial resolution proof example

Where $\pi : F \vdash \bot$ and π in the list form is:

$$\begin{array}{lll} 1. & x_2^0 \vee x_3^1 & \text{Clause from } F \\ 2. & x_3^0 & \text{Clause from } F \\ 3. & x_2^0 & \text{Res(1, 2)} \\ 4. & x_1^1 \vee x_2^1 & \text{Clause from } F \\ 5. & x_1^1 & \text{Res(4,3)} \\ 6. & x_1^0 & \text{Clause from } F \\ 7. & \bot & \text{Res(5, 6)} \end{array} \tag{2.7}$$

Looking at the list it is apparent that π is of the input form and it can be seen that it is regular by looking at G_π (figure 2). Thus, π is indeed a trivial resolution refutation. Looking back at section 3, unit propagation is also able to refute the formula by the following steps:

initial state	$(x_1^1 \vee x_2^1) \wedge (x_2^0 \vee x_3^1) \wedge x_3^0 \wedge x_1^0$
Unit Propagate x_3^0	$(x_1^1 \vee x_2^1) \wedge x_2^0 \wedge x_1^0$
Unit Propagate x_2^0	$x_1^1 \wedge x_1^0$
Unit Propagate x_1^1	$\bot$

(2.8)

This connection is not coincidental since the two are closely related. To reach a refutation in a trivial resolution proof, F must contain at least one unit clause. As mentioned earlier, the only way to get the empty clause as a resolvent, is to resolve over two unit clauses of the same variable with opposite truth signs x_i and $\neg x_i$. The index form of π restricts D_{2i} to be an

axiom so for $\pi : F \vdash \bot$ a unit clause is needed on D_{L-1} and D_{L-2}. Since $D_L = \bot$ is the result of an application of the resolution rule, D_{L-1} has to be an axiom and a unit clause.

Proposition 2.2. *If F has a unit resolution refutation, then F has a trivial resolution refutation. If F has a trivial resolution refutation, then F has a unit resolution refutation.*

Proof. Assume that F is unsatisfiable and has a $\pi = (D_1, D_2, ..., D_L)$ such that $\pi : F \vdash \bot$. Then as argued, step D_{L-1} and D_{L-2} must be unit clauses and $D_{L-1} \in F$. Let's call the single literal in these clauses for x so $D_{L-1} = x^\sigma$ and $D_{L-2} = x^{1-\sigma}$. For D_{L-2} to be a unit clause there must exist a D_k consisting of exactly two literals x_k^σ and $x^{1-\sigma}$ for the resolvent to be a single literal. When we unit propagate on x^σ, x_k^σ will then become a unit clause. Inductively this will continue until F contains conflicting unit clauses. Since we know that F is unsatisfiable from the premise, unit resolution can't reduce F to the empty set. □

Building on proposition 2.2 and the properties of unit resolution and trivial resolution [3], we have $\pi : F \vdash C$ and $C = a_1^\sigma \vee a_2^\sigma \vee ... \vee a_k^\sigma$ iff F is refutable by unit resolution of $F \cup \{a_1^{1-\sigma}, a_2^{1-\sigma}, ..., a_k^{1-\sigma}\}$. This is the property used in conflict analysis in CDCL. When a conflict is found, a trivial resolution proof is constructed backwards through the clauses in F leading to the conflict, until a clause C with certain properties are found. The current truth assignment is falsified by C ensuring that the algorithm won't evaluate known unsatisfiable states multiple times. This is expanded on later.

2.5.2 Clause Learning

2.5.2.1 First unique implication point

Working on the intuition of CDCL, consider the following example CNF formula:

$$\begin{aligned} F := \{&x_1^1 \vee x_2^1 \vee x_3^1,\ x_1^1 \vee x_3^0,\ x_2^0 \vee x_7^0 \vee x_4^0, \\ &x_5^0 \vee x_4^1 \vee x_2^0,\ x_5^1 \vee x_6^0 \vee x_8^0,\ x_4^1 \vee x_5^1 \vee x_6^1 \vee x_7^0\} \end{aligned} \tag{2.9}$$

The formula is a state in execution of DPLL solving and the decision $x_1^1 \xleftarrow{dec} 1$ is made. At this state $\rho = \{x_1^1,\ x_7^1,\ x_8^1\}$. The following unit propagation will result in conflict after propagating $x_1^1,\ x_3^0,\ x_2^1,\ x_4^0,\ x_5^0,\ x_6^0$. To illustrate, we

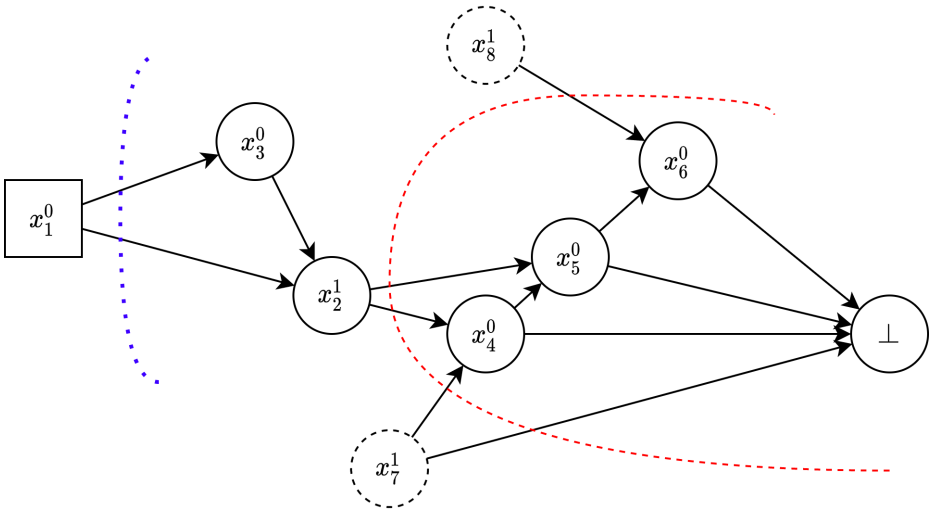

Figure 3: Conflict graph of equation 2.9

can construct a *conflict graph* (figure 3) of the process where each vertex is a unit assignment and edges are drawn from the literals contributing to propagating literal.

Definition 2.6. *For a variable $x \in \rho$ we define its decision level $lev(x)$ as the amount of decisions made at the stage of assignment.*

Let the most recent decision variable be x_{dec} and the current decision level be $lev(x_{dec})$. The circular nodes represent a unit assignment made at the current decision level, the square node is the decision variable and the dashed nodes are literals asserted in a prior decision level, e.g. asserted variable with $lev(x) < lev(x_{dec})$. Marked on the graph are two cuts. These mark the sections which are entered through unique implication points.

Definition 2.7. *A vertex in a conflict graph is a unique implication point (UIP) if path traversals from the decision node to the conflict node pass through that vertex.*

In figure 3 the two UIP's are x_1^0 and x_2^1. The decision variable is guaranteed to be a UIP by definition. The *first unique implication point* (1UIP) is the UIP closest to the conflict node, in this case x_2^1. The objective of the 1UIP learning scheme is to find a clause $C = a_1^\sigma \vee ... \vee a_k^\sigma$ where $C|_\rho$ is falsified and there is a unit resolution refutation of $F \cup \{a_1^{1-\sigma}, ..., a_k^{1-\sigma}\}$, where one of the

literals in C is the literal at the 1UIP. The fact that the decision variable always comes with a UIP ensures that such a C is always available during conflict analysis. So the clause C of k length has to consist of $k-1$ literals asserted outside of the current decision level and one literal inside the current decision level namely the 1UIP. There are several different ways of creating the C clause, for example by selecting the UIP differently, or selecting the cut with the minimum amount of intruding edges, hence minimizing the size of C. Zhang, Madigan, Moskewicz nad Malik [13] investigate the performance consequence of different clause learning schemes, and 1UIP performs very well.

