HitPBO: An Implicit Hitting Set Solver for Pseudo-Boolean Optimization

Hannes Ihalainen
Department of Computer Science, University of Helsinki, Finland

Dieter Vandesande
Department of Computer Science, Vrije Universiteit Brussel, Belgium
Department of Computer Science, KU Leuven, Belgium

André Schidler
Chair of Computer Architecture, University of Freiburg, Germany

Jeremias Berg
Department of Computer Science, University of Helsinki, Finland

Matti Järvisalo
Department of Computer Science, University of Helsinki, Finland

Abstract

We describe HitPBO 1.0, a from-scratch open-source C++ implementation of the implicit hitting set (IHS) approach to pseudo-Boolean optimization. Compared to earlier implementations, HitPBO adds a range of functionalities and search techniques, certificates, and support for various alternative solvers within IHS. We give an overview of the solver's architecture and its functionalities.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Constraint and logic programming

Keywords and phrases Pseudo-Boolean optimization, implicit hitting set approach, solvers

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.34

Category Tool Paper

Supplementary Material *Software (Source Code)*: `https://bitbucket.org/coreo-group/hitpbo` archived at `swh:1:dir:286aa67b406061f690777757a8b0a716c956b310`

Funding *Hannes Ihalainen*: Funded by University of Helsinki Doctoral Program in Computer Science and Research Council of Finland grant 356046.
Dieter Vandesande: Funded by the Fonds Wetenschappelijk Onderzoek – Vlaanderen fellowship 11A5J25N.
André Schidler: Funded by German Research Foundation (DFG) through grant INST 35/1597-1 FUGG.
Jeremias Berg: Funded by Research Council of Finland grant 362987.
Matti Järvisalo: Funded by Research Council of Finland grant 356046.

Acknowledgements The authors wish to thank the Finnish Computing Competence Infrastructure (FCCI) and the state of Baden-Württemberg through bwHPC for supporting this project with computational and data storage resources.

1 Introduction

Pseudo-Boolean optimization (PBO) refers to minimizing a linear objective function subject to linear constraints over Boolean variables with integer coefficients, and is also known as 0-1 integer programming. Situated between integer-linear programming (ILP) and maximum satisfiability (MaxSAT) as an important subclass of the former and a key generalization

of the latter, PBO finds motivations through various real-world applications, and can be approached by ILP methods [13, 37], CDCL-like reasoning-based methods [19, 17, 34, 35], and translations to propositional clauses [33, 7]. Interest in developing new solvers for PBO is on the rise, as witnessed, e.g., by the recent PB Competitions [28]. We describe HitPBO 1.0, arguably a state-of-the-art implementation of the implicit hitting set (IHS) approach to PBO. HitPBO is a from-scratch, open-source C++ reimplementation of the earlier Python-based PBO-IHS [34, 35] solver, and also goes beyond its earlier beta versions [25, 26].

A rudimentary IHS approach works by (i) computing constraints over the objective variables implied by the constraints of a given instance and (ii) computing optimal ways of satisfying the so-far extracted implied constraints, until an optimal solution to the original instance is found. The negations of the computed implied constraints are typically called unsatisfiable cores – hence, we refer to (i) as *core extraction*. On the other hand, (ii) is referred to as *hitting set computation* since in the special case that all implied constraints are at-least-one constraints over positive terms with unit coefficients – i.e., purely positive propositional clauses – the constraints constitute a hitting set problem. HitPBO implements a noticeably more complex IHS approach, integrating various search techniques typical to IHS implementations in different settings [32, 15, 5], as well as novel ways – from using ILP and PBO solvers to greedy approximation algorithms and stochastic local search – for hitting set computations. HitPBO also offers IHS-specific techniques for discovering symmetric cores, avoiding unnecessary core extraction, and can provide proof logs in the VeriPB 3 format [22, 9] as optimality certificates. Finally, HitPBO integrates various alternative solvers for use in core extraction and in hitting set computation. We provide a comprehensive overview of the features and functionalities of HitPBO as well as the interface and various command line options HitPBO reveals for controlling its internal search. HitPBO is available in open source at `https://bitbucket.org/coreo-group/hitpbo`.

