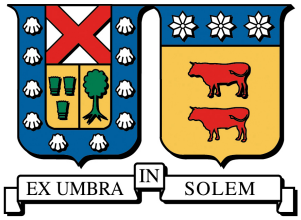
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso, Chile



Backbones in Pseudo-Boolean Optimization: Extraction and Analysis

Matías Francia-Carramiñana

THESIS FOR THE DEGREE OF MASTER IN COMPUTER SCIENCE

THESIS COMMITTEE:

Dr. Roberto Asín-Achá — Advisor, UTFSM
Dr. Ricardo Ñanculef — Co-Advisor, UTFSM
Dra. Elizabeth Montero — Internal Evaluator, UTFSM
Dr. Albert Oliveras — External Evaluator, UPC Barcelona

2026

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☐ Memoria o trabajo de título ☑ Tesis de Postgrado

Título del trabajo: Backbones in Pseudo-Boolean Optimization: Extraction and Analysis

Nombre del candidato(a): Matías Elian Francia Carramiñana

Carrera / Grado: Ingeniero Civil Informático

Campus: Casa Central Departamento: Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Roberto Asín Achá, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☑ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por **(marcar una opción)**:

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 16/03/2026 **Firma:** 

Estudiante o Candidato(a):

Fecha: 16/03/2026 **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

Acknowledgements

The research of the author was supported in part by the AI4OPT institute NSF award 2112533.

Abstract

Backbone, the set of variables that are fixed across all optimal solutions, captures key structural properties of combinatorial optimization problems. The backbone provides interpretable indicators of problem hardness and, as demonstrated by recent research in SAT, valuable supervision targets for learning-based methods. Yet, the Pseudo-Boolean Optimization (PBO) domain has lacked dedicated tools for its extraction. Here we introduce three new backbone extractors for Pseudo-Boolean Optimization (PBO): RoundingBack, a PBO-native extractor built on top of RoundingSAT; GuroBack, an extractor built on top of the MILP solver Gurobi; and NapBack, a pipeline that converts PBO instances into SAT using NaPS and delegates the backbone extraction to CadiBack, a backbone extractor for SAT. Additionally, we propose two heuristic variants of RoundingBack: RB-WP (weighted-propagation ordering) and RB-PG (diversity-driven phase guidance). Each extractor demonstrates distinct strengths tailored to different domain applications. On the PBO Competition 2024 OPT-Lin benchmark, RB-PG achieves the highest extraction coverage (223 of 335 instances, 67%), outperforming GuroBack and NapBack. Our experimental evaluation also shows that RoundingBack, GuroBack and NapBack are complementary, and it is often the case that when an extractor fails, another is fit. Beyond algorithmic performance, a large-scale analysis over more than eight thousand instances shows a bimodal distribution distinguishing flexible and rigid problem classes. We also show that the correlation between backbone density and problem hardness is domain-specific. All extracted backbones are publicly released to foster future research in learning-based PBO and structural instance analysis.

Keywords: Backbone, Pseudo-Boolean Optimization, Combinatorial Optimization.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Combinatorial optimization problems appear throughout science and engineering, from cryptanalysis [37] and software testing [4] to chemical process design [51] and autonomous navigation [3]. These problems share a common structure: selecting discrete configurations from exponentially large solution spaces to optimize some objective function while satisfying given constraints. Their computational difficulty, often NP-hard, has motivated decades of research into both exact and approximate solving techniques [58, 5].

Pseudo-Boolean Optimization (PBO) sits between Boolean Satisfiability (SAT) and Mixed-Integer Linear Programming (MILP). PBO extends SAT's propositional logic with linear objective functions and pseudo-Boolean constraints—linear inequalities over Boolean variables—allowing natural expression of optimization problems in circuit verification [57], resource allocation [47], and covering problems [15, 34]. PBO inherits efficient conflict-driven clause learning (CDCL) techniques from SAT [36, 39] while also using cutting planes and branch-and-bound methods from integer programming [33, 43].

Modern PBO solvers such as RoundingSAT [20, 17] and SCIP [2, 8] show that native handling of pseudo-Boolean constraints can outperform both pure SAT approaches (which require expensive encodings [19]) and generic MILP solvers on many problem classes. Recent advances in pseudo-Boolean propagation [42] and core-guided optimization [17] have strengthened the case for dedicated PBO techniques. The Pseudo-Boolean Competition continues to drive progress in this area; its 2024 edition provides the benchmark suite used in this work.

Beyond efficient solving algorithms, the *structural* properties of optimization problems help explain computational difficulty. An important structural concept is the *backbone*: the set of variables that take the same value across all optimal solutions. Schneider et al. [50] introduced backbones for the Traveling Salesman Problem, capturing a notion of structural rigidity or "frozen core" within a problem instance. Backbone variables represent decisions determined by the problem structure rather than incidental choices among equivalent alternatives.

Backbones have several uses. In SAT, backbone variables correlate with problem

hardness, especially near phase transitions [38, 44]. Large backbones indicate that many variable assignments are tightly constrained, often making instances harder to solve. Conversely, small backbones mean more flexibility in the solution space. This hardness correlation holds empirically across various SAT benchmark families [32]. For solver enhancement, Slaney and Walsh [52] showed that backbone information can guide search heuristics, while Wang et al. [56] trained neural networks to predict backbone variables and use these predictions to accelerate CDCL solving.

For optimization problems, backbones take on added importance. Slaney and Walsh [52] extended backbones from satisfiability to optimization, defining them as variables fixed across all optimal solutions. This requires first finding the optimal objective value, then identifying variables invariant within that optimal solution space. The resulting backbone provides a domain-independent characterization of problem structure that complements domain-specific features used for algorithm selection [53, 54] and instance space analysis.

Combining backbones with machine learning for combinatorial optimization is a promising direction. Learning-based methods for MILP—including learn-to-branch [31, 22], predict-and-search [26, 28], and contrastive learning approaches [27, 11]—show that structural features can guide optimization algorithms. Backbones are a natural target for such learning: they represent the "essential" part of optimal solutions that must be identified, while non-backbone variables offer flexibility for heuristics. This motivates the need for efficient backbone extraction methods that can generate training data at scale.

Despite the utility of backbones in SAT and their extension to optimization, a gap exists in the literature: *no dedicated backbone extraction methods have been developed for Pseudo-Boolean Optimization or Mixed-Integer Linear Programming.* Backbone extraction is fundamentally a SAT/CSP concept, and existing extractors such as CadiBack [6] target pure SAT instances. Applying them to PBO requires converting the PBO instance to SAT through potentially expensive encodings [19], which introduces overheads that may limit extraction coverage or efficiency.

We address this gap by developing native PBO backbone extraction methods. We introduce RoundingBack, a backbone extractor built on the RoundingSAT PBO solver, along with two heuristic variants: RB-WP (weighted-propagation ordering) and RB-PG (diversity-driven phase guidance). For comparison, we also implement GuroBack—a novel MILP-based backbone extractor using Gurobi, which is itself a contribution since no native MILP backbone extractors previously existed—and NapBack (a SAT-conversion pipeline using NaPS and CadiBack).

Our experimental evaluation on 8,351 instances from the PBO Competition 2024 OPT-Lin benchmark shows that the RB-PG variant achieves 67% overall extraction coverage, outperforming both baseline approaches. We also find complementarity patterns among extractors: different methods succeed on different instance families, suggesting that portfolio approaches may further improve coverage. We release the complete dataset of extracted backbones to support future research in PBO solving,

algorithm selection, and machine learning for combinatorial optimization.

1.1 Hypothesis

Implementing a native backbone extractor for PBO reduces extraction time compared to methods that require prior conversion of the problem to SAT. Additionally, variable ordering heuristics reduce computation time compared to the original algorithm without specific ordering.

1.2 Objectives

The general objective of this work is to develop and implement a backbone extraction mechanism for Pseudo-Boolean Optimization (PBO) instances, integrating it into the RoundingSAT solver, to generate representative and structurally informative data for training machine learning models and analyzing instance structure.

The specific objectives associated with this work are the following:

1. Design and implement the backbone extraction algorithm for PBO instances, adapting CadiBack's method to optimization problems with binary variables.
2. Extend the mechanism to support both optimization (with objective function) and decision problems (without objective function).
3. Develop and explore heuristics that speed up extraction, based on internal extractor modifications.
4. Implement a pipeline that converts PBO to SAT, extracts the SAT backbone, and converts back to PBO.
5. Evaluate heuristic effectiveness in reducing extraction time, comparing with the original PBO-adapted algorithm and the SAT conversion pipeline.
6. Report the contribution through a conference paper.

A preliminary version of this work was published at the 18th International Conference on Agents and Artificial Intelligence (ICAART 2026) [21]. The present thesis extends the conference paper with a broader background study and a more detailed literature review.

1.3 Structure

Chapter 2 provides background on Boolean Satisfiability, Conflict-Driven Clause Learning, Pseudo-Boolean Optimization, and backbones. Chapter 3 reviews related work on backbone extraction and PBO solving. Chapter 4 describes our proposed backbone extraction methods. Chapter 5 presents the experimental evaluation. Chapter 6 summarizes conclusions and discusses future research directions.

Chapter 2

Background

This chapter covers Boolean Satisfiability (Section 2.1), Conflict-Driven Clause Learning (Section 2.2), Pseudo-Boolean Optimization (Section 2.3), backbones (Section 2.4), backbone extraction algorithms (Section 2.5), and PBO solver paradigms (Section 2.6).

2.1 Boolean Satisfiability

Boolean Satisfiability (SAT) is the canonical NP-complete problem [7]. We present the propositional logic basics needed to formulate SAT.

2.1.1 Propositional Logic Fundamentals

Definition 2.1.1 (Propositional Variable)**.** A *propositional variable* (or Boolean variable) is a variable x that takes values from the Boolean domain $\mathbb{B} = \{0, 1\}$, where 0 represents *false* and 1 represents *true*. We denote a finite set of propositional variables as $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$.

Definition 2.1.2 (Literal)**.** A *literal* is either a propositional variable x (called a *positive literal*) or its negation $\neg x$ (called a *negative literal*). We use ℓ to denote a literal and $\bar{\ell}$ to denote its complement, i.e., if $\ell = x$ then $\bar{\ell} = \neg x$, and if $\ell = \neg x$ then $\bar{\ell} = x$.

Example 2.1.1. Consider variables x_1 and x_2. The set of literals over these variables is $\{x_1, \neg x_1, x_2, \neg x_2\}$. Here, x_1 and $\neg x_1$ are complements of each other.

2.1.2 Clauses and Conjunctive Normal Form

Definition 2.1.3 (Clause)**.** A *clause* C is a disjunction (logical OR) of literals:

$$C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k) \tag{2.1}$$

where ℓ_i denotes the i-th literal in the clause. The number of literals k in a clause is called its *width*. A clause with exactly k literals is called a *k-clause*.

Definition 2.1.4 (Conjunctive Normal Form)**.** A propositional formula φ is in *Conjunctive Normal Form* (CNF) if it is a conjunction (logical AND) of clauses:

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m = \bigwedge_{j=1}^{m} C_j \tag{2.2}$$

where each C_j is a clause. A CNF formula over n variables with m clauses is often characterized by its *clause-to-variable ratio* m/n.

Any propositional formula, including those with implications ($\rightarrow$) and biconditionals ($\leftrightarrow$), can be transformed to an equisatisfiable CNF formula using the Tseitin transformation [55]. The transformation introduces auxiliary variables to avoid exponential blow-up, producing a formula whose size is linear in the original.

Example 2.1.2. Consider the formula $\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_3)$. This is a CNF formula with three clauses: $C_1 = (x_1 \vee \neg x_2)$ is a 2-clause, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$ is a 3-clause, and $C_3 = (\neg x_3)$ is a 1-clause (also called a *unit clause*).

2.1.3 The SAT Decision Problem

Definition 2.1.5 (Assignment)**.** A *(truth) assignment* $\sigma : \mathcal{X} \rightarrow \mathbb{B}$ is a mapping from propositional variables to Boolean values. An assignment σ naturally extends to literals by defining $\sigma(\neg x) = 1 - \sigma(x)$.

Definition 2.1.6 (Satisfying Assignment)**.** An assignment σ *satisfies* a clause $C = (\ell_1 \vee \cdots \vee \ell_k)$ if at least one literal in C evaluates to true under σ, i.e., $\exists i \in \{1, \ldots, k\} : \sigma(\ell_i) = 1$. An assignment σ *satisfies* a CNF formula $\varphi = C_1 \wedge \cdots \wedge C_m$ if it satisfies all clauses simultaneously, i.e., $\forall j \in \{1, \ldots, m\} : \sigma$ satisfies C_j. Such an assignment is called a *satisfying assignment* or a *model* of φ.

Definition 2.1.7 (Boolean Satisfiability Problem (SAT))**.** The *Boolean Satisfiability problem* (SAT) is defined as follows:

- **Instance:** A propositional formula φ in CNF.
- **Question:** Does there exist an assignment σ that satisfies φ?

If such an assignment exists, φ is called *satisfiable*; otherwise, it is *unsatisfiable*.

Example 2.1.3. Consider the formula $\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2)$ over variables $\{x_1, x_2\}$. The assignment $\sigma(x_1) = 0, \sigma(x_2) = 1$ satisfies the first two clauses but falsifies the third clause $(\neg x_2)$. In fact, no assignment satisfies all three clauses simultaneously, hence φ is unsatisfiable.

Example 2.1.4. For the formula $\psi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$, the assignment $\sigma(x_1) = 1, \sigma(x_2) = 1$ satisfies both clauses: $\sigma(x_1) = 1$ makes the first clause true, and $\sigma(x_2) = 1$ makes the second clause true. Thus, ψ is satisfiable with model σ.

2.1.4 NP-Completeness of SAT

SAT was the first problem proven NP-complete.

Theorem 2.1.1 (Cook's Theorem [13])**.** *The Boolean Satisfiability problem (SAT) is NP-complete.*

Cook's theorem established that SAT is intractable assuming $\mathsf{P} \neq \mathsf{NP}$:

1. **SAT $\in$ NP:** Given a candidate assignment σ, one can verify in polynomial time whether σ satisfies the CNF formula φ by evaluating each clause.
2. **SAT is NP-hard:** Any problem in NP can be reduced to SAT in polynomial time. Cook's proof shows that any nondeterministic Turing machine computation can be encoded as a SAT instance such that the machine accepts its input if and only if the corresponding formula is satisfiable.