2.5.2.2 Finding the 1UIP

To construct C with the described properties, we return to proposition 2.2 and the observation that if $F \cup \{a_1^{1-\sigma}, ..., a_k^{1-\sigma}\}$ is refutable by unit propagation then there is a π for which $\pi : F \vdash C$ holds. This is exactly the case when we falsify F by unit propagation where $\{a_1^{1-\sigma}, ..., a_k^{1-\sigma}\}$ are the literals outside of the current decision level contributing to reaching the conflict node in the graph and the 1UIP. In other words, we want to find all the literals pointing through our cut in figure 3 together with the 1UIP since we know that assigning all of these will lead to unsatisfiability. This objective can be reached by using the resolution rule to resolve repeatedly over the literals asserted in the current decision level until the resolvant adheres to the restrictions on C described above. Since the goal is to have the 1UIP as a literal in C, the first clauses resolved are the last two resolved by unit propagation. In essence, a trivial resolution proof of C is constructed backwards from the list of clauses causing assignments during unit propagation.

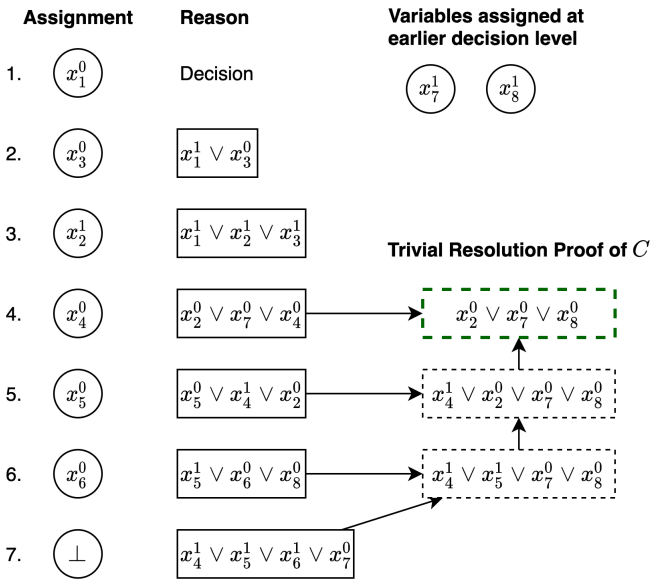

Figure 4: Conflict resolution of conflict in equation 2.9. The column on the left contains assignments made by unit propagation after the decision x_1^0. The reason column states the clause from which the assignment is derived. The trivial resolution proof should be read bottom up and proves C marked with green.

Figure 4 shows this process executed on equation 2.9. Starting from the bottom of the reason column, the resolution rule is used on every clause to construct the proof $\pi : F \vdash x_2^0 \vee x_7^0 \vee x_8^0$. The trivial resolution proof terminates here since x_2 is the 1UIP and $lev(x_7),\ lev(x_8) < lev(x_{dec})$. It also holds that $F \cup \{x_2^1,\ x_7^1,\ x_8^1\}$ is refutable by unit resolution since these propagations have already been evaluated in step 3-7 of assignments made.

2.5.3 Non-chronological backtracking

Consider the properties of the derived clause C. Let the most recent decision variable be denoted by x_{dec}. The variables in C all have an assignment level strictly lower than the level of the 1UIP, e.g. $lev(x) < lev(x_{dec})$ where $lev(x)$ is the level of any variable x in C other than the 1UIP literal. That means with clause C added, the 1UIP literal can be propagated at $BackjumpLevel = \max_{x \neq x_{dec}} lev(x)$. CDCL uses this fact to backtrack to $BackJumpLevel$ instead of just going to directly explore $x_{dec}^{1-\sigma}$.

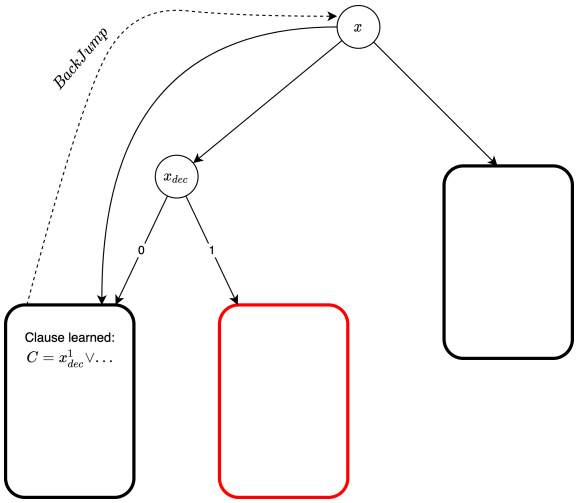

Figure 5: Illustration of backjump. The red section represents the skipped evaluation.

The intuition for this is to go back as far as we can while still using the observation that $x_{dec}^{1-\sigma}$ has to be set, instead of going down a potential long calculation to conclude if this assignment leads to unsatisfiability. This is illustrated in figure 5 where the red section represents the skipped evaluation. It has to be noted, as presented by Nadel and Ryvchin [10], that a CDCL solver with chronological backjump, e.g. going back to the decision variable, performs well on some benchmarks.

2.5.4 Restarting

During execution of CDCL in practice it is beneficial to restart the algorithm during evaluation of a problem. A restart involves setting $\rho \leftarrow \emptyset$ and choosing new decision variables for the next run. In other words, a *backjump* to assignment level 0 is performed. The benefit of a restart is that the algorithm benefits from learning derived clauses from one branch of the execution tree in the rest of the execution tree. The new clauses contribute to unit propagations not possible at the initialisation of the algorithm. The scheme for when a restart is performed can be very simple such as once every n seconds or conflicts. Different restart schemes have different benefits. An aggressive restarting scheme might be better for unsatisfiable formulae

since it enables more unit propagations at early decision levels [7]. A passive restarting scheme might be necessary for satisfiable problems since the algorithm needs to assign all variables in one run through.

2.5.5 Pseudocode for the CDCL algorithm

Now all the pieces can be brought together (algorithm 6) as follows. First, much like DPLL, unit propagation is executed for as long as possible. Then, if F is neither satisfied nor falsified, a decision variable is chosen and a decision is made. Notice that the recursive call made in DPLL is gone, since $backjump$ decides the next evaluated state. In an implementation, ρ should be constructed in a way such that the assertion level of each assignment is known. This will allow $backjump$ to set ρ to the correct state. The implementation differs from that in DPLL where F is directly modified and the assignments are tested recursively. The recursive step is no longer necessary since the added clause C will cause the decision variable to propagate immediately by design. Unit propagation in this implementation of CDCL is simpler than the DPLL counter part but methods for checking if $\models F|_\rho$ has to be implemented for the if statements to trigger.

Algorithm 6: CDCL

```
CDCL (F)
    L ← 0 // init decision level var
    ρ ← ∅ // init ρ
    while true do
        Unit Propagate updating ρ
        if ρ falsifies F and L = 0 then
            return unsat
        if ρ falsifies F then
            Do conflict analysis and add C to F
            // Backjump
            Find Backjump level L_back from ρ
            L ← L_back
            Remove all assignments in ρ with lev(x) ≤ L
            continue // With next iteration
        else
            if All variables ∈ ρ then
                Exit and return true and the assignment ρ
            L ← L + 1
            X ← a variable from F where ρ(X) = ∗
            ρ ← X^0 // Set decision var to false
            continue // With next iteration
```

3 Pseudo-Boolean Solving and 0-1 Integer Linear Programming

3.1 Pseudo-Boolean formulae

Now that a method of solving satisfiability problems is established, let's examine another class of problems, *integer linear programs* (ILP) and how to solve the variant *0-1 integer linear programs* or *pseudo-boolean formulas* with the same approach as CDCL.