PBO. A literal ℓ is a Boolean $(0-1)$ variable or its negation. An objective $\mathcal{O} \equiv \sum_i c_i\ell_i + L$ is a sum of terms and an integer constant L (each term consisting of a literal ℓ_i and a positive coefficient c_i). A pseudo-Boolean formula F is a set of pseudo-Boolean constraints, i.e., linear inequalities over literals, each of which can be assumed to be of the form $C \equiv \sum_i c_i\ell_i \geq B$ for positive constants c_i and B and literals ℓ_i. C is satisfied by an assignment τ (denoted $\tau(C) = 1$) if $\sum_i c_i\tau(\ell_i) \geq B$; τ is a solution of F if it satisfies each constraint in F. Pseudo-Boolean optimization (PBO) is the problem of minimizing $\mathcal{O}$ over the solutions of F. A constraint $C \equiv \sum_i c_i\ell_i \geq B$ is a *core* (constraint) of a PBO instance $(F, \mathcal{O})$ if (i) each ℓ_i is over a variable that appears in $\mathcal{O}$, and (ii) C is entailed by F, i.e., satisfied by all solutions of F.

2 Overview of the HitPBO Solver and its Architecture

The implicit hitting set (IHS) approach to computing an optimal (minimum-cost) solution to a PBO instance $(F, \mathcal{O})$ iteratively uses two subroutines, a *(hitting set) optimizer* and a *core extractor*. On each iteration, the hitting set optimizer computes a solution (a "hitting set") δ to a set κ of cores of $(F, \mathcal{O})$. Since κ is a relaxation of F, the cost of any minimum-cost (wrt. $\mathcal{O}$) hitting set constitutes a lower bound on the optimal (minimum) cost of solutions of F [34]. Then, the core extractor is queried for an extension τ of δ to a solution of F; if one exists, it has the same cost as δ. Since δ is minimum-cost, τ is optimal for $(F, \mathcal{O})$. Otherwise, the core extractor returns a new core C of $(F, \mathcal{O})$ that is not satisfied by δ. Then C is added to κ, and the optimizer is called again. Towards practical efficiency, as we will

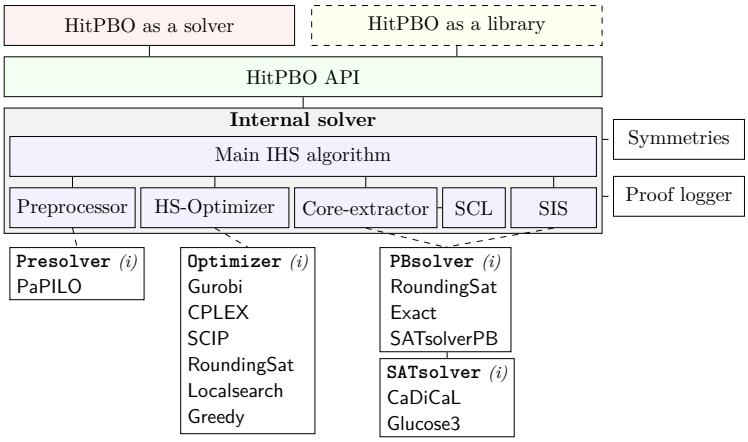

Figure 1 The architecture of the HitPBO solver. Interfaces are marked with (i).

describe, HitPBO extends IHS with a range of techniques and refinements that allow, e.g., the optimizer to compute in different ways non-optimal hitting sets that result either in more cores or improvements to the best-known solution of $(F, \mathcal{O})$; computation of multiple cores per hitting set; as well as preprocessing, further core simplification and certified solving.

Figure 1 overviews the structure of HitPBO. The design aims for modularity, extensibility and simple exchange of different components of the IHS search for PBO. HitPBO can be used as a library and as a stand-alone executable. The internal solver is in both cases invoked via the HitPBO interface (Section 6). The internal solver is divided into components, each implementing specific search techniques and data structures, and managing their backend solvers. **Main IHS algorithm** controls the whole search. **Core-extractor** and **SCL** (Section 3) handle core extraction extended with symmetric core learning (SCL) [25], **HS-Optimizer** (Section 4) handles hitting set optimization; **Preprocessor** (Section 5) simplification and reformulation techniques before the main search and solution reconstruction after search; and **SIS** solution-improving search (Section 5) heuristically invoked during IHS search [35]. The main external libraries and backend solvers are listed in the transparent boxes of Figure 1. The interfaces through which the main components interact with the backend solvers are marked with *(i)*. For example, the **Core-extractor** component interacts with instantiations of the `PBsolver` interface. Proofs of optimality in "Proof logger" and the computation of symmetries for SCL and symmetry breaking in "Symmetries" are implemented via external libraries (Section 5).