Unless $\mathsf{P} = \mathsf{NP}$, no polynomial-time algorithm exists for SAT in the worst case. Nevertheless, modern CDCL-based SAT solvers can solve many practical instances with millions of variables [7].

2.2 Conflict-Driven Clause Learning

Conflict-Driven Clause Learning (CDCL) is the algorithmic paradigm underlying modern SAT solvers [7, 36]. When a solver encounters a contradiction, it analyzes the conflict cause and learns a new clause that prevents similar contradictions. Understanding CDCL is necessary for pseudo-Boolean solvers that extend these techniques to linear inequalities.

Algorithm 2.1 presents the high-level structure of CDCL. The solver alternates between propagation (deriving forced assignments), decision (choosing variable values), and conflict handling (learning from contradictions and backtracking).

2.2.1 Unit Propagation

CDCL solvers use *unit propagation* (also called Boolean Constraint Propagation or BCP) as their main inference mechanism. In a clause where all but one literal are assigned false, the remaining literal must be true for the clause to be satisfied.

Definition 2.2.1 (Unit Clause under Partial Assignment [7])**.** Let $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$ be a clause and σ be a partial assignment. The clause C is *unit* under σ if exactly one literal ℓ_i is unassigned and all other literals ℓ_j $(j \neq i)$ are falsified by σ. The unassigned literal ℓ_i is called the *unit literal* or *implied literal*.

Algorithm 2.1 CDCL SAT Solving [7]

```
1: procedure CDCL(φ)
2:     d ← 0                                              ▷ Decision level
3:     loop
4:         UnitPropagate(φ)                               ▷ Derive forced assignments
5:         if conflict detected then
6:             if d = 0 then return UNSAT                 ▷ Conflict without decisions: unsatisfiable
7:             end if
8:             (C_L, d′) ← ConflictAnalysis               ▷ Learn clause, find backtrack level
9:             Add learned clause C_L to φ                ▷ Prevent same conflict
10:            Backtrack(d′)                              ▷ Undo assignments above level d′
11:            d ← d′                                     ▷ Continue; C_L will propagate
12:        else if all variables assigned then return SAT
13:        else
14:            d ← d + 1                                  ▷ New decision level
15:            (x, v) ← PickBranchingVariable             ▷ Select variable and value
16:            Assign x ← v                               ▷ Make decision
17:        end if
18:    end loop
19: end procedure
```

Definition 2.2.2 (Unit Propagation)**.** *Unit propagation* is the process of repeatedly identifying unit clauses and extending the partial assignment by setting each unit literal to true. Given a CNF formula φ and partial assignment σ, unit propagation computes the unique fixed point where no unit clauses remain.

Example 2.2.1. Consider the formula $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (\neg x_4 \vee x_5) \wedge (\neg x_2)$ with partial assignment $\sigma = \{x_1 \mapsto 1, x_3 \mapsto 0\}$. The clause $(\neg x_1 \vee x_4)$ becomes unit since $\neg x_1$ is false, implying $x_4 = 1$. This assignment makes $(\neg x_4 \vee x_5)$ unit, implying $x_5 = 1$. Unit propagation thus extends σ to $\{x_1 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto 1, x_5 \mapsto 1\}$.

Unit propagation is computationally inexpensive—modern solvers use the *two-watched-literal scheme* to achieve amortized constant time per propagation [39]. A variable's *propagation activity* measures how frequently its assignment triggers propagations of other variables; this metric is used by some solver heuristics to identify influential variables.

2.2.2 Decision Variables and the Search Tree

When unit propagation reaches a fixed point without finding a conflict or a complete assignment, the solver must make a *decision*: it selects an unassigned variable and

assigns it a truth value.

Definition 2.2.3 (Decision Level)**.** Each decision variable is associated with a *decision level* $d \geq 1$, indicating the depth in the search tree. Variables assigned through unit propagation inherit the decision level of the decision that triggered them. Decision level 0 is reserved for literals that can be derived from the original formula without any decisions (initial unit propagation).

Definition 2.2.4 (Decision Variable)**.** A *decision variable* is a variable whose value is chosen by the solver's branching heuristic rather than being implied through unit propagation. The sequence of decisions forms a search tree, where each path from the root represents a partial assignment.

Decision ordering affects solver performance. After each decision, unit propagation extends the assignment until either: (1) all clauses are satisfied (solution found), (2) some clause becomes falsified (conflict detected), or (3) no more propagations are possible (another decision is needed).

2.2.3 Conflict Analysis

A *conflict* occurs when a clause has all its literals falsified under the current partial assignment. Rather than simply backtracking to the most recent decision, CDCL solvers perform *conflict analysis* to understand which decisions caused the conflict [36].

Definition 2.2.5 (Conflict Clause)**.** A clause C is a *conflict clause* (or *falsified clause*) under assignment σ if every literal in C is assigned false by σ.

Definition 2.2.6 (Implication Graph [36])**.** The *implication graph* is a directed acyclic graph that records the causal relationships between variable assignments. Each node represents a literal and its decision level. An edge from ℓ_1 to ℓ_2 indicates that ℓ_1 was one of the antecedent literals that implied ℓ_2 through unit propagation. Decision literals have no incoming edges.

Example 2.2.2. Consider clauses $C_1 = (\neg x_1 \vee x_2)$, $C_2 = (\neg x_1 \vee x_3)$, $C_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$, and $C_4 = (\neg x_4 \vee \neg x_5)$. Suppose $x_1 = 1$ at decision level 1 and $x_5 = 1$ at decision level 2. From C_1: $x_2 = 1$ (level 1). From C_2: $x_3 = 1$ (level 1). From C_3: $x_4 = 1$ (level 1). From C_4: conflict, since both x_4 and x_5 are true. The implication graph shows that x_1 and x_5 together lead to this conflict through the propagation chain.

Conflict analysis traverses the implication graph backwards from the conflict, identifying a *cut* that separates the decision variables from the conflict. The most common strategy identifies the *first Unique Implication Point (UIP)*—the first node on the path from the most recent decision to the conflict that appears on every such path.

2.2.4 Clause Learning

CDCL learns a new clause from each conflict, preventing the same conflict pattern from recurring [36].

Definition 2.2.7 (Learned Clause)**.** Given a conflict and its implication graph, a *learned clause* C_L is a clause that:

1. Is logically implied by the original formula (i.e., adding C_L does not change satisfiability),
2. Is falsified by the current assignment (i.e., represents the conflict),
3. Contains at most one literal from the current decision level (the UIP scheme).

The learned clause is constructed by *resolution*: starting from the conflict clause, repeatedly resolve with the antecedent clause of each literal until reaching the UIP. Resolution of clauses $C_1 = (A \vee x)$ and $C_2 = (B \vee \neg x)$, where A and B are disjunctions of literals and x is a variable appearing positively in C_1 and negatively in C_2, produces the resolvent $C = (A \vee B)$.

Example 2.2.3. Continuing the previous example, the conflict involves $C_4 = (\neg x_4 \vee \neg x_5)$ with both literals false (since $x_4 = x_5 = 1$). Resolving with the antecedent of x_4, which is $C_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$, yields $(\neg x_2 \vee \neg x_3 \vee \neg x_5)$. Further resolution eventually produces the learned clause $(\neg x_1 \vee \neg x_5)$, which states that x_1 and x_5 cannot both be true.

Learned clauses accumulate during search, pruning the search space by preventing the solver from revisiting equivalent conflict situations. Modern solvers periodically delete less useful learned clauses to manage memory.

2.2.5 Non-chronological Backtracking

Traditional backtracking undoes the most recent decision when a conflict occurs. CDCL employs *non-chronological backtracking* (also called *backjumping*), which can skip multiple decision levels [36].

Definition 2.2.8 (Backtrack Level)**.** Given a learned clause C_L with literals at various decision levels, the *backtrack level* is the second-highest decision level among literals in C_L. After backtracking to this level, exactly one literal in C_L remains unassigned (the one from the highest level) while all other literals are falsified by the remaining assignment. This makes C_L *unit under the current partial assignment*—also called an *asserting clause*—forcing propagation of the single unassigned literal.

Example 2.2.4. If the learned clause is $(\neg x_1 \vee \neg x_5)$ where x_1 was decided at level 1 and x_5 at level 2, the backtrack level is 1. The solver undoes all assignments at

levels 2 and above. At level 1, $x_1 = 1$ still holds, so $\neg x_1$ is false; thus C_L has one falsified literal and one unassigned literal ($\neg x_5$), making it unit under the current assignment. This immediately implies $x_5 = 0$, preventing the conflict from recurring.

Non-chronological backtracking improves efficiency by jumping directly to the decision level where the conflict can be resolved, skipping irrelevant decisions.

2.2.6 Variable Selection Heuristics: VSIDS

Decision variable choice affects solver performance. The *Variable State Independent Decaying Sum* (VSIDS) heuristic, introduced in Chaff [39], focuses on recently active variables.

Definition 2.2.9 (VSIDS Heuristic)**.** The *VSIDS* (Variable State Independent Decaying Sum) heuristic maintains an activity score for each variable. When a conflict occurs, scores of variables appearing in the learned clause are incremented. Periodically, all scores are decayed (multiplied by a constant < 1). The decision heuristic selects the unassigned variable with the highest score.

VSIDS is conflict-driven (variables in recent conflicts receive higher priority, focusing search on the "hard" part of the formula), uses exponential decay (recent conflicts matter more than older ones, allowing the heuristic to adapt as search progresses), and is efficient (scores are simple floating-point values, making updates and lookups fast).

Modern variants refine VSIDS; for example, some solvers also track propagation activity—how often a variable's assignment causes other variables to be propagated—as an additional signal for variable importance.

2.2.7 Relevance to Pseudo-Boolean Solving

CDCL extends to pseudo-Boolean optimization. Solvers like RoundingSAT integrate conflict-driven search with *cutting planes reasoning*, learning linear inequalities instead of clauses [20]:

Propagation generalizes: instead of unit propagation on clauses, pseudo-Boolean solvers propagate on linear constraints, and when a constraint's slack becomes zero (all unfixed terms are required to satisfy the bound), variable assignments are implied. Conflict analysis also generalizes: when a constraint becomes infeasible, the solver learns a new linear inequality derived through cutting planes operations (such as saturation and division), analogous to resolution for clauses. Heuristics adapt as well—variable selection can incorporate propagation activity, measuring how often a variable's assignment triggers propagations of other variables, to identify influential variables in the constraint system.

The extension preserves CDCL structure while using the additional reasoning power of linear constraints.

2.3 Pseudo-Boolean Optimization

Linear Pseudo-Boolean Optimization (Linear PBO) lies between propositional satisfiability and Mixed Integer Programming. Linear PBO involves integer linear programming restricted to binary variables, using notation and solving techniques from the SAT community. More general PBO allows nonlinear terms (products of Boolean variables), but this thesis focuses on the linear case.

2.3.1 Pseudo-Boolean Constraints

Definition 2.3.1 (Pseudo-Boolean Function [9])**.** A *pseudo-Boolean function* is a function $f : \mathbb{B}^n \to \mathbb{R}$ mapping Boolean vectors to real numbers. Any pseudo-Boolean function can be uniquely represented as a multilinear polynomial over Boolean variables.

Definition 2.3.2 (Linear Pseudo-Boolean Constraint)**.** A *linear pseudo-Boolean (PB) constraint* is a linear inequality of the form

$$\sum_{i=1}^{n} c_i \ell_i \geq b \tag{2.3}$$

where:

- ℓ_i are Boolean literals, i.e., either a propositional variable x_i or its negation $\neg x_i$,
- $c_i \in \mathbb{N}$ are non-negative integer coefficients, and
- $b \in \mathbb{N}$ is the bound (or threshold).

For evaluation purposes, a literal ℓ under assignment σ takes value $\sigma(\ell) = \sigma(x)$ if $\ell = x$, and $\sigma(\ell) = 1 - \sigma(x)$ if $\ell = \neg x$. A PB constraint is satisfied by assignment σ when the weighted sum of its literals meets or exceeds the bound.

More generally, pseudo-Boolean constraints can include nonlinear terms—products of literals with integer coefficients. Such constraints arise in quadratic pseudo-Boolean optimization and can be linearized through auxiliary variable introduction, though this may increase problem size.

Linear PB constraints are linear inequalities where variables are restricted to $\{0, 1\}$ rather than continuous values. Nonlinear PB constraints, involving products of Boolean variables, also exist and can be handled through linearization or direct nonlinear methods [9]. While LP is solvable in polynomial time, PB decision problems are NP-complete.

Example 2.3.1. Consider the PB constraint $2x_1 + 3x_2 + x_3 \geq 4$. This constraint is satisfied when the weighted combination of true variables reaches at least 4. For instance, $\sigma = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0\}$ yields $2(1) + 3(1) + 1(0) = 5 \geq 4$, satisfying the constraint. However, $\sigma' = \{x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 1\}$ yields $2(1) + 3(0) + 1(1) = 3 < 4$, violating it.

2.3.2 Relationship to Propositional Satisfiability

PB constraints strictly generalize propositional clauses. Any clause can be encoded as a PB constraint with unit coefficients and threshold one.

Example 2.3.2. The clause $(x_1 \vee x_2 \vee \neg x_3)$ is logically equivalent to the PB constraint

$$x_1 + x_2 + (1 - x_3) \geq 1 \tag{2.4}$$

which simplifies to $x_1 + x_2 + \bar{x}_3 \geq 1$ using the notation $\bar{x}$ for the negated literal. The constraint requires at least one of the three literals to be true, exactly capturing the clause semantics.

However, PB constraints offer strictly greater expressiveness. Cardinality constraints ("at least k of n variables must be true") and weighted threshold constraints have compact PB representations but require exponentially many clauses in CNF [9].

Example 2.3.3. The constraint "at least 2 of $\{x_1, x_2, x_3\}$ must be true" is directly expressed as

$$x_1 + x_2 + x_3 \geq 2 \tag{2.5}$$

Converting this to CNF requires enumerating all pairs: $(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3)$. For general constraints $\sum_{i=1}^{n} c_i x_i \geq b$ with arbitrary coefficients, the CNF encoding can grow exponentially in the number of variables.

The expressiveness gap has proof-theoretic implications. Resolution (underlying CDCL SAT solvers) can require exponentially many steps to derive unsatisfiability for certain PB constraints, whereas cutting planes admits polynomial-length proofs for the same instances [25]. PB solvers are thus preferable when the problem involves cardinality or weighted constraints.