Linear programming [5] is concerned with maximizing or minimizing an objective while satisfying of a series of linear constraints. The constraints manifest as either an equality or an inequality that a certain solution or optimization has to comply to. The nature of 0-1 integer linear programming (0-1ILP) is that the variables also are restricted to be 0 or 1. This is very similar to truth assignment which is what this paper has been concerned with so far. This is an example of such constraints:

$$2x_1 + 3x_2 - 4x_3 \geq 1$$
$$2x_3 - 3x_1 + 1x_2 \geq 2$$

This problem can be satisfied trivially in the setting of ILP with no 0-1 restrictions by setting $x_2 = 2$ and the other variables $x_1,\ x_3 = 0$. Under the 0-1 restriction however, there is no satisfying assignment. This becomes apparent when $x_3 \leftarrow 1$ and $x_1 \leftarrow 0$ to satisfy the second constraint. Substituting these necessities into the first constraint we get $0 + 3x_2 - 4 \geq 1$ which can not be satisfied. In the case of maximization and minimization an objective function is also provided. It is then the goal to find a set of variable assignments ρ with the respectively minimum or maximum value that also satisfies all constraints.

3.1.1 Normalized Form

Before going in to *cutting planes* we introduce *normalized form* of Pseudo-Boolean formulas as defined in the *Handbook of Satisfiability* [3]. Working on formulas in normalized form allow us to describe the rules of derivation in a simple fashion and introduce the term *slack*. Specifically, normalized

formulation concerns the structure of constraints which has to adhere to

$$\sum_{i\in[n],\ \sigma\in\{0,1\}} a_i^\sigma x_i^\sigma \geq A$$

Where all $a_i^\sigma \in \mathbb{N}^+$ and $A \in \mathbb{N}^+$ and the constraint has to be a *greater-than constraint*. A is referred to as *the degree of falsity* of the constraint. Note that if $A \leq 0$ the constraint is always satisfied in this form. To normalize any constraint, the following tools are needed:
If the constraint is a *lesser-than constraint*, we can multiply it with -1 to flip the inequality as seen in equation 3.1.

$$\begin{aligned} \sum_{i\in[n],\sigma\in\{0,1\}} a_i^\sigma x_i^\sigma \leq A \quad &\Leftrightarrow \\ \sum_{i\in[n],\sigma\in\{0,1\}} -a_i^\sigma x_i^\sigma \geq -A \quad &(3.1) \end{aligned}$$

If the constraint is an *equality constraint*, we split the constraint into two, one lesser-than and one greater-than constraint as seen in equation 3.2.

$$\begin{aligned} \sum_{i\in[n],\sigma\in\{0,1\}} a_i^\sigma x_i^\sigma = A \quad &\Leftrightarrow \\ \sum_{i\in[n],\sigma\in\{0,1\}} a_i^\sigma x_i^\sigma \geq A, \sum_{i\in[n],\sigma\in\{0,1\}} a_i^\sigma x_i^\sigma \leq A \quad &(3.2) \end{aligned}$$

And lastly, to change any coefficient from negative to positive, we use that

$$a^\sigma x^\sigma = a^\sigma (1 - x^{1-\sigma}) \quad (3.3)$$

This holds since any variable x_i^σ can only take 0-1 values and when $x_i^1 = 1$ then $x_i^0 = 0$ and vice versa. So to normalize any constraint, follow these steps:

1. If the constraint is an equality use equation 3.2 to split the constraint
2. If the constraint is a lesser-than constraint use 3.1 to factor by -1
3. For all coefficient variable pair $a_i^\sigma x_i^\sigma$ where $a_i^\sigma \leq -1$ rewrite using equation 3.3

An example of such a conversion of the following constraint follows

$$\begin{aligned} 3x_1 - x_2 - 5x_3 + 4x_4 &\leq -3 \\ 3x_1^1 - x_2^1 - 5x_3^1 + 4x_4^1 &\leq -3 \\ -3x_1^1 + x_2^1 + 5x_3^1 - 4x_4^1 &\geq 3 \\ -3(1 - x_1^0) + x_2^1 + 5x_3^1 - 4(1 - x_4^0) &\geq 3 \\ -3 + 3x_1^0 + x_2^1 + 5x_3^1 - 4 + 4x_4^0 &\geq 3 \\ 3x_1^0 + x_2^1 + 5x_3^1 + 4x_4^0 &\geq 10 \end{aligned} \tag{3.4}$$

3.1.2 CNF Problems as Pseudo-Boolean Formulae

CNF formulae as presented as input for the DPLL and CDCL algorithms, can be described as Pseudo-Boolean formulae. A CNF formula is a series of clauses consisting of literals for which at least one has to evaluate true for each clause. This is a special case of a Pseudo-Boolean model where each constraint corresponds to one clause of the CNF formula. A CNF formula F consisting of n clauses ϕ_n is equivalent to a set of Pseudo Boolean constraints as follows:

$$F = \bigwedge_{i=1}^{n} (x_1^\sigma \vee ... \vee x_{k_i}^\sigma) \quad \text{as PB constraints:} \quad \bigcup_{i=1}^{n} \{x_1^\sigma + ... + x_{k_i}^\sigma \geq 1\}$$

The benefit of formulating problems in PB format comes with conciseness and an improved potential of performing calculations over the set of constraints. Pseudo-Boolean Formulae can be formulated in CNF, but the size of the resulting CNF formula can be exponentially larger [3]. For instance the single Pseudo-Boolean constraint

$$-x_1 + x_2 - x_3 + 2x_4 + x_5 \geq 1$$

corresponds to the following 7 CNF clauses

$$\begin{aligned} &(x_1^0 \vee x_2^1 \vee x_4^1) \wedge (x_1^0 \vee x_3^0 \vee x_4^1) \wedge (x_1^0 \vee x_5^1 \vee x_4^1) \\ \wedge &(x_2^1 \vee x_3^0 \vee x_4^1) \wedge (x_2^1 \vee x_5^1 \vee x_4^1) \wedge (x_3^0 \vee x_5^1 \vee x_4^1) \\ \wedge &(x_1^0 \vee x_2^1 \vee x_3^0 \vee x_5^1) \end{aligned}$$

Constraints where all $a_i^\sigma \in \{0, 1\}$ are called *cardinality constraints*. Cardinality constraints dictate that A literals of the constraint has to be 1. A CNF

clause can therefore be expressed as a a cardinality constraint where $A = 1$. There are other methods of encoding Pseudo-Boolean formulae in CNF, but this specific case illustrates the conciseness of Pseudo-Boolean formulation. New variable introduction, substitution and other encodings such as the one proposed by Bailleux, Boufkhad, and Roussel [2] can support encoding to CNF in a more efficient way, though it either still has worst case exponential size consequences or in the case of substitution comes with an overhead of defining new variables.

3.1.3 Solutions to Pseudo-Boolean Formulae

To visualize the solution space of a Pseudo-Boolean formula consider this model:

$$\begin{aligned} 2x_1 + 2x_2 &\geq 1 \\ -2x_1 + 5x_2 &\geq -3 \\ x_1 - 4x_2 &\geq -9 \\ -6x_1 - 3x_2 &\geq -17 \\ 3x_1 - x_2 &\geq 0 \end{aligned} \tag{3.5}$$

To satisfy this system with a Pseudo-Boolean solution, x_1 and x_2 has to comply with all the constraints and $x_i \in \{0,\ 1\}$. Since there are just 2 variables in this example, each constraint can be plotted as a linear function in a coordinate system as seen in figure 6. The solution space is highlighted by marking the area where all the linear constraints are satisfied. For this example there are 2 solutions under the 0-1 constraint at the coordinates (1,0) and (1,1) (marked with red dots). This visualization can be used to understand how the solution space corresponds to a polytope with potential 0-1 integer solutions for any number of variables. The polytope can have no 0-1 integer points or be infeasible, in which case the system is unsatisfiable. To check if a solution space has 0-1 solutions, the polytope has to be *cut* until a 0-1 integer solution is in a corner of the polytope. This is called *cutting planes*.