3 Core Extraction

HitPBO implements several different strategies for extracting and simplifying multiple cores that are falsified by a given hitting set. This core extraction is divided into two nested procedures. *The outer procedure* maintains and updates a set of assumptions (initialized to the given hitting set) under which cores are extracted. *The inner procedure* invokes a decision procedure for computing cores under the given assumptions and returns either a set of new cores or a solution that extends the assumptions. When a set of cores is found, the inner procedure ends with a post-processing step that refines the set and simplifies the cores, before returning to the *outer procedure* that refines the assumptions based on the processed set of cores. Core extraction terminates when no new cores are found after an update to the assumptions.

Weight-Aware Core Extraction for PBO. The strategy for updating assumptions in the outer procedure can be configured using the parameter `--ce-strategy=<strategy>`. The default strategy is – to the best of our understanding – the first PBO-lifting of weight-aware core extraction (WCE) from MaxSAT [6]. (While WCE for PBO was mentioned in [17], the approach there is simpler.) Under WCE, the outer procedure maintains a positive score for each assumption, initialized to the corresponding coefficient in the objective function. Intuitively, the scores approximate the additional cost that can be incurred on an assumption by hitting sets that satisfy the cores already computed in the current iteration. Given a new set κ of cores, the outer procedure reduces the scores of some assumptions and removes score-0 ones. Score reduction is based on approximating for each $C \in \kappa$ the marginal cost incurred by satisfying C. For a clausal (at-least-one) core C, the scores of its literals are reduced by the minimum (current) score of each literal in the core. For a PB constraint C, we compute a clausal core C' for which (i) all literals in C' have a positive score, (ii) C' is implied by C, and (iii) C' has a minimum cardinality among the clauses that satisfy (i) and (ii). C' is then used to reduce the scores. C is processed until no more clausal cores implied by C with positive scores are found. Specifically, consider a core $C = \sum_{i=1}^{k} c_i\ell_i + \sum_{i=k+1}^{n} c_i\ell_i \geq b$. Assume that (i) ℓ_i is 0-score for $i \leq k$, (ii) ℓ_i has a positive score for $i \geq k+1$, and (iii) $c_i \leq c_{i+1}$ for $i \geq k+1$. Let j be the smallest index for which $\sum_{i=1}^{j} c_j \geq b$. If $j \leq k$, C does not imply any clauses that have a positive score. Otherwise, the clausal core we use to update scores is $\sum_{i=j}^{n} \ell_i \geq 1$.

Assumption Shuffling. The inner procedure of core extraction makes use of shuffling [35] to obtain several cores falsified by the current assumptions. Concretely, after obtaining a core, the core-extracting decision procedure is invoked again under a different randomized order of the same assumptions until *nb_shuffles* (parameter `--ce-shuffles=<int>`, set to 10 by default) new cores have been obtained.

Core Processing and Simplification. After assumption shuffling, the inner procedure performs a post-processing step on the set of cores obtained. The post-processing aims to reduce the sizes of coefficients in the cores and to remove redundant terms using a range of simplification rules (detailed formally in Appendix A), including, e.g., extended at-least-k detection, division of coefficients by the greatest common divisor, and an application of the Chvátal-Gomory cut rule [11, 2]. After processing all cores, the cores that are *subsumed* by others are removed: for two cores $C_1 = \sum_{\ell \in S_1} a^1_\ell \ell \geq B_1$ and $C_2 = \sum_{\ell \in S_2} a^2_\ell \ell \geq B_2$, C_1 *subsumes* C_2, if (i) $S_1 \subseteq S_2$, (ii) $B_1 \geq B_2$, and (iii) $a^1_\ell \leq a^2_\ell$ for all $\ell \in S_1$. Finally, the remaining set of cores is shrunk by removing the cores with the highest number of terms until *nb_keep* (by default 5) cores are left, or all the cores have at most *keep_always* (default 2) terms.

Symmetric Core Learning. HitPBO implements symmetric core learning (SCL; see [25] for details) for optionally generating symmetric cores at each core-extraction iteration based on instance symmetries computed with BreakID [16] before the main search. Two variants are available: explicit SCL (enabled with `--explicit-scl`) that directly generates symmetric cores, and compact SCL (enabled with `--compact-scl`) that employs different encodings to represent compactly sets of symmetric cores.