2.3.3 Decision and Optimization Problems

Definition 2.3.3 (PB Decision Problem)**.** Given a set $\mathcal{C}$ of PB constraints over Boolean variables $\mathcal{X}$, the *PB decision problem* asks whether there exists an assignment $\sigma : \mathcal{X} \to \mathbb{B}$ satisfying all constraints in $\mathcal{C}$.

Definition 2.3.4 (Pseudo-Boolean Optimization)**.** Given a set $\mathcal{C}$ of PB constraints over Boolean variables $\mathcal{X}$ and a linear objective function

$$f(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i \tag{2.6}$$

with integer weights $w_i \in \mathbb{Z}$, the *Pseudo-Boolean Optimization (PBO) problem* seeks an assignment σ^* that minimizes f while satisfying all constraints:

$$\sigma^* = \arg\min_{\sigma \models \mathcal{C}} f(\sigma) \tag{2.7}$$

The PB decision problem is NP-complete since it generalizes SAT. Consequently, PBO is NP-hard.

Optimization reduces to a sequence of decision problems. Given an optimal solution σ^* with objective value $f(\sigma^*) = S^*$, adding the constraint $f(\mathbf{x}) = S^*$ transforms the optimization instance into a decision problem. This reduction is useful when enumerating all optimal solutions or proving properties about the optimal solution set.

Example 2.3.4. Consider minimizing $f(x_1, x_2, x_3) = 2x_1 + x_2 + 3x_3$ subject to $x_1 + x_2 + x_3 \geq 2$. The optimal solution $\sigma^* = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0\}$ achieves $f(\sigma^*) = 3$. To enumerate all optimal solutions, we add constraint $2x_1 + x_2 + 3x_3 = 3$ and solve the resulting decision problem.

2.3.4 Connection to Integer Linear Programming

Linear PBO is equivalent to 0-1 Integer Linear Programming (0-1 ILP), differing in notation and solving methodology. ILP solvers typically use branch-and-bound with LP relaxations, while PB solvers adapt conflict-driven techniques from SAT solving. The SAT-based approach can be advantageous for highly combinatorial instances where LP relaxation provides weak bounds.

The choice between PBO and 0-1 ILP formulations depends on problem structure: PBO solvers excel when constraints encode logical relationships expressed through Boolean combinations, while ILP solvers may perform better when LP relaxation closely approximates the integer optimum [9].

2.4 Backbones

A *backbone* characterizes constraint satisfaction and optimization problems structurally. Originally studied in SAT, backbones capture variables that are "frozen" across all solutions, offering insight into problem structure and computational hardness.

2.4.1 Backbones in Boolean Satisfiability

Consider a Boolean formula φ in conjunctive normal form (CNF) over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. A *literal* is either a variable x_i or its negation $\neg x_i$. A *model* (or satisfying assignment) of φ is an assignment of truth values to all variables in X that makes φ evaluate to true. We denote the set of all models of φ as $\mathcal{M}(\varphi)$.

Definition 2.4.1 (Backbone of a SAT instance [29])**.** Let φ be a satisfiable Boolean formula. The *backbone* of φ, denoted $\mathcal{B}(\varphi)$, is the set of literals that are true in every model of φ:

$$\mathcal{B}(\varphi) = \{\ell \mid \forall \mu \in \mathcal{M}(\varphi) : \mu \models \ell\} \tag{2.8}$$

where $\mu \models \ell$ denotes that the assignment μ satisfies literal ℓ.

Equivalently, a literal ℓ belongs to the backbone if and only if $\varphi \wedge \neg\ell$ is unsatisfiable. To test whether ℓ is in the backbone, check whether augmenting the formula with $\neg\ell$ makes it unsatisfiable.

Example 2.4.1. Consider the formula $\varphi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$. The models of φ are:

- $\mu_1 : x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{true}$
- $\mu_2 : x_1 = \text{true}, x_2 = \text{false}, x_3 = \text{true}$

In both models, x_1 and x_3 are true, while x_2 varies. Therefore, $\mathcal{B}(\varphi) = \{x_1, x_3\}$.

The backbone can also be understood in terms of variable-value pairs. For decisional problems such as SAT, the backbone comprises pairs $\langle \text{variable}, \text{value} \rangle$ that appear in every satisfying assignment [32]. A variable x_i is called a *backbone variable* if either $x_i \in \mathcal{B}(\varphi)$ or $\neg x_i \in \mathcal{B}(\varphi)$.

2.4.2 Extension to Optimization Problems

The backbone concept extends to combinatorial optimization problems, where it consists of variable-value pairs present in all optimal solutions [52].

Definition 2.4.2 (Backbone of an optimization problem)**.** Let $\mathcal{P}$ be a combinatorial optimization problem with objective function f to be minimized (or maximized), and let $\mathcal{S}^*$ denote the set of all optimal solutions. The *backbone* of $\mathcal{P}$ is:

$$\mathcal{B}(\mathcal{P}) = \{\langle x_i, v \rangle \mid \forall s \in \mathcal{S}^* : s(x_i) = v\} \tag{2.9}$$

where $s(x_i)$ denotes the value of variable x_i in solution s.

For optimization variants of SAT, such as Maximum Satisfiability (MaxSAT), the backbone comprises literals that must hold in every optimal solution. Similarly, for constraint optimization problems, the backbone identifies variable assignments that are invariant across all optima.

2.4.3 Feasibility Backbones versus Optimality Backbones

When moving from decision problems to optimization problems, the backbone concept splits into two related but distinct notions. Consider a PBO instance with a set of constraints φ over Boolean variables $\mathbf{x} = (x_1, \ldots, x_n)$ and an objective function $f(\mathbf{x})$ to be minimized.

The *feasibility backbone* considers all solutions that satisfy the constraints, regardless of their objective value. It is the set of literals true in every satisfying assignment—precisely Definition 2.4.1:

$$\mathcal{B}_{\text{feas}}(\varphi) = \{\ell \mid \varphi \models \ell\}.$$

The *optimality backbone* is more restrictive: it considers only the optimal solutions, i.e., those that both satisfy φ and minimize f. This corresponds to Definition 2.4.2, where $\mathcal{S}^*$ denotes the set of all optimal solutions.

Since optimal solutions are a subset of all feasible solutions, any literal that is forced across *all* feasible solutions must also be forced across the (smaller) set of optimal ones. Therefore, $\mathcal{B}_{\text{feas}}(\varphi) \subseteq \mathcal{B}_{\text{opt}}(\varphi, f)$. The converse does not hold: a variable might take different values across feasible solutions in general, yet always take the same value whenever the objective is optimal.

This relationship enables a practical reduction. Suppose the optimal objective value z^* is known (e.g., from a prior optimization phase). Adding the constraint $f(\mathbf{x}) \leq z^*$ to φ yields a new formula φ' whose feasible solutions are exactly the optimal solutions of the original problem. Computing the feasibility backbone of φ'—a standard backbone computation on a decision problem—therefore yields the optimality backbone of the original instance:

$$\mathcal{B}_{\text{feas}}(\varphi') = \mathcal{B}_{\text{opt}}(\varphi, f).$$

This is the OPT-to-DEC transformation that the extraction algorithms in this thesis rely on (Section 4.1): first solve to optimality to obtain z^*, then extract the backbone of the resulting decision problem. In this way, any backbone extractor designed for satisfiability problems can be applied to optimization problems, provided the optimal value is available.

The inclusion $\mathcal{B}_{\text{feas}}(\varphi) \subseteq \mathcal{B}_{\text{opt}}(\varphi, f)$ also suggests a potential strategy: literals identified as feasibility backbone members—without requiring knowledge of z^*—are guaranteed to belong to the optimality backbone, potentially reducing the number of candidates that need to be tested after the optimization phase. In practice, however, this is unlikely to yield significant savings: computing the feasibility backbone still requires multiple solver calls on the full constraint set, the optimization phase cannot be skipped since z^* is needed for the remaining candidates, and the feasibility backbone tends to be small relative to the optimality backbone, since most fixed-variable structure arises from the optimization constraint rather than the feasibility constraints alone.

A related observation appears in the study of *backdoors*—sets of variables whose instantiation simplifies a problem to a tractable form. Dilkina et al. [18] showed that in mixed-integer programming, the backdoor sets needed for proving optimality can differ substantially in size from those needed for finding a feasible solution, highlighting that the feasibility-optimality distinction is a recurring theme in the structural analysis of combinatorial problems.

2.4.4 Computational Complexity

The computational complexity of backbone-related decision problems has been extensively studied [29].

Theorem 2.4.1 (Complexity of backbone detection [29])**.** *Given a satisfiable Boolean formula φ and a literal ℓ, determining whether $\ell \in \mathcal{B}(\varphi)$ is* coNP*-complete.*

Computing the complete backbone requires determining membership for each literal, yielding a problem at least as hard as solving multiple SAT instances.

2.5 Backbone Extraction Algorithms

We now present algorithmic techniques for computing backbones, which form the foundation for our proposed methods.

2.5.1 Iterative Literal Testing

The most direct approach follows from the definition: a literal ℓ belongs to the backbone if and only if $\varphi \wedge \neg\ell$ is unsatisfiable. This yields the *iterative literal testing* algorithm [29].

The algorithm maintains a candidate set initialized with all literals. For each candidate ℓ, it tests whether $\varphi \wedge \neg\ell$ is satisfiable. If unsatisfiable, ℓ is added to the backbone. While correct, this approach requires $O(n)$ SAT calls in the worst case—one for each variable polarity that could potentially be in the backbone. For formulas with large backbones or difficult satisfiability checks, this becomes computationally prohibitive. Nevertheless, iterative testing establishes the foundation upon which all subsequent refinements build.

2.5.2 Model-Based Filtering

If a literal takes different values in two satisfying assignments, it cannot be a backbone literal. This observation underlies *model-based filtering* [29].

Definition 2.5.1 (Model-Based Filtering)**.** Given two models $\mu_1, \mu_2 \in \mathcal{M}(\varphi)$ and a candidate set $\mathcal{C}$, the filtered candidate set is:

$$\mathcal{C}' = \{\ell \in \mathcal{C} \mid \mu_1 \models \ell \text{ and } \mu_2 \models \ell\}$$

The algorithm maintains a set of candidate backbone literals, initially populated from an arbitrary satisfying assignment. Whenever the solver returns a new model—either from a satisfiability check or as a byproduct of other operations—the candidate set is filtered to retain only those literals satisfied by all observed models. This filtering is computationally inexpensive (linear in the number of variables) yet can eliminate large portions of the candidate set with each new model.

Model-based filtering effectiveness depends on the diversity of models encountered. Formulas with many differing satisfying assignments benefit most from this technique.

2.5.3 Chunk-Based Probing

Rather than testing literals individually, *chunk-based probing* tests multiple candidates simultaneously [29]. Given a chunk $\Gamma \subseteq \mathcal{C}$ of k candidate literals, we ask whether at least one can be negated while maintaining satisfiability.

Definition 2.5.2 (Chunk Constraint)**.** For a chunk $\Gamma = \{\ell_1, \ldots, \ell_k\}$ of candidate literals, the chunk constraint is:

$$\bigvee_{\ell \in \Gamma} \neg \ell \quad \equiv \quad \sum_{\ell \in \Gamma} \bar{\ell} \geq 1$$

If φ conjoined with this constraint is unsatisfiable, then negating any literal in Γ renders the formula unsatisfiable—hence all literals in Γ are backbone members. This identifies k backbone literals with a single SAT call, a substantial improvement when backbones are large.

Conversely, if the constrained formula is satisfiable, the resulting model provides filtering information: any candidate literal falsified in this model is definitively not in the backbone. The algorithm then continues with the remaining candidates.

Adaptive chunking dynamically adjusts the chunk size based on solver responses. After successfully identifying backbone literals (UNSAT response), the chunk size doubles, exploiting the apparent density of backbone literals in that region. After a SAT response, the chunk size resets to one, reflecting that the current chunk contained non-backbone literals.

2.5.4 The CadiBack Algorithm

CadiBack [6] is a standalone backbone extractor that leverages the CaDiCaL SAT solver. It is the state-of-the-art backbone extractor for SAT, combining iterative testing, model-based filtering, and adaptive chunking with optimizations that exploit solver internals.

CadiBack's main contribution is *improved model rotation via watched literals*. When testing whether a literal can be flipped, CadiBack uses the solver's watched

literal data structures to determine which clauses would become unsatisfied. This allows rapid local search for alternative models without full SAT calls.

CadiBack also employs incremental SAT solving, maintaining learned clauses and solver state across the many queries required for backbone extraction. This amortizes the cost of preprocessing and early inference across all tests.

2.5.5 Phase Selection for Diverse Solutions

Model-based filtering works best with diverse satisfying assignments. The *pguide* heuristic [40] biases search toward unexplored regions of the solution space.

During CDCL search, when selecting a value for a decision variable, the solver must choose a polarity (true or false). Standard heuristics often use cached polarities or fixed defaults. Pguide instead tracks the polarity assigned to each variable across all previously found solutions.

Definition 2.5.3 (Polarity Potential)**.** For a variable u, let p_u denote the number of solutions where $u = 1$ and n_u where $u = 0$. The polarity potential is $\Pi_u = p_u - n_u$.

When selecting the polarity for variable u, pguide chooses $u = 0$ if $\Pi_u > 0$ and $u = 1$ otherwise—biasing against the majority polarity observed so far. This encourages the solver to explore different regions of the solution space, increasing the likelihood of finding counterexamples that demonstrate a literal is not in the backbone.

In the context of backbone extraction, diverse solutions accelerate candidate filtering. If most solutions assign variable x to true, pguide will bias toward trying $x = 0$, potentially discovering a model that eliminates x from the candidate set.

2.6 PBO Solver Paradigms

PBO is NP-hard, and no single solver outperforms all others across all instance types. Performance depends on instance characteristics such as constraint density, coefficient magnitudes, and problem structure. Different solving paradigms have emerged with distinct strengths. We present three approaches used in our experiments: RoundingSAT (native PBO with cutting planes), Gurobi (commercial MIP solver), and NaPS (PBO-to-SAT translation).

2.6.1 RoundingSAT: Native PBO with Cutting Planes

RoundingSAT [20] integrates conflict-driven search with cutting planes reasoning. Unlike CDCL solvers that learn clauses, RoundingSAT learns linear inequalities over Boolean variables. Linear inequalities are more expressive than clauses, capturing relationships that would require exponentially many clauses to express.