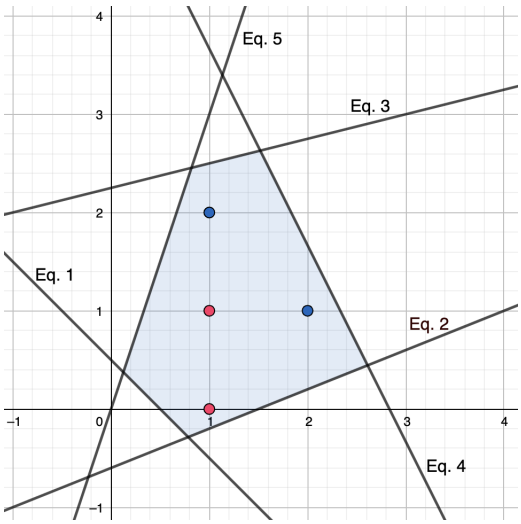

Figure 6: Plot of the constraints in equation 3.5. The first axis is x_1 and the second axis is x_2. The blue section is the solution space, the blue dots mark integer solutions and the red dots mark 0-1 integer solutions. Each constraint is labelled "$Eq.\ i$" for $0 \leq i \leq m$ constraints.

3.2 Cutting Planes

3.2.1 Derivation Rules of Cutting Planes

Before covering rules specific to integer linear programs, there are 3 simple rules that can be used to manipulate Pseudo-Boolean formulas without changing the solution space. These rules are defined as Biere, Heule, van Maaren and Walsh [3] present them. The first rule of reasoning is:

$$\text{Literal Axioms:} \quad \frac{}{x_i^\sigma \geq 0} \quad \text{and} \quad \frac{}{x_i^\sigma \leq 1}$$

Which means that any 0-1 variable can be introduced as either larger than 0 or smaller than 1.

$$\text{Multiplication:} \quad \frac{\sum_i a_i^\sigma x_i^\sigma \geq A}{\sum_i c a_i^\sigma x_i^\sigma \geq cA} \text{ for } c \in \mathbb{N}^+$$

The multiplication rule is extended to include a method of changing a less-than constraint to a greater-than constraint. This rule is needed as previously mentioned to reformulate constraints to normalized form.

$$\text{Multiplication (negative):} \quad \frac{\sum_i a_i^\sigma x_i^\sigma \leq A}{\sum_i k a_i^\sigma x_i^\sigma \geq kA} \text{ for } k \in \mathbb{Z}^-$$

The rule is sound since $\sum_i a_i^\sigma x_i^\sigma$ and k are simply integer values and for any real numbers a and b it holds that $a \leq b \Leftrightarrow -a \geq -b$.

3.2.2 The Chvátal-Gormory Cut

The fundamental rule for cutting real solutions from the solution space is division (Also called Cvátal-Gomory cut rule [3]).:

$$\text{Division:} \quad \frac{\sum_i c a_i^\sigma x_i^\sigma \geq A}{\sum_i a_i^\sigma x_i^\sigma \geq \lceil A/c \rceil} \text{ for } c \in \mathbb{N}^+$$

This rule can alter the solution space, essentially adding a *cut* removing real solutions. The rule is sound since for any integer K it holds if $cK \geq A$ then $K \geq A/c$ and then $K \geq \lceil A/c \rceil$ follows. For example if $A/c = 2.3$ then K has to be at least $\lceil 2.3 \rceil = 3$ to satisfy the inequality. General division is an extension of the rule and states the following:

$$\text{General Division:} \quad \frac{\sum_i a_i^\sigma x_i^\sigma \geq A}{\sum_i \lceil a_i^\sigma / c \rceil \, x_i^\sigma \geq \lceil A/c \rceil} \text{ for } c \in \mathbb{N}^+$$

General division can be derived using the rule of axioms and division. Consider the constraint $\sum_i a_i^\sigma x_i^\sigma \geq A$ if we want to divide by a divisor not common between all a_i^σ, we can use the rule of axioms to construct new constraints $x_i^\sigma \geq 0$ and add them to the first constraint until all coefficient has the common divisor. This is General Division. Let us consider again example 3.5 and derive a couple of cuts using the division rule. For any constraint where the coefficients have a common denominator, divide by the greatest common denominator. For the first constraint in 3.5:

$$\begin{aligned} 2x_1 + 2x_2 &\geq 1 && \text{Division by 2} \\ x_1 + x_2 &\geq 1 \end{aligned} \tag{3.6}$$

For the 4th constraint 3.5:

$$\begin{aligned} -6x_1 - 3x_2 &\geq -17 && \text{Division by 3} \\ -(6/3)x_1 - (3/3)x_2 &\geq \lceil -17/3 \rceil \\ -2x_1 - x_2 &\geq -5 \end{aligned} \tag{3.7}$$

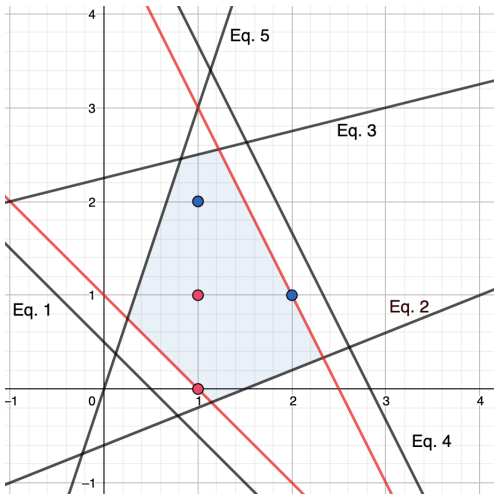

Figure 7: The Chvátal-Gomory cuts 3.6 and 3.7

These derived constraints are plotted in figure 7 and we can see the reduced solution space. If it is possible to get the inequality $0 \geq 1$ using only the rules reasoning listed here, no 0-1 integer solution exist, since division doesn't remove 0-1 integer solutions.

3.3 Conflict Driven Solving of PB Formulae

To apply the CDCL procedure to a Pseudo-Boolean formula some elements are needed, firstly a way to determine conflicts, secondly a way to unit propagate and thirdly a way to perform resolution to learn new constraints from conflicts. The purpose of this section is to introduce conflict driven Pseudo-Boolean solving with the objective of understanding the solving process well enough to see why presolving can be beneficial. We will be focusing on Conflict Driven Solving based on division instead of the saturation rule as this is how RoundingSat is implemented [6]. RoundingSat is the Solver used to conduct the experiments.

3.3.1 Recognizing Conflicts

To determine when a constraint is violated, the term *slack* is introduced. To utilize Slack, the coefficients in a constraint have to be positive. Therefore, normalized form is assumed for all constraints, and if the constraint is not in

normalized form, the conversion mentioned above, is executed before calculating slack. Slack is a measure of how far a constraint is from being falsified by ρ. For a constraint C of n literals and a truth assignment ρ, the slack is defined as such:

$$slack(C, \rho) = \left(\sum_{i \in [n],\ \sigma \in \{0,1\},\ \rho(x_i^1) \neq 0} a_i^\sigma \right) - A \tag{3.8}$$

Another way of describing slack is that it's the sum of the coefficients from all true and unassigned literals minus the degree of falsity. We call $slack(C, \rho)$ the slack of the constraint C under the assignment ρ. Since the coefficients are positive then if the slack of a constraint $slack(C, \rho) \leq 0$ the truth assignment ρ falsifies C. This fact can be used to *propagate* variables if their coefficient is larger than the slack.

Propagation in conflict driven Pseudo-Boolean solving:

- If a variable x_i^σ has a coefficient $a_i^\sigma > slack(C, \rho)$ add $\rho \leftarrow x_i^\sigma$

Thus, we have the first ingredient of conflict driven solving of Pseudo Boolean formulas; *propagation*. As an example let's see how propagation can be used on constraint 3.4 from section 3.1.1. We start by calculating the slack for the constraint where no variables are assigned in ρ.

$$slack(3x_1^0 + x_2^1 + 5x_3^1 + 4x_4^0 \geq 10,\ \emptyset) = (3 + 1 + 5 + 4) - 10 = 3$$

So this means that we cannot falsify a variable with a coefficient greater than 3 without falsifying the constraint resulting in conflict. Therefore, we can immediately propagate $\rho \leftarrow x_3^1$ and $\rho \leftarrow x_4^0$. Note that this doesn't change the slack of the constraint. If however $\rho(x_4^0) = 0$ when the propagation step is executed, we get the slack

$$slack(3x_1^0 + x_2^1 + 5x_3^1 + 4x_4^0 \geq 10,\ \rho = \{x_4^1\}) = (3 + 1 + 5) - 10 = -1$$

This is a conflict and thus under this assignment there is no way to satisfy the constraint.