Instantiations of Core Extraction in HitPBO. Decision procedures for core extraction are instantiated through `PBsolver` (Figure 1). The underlying solver can be selected with parameter `--ce-oracle=<solver>`. HitPBO currently offers as PB solvers a modified

version of RoundingSat 2 [19] (forked from [31] commit c548e10, with modifications related to e.g., proof logging, modifying the interface, and using `int64_t` instead of `long long`; the core solver remains untouched) and unmodified Exact [19] ([20] commit 5da02b6) as well as the SAT solvers CaDiCaL 3.0.0 [8] and Glucose 3 [4] for core extraction. Use of SAT solvers is instantiated via SATsolverPB within the `PBsolver` interface, setting parameter `--ce-oracle=satsolver`. SATsolverPB maintains an instantiation of a SAT solver and an instantiation of a PB solver. The PB solver considers the PB constraints F as-is, while the SAT solver works on clausal constraints. SATsolverPB can be configured to identify non-clausal constraints that can be compactly represented as clauses, including at-least-$(n-1)$ constraints as well as specific types of big-M constraints.[1] For each PB constraint that is not represented as clauses exactly, SATsolverPB adds an implied clausal approximation of the constraint to the SAT solver. If all constraints in F are exactly represented in the SAT solver, SATsolverPB relies only on the SAT solver to decide satisfiability and to extract cores and solutions. Otherwise, when given a new set of assumptions as a query, SATsolverPB first invokes the SAT solver with those assumptions. The PB decision procedure is then invoked if the SAT solver cannot find cores, and satisfiability is reported when the PB solver reports satisfiability.

4 Hitting Set Optimization

Hitting set computations are realized via various instantiations of the abstract `Optimizer`.

Termination Conditions. HitPBO employs a heuristic early-termination strategy for hitting set optimization. Assume *UB* is the current best-known cost of solutions for the instance being solved. The early termination of any instantiation of `Optimizer` is controlled by the *bound_cost* parameter given by HitPBO to the `Optimizer` in each invocation. In the default configuration *bound_cost* $= UB - 1$, except when an optimal hitting set is sought for. `Optimizer` is allowed to terminate and return its best known hitting set δ when one of the following *termination conditions* is met: (1) δ has cost at most *bound_cost* or (2) δ is minimum-cost, or (3) the lower bound on the cost of any hitting set is proven to be at least *UB*. Conditions (1)–(2) ensure that the next core-extraction step will either result in new cores or a solution with a cost less than *UB*. Condition (3) allows `Optimizer` to terminate without returning a new hitting set when *UB* is the optimal cost of the instance. HitPBO will also periodically force `Optimizer` to return a minimum-cost hitting set by invoking it with a low value for *bound_cost*. By default, optimal hitting sets are required whenever the latest hitting set did not result in new cores, or if `Optimizer` has been called *nonopt_iter_limit* (default 1000) times without it returning an optimal hitting set.

Hitting Set Computation with ILP and PB Solvers. HitPBO offers two types of instantiations of `Optimizer` based on algorithms guaranteed to eventually meet the termination conditions (1)–(3) on any input: ones based on ILP solvers (commonly used by IHS solvers to compute hitting sets), and ones based on different types of optimization algorithms using a PB solver [26]. The ILP solvers supported are Gurobi [23] (set `--optimizer=gurobi`),

[1] Big-M refers to the standard modelling technique (see e.g. [10]) where a term Mx, with M a suitably chosen large integer value ("big-M") and x a binary variable, is used for making a constraint conditional on the value of x. This standard "trick" can be used for e.g. representing logical constraints with linear inequalities. In our context, SATsolverPB can currently identify big-M representations of logical constraints of the form $x \vee (y_1 \wedge y_2 \wedge \cdots \wedge y_n)$.

CPLEX [12] (`--optimizer=cplex`), and SCIP [1, 24] (`--optimizer=scip`) with many of their parameters revealed, including parameters related to integer feasibility tolerance, and a parameter that runs SCIP in numerically exact solving mode [18, 24] (`--scip-exact` when SCIP is the selected optimizer). A key motivation for the PB-based optimization algorithms is that they enable proof logging and thereby certifiable exactness (Section 5). The PB-based algorithms implemented use RoundingSat [19] as a decision procedure: (i) core-guided optimization (`--optimizer=roundingsat-cg`), and two variants of SIS: (ii) one that uses reified constraints to circumvent the need to restart the PB solver between hitting-set computations (`--optimizer=roundingsat-ss`), and (iii) one that instead restarts (`--optimizer=roundingsat-sh`). The SIS variants are from-scratch implementations. For core-guided optimization, HitPBO uses the PBO-OLL-rs solver from PB Competition 2024 [30].