The algorithm extends CDCL to operate on pseudo-Boolean constraints. On conflict, RoundingSAT performs conflict analysis using cutting planes derivations rather than resolution. The two operations are:

- **Addition**: Combining two linear inequalities by adding them coefficient-wise.
- **Division**: Dividing all coefficients by a constant and rounding up (saturation), which corresponds to applying a cutting plane.

These operations define the cutting planes proof system, which is exponentially stronger than resolution. RoundingSAT can derive contradictions in polynomial steps for certain problem classes where resolution requires exponential time.

RoundingSAT uses exact arithmetic over arbitrary-precision integers. This incurs computational overhead compared to floating-point arithmetic but eliminates numerical instability with large coefficients—common in PBO instances from combinatorial problems.

RoundingSAT also incorporates LP relaxations through integration with the SoPlex LP solver. At certain points during search, the solver can compute the LP relaxation of the current subproblem, using the fractional solution to guide branching decisions or derive additional cutting planes.

Propagation in RoundingSAT operates natively on pseudo-Boolean constraints without converting them to clausal form. When a constraint's slack (the difference between the maximum achievable sum and the threshold) becomes insufficient to allow any unassigned literal to be falsified, the solver propagates the implied assignments directly. This preserves the structure of the original problem and avoids the variable explosion that can occur with CNF encodings.

2.6.2 Gurobi: Commercial MIP Solver

Gurobi [24] is a commercial solver that handles PBO as 0-1 Integer Linear Programming. Its algorithmic foundation differs from CDCL-based approaches, building on Branch-and-Bound from linear programming.

Branch-and-Bound maintains a search tree where each node is a subproblem with additional variable bounds. At each node, Gurobi solves the LP relaxation (allowing variables to take fractional values in $[0, 1]$). If the relaxation is infeasible, the node is pruned. If it yields an integer solution, the incumbent is updated. Otherwise, the solver branches on a fractional variable, creating child nodes that fix it to 0 or 1.

Gurobi uses cutting plane techniques to tighten the LP relaxation:

- **Gomory cuts** [23]: Derived from the simplex tableau, these cuts separate fractional LP solutions from the integer hull.
- **Mixed Integer Rounding (MIR) cuts** [35]: Generalize Gomory cuts using rounding arguments applicable to mixed-integer problems.

These cuts strengthen the LP relaxation, providing tighter bounds and reducing search tree size. The interplay between cutting planes and branching—Branch-and-Cut—is central to modern MIP solving.

Unlike RoundingSAT, Gurobi does not perform conflict-driven learning in the CDCL sense. Pruning relies on bound comparisons: a node is pruned when its LP relaxation value cannot improve upon the best known solution.

Gurobi uses floating-point arithmetic for computational efficiency. Modern floating-point implementations achieve remarkable speed, but they can introduce numerical issues when constraint coefficients span many orders of magnitude.

Gurobi's advantages include extensive engineering optimization, parallel processing to explore multiple branches simultaneously, and aggressive presolve routines for problem simplification.

2.6.3 NaPS: PBO-to-SAT Translation

NaPS [49] translates PBO constraints into CNF and delegates solving to a SAT solver, leveraging decades of engineering investment in solvers like MiniSat.

The translation uses Reduced Ordered Binary Decision Diagrams (ROBDDs) [10] to encode pseudo-Boolean constraints as CNF. An ROBDD is a compressed representation of a Boolean function as a DAG. For a pseudo-Boolean constraint, the ROBDD encodes all satisfying assignments, and CNF is derived by introducing auxiliary variables for internal ROBDD nodes.

The ROBDD-based encoding maintains arc consistency: unit propagation on the CNF achieves the same pruning as propagation on the original PB constraint. This is stronger than many alternative encodings that lose propagation strength in translation.

The approach has limitations. ROBDD construction can be expensive for constraints with many variables or large coefficients—ROBDD size depends on variable ordering. The translation also introduces auxiliary variables that obscure the original problem structure.

Once translated, NaPS solves the resulting CNF using standard CDCL techniques: unit propagation, conflict analysis with resolution, and clause learning. The learned clauses are disjunctions of literals over both original and auxiliary variables. This allows NaPS to benefit from CDCL's pruning, but learned clauses cannot express general linear relationships—they remain within resolution's expressive power.

2.6.4 Comparison of Paradigms

Each paradigm brings a fundamentally different perspective to PBO. RoundingSAT stands out for its ability to learn linear inequalities rather than clauses, placing it in the cutting planes proof system which is exponentially stronger than resolution. This theoretical advantage comes with practical trade-offs: exact arithmetic ensures correctness with arbitrary coefficients but runs slower than floating-point. Gurobi

approaches the problem from the operations research tradition, treating PBO as 0-1 ILP and exploiting LP relaxations to compute tight bounds that guide branch-and-bound search. Its strength lies in decades of engineering optimization and the power of continuous relaxations to prune the search space, though floating-point arithmetic can introduce numerical issues when coefficients vary widely. NaPS takes yet another route by translating constraints into CNF and delegating to highly optimized SAT solvers, benefiting from their mature propagation and learning mechanisms while remaining limited to resolution's expressive power.

These differences shape when each solver excels. RoundingSAT tends to perform well on instances where cutting planes reasoning provides significant advantages over resolution—problems with cardinality or weighted constraints that would require exponential clausal encodings. Gurobi dominates when LP relaxations are tight approximations of the integer optimum, as the bounds enable aggressive pruning. NaPS shines on instances where the CNF encoding remains compact and the problem structure aligns well with CDCL's conflict-driven search. No single paradigm dominates across all instance types, which motivates studying solver behavior across diverse benchmarks.

Chapter 3

Related Work

We survey backbone computation techniques for SAT, PBO solving approaches, and backbone applications in combinatorial optimization. Chapter 2 covers the technical foundations; here we focus on the evolution of ideas and research gaps.

3.1 Backbone Extraction in SAT

Backbone research began with investigations into random SAT hardness. Parkes [44] first observed that backbone variables correlate with solution space clustering near the phase transition. Monasson et al. [38] formalized this, showing that backbones emerge in the hard region of random 3-SAT where instances transition from almost surely satisfiable to almost surely unsatisfiable. Achlioptas et al. [1] connected backbone size to solution landscape fragmentation: large backbones correspond to isolated solution clusters with large Hamming distances between them.

Kilby et al. [32] proved that determining backbone membership is coNP-complete, so no polynomial-time algorithm exists unless P = NP. Despite this, practical algorithms perform well on structured instances from real-world applications.

Janota et al. [29] categorized extraction techniques into four strategies: iterative literal testing, which checks if each candidate's negation is satisfiable; chunk-based probing, which tests multiple candidates with cardinality constraints; model-based filtering, which eliminates candidates that vary across satisfying assignments; and core-based identification, which extracts backbone information from unsatisfiability proofs. These techniques are orthogonal and can be combined; effective extractors interleave them based on solver feedback. Chapter 2 presents these algorithms in detail.

CadiBack [6] leverages CaDiCaL and is the current state-of-the-art SAT backbone extractor. It introduces adaptive chunk sizing based on backbone density, improved model rotation using watched literal data structures, and tight integration with incremental SAT solving to amortize preprocessing costs. CadiBack shows that efficient extraction requires exploiting solver internals—particularly search data

structures—rather than treating the solver as a black box.

NeuroBack [56] uses graph neural networks to predict backbone variables in SAT, guiding CDCL variable selection. It trains on solved instances to predict likely backbone members, then biases branching toward these variables. Results show improved solving times on structured benchmarks—even imperfect predictions help. These learning-based approaches need training data from a specific problem distribution; generating this data is expensive since computing exact backbones requires solving instances multiple times.

3.2 PBO Solving Approaches

PBO solving is methodologically diverse. Unlike SAT where CDCL dominates, PBO admits different paradigms with complementary strengths across instance classes. This affects backbone extraction: algorithms effective for one solver type may not work for another.

Three main paradigms exist. Native PBO solvers like RoundingSAT [20] extend CDCL to operate directly on pseudo-Boolean constraints, learning linear inequalities through cutting planes. MIP solvers like Gurobi [24] use branch-and-bound with LP relaxations and cutting plane techniques. Translation-based solvers like NaPS [49] encode PBO constraints into CNF using structures like ROBDDs and leverage optimized SAT solvers.

Recent work on automatic algorithm selection for PBO [45] shows that no single solver dominates across all instance types. Constraint density, coefficient magnitudes, and structural properties determine which paradigm performs best. This suggests that backbone extraction for PBO must either target specific solver families or be solver-agnostic.

Section 2.6 covers the technical details. The key point is that PBO solver diversity presents both a challenge and opportunity: different solver internals expose different information that could accelerate extraction.

3.3 Backbones in Combinatorial Optimization

Backbones have applications beyond SAT. In optimization problems, the backbone comprises variable assignments present in all optimal solutions (Definition 2.4.2).

Slaney and Walsh [52] investigated backbones in optimization and found that the relationship between backbone size and hardness is domain-dependent. Some problem classes show positive correlation (large backbones indicate hard instances), others show inverse relationships.

Schneider et al. [50] used backbones for the Traveling Salesman Problem, identifying edges in all optimal tours to decompose instances into smaller subproblems. Fixing backbone edges and solving the reduced problem gave computational speedups,

illustrating how backbones enable divide-and-conquer in optimization.

Beyond TSP, backbones have been studied across combinatorial optimization. Zhang and Zhang [60] proposed backbone-guided local search for MAX-SAT, using approximate backbone information to bias search toward promising regions. In integer programming, a related concept is *reduced cost fixing*: when the reduced cost of a variable exceeds the optimality gap, the variable can be fixed to its LP bound value in all optimal solutions [58]. This technique, standard in MIP solvers, identifies a subset of backbone variables without explicitly computing the full backbone.

3.4 Backbone Extraction Beyond SAT

While backbone extraction algorithms have been extensively developed for SAT [29, 6], the situation is different for optimization problems, where exact backbone computation has received far less attention.

In Maximum Satisfiability (MaxSAT), backbone information has been used to guide local search rather than extracted exactly. Zhang et al. [59] proposed Backbone Guided Local Search (BGLS), which runs multiple short WalkSAT executions and records how often each literal appears in the resulting local minima. These frequencies—called *pseudo-backbone frequencies*—serve as heuristic estimates of backbone membership: a literal that consistently appears across many local optima is likely to be a backbone variable. BGLS then biases subsequent search decisions toward these high-frequency literals, improving solution quality on SATLIB instances. Crucially, this is an estimation heuristic, not an exact extraction method—there are no formal guarantees that the identified variables are true backbone members. Jiang et al. [30] provided a theoretical justification for why exact extraction has not been pursued in this setting: they proved that retrieving the full backbone in weighted MAX-SAT is NP-hard, and that even retrieving a fixed fraction of the backbone remains NP-hard, establishing a fundamental barrier for exact extraction.

Beyond propositional domains, Previti et al. [46] generalized backbone computation to Satisfiability Modulo Theories (SMT), proposing algorithms for theories such as Linear Integer Arithmetic and Bitvectors. Their work extends the concept of backbone variables to non-Boolean domains with potentially infinite value sets, but targets the satisfiability (decision) variant rather than optimization objectives.

To the best of our knowledge, no prior work has proposed general-purpose algorithms for exact backbone extraction in Pseudo-Boolean Optimization. The methods presented in this thesis are the first to address this gap, operating directly on arbitrary PBO instances without restrictions on constraint structure or objective function form.

3.5 Partial Backbones and Their Utility

A partial backbone is the subset of backbone literals identified within a given computational budget, without necessarily determining the status of all variables. All iterative backbone extraction algorithms—including CadiBack [6] and the methods proposed in this thesis—are inherently anytime: they can be interrupted at any point, returning the backbone literals confirmed so far. This property is particularly relevant for large-scale PBO instances where complete extraction may exceed time limits.

Partial backbone information has proven useful across multiple settings. De Haan et al. [16] introduced *local backbones* (k-backbones): backbone literals that can be determined from subsets of at most k clauses. Their empirical results show that a large fraction of backbones in structured SAT instances are local, in contrast to random instances—meaning they are computationally inexpensive to identify. Chen and Whitley [12] demonstrated that *pseudo backbones*—approximate backbone variables estimated via local search—can vastly simplify SAT instances, decomposing them into thousands of independent connected components and enabling divide-and-conquer solving.

From a machine learning perspective, the utility of partial backbone information is supported by the broader literature on learning from imperfect data. Rolnick et al. [48] demonstrated that deep neural networks maintain over 90% test accuracy on MNIST even when each clean training example is diluted with 100 randomly-labeled examples, suggesting that partial or noisy structural signals can be effectively leveraged given sufficient data. Natarajan et al. [41] provided theoretical guarantees for empirical risk minimization under random label noise, establishing that learning is provably possible even with substantial label corruption. These results suggest that partial backbone labels—even when incomplete or approximate—can serve as useful training signals for machine learning models applied to combinatorial optimization, as proposed for the dataset released with this thesis.

The algorithms proposed in this thesis naturally support partial backbone extraction. Since the extraction process iterates over candidate variables, testing each for membership in the backbone, any intermediate state constitutes a valid partial backbone. This anytime property, combined with the evidence that partial structural information is sufficient for both solver guidance and machine learning applications, makes the proposed methods practical even when complete extraction is infeasible.

Gap: PBO-Specific Backbone Tools. Despite backbone utility in SAT and other domains, backbone extraction for PBO remains unexplored. CadiBack and related SAT tools do not apply directly: PBO constraints are not clauses, PBO solvers have different architectures, and the optimization objective adds structure absent in decision problems. This gap motivates our work on backbone extraction methods for pseudo-Boolean optimization.

Chapter 4

Proposed Methods

We develop three backbone extractors for Pseudo-Boolean Optimization: RoundingBack, a native PBO extractor built on RoundingSAT (Section 4.2); GuroBack, using MILP solving; and NapBack, using SAT-based extraction (Section 4.4). All three use the same general algorithm based on adaptive chunking (Section 4.1), with RoundingBack having two heuristic variants for candidate ordering and solution diversity (Section 4.3).

4.1 General Backbone Extraction Algorithm

Our backbone extraction algorithm extends SAT techniques (Chapter 2, Section 2.5) to PBO. The main challenge is that PBO backbones are defined over optimal solutions only—a literal belongs to the backbone if and only if it holds in every optimal solution. We must first establish optimality before extraction can proceed.

4.1.1 Optimization to Decision Transformation

The first step transforms the optimization problem into a decision problem that captures exactly the set of optimal solutions. Let $\mathcal{P}$ be a PBO instance with objective function $f(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i$ to minimize, subject to constraints $\mathcal{C}$.