3.3.2 Conflict Analysis

When a conflict is found, much like CDCL, conflict analysis is performed. The difference is, that the variables we can resolve over have coefficients and

that we need to be sure that only integer solutions are considered. Let's start by introducing the resolution rule for Pseudo-Boolean instances:

$$\frac{a_j x_j^\sigma + \sum_{i \neq j} a_i^\sigma x_i^\sigma \geq A \qquad b_j x_j^{1-\sigma} + \sum_{i \neq j} b_i^\sigma x_i^\sigma \geq B}{(\sum_{i \neq j} a_i^\sigma + b_i^\sigma) x_i^\sigma \geq A + B - a_j} \text{ for } a_i^\sigma = b_j^\sigma$$

Which can be derived ny adding two constraints with variables x_i^σ and $x_i^{1-\sigma}$ as well as same coefficients. To get two constraints to have the same coefficient on the variable x_j we want to resolve over, we find the greatest common denominator of the two coefficients of x_j and call it $d = gcd(a_i, b_j)$. Then we multiply the first constraint with $k_A = d/b_j$ and the second constraint by $k_B = d/a_j$. By example:

$$\begin{aligned}
6x_1^1 + 3x_2^1 + 5x_3^1 &\geq 6 && \text{Constraint } A \\
9x_1^0 + 2x_2^1 + 2x_4^1 &\geq 4 && \text{Constraint } B \\
gcd(6, 9) &= 3 && \text{Enable resolve over } x_1 \\
18x_1^1 + 9x_2^1 + 15x_3^1 &\geq 18 && A * (9/3) \\
18x_1^0 + 4x_2^1 + 4x_4^1 &\geq 8 && B * (6/3) \\
(9 + 4)x_2^1 + 15x_3^1 + 4x_4^1 &\geq 18 + 8 - 18 && \text{Resolving over } x_1 \\
13x_2^1 + 15x_3^1 + 4x_4^1 &\geq 8 && (3.9)
\end{aligned}$$

To resolve over a variable x_i with constraint $a_j x_j^\sigma + \sum_{i \neq j} a_i^\sigma x_i^\sigma \geq A$ and $b_j x_j^{1-\sigma} + \sum_{i \neq j} b_i^\sigma x_i^\sigma \geq B$ where $a_j \neq b_j$:

- Find $gcd(a_j, b_j)$
- Multiply constraint A with k_A and constraint B with k_B
- Use the resolution rule (add the constraints together)

Now that we have a way of resolving constraints the approach is very similar to the CDCL conflict analysis. Resolve the conflicting constraint with the reason for the conflict. The issue we can run into is that the constraint derived from conflict analysis is not falsified by ρ. This is essential, as described in the CDCL section, since the new constraint has to propagate at the decision level we jump back to. For example consider the following constraints:

$$2x_1^1 + x_2^1 + 2x_3^0 \geq 2 \tag{3.10}$$
$$2x_1^0 + x_4^1 \geq 2 \tag{3.11}$$

After $\rho \leftarrow x_3^1$ the first constraint propagates $\rho \leftarrow x_1^1$ and the slack of the second constraint is -1. When we resolve over the conflicting constraint 3.11 and the reason 3.10 we get:

$$\frac{2x_1^1 + x_2^1 + 2x_3^0 \geq 2 \qquad 2x_1^0 + x_4^1 \geq 2}{x_2^1 + 2x_3^0 + x_4^1 \geq 2}$$

Which isn't falsified under ρ and doesn't propagate. To solve this issue, we need to utilize cutting planes (division) and weakening.

$$\text{Weakening:} \quad \frac{\sum_i a_i^\sigma x_i^\sigma \geq A}{\sum_{i \neq j} a_i^\sigma x_i^\sigma \geq A - \max(a_j^1,\ a_j^0)}$$

When a constraint is weakened the greatest contribution of a variable is assumed to be valid, thus not changing the slack. Weakening can be derived by using the axiom rule to infer the constraint $x_j^{1-\sigma} \geq 0$, multiplying with a_j^σ and adding the axiom constraint to the constraint we wish to weaken. Using the method proposed in *Handbook of Satisfiability* [3] the following steps can be iterated over until we have a derived constraint which is falsified by ρ:

- Let the variable responsible of falsifying the constraint C_{confl} under ρ be called x_{confl}^σ. Then weaken the constraint C_{reason} on a non-falsified literal l' which has a coefficient $a_{l'}$ not divisable by $a_{confl}^{1-\sigma}$. This gets $\texttt{weaken}(C_{reason}, l')$
- Divide this by a_{confl}^σ and obtain $\texttt{divide}(\texttt{weaken}(C_{reason}, l'), a_{confl}^\sigma)$.
- Resolve the conflict constraint over this modified constraint, getting $\texttt{resolve}(C_{confl}, \texttt{divide}(\texttt{weaken}(C_{reason}, l'), a_{confl}^\sigma))$.

The intuition for this method is to remove all non 0-1 solutions from the solution space before inferring a new constraint. The proof of correctness for this method can be found in [3], and will not be covered here.

If we apply this method to the above example, we get:

$$\begin{aligned}
\rho &= \{x_3^1, x_1^1\} \\
C_{confl} &= (2x_1^0 + x_4^1 \geq 2) \\
C_{reason} &= (2x_1^1 + x_2^1 + 2x_3^0 \geq 2) \\
a_{confl}^{\sigma} x_{confl}^{\sigma} &= 2x_1^1 \\
l' &= x_2^1 \\
\texttt{weaken}(C_{reason}, l') &= (2x_1^1 + 2x_3^0 \geq 1) \\
\texttt{divide}(\texttt{weaken}(C_{reason}, l'), a_{confl}^{\sigma}) &= (x_1^1 + 2x_2^0 \geq 1) \\
\texttt{resolve}(\texttt{divide}(\texttt{weaken}(C_{reason}, l'), a_{confl}^{\sigma})) &= (x_4^1 + 2x_3^0 \geq 2)
\end{aligned}$$

Which is falsified under ρ.

3.4 Optimization Problems

In many cases the objective of solving Pseudo-Boolean instances is to obtain an optimal solution in respect to an objective function. The goal is to either minimize or maximize the value calculated by inserting a solution in the objective. An objective function looks like this:

$$max(\sum_i a_i l_i) \quad \text{or} \quad min(\sum_i a_i l_i)$$

Where a_i is the coefficient of the literal l_i. If we insert a satisfying assignment ρ we call the value of the objective function for the *objective value* A_ρ.

$$A_\rho = \sum_i a_i(\rho(l_i))$$

The solving procedure for optimization problems is almost identical to the one for PB satisfiability problems. But instead of returning when a satisfying assignment is found, the solution is used to construct a new constraint

$$\sum_i a_i l_i \leq A_\rho - 1$$

if it is a minimization objective or

$$\sum_i a_i l_i \geq A_\rho + 1$$

if it is a maximization objective. Then the satisfying assignment is saved ρ and we run the solver again. Consider the following example

$$\begin{aligned} & \rho = \{x_1^0, x_2^0, x_3^1, x_4^1\} \\ \text{Obj function:} \quad & min(2x_1^0 + x_2^0 - x_3^1 + 3x_4^0) \\ & A_\rho = 2(1) + 1(1) - 1(1) + 3(0) = 2 \\ \text{New constraint introduced:} \quad & 2x_1^0 + x_2^0 - x_3^1 + 3x_4^0 \leq 1 \end{aligned}$$

When the solver reaches `unsat` the last satisfying assignment ρ is returned, thus returning the optimized solution. This method is called *linear search*. Experiments using binary search to find the objective value has been attempted, but linear search seems to be the fastest solution in most cases [11].