Heuristic Hitting Set Computation by Local Search or a Greedy Algorithm. HitPBO implements two heuristic instantiations of `Optimizer`: Localsearch based on stochastic local search, and Greedy as a greedy approach. The heuristic algorithms can only detect termination condition (1), i.e., terminate when the current hitting set has a cost lower than *bound_cost*. To guarantee termination on any input, Localsearch and Greedy internally invoke another instantiation of `Optimizer` as a fallback, in cases they fail to find a hitting set that meets the termination criterion. The fallback optimizer is user-configurable. Localsearch implements a lightweight stochastic local search (SLS) that is an adaptation of NuPBO [38] geared towards quickly adapting an existing hitting set to newly added constraints and seeking diverse (measured by Hamming distance) hitting sets that yield diverse cores. Localsearch offers several parameters that allow fine-tuning SLS behaviour and parameters that adjust the interaction between SLS and its fallback optimizer, e.g., how much time is invested in SLS and how often the fallback optimizer is called. Greedy implements a simple greedy approach that computes a score for each literal that balances (i) the extent to which an assignment makes the hitting set instance closer to being satisfied and (ii) how much extra cost is incurred by the assignment. Search proceeds by greedily picking an assignment that maximizes the score and updating the scores after each new assignment.

5 Further Search Techniques and Functionalities

Constraint Seeding. HitPBO applies constraint seeding [34] to initialize `Optimizer` before the main search with all input constraints that only contain objective variables. In addition, HitPBO also seeds constraints that can be weakened to cores by removing non-objective variables: e.g., if the objective function has variables x_1 and x_2, but not x_3, and the instance has a constraint $x_1 + x_2 + x_3 \geq 2$, HitPBO would seed $x_1 + x_2 \geq 1$ to `Optimizer`. Constraint seeding can be disabled with `--seeding=false`.

Hardening [3, 27] and Reduced Cost Fixing [5, 14, 13]. HitPBO uses hardening and reduced cost fixing to fix variables during search. Consider a literal ℓ that has a coefficient c in the objective. Let UB be the cost of the currently best solution, and L the constant term in the objective. Hardening fixes ℓ to 0 if $c + L \geq UB$. Given an optimal solution τ^{LP} with cost optLP of the LP relaxation of the current hitting set instance as ILP, the corresponding reduced cost rc_ℓ represents a minimum increase in cost that follows from flipping a variable in the LP relaxation. In HitPBO, reduced cost fixing fixes $\ell = 1$ ($\ell = 0$) if (1) optLP $- rc_\ell \geq UB$ (optLP $+ rc_\ell \geq UB$), and (2) $\tau^{LP}(\ell) = 1$ ($\tau^{LP}(\ell) = 0$). This

in fact generalizes reduced cost fixing from previous work on IHS [5] which had a more restrictive variant of (1), requiring that the currently best-known solution for the PBO instance, witnessing the current UB, also assigns ℓ to 1 (0) for the case optLP $- rc_\ell = UB$ (optLP $+ rc_\ell = UB$). While the more restrictive condition guarantees the preservation of an optimal solution, HitPBO instead implements a slightly adapted termination procedure that guarantees that an optimal solution is returned even if reduced cost fixing has changed the optimal cost or feasibility of the instance. In practice, the possible changes resulting from reduced cost fixing as implemented in HitPBO are either that (a) the constraints become infeasible, or (b) the optimal cost of the new instance exceeds the known upper bound. In both cases, the best known solution is returned as an optimal one. Similarly, hardening in HitPBO employs $\geq$ instead of $>$ as typically in core-guided search [3]. Hardening and reduced cost fixing are on by default and controlled via `--hardening` and `--rc`.

Solution-Improving Search (SIS). Following [35], HitPBO heuristically invokes SIS during search whenever the main IHS search does not improve the lower bound LB for the optimum for sis_limit iterations (by default 5, controlled via `--sis-iters`). Implemented using a PB solver, SIS is terminated after the PB solver has seen a defined number of conflicts. The number of allowed conflicts per invocation is increased in proportion to the number of total conflicts in the oracle so far, resulting in a behaviour where the calls to SIS will be more aggressive when the search has continued longer. SIS uses, by default, a separate PB solver instantiation from the instantiation used for core extraction, but HitPBO can also be configured to use the same instantiation for both.