Definition 4.1.1 (OPT-to-DEC Transformation)**.** Given a PBO instance $\mathcal{P}$ with optimal objective value S^*, the *decision transformation* $\mathcal{D}(\mathcal{P}, S^*)$ is the decision problem consisting of constraints $\mathcal{C}$ augmented with:

$$f(\mathbf{x}) = S^* \tag{4.1}$$

Equivalently: $f(\mathbf{x}) \leq S^*$ and $f(\mathbf{x}) \geq S^*$.

The satisfying assignments of $\mathcal{D}(\mathcal{P}, S^*)$ are exactly the optimal solutions of $\mathcal{P}$. This reduces PBO backbone extraction to decision-problem backbone extraction, so we can apply iterative testing, model-based filtering, and chunk-based probing.

Example 4.1.1. Consider minimizing $f(x_1, x_2, x_3) = 2x_1 + x_2 + 3x_3$ subject to $x_1 + x_2 + x_3 \geq 2$. If the optimal value is $S^* = 3$, the decision transformation adds constraint $2x_1 + x_2 + 3x_3 = 3$. The optimal solutions $\{x_1 = 1, x_2 = 1, x_3 = 0\}$ and potentially others now form the solution space for backbone extraction.

4.1.2 Notation and Flippability

We adopt the following notation throughout this chapter. A *literal* ℓ is either a variable x or its negation $\neg x$. We write $\bar{\ell}$ for the complement: if $\ell = x$ then $\bar{\ell} = \neg x$, and vice versa. An assignment σ maps variables to $\{0, 1\}$.

Definition 4.1.2 (Flippable Literal)**.** A literal ℓ is *flippable* with respect to a satisfiable formula φ if there exists a satisfying assignment $\sigma \in \mathcal{M}(\varphi)$ such that $\sigma \models \bar{\ell}$. Equivalently, ℓ is flippable if $\varphi \wedge \bar{\ell}$ is satisfiable.

A literal belongs to the backbone if and only if it is not flippable. The extraction algorithm tests flippability of candidate literals to determine backbone membership.

4.1.3 Adaptive Chunking Strategy

Our adaptive chunking strategy balances batch testing efficiency against wasted solver calls. Rather than testing literals individually (expensive when backbones are large) or in fixed-size chunks (inefficient when mismatched to backbone density), chunk size adjusts dynamically based on solver feedback.

Definition 4.1.3 (Chunk Constraint for PBO)**.** For a chunk $\Gamma = \{\ell_1, \ldots, \ell_k\}$ of candidate literals, the chunk constraint requires at least one literal to be flipped:

$$\sum_{\ell \in \Gamma} \bar{\ell} \geq 1 \tag{4.2}$$

This is a pseudo-Boolean constraint requiring the sum of complemented literals to be at least one.

When the solver returns UNSAT for the formula augmented with this constraint, all literals in Γ are backbone members—none can be flipped while maintaining satisfiability. When the solver returns SAT, the new model provides filtering information and demonstrates that at least one literal in the chunk is not in the backbone.

After each UNSAT result, chunk size doubles (exploiting backbone density); after SAT results, it resets to one (avoiding wasted tests on non-backbone regions). This achieves logarithmic behavior in regions with contiguous backbone literals while staying correct when backbone density varies.

Algorithm 4.1 PBO Backbone Extraction with Adaptive Chunking

Require: PBO instance $\mathcal{P}$ with constraints $\mathcal{C}$ and objective f
Ensure: Backbone $\mathcal{B}(\mathcal{P})$

$S^* \leftarrow \textsc{SolveToOptimality}(\mathcal{P})$ ▷ Get optimal value
$\sigma \leftarrow$ optimal model from solver
$\mathcal{C}' \leftarrow \mathcal{C} \cup \{f(\mathbf{x}) = S^*\}$ ▷ OPT-to-DEC transformation
$\Lambda \leftarrow \{\ell \mid \sigma \models \ell\}$ ▷ Initialize candidates from model
$\mathcal{B} \leftarrow \emptyset$
$k \leftarrow 1$ ▷ Initial chunk size
while $\Lambda \neq \emptyset$ **do**
    $\Gamma \leftarrow$ first $\min(k, |\Lambda|)$ literals from Λ
    $\varphi \leftarrow \mathcal{C}' \wedge \left(\sum_{\ell \in \Gamma} \bar{\ell} \geq 1\right)$
    if $\textsc{Solve}(\varphi) = \textsc{unsat}$ **then**
        $\mathcal{B} \leftarrow \mathcal{B} \cup \Gamma$ ▷ All in chunk are backbone
        $\Lambda \leftarrow \Lambda \setminus \Gamma$
        $k \leftarrow 2k$ ▷ Double chunk size
    else
        $\sigma' \leftarrow$ model from solver
        $\Lambda \leftarrow \{\ell \in \Lambda \mid \sigma' \models \ell\}$ ▷ Filter by new model
        $k \leftarrow 1$ ▷ Reset chunk size
    end if
end while
return $\mathcal{B}$

4.1.4 Complete Algorithm

Algorithm 4.1 presents the complete backbone extraction procedure for PBO.

The algorithm has two phases: first solve the PBO instance to optimality, obtaining both S^* and an optimal model σ; then perform iterative backbone extraction on $\mathcal{D}(\mathcal{P}, S^*)$.

The candidate set Λ is initialized with all literals satisfied by the first solution obtained after adding the optimality constraint—these are the only possible backbone members, since any literal falsified by this model has a counterexample. The main loop selects chunks of candidates, tests their flippability via the chunk constraint, and updates both the backbone and candidate sets based on solver responses.

Theorem 4.1.1 (Correctness)**.** *Algorithm 4.1 correctly computes the backbone of a PBO instance.*

Proof. Consider a literal $\ell \in \Gamma$ for some chunk Γ. The literal ℓ is added to $\mathcal{B}$ only when the chunk constraint $\sum_{\ell' \in \Gamma} \bar{\ell}' \geq 1$ is unsatisfiable. This means no satisfying assignment can flip any literal in the chunk, so in particular ℓ is not flippable and belongs to the backbone.

Conversely, if ℓ is in the true backbone, it can never be filtered from Λ: filtering removes only literals falsified by some model, but no model can falsify a backbone literal. Eventually ℓ will be tested in some chunk Γ, and the UNSAT result will add it to $\mathcal{B}$. □

4.1.5 Complexity Analysis

The number of solver calls depends on backbone size and the effectiveness of filtering. In the worst case (empty backbone, no filtering), each candidate requires one SAT call, yielding $O(n)$ calls. In the best case (all variables are backbone, no filtering needed), adaptive chunking achieves $O(\log n)$ calls—this logarithmic bound holds specifically when using only the basic chunk constraint (a single negated literal per test) without additional filtering strategies.

In practice, performance depends on backbone density (denser backbones benefit more from chunk doubling), model diversity (more diverse models eliminate more candidates per filtering step), and candidate ordering (testing backbone literals first reduces wasted SAT calls). These last two factors present an explore-vs-exploit trade-off: ordering candidates to test likely backbone members first reduces solver calls (exploitation), while pursuing model diversity improves filtering effectiveness (exploration). Better instance-specific strategies could potentially improve this balance.

The heuristic variants in Section 4.3 address candidate ordering and model diversity.

4.2 RoundingBack: Native PBO Extractor

RoundingBack extracts backbones directly from pseudo-Boolean constraints, without CNF translation or external solvers. It implements Algorithm 4.1 within RoundingSAT, using the solver's native PB reasoning.

4.2.1 RoundingSAT Foundation

As described in Section 2.6, RoundingSAT combines CDCL-style search with cutting planes reasoning and LP relaxations. For backbone extraction, the relevant inherited properties are native PB propagation (avoiding CNF translation overhead), cutting planes learning (enabling the solver to reason natively about chunk constraints and the optimality constraint as linear inequalities), and exact integer arithmetic (ensuring numerical stability with large coefficients).

4.2.2 Implementation Details

RoundingBack extends RoundingSAT with backbone-specific functionality:

Incremental solving. Chunk-based extraction requires many related solver calls on formulas differing only in the chunk constraint. RoundingBack maintains learned constraints and solver state across calls. The chunk constraint is added as a retractable assumption, avoiding re-processing the base formula.

Model access. On SAT results, RoundingBack extracts the satisfying assignment for candidate filtering. The model maps original problem variables to Boolean values, excluding auxiliary variables introduced during solving.

Optimality constraint integration. The OPT-to-DEC transformation adds an equality $f(\mathbf{x}) = S^*$. This constraint is added permanently after the optimization phase completes, so all subsequent queries consider only optimal solutions.

The implementation is available as open-source software at `https://github.com/matiasfrancia/roundingback`.

4.2.3 Advantages of Native PBO Extraction

Working directly with pseudo-Boolean constraints avoids the overhead of CNF translation, which can introduce many auxiliary variables and clauses for constraints with large coefficients. More importantly, the original constraint structure remains visible to the solver, allowing it to exploit problem-specific patterns that would be obscured after encoding. Since both optimization and extraction happen within the same solver instance, learned constraints from optimization carry over to extraction queries, and there is no interface overhead between components.

4.2.4 Relationship to CadiBack

The general extraction strategy in Algorithm 4.1 shares its high-level structure with CadiBack [6], the state-of-the-art SAT backbone extractor. Both employ iterative literal testing with model-based candidate filtering, and both use chunking to test multiple candidates simultaneously. However, the two approaches differ in several fundamental ways that reflect the underlying solver paradigms.

Proof system. CadiBack relies on resolution-based clause learning within CaDiCaL, while RoundingBack uses the cutting planes proof system of RoundingSAT. Cutting planes can produce exponentially shorter proofs than resolution for certain constraint families [14], which is particularly relevant for PBO instances with large coefficients.

Constraint representation. CadiBack operates on clauses (CNF), requiring any non-clausal input to be encoded into SAT. RoundingBack operates natively on pseudo-Boolean constraints, avoiding the potentially exponential blowup of CNF encodings [19].

Model rotation. Both CadiBack and RoundingBack implement model rotation to filter non-backbone candidates without additional solver calls. In CadiBack, flippable literals are identified efficiently using watched literal data structures, traversing

only clauses watched by backbone candidates. In RoundingBack, flippability is determined by checking the slack of each constraint: a variable can be flipped in the current model if no constraint's slack prevents the value change. While both approaches achieve the same goal—identifying variables that can be safely flipped—CadiBack's watched-literal scheme avoids traversing all clauses, providing an algorithmic advantage. Adapting watched-literal techniques to pseudo-Boolean constraints, where flipping a variable affects constraints through weighted sums rather than simple clause satisfaction, remains an open direction for improving RoundingBack's efficiency.

Arithmetic. CadiBack works with Boolean values; RoundingBack operates with exact integer arithmetic over arbitrary-precision coefficients, preserving the numerical properties of PBO instances without floating-point rounding errors.

4.3 Heuristic Variants

The base algorithm leaves two aspects unspecified: the order in which candidates are tested, and how to find diverse models during filtering. We present two heuristic variants addressing these.

4.3.1 RoundingBack-WP: Weighted Propagation Ordering

Candidate ordering affects the number of solver calls. Testing backbone literals first yields UNSAT results that identify multiple backbone members and trigger chunk doubling. Testing non-backbone literals first wastes calls that yield only filtering information.

RoundingBack-WP (Weighted Propagation) orders candidates by their propagation activity during initial optimization. Variables causing many propagations are central to the constraint structure and more likely constrained to fixed values in optimal solutions.

Definition 4.3.1 (Propagation Score)**.** During solving, when a constraint C implies variable v through propagation, the propagation score of v is updated as:

$$\text{score}(v) \mathrel{+}= \frac{\text{coefficient}(v, C)}{\max_u(\text{coefficient}(u, C))} \tag{4.3}$$

where $\text{coefficient}(v, C)$ is the coefficient of v in constraint C.

Normalizing by maximum coefficient prevents propagations from large-coefficient constraints from dominating small-coefficient ones. A score increment of 1 means the variable was dominant in its propagating constraint; near 0 means a minor contribution.

After optimization, candidates are sorted by propagation score in descending order. Variables with high scores—frequently involved in propagations with significant coefficients—are tested first.

Algorithm 4.2 RoundingBack-WP: Weighted Propagation Ordering

Require: PBO instance (φ, f), optimal value z^*
Ensure: Backbone $\mathcal{B}$
1: Initialize score$(u) \leftarrow 0$ for all variables u
2: $\sigma \leftarrow$ solve (φ, f) to optimality, where on each propagation by constraint C:
3: **for** each variable u in C (excluding the propagated variable) **do** ▷ During solving
4: score(u) += coefficient(u, C) / $\max_{w \in C}$ coefficient(w, C)
5: **end for**
6: $\Lambda \leftarrow$ sort candidates by score in descending order
7: $\mathcal{B} \leftarrow$ ExtractBackbone$(\varphi \wedge (f(\mathbf{x}) \leq z^*), \sigma, \Lambda)$ ▷ Algorithm 4.1
8: **return** $\mathcal{B}$

Example 4.3.1. Consider constraints $3x_1 + x_2 + x_3 \geq 4$ and $x_1 + x_4 \geq 1$. If propagation from the first constraint implies x_1, the score increment is $3/3 = 1$. If the second constraint later propagates x_4, its increment is $1/1 = 1$. Despite different coefficient magnitudes, both propagations contribute equally to their respective variables' scores.

This heuristic is lightweight—scores are computed during normal solving with minimal overhead—and gives a principled ordering based on constraint structure rather than arbitrary variable numbering.

4.3.2 RoundingBack-PG: Diversity-Driven Phase Selection

Model-based filtering eliminates candidates falsified by some optimal solution. Filtering works better when models are diverse—if all models found are similar, few candidates get eliminated.

RoundingBack-PG uses the pguide heuristic from Nadel [40] to encourage diversity. The idea is to bias decision polarities against the majority polarity observed in previous solutions.

Definition 4.3.2 (Polarity Potential)**.** For each variable u, maintain counters p_u (solutions with $u = 1$) and n_u (solutions with $u = 0$). The polarity potential is:

$$\Pi_u = p_u - n_u \tag{4.4}$$

During CDCL search, when the solver must choose a polarity for decision variable u:

- If $\Pi_u > 0$, choose $u = 0$ (bias against majority positive).
- If $\Pi_u < 0$, choose $u = 1$ (bias against majority negative).