3.5 Pseudo Boolean Solving Procedure

On the basis of propagation and conflict analysis, the following procedure is put forward to solve Pseudo Boolean instances.

- Propagate until either a conflict or no propagations are possible.
- If the problem is satisfied, return `sat` or in the case of an optimization problem, save ρ and add the new constraint and run the solver again.
- If there is no conflict, make a decision on an unassigned variable.
- If there is a conflict and decision level is 0, return `unsat`. Else do conflict analysis and add derived constraint to the set of constraints.

3.6 Presolving Methods

Presolving for mixed integer programs (MIP) is a series of transformations and reductions aiming to reduce the amount of redundant information and restructuring problems to be easier to solve [1]. In some cases for MIP problems presolving increase the solving speeds by magnitudes and can be the difference of solving a problem and timing out. In the paper *Presolve Reductions in Mixed Integer Programming* [1], an experiment on 3047 models showed that 504 instances couldn't be solved within a given time limit without utilizing presolving. Although, presolving in many cases is beneficial, in

16 of the 3047 test cases, presolving actually increased the solve time and made them intractable.
The presolvers utilized in this paper are supplied by the presolving library PaPILO [14]. PaPILO is built for MIP, and it therefore follows that several of the presolving methods are designed to optimize with continuous and non-binary integer variables in mind. However, a Pseudo-Boolean instance can be seen as a subproblem to a MIP and since PaPILO is sound for all MIP problems, it follows that it is sound for Pseudo-Boolean instances. All of the presolving methods are described in *Presolve Reductions in Mixed Integer Programming* [1], but several of the methods are not relevant to PB solving, since the focus is on replacing general integer or real variables with restricted variables.

3.6.1 Probing

The intuition for probing is to bound variables depending on a binary variable. This is very effective for MIP but since all bounds are predetermined to be 0-1 for PB instances, stronger bounds can not be derived without fixing a variable to a 0-1 value. However, fixing a variable can be immensely valuable if it can be done cheaply, since it halves the amount of possible solutions. The application of probing on PB instances is therefore to attempt to fix variables to a each other, before solving is initiated. Furthermore, *simple probing* attempts to substitute dependent variables. Consider the two constraints:

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 2 \\ x_1 - x_2 &= 0 \end{aligned}$$

As we can see from the second constraint if $x_1 = 1 \rightarrow x_2 = 1$ and $x_1 = 0 \rightarrow x_2 = 0$. Therefore, x_2 is dependant on x_1 and x_2 can be substituted for x_1 in the first constraint.

3.6.2 Non-Zero Cancellation

Non-Zero Cancellation (referred to as *sparsify* in PaPILO) concerns finding constraints with parallel variables and subtracting one constraint from another to reduce the amount of non-zero entries. Non-Zero entries have a great effect on many subroutines in MIP solvers [1], so we will investigate the effect on conflict driven PB solving. Non-Zero entries are the amount of variables for which $a_i \neq 0$.

3.6.3 Conflict Analysis as a Presolve Reduction

Conflict analysis (referred to as *propagation* in PaPILO) can also be used to derive conflict constraints from infeasible subproblems even before solving is initiated [1]. Conflict analysis is used to transform an initial set of conflict constraints in to a single conflict constraint. In MIP, this presolve reduction provides a 5% speed-up in the "$\geq$ 10 sec" bracket in the experiments conducted by Achterberg et al.[1]. Since conflict analysis is one of the key elements in conflict driven Pseudo-Boolean solving, it might be very beneficial to have these reductions calculated independently of a specific truth assignment ρ. Allowing for conflicts to occur earlier in the propagation graph might reduce the depth of the tree significantly, resulting in faster solve speed.

4 Implementation

This section will cover the implementation of a presolve step using the mixed integer linear presolver PaPILO [14] to optimize Pseudo-Boolean formulae before solving them with the solver RoundingSat [15]. The project is called `PresolveOpb`. The Opb Presolver is built to handle benchmark cases from the 2016 Pseudo-Boolean competition [12], and is designed to accommodate the requirements set for these solvers. The project is located at this git repository:

https://github.com/AKjeld/PresolveOpb

`PresolveOpb` has the following features:

- Presolving any Pseudo-Boolean instance in `.opb` file format.
- Utilizing state of the art presolving methods provided by the PaPILO library.
- Runs straight from the `./PresolveOpb` binary.
- Allows for changing settings such as time limit and enabling/disabling various presolve reductions.
- Provides postsolving to retrieve the solution from the presolved instance in the original problem space.

4.1 Requirements

The goal of the program is to read a problem instance in the `.opb` file format and apply presolving methods from the PaPILO presolver, while still conserving the constraints of a Pseudo-Boolean problem. That is, conserving the 0-1 integer constraints across all variables. The program should supply the presolved instance in the same format as the original as well as a postsolve instance. Furthermore, the program should have a postsolve feature, where the user supplies a solution file along with the postsolve instance, to get the solution in the original problem space.

4.2 Pseudo-Boolean Instances

Some limitations apply to the file format of the `.opb` instances which affect the structure of the problems compared to normalized form. It is necessary to have a conversion from the output of the PaPILO presolver to the Pseudo-Boolean instance format. This description of Pseudo-Boolean instances is based on the definition from *Input/Output Format and Solver Requirements for the Competitions of Pseudo-Boolean Solvers* [12].
We name the variables in a constraint $\{x_0, x_2, x_3, ..., x_n\}$ then any Pseudo Boolean constraint can be reformulated to adhere the following:

$$\sum_{i\in[n]} a_i^1 x_i^1 \geq A \quad \text{or} \quad \sum_{i\in[n]} a_i^1 x_i^1 = A \tag{4.1}$$

where a_i^1 denotes the coefficient for x_i^1. Since we're working on Pseudo-Boolean Constraints, $x_i^1 \in \{0, 1\}$, $A \in \mathbb{Z}$ and $a_i^1 \in \mathbb{Z}$. The `.opb` format dictates that all constraints are written using only x_i^1. To rewrite a negated variable we use equation 3.3, and if the constraint is in `.opb` format we write x_i^1 interchangeably with x_i To rewrite a constraint

$$\sum_{i\in[n],\sigma\in\{0,1\}} a_i^\sigma x_i^\sigma \leq A \tag{4.2}$$

to the desired formulation, we can factor the constraint with -1. This transformation is used to change less than constraints yielded by the presolve step. By example:

$$\begin{aligned}
3x_1^1 + 2x_2^0 - 3x_3^1 &\leq 4 && \Leftrightarrow \\
-3x_1^1 - 2x_2^0 + 3x_3^1 &\geq -4 && \Leftrightarrow \\
-3x_1^1 - 2(1 - x_2^1) + 3x_3^1 &\geq -4 && \Leftrightarrow \\
-3x_1^1 - 2 + 2x_2^1 + 3x_3^1 &\geq -4 && \Leftrightarrow \\
-3x_1^1 + 2x_2^1 + 3x_3^1 &\geq -2
\end{aligned}$$

So to rewrite a formula to the `.opb` format the following steps are followed:

1. If the constraint is in the form of equation 4.2 factor by -1
2. For all coefficient variable pairs $a_i^\sigma x_i^\sigma$ where $\sigma = 0$
   rewrite to $a_i^\sigma - a_i^\sigma x_i^{1-\sigma}$ and subtract a_i^σ from each side of the constraint

```
* #variable= 4 #constraint= 2
min: 2 x1 +2 x2 -1 x4 ;
1 x1 -2 x2 +1 x3 +2 x4 >= 2 ;
-2 x1 + 2 x2 = 2 ;
```

Figure 8: Example `.opb` file

A `.opb` file covers two different problems, namely, Pseudo-Boolean Optimization (PBO) and Pseudo-Boolean Satisfaction (PBS) [12]. They differ in that a PBO instance has a minimize objective function as the first non-comment line of the file. An example of a `.opb` PBO file can be seen in figure 8. Any line beginning with a `"*"` is a comment line, and the very first comment is a header including values for the number of variables and constraints in the problem.