Proof Logging. HitPBO supports proof logging as described in [26], using a version of the external "VeriPB_Prooflogger" library [36] upgraded to support the VeriPB 3 proof format [22, 9]. Currently, with proof logging enabled (by specifying an outputfile for the proof via `--prooffile=<filename>`), HitPBO supports the use of RoundingSat-based instantiations of `Optimizer` for hitting set optimality proofs, RoundingSat-based SIS, and core-extracting with both RoundingSat and CaDiCaL (in plain mode). Proof logging does not currently support symmetry breaking, symmetric core learning, reduced cost fixing, ILP presolving, or core simplifications. However, proofs of optimality of hitting sets while using a non-proof logging and potentially numerically inexact (i.e., typical) ILP solver for (most) hitting set computations can be realized by specifying a separate "exact optimizer" (`--exact-optimizer=<solver>`) to verify the optimality of the hitting sets when needed. In the default configuration, when an exact optimizer is used, HitPBO invokes the exact optimizer when the cost of a solution matches the (claimed) lower bound for the optimum. Other ways of scheduling the calls to the exact optimizer include `--exopt-verify-claimed` (`--exopt-on-forced`), which invokes the exact optimizer when the ILP solver claims optimality (is forced to compute an optimal solution). The "exact optimizer" feature can also be used to increase trust when proof logging is not enabled; SCIP in numerically exact mode [18, 24] (that does not produce proofs in the VeriPB proof format) can also be used as the exact optimizer to verify the optimality of the hitting sets.

Symmetry Breaking. HitPBO also has the option to use BreakID [16] to generate lex-leader constraints to break the symmetries of the PBO instance. Depending on the configuration, the symmetry-breaking constraints are added to the core-extractor, hitting set optimizer, or oracle used in SIS, or any combination of these. Symmetry breaking is enabled via parameter `--break-symmetries`.

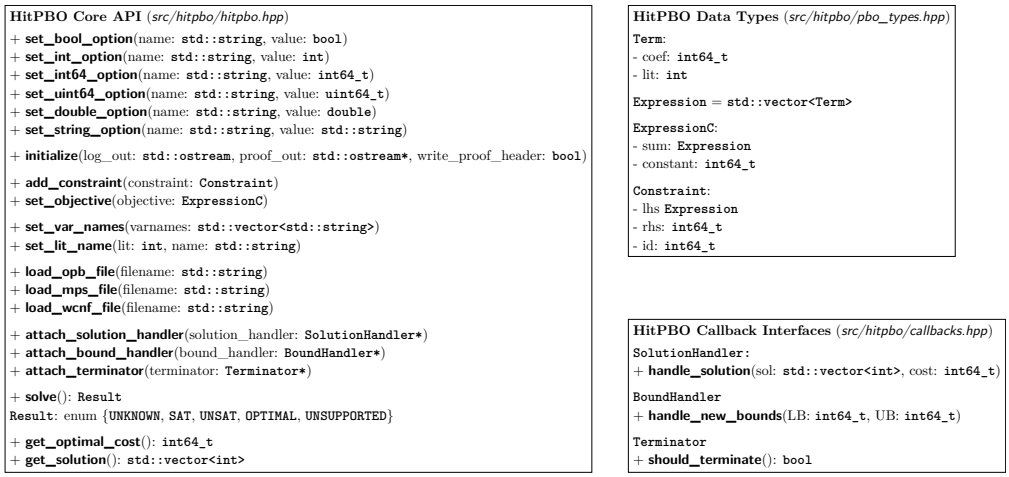

Figure 2 Overview of the HitPBO interface: core functions, central data types, and callback interfaces.

Preprocessing. HitPBO supports preprocessing via the PaPILO ILP presolver [21] v3.0.1.0 using selected techniques, with options for setting a time limit for preprocessing and selecting the employed preprocessing techniques. Preprocessing is enabled via `--presolve`.

6 The HitPBO Interface

HitPBO can also be used as a library through the interface defined in the *src/hitpbo/hitpbo.hpp* file in the repository (see Figure 2 for an overview). To configure HitPBOs parameters, the interface provides functions **set_*_option()** (where * is either **bool**, **int**, **int64**, **uint64**, **double**, or **string**). The parameter names match the names of command-line options. The input instance is given to the solver via **add_constraint()** and **set_objective()** functions, represented using the following types:

- `Term`: A Boolean literal (represented as `int`; a negative number indicates a negated literal) with an integer (`int64_t`) coefficient.
- `Expression`: A sum of pseudo-Boolean terms represented as an `std::vector<Term>`.
- `ExpressionC`: An `Expression` with an additional constant term of type `int64_t`.
- `Constraint`: A pseudo-Boolean constraint $lhs \geq rhs$ with an identifier *id* (for proof-logging) represented by an `Expression` *lhs*, an `int64_t` *rhs*, and an `int64_t` *id*.