Algorithm 4.3 RoundingBack-PG: Diversity-Driven Phase Selection

Require: PBO instance (φ, f), optimal value z^*
Ensure: Backbone $\mathcal{B}$
Initialize polarity counters: $p_u \leftarrow 0$, $n_u \leftarrow 0$ for all variables u
$\sigma \leftarrow$ solve (φ, f) to optimality
Update counters from σ: $p_u \mathrel{+}= [\sigma(u) = 1]$, $n_u \mathrel{+}= [\sigma(u) = 0]$
for each decision on variable u during search **do** ▷ PGUIDE polarity selection
    $\Pi_u \leftarrow p_u - n_u$
    if $\Pi_u > 0$ **then**
        choose $u = 0$ ▷ Bias against majority
    else if $\Pi_u < 0$ **then**
        choose $u = 1$
    else
        use default polarity
    end if
end for
On each new model σ': update p_u, n_u from σ'
$\mathcal{B} \leftarrow$ ExtractBackbone$(\varphi \wedge (f(\mathbf{x}) \leq z^*), \sigma, \Lambda)$ ▷ Algorithm 4.1
return $\mathcal{B}$

- If $\Pi_u = 0$, use the default polarity heuristic.

This biasing pushes the solver toward different solution regions. If most solutions so far set $x = 1$, the solver will try $x = 0$, potentially finding a counterexample that eliminates x from candidates.

Polarity counters update each time the solver returns a satisfying assignment (from optimization or SAT chunk queries). Over extraction, pguide steers search toward increasingly diverse solutions.

Example 4.3.2. After finding three solutions where $x_1 = 1$ and one where $x_1 = 0$, we have $\Pi_{x_1} = 3 - 1 = 2 > 0$. The next decision on x_1 will bias toward $x_1 = 0$, increasing the chance of finding a fourth solution that also has $x_1 = 0$—potentially eliminating x_1 from candidates if it was only tentatively included.

RoundingBack-PG combines naturally with RoundingBack-WP. The heuristics address orthogonal aspects: WP determines which candidates to test first; PG influences how the solver searches for counterexamples. Both can be enabled simultaneously.

4.4 Baseline Extractors

We implemented two baseline extractors for comparison: one MILP-based, one SAT-based. Both use the same general algorithm (Algorithm 4.1) but with different

underlying solvers.

4.4.1 GuroBack: MILP-Based Extraction

GuroBack uses Gurobi for backbone extraction, treating PBO as 0-1 ILP. As described in Section 2.6, Gurobi combines Branch-and-Bound with LP relaxations and cutting planes.

PBO to MIP Transformation

PBO to MIP transformation is straightforward: pseudo-Boolean constraints become linear constraints over binary variables. For a PBO constraint $\sum_i c_i \ell_i \geq b$ where literals may be negated, substitute $\bar{x} = 1 - x$ and rearrange:

$$\sum_{i:\ell_i = x_i} c_i x_i + \sum_{i:\ell_i = \neg x_i} c_i (1 - x_i) \geq b \tag{4.5}$$

The objective transforms similarly. Gurobi variables are declared binary ($x_i \in \{0, 1\}$), and constraints are added through the Gurobi C++ API.

Chunk Constraint Implementation

For a chunk Γ, the constraint $\sum_{\ell \in \Gamma} \bar{\ell} \geq 1$ requires at least one literal to differ from the current model. GuroBack adds a temporary linear constraint:

$$\sum_{\ell \in \Gamma : \sigma \models \ell} (1 - x_\ell) + \sum_{\ell \in \Gamma : \sigma \not\models \ell} x_\ell \geq 1 \tag{4.6}$$

where x_ℓ is the variable underlying literal ℓ. The constraint is removed after each query for incremental solving.

Characteristics

For backbone extraction specifically, Gurobi's LP relaxations help quickly establish infeasibility of chunk constraints when literals are backbone members. However, Gurobi does not learn from conflicts in a way that persists across search, and its floating-point arithmetic can cause numerical issues with extreme coefficient ratios.

The implementation is available at `https://github.com/bryan-alvarado-ulloa/guroback`.

4.4.2 NapBack: SAT-Based Extraction

NapBack reduces PBO backbone extraction to SAT backbone extraction through a pipeline: a PBO solver for optimization, NaPS for PB-to-CNF translation, and CadiBack for SAT backbone extraction. As described in Section 2.6, SAT solvers use CDCL with resolution-based learning, which differs fundamentally from the cutting planes reasoning in native PBO solvers.

Pipeline Architecture

The pipeline first solves the PBO instance to get S^* (NapBack accepts the optimal value as input, allowing flexibility in solver choice), then adds the optimality constraint $f(\mathbf{x}) = S^*$ to obtain the decision problem over optimal solutions. NaPS converts all pseudo-Boolean constraints to CNF using ROBDD-based encoding, CadiBack computes the backbone, and results are mapped back to original PBO variables.

NaPS Translation

NaPS [49] translates PB constraints to CNF using Reduced Ordered Binary Decision Diagrams (ROBDDs). For each PB constraint, NaPS builds an ROBDD representing all satisfying assignments, then encodes it as CNF clauses with auxiliary variables for internal nodes.

ROBDD-based encoding ensures unit propagation on the CNF achieves the same inferences as propagation on the original PB constraint (arc consistency). This is stronger than many alternative encodings that lose propagation strength.

However, ROBDD construction can be expensive: diagram size depends on variable ordering and can grow exponentially for constraints with many variables or large coefficients. NaPS includes ordering heuristics but cannot guarantee polynomial-size diagrams.

CadiBack Integration

CadiBack [6] is the state-of-the-art SAT backbone extractor, integrated into CaDiCaL. It provides model rotation via watched literals, incremental solving with persistent learned clauses, and adaptive chunking with refined size selection. Delegating to CadiBack gives NapBack the benefit of years of SAT backbone engineering, though at the cost of CNF translation overhead and restriction to clause-based reasoning.

Characteristics

The modular pipeline allows components to be replaced independently. However, ROBDD translation can be expensive for constraints with many variables or large coefficients, and the original PBO structure becomes obscured in the resulting CNF.

The implementation is available at `https://github.com/bryan-alvarado-ulloa/napback`.

4.4.3 Comparison of Approaches

Table 4.1 summarizes the key differences between our backbone extractors.

Chapter 5 compares these approaches across diverse PBO benchmarks, showing how these trade-offs manifest in practice.

Table 4.1: Comparison of backbone extraction approaches.

Property	**RoundingBack**	**GuroBack**	**NapBack**
Underlying solver	RoundingSAT	Gurobi	CaDiCaL
Constraint representation	Native PB	Linear (MIP)	CNF
Learning	Cutting planes	None	Resolution
Arithmetic	Exact integers	Floating-point	Exact (Boolean)
LP relaxations	Yes (SoPlex)	Yes (native)	No
Translation required	No	Minimal	ROBDD-based

Chapter 5

Experimental Evaluation

This chapter evaluates the backbone extraction methods from Chapter 4 on instances from the Pseudo-Boolean Competition 2024. We ask three questions: (1) How many benchmark instances can we extract backbones for? (2) How does backbone density relate to problem hardness? (3) How do our five extractors compare in coverage and per-domain performance?

Section 5.1 describes the experimental setup. Section 5.2 presents the extraction effort that produced a public dataset of over 8,000 PBO backbones. Section 5.3 analyzes backbone density versus solving difficulty. Section 5.4 compares the five extraction methods on the official evaluation benchmark.

5.1 Experimental Setup

5.1.1 Hardware Infrastructure

We ran all experiments on compute nodes from the Savio computational cluster at UC Berkeley, Department of Industrial Engineering and Operations Research (IEOR), with Intel Xeon E5-2670 v3 processors (12 cores at 2.30 GHz base, 3.10 GHz turbo). Each node has two processors (24 physical cores) and 64 GB of DDR4 RAM.

5.1.2 Software Environment

We used RoundingSAT commit d34b6be, which includes the RoundingBack extension implementing the three variants (RB-Base, RB-WP, RB-PG) described in Chapter 4. Gurobi 10.0.0rc2 was configured with default parameters except for thread count, set to match available cores. NaPS 1.02b5 performed ROBDD-based translation from PBO to CNF. CadiBack 0.2.1, built against CaDiCaL 2.0.0, provides the SAT backbone extraction capability used by NapBack.

For the GuroBack and NapBack baseline extractors, we used the implementations described in Chapter 4, available at the repositories listed there.

5.1.3 Benchmark Selection

Our experiments use instances from the Pseudo-Boolean Competition 2024 (PB24), specifically the OPT-LIN track. This track comprises optimization instances with only linear pseudo-Boolean constraints—no nonlinear terms such as products of variables appear in the objective or constraints. The restriction to linear instances matches the scope of our extraction methods, which assume linear PBO as input.

The complete OPT-LIN dataset contains 14,849 instances spanning diverse application domains. These instances range from small (tens of variables) to very large (millions of variables and constraints). Problem domains include combinatorial optimization problems, constraint satisfaction reductions, hardware verification, and various academic benchmarks.

For the detailed extractor comparison, we use 335 instances from the official PB24 evaluation benchmark. The full evaluation set was used by competition organizers to rank participants; the 335 instances are the subset for which we could compute optimal objective values using our extractors (RoundingBack, GuroBack, NapBack). This allows us to isolate backbone extraction performance from optimization performance.

5.1.4 Time Limits and Resource Constraints

We used a 10-minute limit (600 seconds wall-clock time) for the optimization phase, which determines whether an instance can be solved to optimality within this time limit. The extraction phase received a 1-hour limit (3,600 seconds), as backbone extraction often requires many solver calls.

Memory was limited to 15 GB per instance, running 4 instances in parallel per machine. Instances exceeding either the time or memory limit were marked as unsolved for that experimental phase.

5.2 Backbone Extraction Coverage

We extracted backbones from as many OPT-LIN instances as possible to create a public dataset. This section describes the three-phase methodology and resulting statistics.

5.2.1 Phase 1: Solving to Optimality

The first phase identifies instances that can be solved to proven optimality within practical time limits. Backbone extraction requires knowledge of the optimal objec-

tive value; without it, we cannot constrain the search to optimal solutions only.

We evaluated three solvers—Gurobi, NaPS, and RoundingSAT—on all 14,849 OPT-LIN instances. Each solver was given a 10-minute time limit per instance. An instance was marked as "solved" if at least one solver returned a proven optimal solution within the limit.

This phase identified 10,224 instances (69% of the dataset) as solvable to optimality. The remaining 4,625 instances either timed out on all three solvers, ran out of memory, or encountered solver errors.

5.2.2 Phase 2: Backbone Extraction

For each of the 10,224 solved instances, we assigned the fastest solver to perform backbone extraction using its corresponding extractor: GuroBack for instances where Gurobi was fastest, NapBack for NaPS, and RB-Base for RoundingSAT. The intuition behind this assignment is that the fastest solver likely has internal state (learned clauses, preprocessed constraints, heuristic scores) that will benefit extraction.

Each extractor was given a one-hour time limit per instance. The extraction phase starts from the known optimal value, so the reported times reflect pure backbone extraction without re-solving for optimality.

Of the 10,224 instances attempted, extraction completed within the time limit for 8,351 instances. This represents an 82% success rate on solved instances, or 56% of the original dataset. The remaining 1,873 instances timed out during extraction, typically because the number of backbone candidates was large and many solver calls were required or because each of them took too much time to complete.

5.2.3 Phase 3: Dataset Publication

The resulting backbone dataset is publicly available at:

`https://github.com/matiasfrancia/dataset-pbo-backbones`

For each of the 8,351 instances with extracted backbones, the dataset includes the backbone itself (represented as a set of literals, i.e., variable-polarity pairs), the optimal objective value used for extraction, and metadata including extraction time, extractor used, and instance path within the competition folder structure.

To our knowledge, this is the first large-scale public dataset of PBO backbones. Previous backbone research has focused on SAT instances, and the extension to optimization problems with linear objectives opens new research directions in problem structure analysis and solver engineering.

5.3 Backbone Density Analysis

Backbone density—the proportion of variables in the backbone—reflects problem structure. This section analyzes how backbone density relates to solving difficulty across problem domains.

5.3.1 Methodology

We computed Spearman correlation coefficients between backbone density and solving time for each domain. Spearman correlation is appropriate because solving times have heavy-tailed distributions (robust to outliers) and we care about monotonic relationships rather than linear ones.

For each domain, we selected the best solver among RoundingSAT, Gurobi, and NaPS based on average solving time. This yielded over 8,000 data points with both backbone and runtime information. When the best solver failed on a particular instance but another succeeded, we included the available data.

We used two-tailed tests with $\alpha = 0.05$. Domains with fewer than 10 instances were excluded, leaving 31 domains for analysis.

5.3.2 Results Overview

Of the 31 domains examined, 13 show statistically significant correlations ($p < 0.05$) between backbone density and solving time. These divide into 7 positive correlations (higher density corresponds to longer solving times) and 6 negative correlations (higher density corresponds to shorter solving times).

Figure 5.1 presents these results as a forest plot. Each row represents one domain, with the point estimate showing the Spearman correlation coefficient ρ and error bars indicating 95% confidence intervals. Domains are sorted by correlation strength, with positive correlations at the top and negative at the bottom. Solid markers indicate statistically significant correlations; hollow markers indicate non-significant results.

5.3.3 Domains with Positive Correlation

Several domains show strong positive correlations (denser backbones = longer solving times):

Haplotype Inference ($\rho = 0.51$, $p < 0.001$): Instances with more constrained solution spaces (larger backbones) are harder. When most variables must take fixed values, the solver navigates a narrow feasible region with little flexibility.

Network Routing ($\rho = 0.49$, $p < 0.001$): Routing instances with larger backbones have tightly constrained optimal solutions, possibly corresponding to topologies with few viable routing options.

In these domains, backbone density serves as a proxy for how "tight" the optimization landscape is.