4.3 Parsing

The first step of presolving is to parse the instance. The PaPILO presolver comes with a command-line interface designed to parse, presolve and save the presolved instance to a new file. However, PaPILO doesn't handle Pseudo-Boolean problem instances by default [14]. Therefore, `PresolveOpb` comes with a parser which reads the input file and utilizes the problem builder provided by PaPILO to hold the parsed constraints. Before loading the constraints, sufficient memory is allocated using information from the header of the instance file which contains information on the amount of variables and constraints. According to the documentation of the Pseudo-Boolean competition benchmarks, some problems might have very long constraints, thus the parser should avoid reading whole lines at once [12]. The problem parser in `PresolveOpb` reads the file using a tokenized input stream processing one coefficient or variable at a time to avoid this issue. The parser also sets all the bounds for the variables to 0-1 and loads the objective function if one is provided. PaPILO stores the problems in a matrix where each column represents a variable and each row represents a constraint. `PresolveOpb` creates a mapping of variables to columns and places values in the matrix accordingly. Table 2 shows the matrix retrieved from parsing the example in figure 8. The degree of falsity is defined by the row's upper and lower limit. If a limit is infinite (inf) the inequality is open in the corresponding direction.

Lower Limit	x_1	x_2	x_3	x_4	Upper Limit
2	1	-2	1	2	inf
2	-2	2			2

Table 2: Constraint matrix of `.opb` file in figure 8

4.4 Presolving

`PresolveOpb` now sets up PaPILO to use the settings provided in the parameter file, and calls the presolver from PaPILO, presolving the instance. The essential part is that PaPILO is forced to preserve the binary quality of the variables, so `PresolveOpb` disables the presolving methods that would substitute binary variables with general integer variables. After the presolving has been executed, `PresolveOpb` validates the output by checking that the name convention and the 0-1 restriction on the variables is conserved. This validation step is important in case a different version of PaPILO is used, since the default presolvers might be changed in future versions. The diagram (figure 9) shows the sequence of messages passed around when a user requests presolving of a test instance. The format of a parameter file along with the definitions of each parameter is described in the repository.

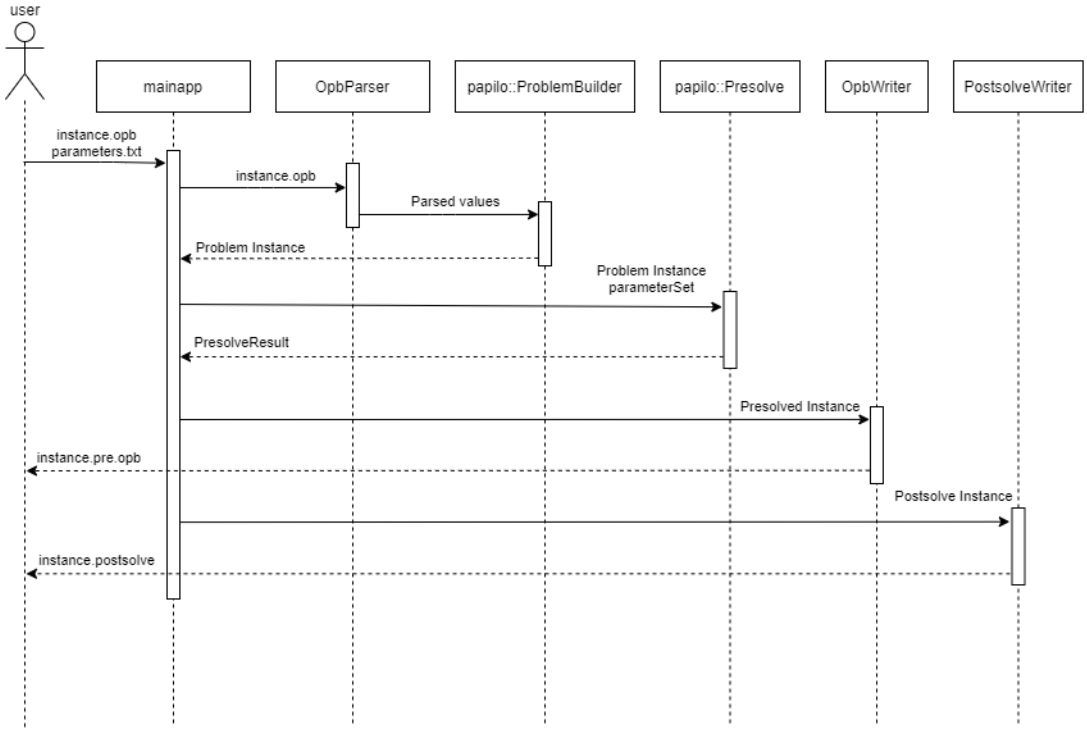

Figure 9: UML sequence diagram showing the presolve step

4.4.1 Retrieving the Presolved Instance

The last step is to retrieve the presolved instance. Since PaPILO does not deliver the constraints under the `.opb` file limitations, the conversion from Section 4.2 is applied to all constraints not adhering to this format. Then the constraints are written in the `.opb` format to the output file. If the instance has an optimization objective, a postsolve file is also written. After optimizing, this file can be used to postsolve a solution.

4.5 Post-solving

After solving the instance using any desired solver, the user may supply the solution along with the postsolve file to retrieve the solution in the original problem space. Presolving instances will result in columns in the constraint matrix to be removed, causing the presolved instance to have a different amount of variables than the original problem. The postsolve instance contains the information on variable mappings between the original and the presolved instance. `PresolveOpb` reads the solution from the presolved instance and applies the variable mapping to get the solution in the original problem space. The postsolve file is an instance of the PaPILO postsolve object, which comes with a method to achieve this. The solution is then verified to be feasible in the original problem space, before being supplied to the user.

5 Testing

5.1 Experiment setup

The Pseudo-Boolean solver used for the experiments is RoundingSat [15] on the git commit 67a39421 from 5th March 2021 (RoundingSat does not have a version label). RoundingSat does also support Linear Programming methods provided by SoPlex, but these were not used, since the goal of the experiments is to investigate the impact of presolving on Conflict Driven Pseudo-Boolean solving. The experiments were conducted on a 2.0 GHz Quad-Core Intel i5 processor with 16 GB RAM. The OS of the testing system is macOS Catalina v. 10.15.7. For each instance a baseline solve-time was obtained by running RoundingSat on the non-presolved instance referred to as `<instance>.opb` and compared to the solve-time of the presolved instance referred to as `<instance>.pre.opb`.

5.2 Test instances

Below are the different datasets described. The tests were run with different timeouts depending on the amount of instances (N) and the average solvetime. The names of the datasets are obtained from the PB competition folders.

Dataset name	Description	Timeout (s)	N
OPT-BIGINT-LIN	Optimization instances from PB16 competition	3600	226
d_n_k	Decision instances from PB16 competition	3600	84
nossum	Decision instances from PB16 competition	2400	150

Many of the test instances timed out even with a timeout of 7200 seconds. Therefore, due to limitations of the testing setup, some of the instances are excluded from the results since both the original and presolved instance were intractable to solve. Other instances, such as the dataset `sumineq` from the PB16 competition were solved very fast (< 10 seconds). These instances were also excluded since the fluctuations in solve time for both the presolved and original problems were minimal.

In the results, a *faster* time is defined as at least 10% improvement, and a *slower* time is at least 10% slower. The postsolve step takes negligible time to execute across all tests, and is therefore not included in the results.

5.3 Results

5.3.1 Nossum

Testing on these instances revealed that most of the instances were not solvable inside the timeout limit. 10 of the instances were ran with a timeout of 7200 seconds, but these were still not solvable. There were 11 problems that were solvable, and saw a change after presolving.