Constraints are of type `Constraint`, and the objective of type `ExpressionC`. Instances where the sum of coefficients in each constraint and in the objective function fits in a signed 64-bit integer are supported. HitPBO is invoked via the API with **solve()**. After termination, the optimal cost and the found solution can be retrieved via **get_optimal_cost()** (returns `int64_t`) and **get_solution()** (returns an `std::vector<int>` as the set of literals in the optimal solution). The API also allows attaching callbacks to the solver to notify the user when a new solution is found (`SolutionHandler`), when the known bounds for optimum change (`BoundHandler`), and to check with the user if the search should be terminated (`Terminator`). The solver checks the user-provided termination condition after each core-extraction step, optimization step, and solution-improving search step.

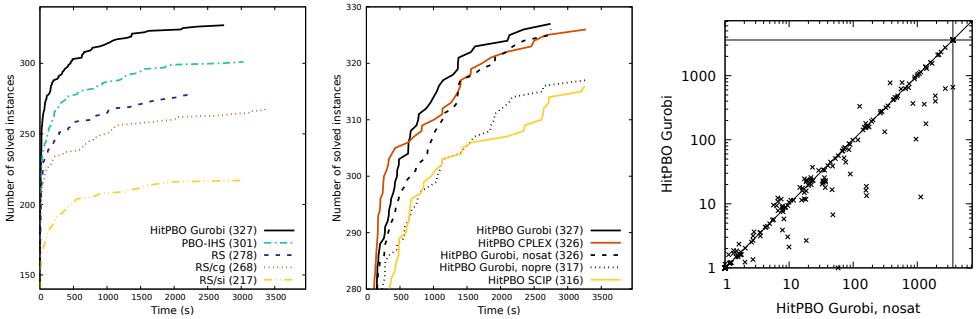

Figure 3 Left: comparison to earlier Python PBO-IHS and variants of RoundingSat; middle: impact of the choice of ILP solver for hitting set computation and switching of ILP preprocessing / SAT-based core extraction; right: with vs without SAT-based core extraction.

7 Empirical Snapshots

Finally, we give selected empirical snapshots to the impact of specific configuration choices in HitPBO. We note that empirical results on further techniques in HitPBO can be found, e.g., for algorithmic choices for hitting set computations and proof logging in [26] and for symmetric core learning in [25]. The results are over the PB Competition 2025 optimization benchmark set (the 555 instances of the "optimization problem, linear constraints" category) available at [28] and obtained using 2.40-GHz Intel(R) Xeon(R) Gold 6148 processors under a per-instance 3600-second time limit and 8-GB memory limit. Figure 3 (left) shows a comparison of HitPBO (employing Gurobi by default) to the earlier Python implementation PBO-IHS [34, 35, 29] (which supports CPLEX but not Gurobi) and to the RoundingSat solver (commit d4edbf7) in the different optimization modes it offers, namely, in its default "hybrid" configuration (RS) [17] and the alternative solution-improving (RS/si) [19] and core-guided (RS/cg) [17] configurations. HitPBO clearly outperforms both PBO-IHS and RoundingSat. Figure 3 (middle) shows the impact of the choice of ILP solver used for hitting set computation, with HitPBO otherwise in its default configuration (SAT-based core extraction and ILP presolving on), and the impact of switching off either ILP presolving (nopre) or SAT-based core extraction (nosat). Both, ILP presolving and SAT-based core extraction improve performance significantly. Figure 3 (right) shows a more fine-grained comparison of the impact of employing SAT-based core extraction, showing that it quite consistently speeds up HitPBO by avoiding to an extent unnecessary and potentially more expensive calls to a PB solver in core extraction.

8 Conclusions

HitPBO is a state-of-the-art implementation of IHS for PBO. While specific technical aspects of the solver (namely, symmetric core learning, proof logging and alternative ways to hitting set computation) have been described in earlier works, we focused on describing the newest version of the solver – including various recent developments – from a system perspective, describing some of the so-far undocumented features and techniques offered.

References

1 Tobias Achterberg. SCIP: Solving constraint integer programs. *Math. Program. Comput.*, 1(1):1–41, 2009. doi:10.1007/S12532-008-0001-1.

2 Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. *INFORMS J. Comput.*, 32(2):473–506, 2020. doi:10.1287/IJOC.2018.0857.

3 Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, Lecture Notes in Computer Science, pages 86–101. Springer, 2012. doi:10.1007/978-3-642-33558-7_9.

4 Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013. doi:10.1007/978-3-642-39071-5_23.

5 Fahiem Bacchus, Antti Hyttinen, Matti Järvisalo, and Paul Saikko. Reduced cost fixing in MaxSAT. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 641–651. Springer, 2017. doi:10.1007/978-3-319-66158-2_41.