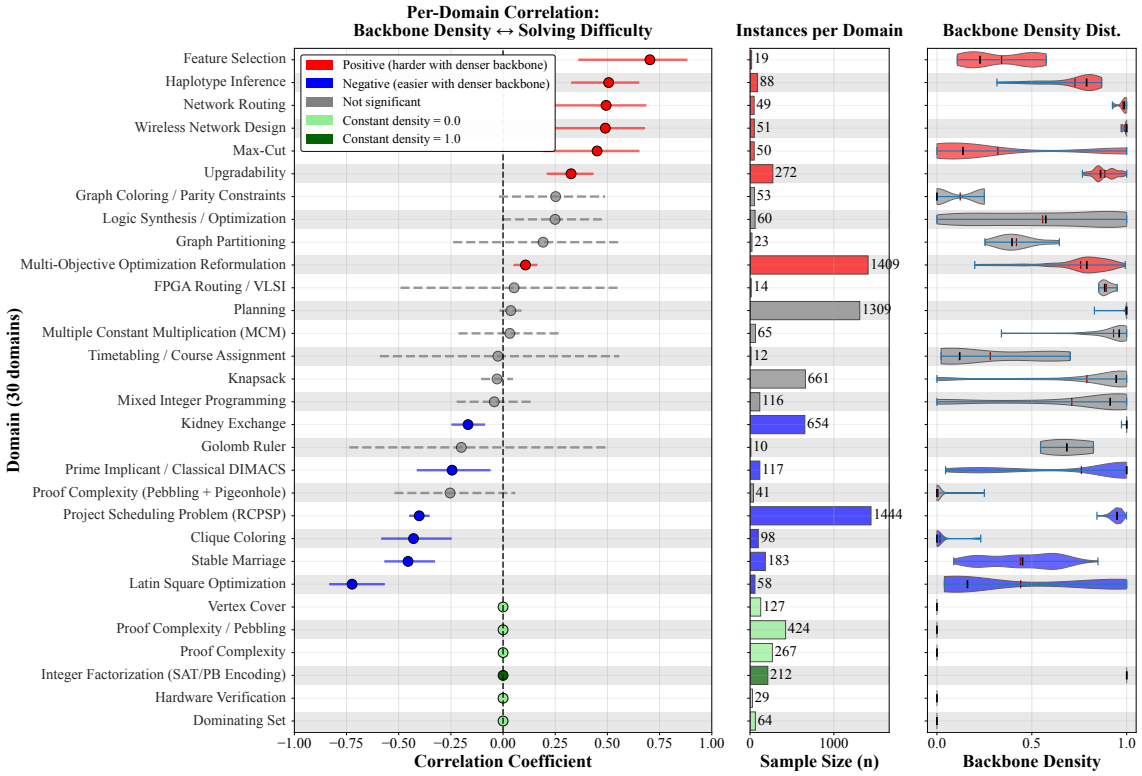


Figure 5.1: Forest plot of Spearman correlations between backbone density and solving time across 31 problem domains. Positive correlations indicate that denser backbones correspond to longer solving times; negative correlations indicate faster solving. Solid markers denote statistically significant correlations ($p < 0.05$). Confidence intervals reflect uncertainty due to finite sample sizes.

5.3.4 Domains with Negative Correlation

Several domains show strong negative correlations (denser backbones = faster solving):

Latin Square Optimization ($\rho = -0.72$, $p < 0.001$): The strongest negative correlation. Latin squares are highly structured, and instances with larger backbones have more redundant constraints that help the solver prune the search space. The solver propagates these implications to derive additional constraints, accelerating convergence.

In these domains, large backbones reflect "easy" structure. The solver exploits constraints through unit propagation and cutting planes, quickly eliminating infeasible regions.

5.3.5 Domains without Significant Correlation

Most domains (18 of 31) show no significant correlation between backbone density and solving time. This does not mean backbone structure is irrelevant—the relationship may be more complex than a monotonic trend.

Some domains appear as non-significant for a trivial reason: constant backbone density. If all instances in a domain have density 0 or density 1, there is no distribution to correlate with solving time. In Figure 5.1, we distinguish these cases visually: light green markers indicate domains where all instances have empty backbones (density = 0), while dark green markers indicate domains where all instances have complete backbones (density = 1). These constant-density domains are "no correlation" by construction, not because the backbone-difficulty relationship is absent.

For domains with varying backbone densities, possible explanations include confounding variables (instance size, constraint density, or coefficient magnitudes may dominate the difficulty relationship, masking any backbone effect), non-monotonic relationships (the true relationship may be U-shaped, with both very sparse and very dense backbones being easy while intermediate densities are hard), or insufficient variation (some domains may have instances with similar backbone densities, providing insufficient variation to detect a correlation).

5.3.6 Bimodal Distribution of Backbone Density

The dataset shows a bimodal distribution of backbone densities. About 11% of instances have empty backbones (density = 0.0), while 14% are fully constrained (density = 1.0). The remaining 75% have intermediate densities, but these cluster toward the extremes rather than around 0.5.

This bimodality suggests two problem classes. Under-constrained problems (empty or near-empty backbones) have many optimal solutions with high flexibility in variable assignments. Any of many variable configurations can achieve optimality, and the backbone captures only those assignments that happen to be shared by all optima. Such instances may arise from loosely specified optimization problems or from encodings where the objective has many global optima. Fully-constrained problems (complete or near-complete backbones) have unique or nearly unique optimal solutions. Every variable (or nearly every variable) is forced to a specific value in any optimal solution. Such instances may arise from tightly specified problems or from encodings where the constraints uniquely determine the optimum.

This bimodality affects extraction algorithm design. Methods optimized for dense backbones (e.g., aggressive chunk doubling) may perform poorly on sparse instances. The adaptive chunking strategy in Chapter 4 addresses this by adjusting to observed backbone density.

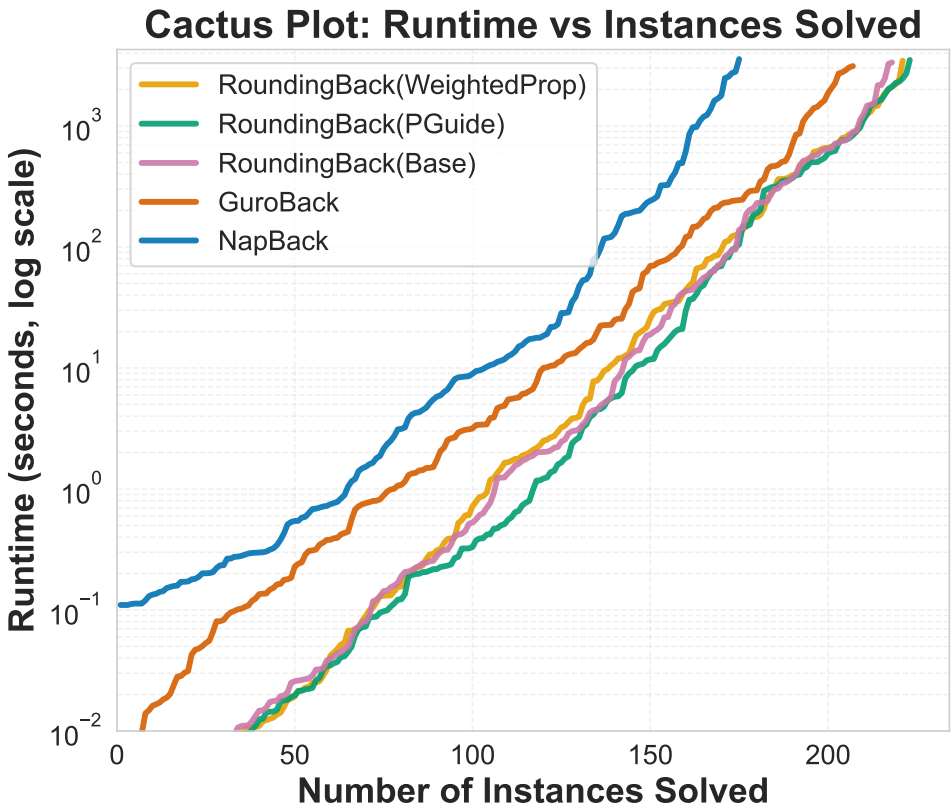


Figure 5.2: Cactus plot comparing backbone extraction performance across five methods. The x-axis shows cumulative instances solved; the y-axis shows cumulative extraction time on a logarithmic scale. RoundingBack variants (RB-PG, RB-WP, RB-Base) achieve the highest coverage, followed by GuroBack and NapBack.

5.4 Extractor Comparison

We tested all five extractors on the 335 PB24 evaluation instances with known optimal values. This benchmark is used to rank competition participants.

We compared RB-Base, RB-WP, RB-PG, GuroBack, and NapBack. Each extractor received the known optimal value at initialization, so timing differences reflect extraction algorithms rather than solver optimization strength.

5.4.1 Overall Coverage Results

Figure 5.2 presents a cactus plot summarizing extraction performance. The x-axis shows the number of instances solved (extraction completed within the one-hour limit), and the y-axis shows cumulative time on a logarithmic scale. Each curve represents one extractor, with steeper curves indicating faster extraction.

The RoundingBack variants achieve the highest coverage: RB-PG leads with 223

instances completed (67% of the benchmark), followed by RB-WP with 221 instances (66%) and RB-Base with 218 instances (65%). The baseline extractors show lower coverage: GuroBack completes 207 instances (62%) and NapBack completes 176 instances (53%).

Gurobi is the fastest solver on this benchmark, but RB-PG (built on RoundingSAT) achieves better extraction coverage. Extraction performance is not just a function of solver strength—the extraction algorithm matters.

The improvements from RB-Base to RB-WP to RB-PG validate the heuristics in Chapter 4. Weighted propagation ordering (WP) and diversity-driven phase selection (PG) each add a few solved instances. While modest, these represent instances at the margin of tractability.

5.4.2 Per-Domain Analysis

Figure 5.3 shows domain-specific performance. Each row is a domain; the x-axis shows the success rate (percentage of instances with completed extraction). Different extractors are distinguished by color in the pie chart markers; marker size indicates instances completed.

Figure 5.3: Per-domain extraction performance across five methods. Each row is a domain; each column an extractor. Marker size indicates the number of instances completed. The plot reveals domain-specific strengths: RoundingBack variants excel in Upgradability and Latin Square, while GuroBack shows unique strength in Max-Cut and Vertex Cover.

RoundingBack variants perform best in several domains. In Upgradability, all RoundingBack variants complete more instances than GuroBack or NapBack, likely

because the constraint structure aligns well with native PB propagation. In Latin Square Optimization, RoundingBack achieves the highest extraction coverage among all extractors; this success is likely attributable to cutting planes reasoning, which exploits the highly structured constraints to derive implications that prune the backbone search space. This is consistent with the strong negative correlation ($\rho = -0.72$) observed in Section 5.3. In Planning, CDCL-style search with linear constraint learning appears well-suited to planning instance structure.

The baseline extractors win in specific domains. In Max-Cut and Vertex Cover, GuroBack completes instances that no RoundingBack variant can handle; these graph optimization problems may benefit from Gurobi's LP relaxations, which can expose backbone implications that SAT-style propagation misses. In Haplotype Inference, NapBack shows strength, consistent with the SAT-friendly structure of these biological inference problems.

5.4.3 Complementarity Analysis

The performance differences show complementarity among extractors. When one method fails, another often succeeds. A portfolio combining RB-PG with GuroBack would cover nearly all observed strengths.

The three RoundingBack variants show marginal differences across domains. All use the same solver and extraction algorithm, differing only in candidate ordering and phase selection. The similar performance suggests these heuristics provide consistent small improvements rather than dramatic domain-specific gains.

5.4.4 High-Variance Domains

Several domains show high variance in extractor performance. Stable Marriage exhibits large performance gaps between RoundingBack and baseline extractors, with neither consistently dominating. Feature Selection shows highly variable extraction times across methods, suggesting instance-specific interactions with algorithm characteristics.

These domains present opportunities for research into why certain approaches succeed or fail, potentially informing specialized extractors or algorithm selection strategies.

5.4.5 Challenging Domains

Some domains remain challenging for all methods, including University of Bologna (MIP) instances, proof complexity constructions (Pebbling + Pigeonhole), and graph benchmarks from BHOSLIB.

These domains represent opportunities for new extraction strategies. The difficulty may stem from inherent hardness, mismatches between instance structure and solver capabilities, or large instance sizes that exhaust the time limit.

5.4.6 Summary of Findings

RoundingBack variants achieve best overall coverage, with RB-PG completing 67% of evaluation instances despite building on a solver that is not the optimization leader. Heuristic improvements provide consistent marginal gains: the progression from RB-Base to RB-WP to RB-PG shows incremental improvements from weighted propagation ordering and diversity-driven phase selection. Extractors exhibit complementary strengths—GuroBack and NapBack complete instances that RoundingBack cannot, particularly in domains where LP relaxations or SAT-based reasoning provide structural advantages. A portfolio combining RB-PG with GuroBack would cover nearly all observed strengths. Some domains remain uniformly challenging, representing opportunities for future algorithmic development.

These findings validate the design choices in Chapter 4. No single approach dominates across all problem types, motivating research into both individual method improvements and portfolio strategies.

Chapter 6

Conclusions and Future Work

This chapter recaps what we did, what we could not do, and what comes next.

6.1 Summary of Contributions

Before this work, no dedicated backbone extractors existed for Pseudo-Boolean Optimization. While backbones have been studied in Boolean Satisfiability, with extractors like CadiBack [6] achieving high coverage on SAT benchmarks, no prior work had developed native PBO extraction techniques. We developed several methods and ran the first large-scale study of backbone properties across PBO domains.

The main contribution is RoundingBack, a backbone extractor built directly on the RoundingSAT PBO solver (Chapter 4, Section 4.2). Unlike approaches that translate PBO to SAT or reformulate as MILP, RoundingBack operates natively on pseudo-Boolean constraints, preserving problem structure, avoiding encoding overhead, and using RoundingSAT's cutting planes reasoning—a proof system exponentially stronger than resolution-based learning in SAT solvers. The implementation uses incremental solving to process the many queries required for backbone extraction, keeping learned constraints across solver calls.

We also developed two heuristic variants that improve extraction efficiency (Section 4.3). RoundingBack-WP orders candidate literals by their propagation activity during optimization, prioritizing variables that frequently participate in constraint propagation with large coefficients; this ordering tends to test backbone literals earlier, enabling more aggressive chunk doubling. RoundingBack-PG uses the pguide heuristic to encourage solution diversity by biasing decision polarities against majority assignments from previous solutions, generating more varied models that speed up candidate filtering. In our experiments, RB-PG achieved the highest extraction coverage among all methods.

To contextualize RoundingBack's performance, we implemented two baseline extractors representing alternative paradigms (Section 4.4). GuroBack uses the commercial Gurobi MILP solver with branch-and-bound and LP relaxations, while Nap-

Back implements a pipeline approach that translates PBO constraints to CNF using NaPS and delegates backbone extraction to CadiBack. These baselines enable comparison across native PBO, MILP, and SAT-based solving paradigms.