N	Solvable (pre)	Solvable (base)	Faster	Slower	Avg. time change	Total time change
150	11	10	5	6	-8.08%	-34.987%

Table 3: Results on the Nossum dataset

The column *Avg. time change* refers to the speedup seen for a single instance in relation to the original solve speed. The *Total time change* column refers to the total solve time of all solvable problems. It was possible to presolve all 150 instances with propagation as the most called presolve reduction. Propagation was the only successful presolve call in the 11 solvable instances.

avg. variables	avg. constraints	avg. variables (pre)	avg. constraints (pre)
4122	8630	3525	7428

Table 4: Nossum: Avg. instance size before and after presolving

5.3.2 Optimization instances

Bracket	Models	Faster	Slower	Avg. time change
$\leq$ 1s	96	16	64	+8780%
$\leq$ 10s	7	5	2	-11.00%
$\leq$ 3600s	20	13	5	-16.6%
Timeout	103			

Table 5: Results on the optimization dataset

Most of the optimization instances were solvable, though some of the instances were solved very fast. Due to the difference in solve time, the result table (table 5) is split in to 3 time brackets. The brackets are exclusive, for instance the second row contains all the models solved in the interval $1s < T \leq 10s$. The results in the fastest bracket show, that models solved almost immediately have very volatile solve times and do not benefit from a presolve step. But for most the models in the $\leq 3600s$ bracket, presolving seems to improve the speed significantly. The most called presolve step was propagation, and the tests yielded similar results when running only using propagation. Out of the 226 instances, 1 model was only tractable when presolved. In table 6 the total solve time of all solvable problems in the dataset is listed. The second row shows the time difference for all models excluding the unstable fast problems and the instance which timed out without presolving.

Models	n	Total solvetime (base)	Total solvetime (pre)
Solvable	123	2468s	1786s
No timeout and $> 1s$	26	1268s	733s

Table 6: Total time results on the optimization dataset

avg. variables	avg. constraints	avg. variables (pre)	avg. constraints (pre)
5677	16797	5127	14750

Table 7: Optimization instances: Avg. instance size before and after presolving

5.3.3 d_n_k Dataset

This dataset contains 19 models with a solve time close to 0.1 second, both with and without presolving. These are omitted from the table 8.

Bracket	Models	Faster	Slower	Avg. time change
$\leq$ 1s	15	5	1	-4,76%
$\leq$ 10s	19	4	4	-5.35%
$\leq$ 3600s	31	13	5	-3.4%

Table 8: Results on the d_n_k dataset

The total solvetime for the d_n_k dataset was 10026 s and with presolve it was 9516 s yielding an average of $\approx 5\%$ speed-up with presolving enabled. The most called presolve reduction was *dualfix*.

5.4 Evaluation

Looking at the results from the tests, presolving does certainly influence the structure of PB instances enough to affect the solve rate significantly. The results indicate that presolving "easy" instances result in volatile solve times. Especially the presolve reduction *propagation* reduces the number of variables and constraints for a majority of the instances. The tests on the Nossum dataset indicate that propagation contributes positively to solve speed, but it has to be noted that the amount of solvable instances was very limited.

Due to the limitations of the testing setup and difficulties obtaining suitable datasets, the tests are inconclusive. It is clear that the tests conducted in this paper have not been optimal. The large spread of solve times resulted in the majority of the instances to either be solved immediately or not be solved within the time limit. This rendered many of the Pseudo-Boolean competition instances unusable, resulting in a very limited amount of test instances. It is clear, that a better understanding of how RoundingSat handles specific cases is needed to obtain suitable test models, and in turn be able to analyse the effect of presolving accurately. Furthermore, a deeper analysis of how MIP Presolve reductions affect Pseudo-Boolean instances, would also be needed to get conclusive results.

Though the results aren't optimal, they do indicate a potential for PB solving. Especially probing seems to reduce the problem size significantly. In relation to solve speed, a presolve step seems to be most effective and consistent in the $\leq 3600s$ bracket. In general, executing the presolve step was very fast and had a negligible impact on solve time in all documented cases. Since this step is so fast to execute, it can also be utilized to gain information about a specific instance before solving.

The total solve time of the suitable instances from the datasets also indicate the benefit of utilizing MIP Presolve reductions in Pseudo-Boolean solving.

6 Conclusion

This paper has given an overview of DPLL, CDCL, conflict driven Pseudo-Boolean solving and a basic introduction to presolve methods used in Mixed Integer Programming. This overview lead to the documentation of `PresolveOpb`, a standalone binary supporting application of MIP Presolve reductions on Pseudo-Boolean instances. Utilizing PaPILO's powerful presolve library, `PresolveOpb` allows experimenting with state-of-the-art MIP Presolvers in the context of Pseudo-Boolean Solving.
The computational experiments indicate that a presolve step is beneficial for Pseudo-Boolean solving, and especially the presolve reduction *probing* performs well. Due to the limitations of this paper, the results are not conclusive, and it is clear that there are still many unanswered questions.
Hopefully, this paper leads to new discoveries and promotes further experimentation and wide application of presolve reductions in Pseudo-Boolean solving.

6.1 Future Work

The development of `PresolveOpb` contributes to the future work on Pseudo-Boolean solvers by enabling application of MIP presolve reductions on PB optimization and satisfiability models. Building on the findings on this paper, further experiments are needed to get conclusive results on the performance of a presolve step. RoundingSat has some integration with LP solver SoPlex to improve its search routine [15]. A natural next step is to test the presolve step with this integration enabled. This was not done here, due to the scope of the paper.

Experimenting with different data sets revealed that individual instances yielded vastly different results. Both in form of solve times and the reductions applied during presolving. Consequently, the drastic inconsistencies rendered many test instances unfit for testing, which impacted the extent of the experiments in this paper negatively. Solving larger test sets with a better performing experiment setup, will also be necessary to get consistent results.

Lastly, theoretical analysis of which presolve reductions are transferable from MIP to PB solving can aid future experimentation in this research field.

7 References

References

[1] Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, Dieter Weninger, *Presolve Reductions in Mixed Integer Programming*, Zuse Institute Berlin, 2016

[2] Bailleux, Olivier, Boufkhad, Yacine, and Roussel, Olivier. *A Translation of Pseudo-Boolean Constraints to SAT*, Journal on Satisfiability pp. 191-200, Boolean Modeling and Computation 2, 2006

[3] Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh, *Handbook of Satisfiability*. Chapter 7. IOS Press, 2020

[4] Davis and Putnam, *A computing procedure for quantification theory*, 7:201-215, Journal of ACM, 1960.

[5] George B. Dantzig, Mukund N. Thapa, *Linear Programming 1: introduction*, Springer Series in Operational Research, Springer-Verlag, 1997

[6] Jan Elffers, Jakob Nordström, *Divide and conquer: Towards faster pseudo-Boolean solving.* In Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18), pp. 1291–1299, July 2018.

[7] Martin Hořeňovský, *Modern SAT solvers: fast, neat and underused (part 3 of N)*, The Coding Nest, codingnest.com/modern-sat-solvers-fast-neat-and-underused-part-3-of-n/, 2019

[8] Michael Huth and Mark Ryan, *Logic in Computer Science*. Cambridge University Press, 2004

[9] J.P. Marques-Silva, Karem A. Sakallah. *GRASP: A Search Algorithm for Propositional Satisfiability* from IEEE Transactions on Computers. pp. 506–521. 1999

[10] Alexander Nadel and Vadim Ryvchin. *Chronological backtracking. In Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of Lecture Notes in Computer Science, pp. 111–121. 2018.

[11] Jakob Nordstrom, *Introduction to presolving in Pseudo-Boolean problems*, Personal communication, 2021

[12] Olivier Roussel, Vasco Manquiho, *Input/Output Format and Solver Requirements for the Competitions of Pseudo-Boolean Solvers*, Pseudo-Boolean Competition 2016 at cril.univ-artois.fr/PB16/

[13] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, Sharad Malik. *Efficient conflict driven learning in a boolean satisfiability solver*. Proc. IEEE/ACM Int. Conf. on Computer-aided design (ICCAD). pp. 279–285. 2001

[14] PaPILO repository: github.com/lgottwald/PaPILO

[15] RoundingSat repository: gitlab.com/miao_research/roundingsat

[16] SCIP Documentation, https://www.scipopt.org/doc/html/