We ran experiments on 8,351 instances from the PBO Competition 2024 OPT-Lin benchmark (Chapter 5), conducting the first systematic study of backbone extraction across PBO domains. RB-PG achieved 67% extraction coverage, outperforming both GuroBack and NapBack. The analysis revealed domain-specific complementarity: different extractors succeed on different instance families, with each method performing best on some domains, suggesting that no single extraction paradigm dominates universally.

Beyond extraction performance, we analyzed the relationship between backbone density and problem characteristics (Section 5.3). Backbone density shows Spearman correlation with solving time for some domains, supporting the theoretical connection between backbone size and problem hardness observed in SAT [38]. We also found a bimodal distribution in backbone densities, with instances clustering into under-constrained and over-constrained regimes.

Finally, we release the backbone dataset for all 8,351 competition instances alongside our implementations. This dataset can support future research in learning-based optimization, algorithm selection, and structural analysis of PBO instances. To our knowledge, this is the largest publicly available collection of PBO backbones.

A preliminary version of this work was published at ICAART 2026 [21]. This thesis extends the conference paper with a broader background study and a more detailed literature review.

6.2 Hypothesis Verification

We now revisit the hypothesis stated in Section 1.1:

> *Implementing a native backbone extractor for PBO reduces extraction time compared to methods that require prior conversion of the problem to SAT. Additionally, variable ordering heuristics reduce computation time compared to the original algorithm without specific ordering.*

The experimental results support both parts of this hypothesis.

Regarding the first claim, RoundingBack operates natively on pseudo-Boolean constraints and avoids the overhead of converting PBO instances to SAT. NapBack, which implements the conversion pipeline using NaPS and CadiBack, requires encoding pseudo-Boolean constraints into CNF—a process that can produce exponentially larger representations [19]. Our experiments show that RoundingBack achieves higher extraction coverage than NapBack on the OPT-Lin benchmark, confirming that native PBO extraction is more effective than the SAT-conversion approach. The advantage is particularly pronounced on instances with large coefficients or complex constraint structures, where SAT encodings become expensive.

Regarding the second claim, both heuristic variants—RB-WP (weighted-propagation ordering) and RB-PG (diversity-driven phase guidance)—outperform the base RoundingBack configuration. RB-PG achieved 67% extraction coverage, the highest among all evaluated methods, by generating more diverse solutions that accelerate candidate filtering through the adaptive chunking mechanism. RB-WP also improves over the baseline by prioritizing high-activity variables, enabling faster backbone identification through more aggressive chunk doubling. These results confirm that variable ordering heuristics meaningfully reduce computation time relative to the unguided algorithm.

In summary, both parts of the hypothesis are verified by the experimental evidence presented in Chapter 5.

6.3 Limitations

We acknowledge several limitations of this work.

Backbone extraction is inherently expensive, requiring multiple solver calls beyond what is needed to find a single optimal solution. Even with adaptive chunking and model-based filtering, extraction can timeout on instances where optimization alone succeeds quickly, limiting practical use as preprocessing in time-critical applications. The 67% coverage achieved by RB-PG, while the highest among evaluated methods, leaves room for improvement on difficult instances.

Our evaluation focuses exclusively on the OPT-Lin track of the PBO Competition 2024, which contains instances with linear pseudo-Boolean constraints and linear objectives. The competition also includes tracks with non-linear constraints and satisfiability variants; backbone behavior on these alternative formulations may differ, and our findings may not generalize to instances with non-linear structure or those drawn from other sources.

The strong domain-specific performance patterns we observed indicate that conclusions about method superiority may not transfer across application areas—an extractor that excels on scheduling instances may perform poorly on circuit verification problems. Finally, all experiments were conducted on a single hardware configuration with fixed timeout parameters; performance rankings could shift on different hardware, and we did not evaluate parallel implementations that might alter relative performance (Gurobi, in particular, supports parallel solving).

6.4 Future Work

Several research directions follow from this work.

The complementarity among extractors suggests portfolio methods that combine multiple approaches, either running extractors in parallel and terminating when any succeeds, or using instance features to select the most promising extractor. Given

that each method performs best on different domain subsets, even simple portfolio strategies may improve overall coverage, while more sophisticated approaches could allocate time budgets dynamically based on early solver feedback.

Machine learning offers several promising directions. The domain-specific performance patterns suggest that instance features (constraint density, coefficient statistics, objective structure) could predict which extractor will succeed, enabling efficient selection without running all methods. Going further, neural models could predict backbone membership directly from instance structure, as has been demonstrated for SAT [56]; extending such approaches to PBO would require handling the richer constraint structure and objective function, but predicted backbones could guide solver search or provide approximate structural indicators even if imperfect. Our density analysis showing correlations between backbone properties and problem difficulty could also support backbone-based hardness classifiers for solver configuration.

Integrating backbone information into solver pipelines could improve optimization performance. If backbone literals can be identified early through prediction or partial extraction, fixing these variables reduces the effective problem size, and backbone-guided branching could prioritize decisions on likely backbone variables. The challenge is achieving speedup without extraction overhead that exceeds the savings. For applications where complete backbones are unnecessary, approximate methods that quickly identify a subset of backbone literals, or incremental extraction that returns partial results as they become available, might sacrifice completeness for practical utility in time-constrained settings.

Expanding experiments to additional PBO tracks (non-linear constraints, decision problems) and other benchmark sources like MIPLIB would test the generality of our methods and might reveal different backbone patterns or identify universal extraction strategies.

The methods and datasets from this thesis provide a starting point for these investigations.

Bibliography

[1] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. *AAAI/IAAI*, 2000:256–261, 2000.

[2] T. Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

[3] A. Albert, F. S. Leira, and L. S. Imsland. Uav path planning using milp with experiments. In *2017 3rd International Conference on Control, Automation and Robotics (ICCAR)*, pages 566–571. IEEE, 2017.

[4] F. Arito, F. Chicano, and E. Alba. On the application of sat solvers to the test suite minimization problem. In *International Symposium on Search Based Software Engineering*, pages 45–59. Springer, 2012.

[5] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[6] A. Biere, N. Froleyks, and W. Wang. Cadiback: Extracting backbones with cadical. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[7] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[8] S. Bolusani, M. Besançon, K. Bestuzheva, A. Chmiela, J. Dionísio, T. Donkiewicz, J. van Doornmalen, L. Eifler, M. Ghannam, A. Gleixner, et al. The scip optimization suite 9.0. *arXiv preprint arXiv:2402.17702*, 2024.

[9] E. Boros and P. L. Hammer. Pseudo-boolean optimization. *Discrete applied mathematics*, 123(1-3):155–225, 2002.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[11] J. Cai, T. Huang, and B. Dilkina. Learning backdoors for mixed integer programs with contrastive learning. *arXiv preprint arXiv:2401.10467*, 2024.

[12] W. Chen and D. Whitley. Decomposing sat instances with pseudo backbones. In *Evolutionary Computation in Combinatorial Optimization (EvoCOP)*, pages 75–90. Springer, 2017.

[13] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[14] W. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.

[15] O. Coudert and J. C. Madre. New ideas for solving covering problems. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 641–646, 1995.

[16] R. de Haan, I. Kanj, and S. Szeider. Local backbones. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 377–393. Springer, 2013.

[17] J. Devriendt, S. Gocht, E. Demirović, J. Nordström, and P. J. Stuckey. Cutting to the core of pseudo-boolean optimization: Combining core-guided search with cutting planes reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3750–3758, 2021.

[18] B. Dilkina, C. P. Gomes, Y. Malitsky, A. Sabharwal, and M. Sellmann. Backdoors to combinatorial optimization: Feasibility and optimality. In *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 56–70. Springer, 2009.

[19] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[20] J. Elffers and J. Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *IJCAI*, volume 18, pages 1291–1299, 2018.

[21] M. Francia-Carramiñana, B. Alvarado-Ulloa, D. Hochbaum, B. Dilkina, R. Ñanculef, and R. Asín-Achá. Backbones in pseudo-boolean optimization: Extraction and analysis. In *Proceedings of the 18th International Conference on Agents and Artificial Intelligence - Volume 5: ICAART*, pages 4697–4705. INSTICC, SciTePress, 2026.

[22] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019.

[23] R. E. Gomory. An algorithm for integer solutions to linear programs. In *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, 1958.

[24] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[25] A. Haken. The intractability of resolution. *Theoretical computer science*, 39:297–308, 1985.

[26] Q. Han, L. Yang, Q. Chen, X. Zhou, D. Zhang, A. Wang, R. Sun, and X. Luo. A gnn-guided predict-and-search framework for mixed-integer linear programming. In *The Eleventh International Conference on Learning Representations*, 2023.

[27] T. Huang, A. M. Ferber, Y. Tian, B. Dilkina, and B. Steiner. Searching large neighborhoods for integer linear programs with contrastive learning. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 13869–13890. PMLR, 2023.

[28] T. Huang, A. M. Ferber, A. Zharmagambetov, Y. Tian, and B. Dilkina. Contrastive predict-and-search for mixed integer linear programs. In *Forty-first International Conference on Machine Learning*, 2024.

[29] M. Janota, I. Lynce, and J. Marques-Silva. Algorithms for computing backbones of propositional formulae. *Ai Communications*, 28(2):161–177, 2015.

[30] H. Jiang, J. Xuan, and Y. Hu. Approximating the backbone in the weighted maximum satisfiability problem. In *Advances in Knowledge Discovery and Data Mining (PAKDD)*, pages 882–889. Springer, 2008.

[31] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[32] P. Kilby, J. Slaney, S. Thiébaux, T. Walsh, et al. Backbones and backdoors in satisfiability. In *AAAI*, volume 5, pages 1368–1373, 2005.

[33] A. H. Land and A. G. Doig. *An automatic method for solving discrete programming problems*. Springer, 2010.

[34] S. Liao and S. Devadas. Solving covering problems using lpr-based lower bounds. In *Proceedings of the 34th annual Design Automation Conference*, pages 117–120, 1997.

[35] H. Marchand and L. A. Wolsey. Aggregation and mixed integer rounding to solve mips. *Operations research*, 49(3):363–371, 2001.

[36] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[37] I. Mironov and L. Zhang. Applications of sat solvers to cryptanalysis of hash functions. In *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*, pages 102–115. Springer, 2006.

[38] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137, 1999.

[39] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[40] A. Nadel. Generating diverse solutions in sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 287–301. Springer, 2011.

[41] N. Natarajan, I. S. Dhillon, P. Ravikumar, and A. Tewari. Learning with noisy labels. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1196–1204, 2013.

[42] R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and R. Zhao. Speeding up pseudo-boolean propagation. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[43] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[44] A. J. Parkes. Clustering at the phase transition. In *AAAI/IAAI*, pages 340–345. Citeseer, 1997.

[45] C. Pezo, D. S. Hochbaum, J. Godoy, and R. J. As'in Ach'a. Automatic algorithm selection for pseudo-boolean optimization with given computational time limits. *Comput. Oper. Res.*, 173:106836, 2025.

[46] A. Previti, A. Ignatiev, M. Järvisalo, and J. Marques-Silva. On computing generalized backbones. In *Proceedings of the 29th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1050–1056. IEEE, 2017.

[47] B. C. Ribas, R. M. Suguimoto, R. A. Montano, F. Silva, L. de Bona, and M. A. Castilho. On modelling virtual machine consolidation to pseudo-boolean

constraints. In *Advances in Artificial Intelligence–IBERAMIA 2012: 13th Ibero-American Conference on AI, Cartagena de Indias, Colombia, November 13-16, 2012. Proceedings 13*, pages 361–370. Springer, 2012.

[48] D. Rolnick, A. Veit, S. Belongie, and N. Shavit. Deep learning is robust to massive label noise. *arXiv preprint arXiv:1705.10694*, 2017.

[49] M. Sakai and H. Nabeshima. Construction of an robdd for a pb-constraint in band form and related techniques for pb-solvers. *IEICE TRANSACTIONS on Information and Systems*, 98(6):1121–1127, 2015.

[50] J. Schneider, C. Froschhammer, I. Morgenstern, T. Husslein, and J. M. Singer. Searching for backbones — an efficient parallel algorithm for the traveling salesman problem. *Computer Physics Communications*, 96(2):173–188, 1996.

[51] G. K. Silori and S. Khanam. Performance analyses of lp and milp solvers based on newly introduced scale: Case studies of water network problems in chemical processes. *Chemical Engineering Research and Design*, 136:417–430, 2018.

[52] J. Slaney, T. Walsh, et al. Backbones in optimization and approximation. In *IJCAI*, pages 254–259, 2001.

[53] K. Smith-Miles, D. Baatar, B. Wreford, and R. Lewis. Towards objective measures of algorithm performance across instance space. *Computers & Operations Research*, 45:12–24, 2014.

[54] S. Strassl and N. Musliu. Instance space analysis and algorithm selection for the job shop scheduling problem. *Computers & Operations Research*, 141:105661, 2022.

[55] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning: 2. Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1983.

[56] W. Wang, Y. Hu, M. Tiwari, S. Khurshid, K. L. McMillan, and R. Miikkulainen. Neuroback: Improving CDCL SAT solving using graph neural networks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[57] R. Wille, H. Zhang, and R. Drechsler. Atpg for reversible circuits using simulation, boolean satisfiability, and pseudo boolean optimization. In *2011 IEEE Computer Society Annual Symposium on VLSI*, pages 120–125. IEEE, 2011.

[58] L. A. Wolsey and G. L. Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 1999.

[59] W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1179–1186, 2003.

[60] W. Zhang and Z. Zhang. A backbone based co-evolutionary heuristic for partial max-sat. In *Artificial Intelligence: Methodology, Systems, and Applications*, pages 75–84. Springer, 2004.

Appendix A

Reproducibility and Code Availability

A.1 Hardware Specifications

All experiments were conducted on compute nodes with Intel Xeon E5-2670 v3 processors (24 cores per node) and 64 GB RAM.

A.2 Software Versions

- RoundingSAT: commit hash d34b6be
- Gurobi: v10.0.0rc2
- NaPS: 1.02b5
- CadiBack: 0.2.1 (CaDiCaL version 2.0.0)

A.3 Code Repositories

The source code for all extractors is publicly available:

- RoundingBack: `https://github.com/matiasfrancia/roundingback`
- GuroBack: `https://github.com/bryan-alvarado-ulloa/guroback`
- NapBack: `https://github.com/bryan-alvarado-ulloa/napback`
- Backbone dataset: `https://github.com/matiasfrancia/dataset-pbo-backbones`