6 Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in SAT-based MaxSAT solving. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, Lecture Notes in Computer Science, pages 652–670. Springer, 2017. doi:10.1007/978-3-319-66158-2_42.

7 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *J. Satisf. Boolean Model. Comput.*, 7(2-3):59–6, 2010. doi:10.3233/SAT190075.

8 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

9 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *J. Artif. Intell. Res.*, 77:1539–1589, 2023. doi:10.1613/JAIR.1.14296.

10 Der-San Chen, Robert G. Batson, and Yu Dang. *Applied Integer Programming: Modeling and Solution*. John Wiley & Sons, 2009.

11 Vasek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discret. Math.*, 4(4):305–337, 1973. doi:10.1016/0012-365X(73)90167-2.

12 IIBM ILOG Cplex et al. V12. 1: User's manual for cplex. *International business machines corporation*, 46(53):157, 2009.

13 Harlan P. Crowder, Ellis L. Johnson, and Manfred W. Padberg. Solving large-scale zero-one linear programming problems. *Oper. Res.*, 31(5):803–834, 1983. doi:10.1287/OPRE.31.5.803.

14 George Bernard Dantzig, Delbert R Fulkerson, and Selmer Martin Johnson. On a linear-programming, combinatorial approach to the traveling-salesman problem. *Operations Research*, 7(1):58–66, 1959. doi:10.1287/opre.7.1.58.

15 Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, Lecture Notes in Computer Science, pages 247–262. Springer, 2013. doi:10.1007/978-3-642-40627-0_21.

16 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016. doi:10.1007/978-3-319-40970-2_8.

17 Jo Devriendt, Stephan Gocht, Emir Demirovic, Jakob Nordström, and Peter J. Stuckey. Cutting to the core of pseudo-boolean optimization: Combining core-guided search with cutting planes reasoning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3750–3758. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16492.

18 Leon Eifler and Ambros M. Gleixner. A computational status update for exact rational mixed integer programming. *Math. Program.*, 197(2):793–812, 2023. doi:10.1007/S10107-021-01749-5.

19 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1291–1299. ijcai.org, 2018. doi:10.24963/IJCAI.2018/180.

20 Exact. https://gitlab.com/nonfiction-software/exact.

21 Ambros M. Gleixner, Leona Gottwald, and Alexander Hoen. PaPILO: A parallel presolving library for integer and linear optimization with multiprecision support. *INFORMS Journal on Computing*, 35(6):1329–1341, 2023. doi:10.1287/IJOC.2022.0171.

22 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-boolean proofs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3768–3777. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16494.

23 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL: https://www.gurobi.com.

24 Christopher Hojny, Mathieu Besançon, Ksenia Bestuzheva, Sander Borst, Joao Dionísio, Johannes Ehls, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Adrian Göß, Alexander Hoen, Jacob von Holly-Ponientzietz, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Marco Lübbecke, Stephen J. Maher, Paul Matti Meinhold, Gioni Mexi, Til Mohr, Erik Mühmer, Krunal Kishor Patel, Marc E. Pfetsch, Sebastian Pokutta, Chantal Reinartz Groba, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Matthias Walter, Dieter Weninger, and Liding Xu. The SCIP Optimization Suite 10.0, 2025. arXiv:2511.18580.

25 Hannes Ihalainen, Jeremias Berg, Matti Järvisalo, and Bart Bogaerts. Symmetric core learning for pseudo-boolean optimization by implicit hitting sets. In Maria Garcia de la Banda, editor, *31st International Conference on Principles and Practice of Constraint Programming, CP 2025, Glasgow, Scotland, August 10-15, 2025*, volume 340 of *LIPIcs*, pages 15:1–15:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.CP.2025.15.

26 Hannes Ihalainen, Dieter Vandesande, André Schidler, Jeremias Berg, Bart Bogaerts, and Matti Järvisalo. Efficient and reliable hitting-set computations for the implicit hitting set approach. In Sven Koenig, Chad Jenkins, and Matthew E. Taylor, editors, *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*, pages 14251–14260. AAAI Press, 2026. doi:10.1609/AAAI.V40I17.38439.

27 João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: Algorithms & applications. *Ann. Math. Artif. Intell.*, 62(3-4):317–343, 2011. doi:10.1007/S10472-011-9233-2.

28 Pseudo-Boolean Competition 2025. https://www.cril.univ-artois.fr/PB25/.

29 PBO-IHS. Pseudo-Boolean optimization (PBO) solver that utilizes the implicit hitting set (IHS) approach. https://bitbucket.org/coreo-group/pbo-ihs-solver/.

30 PBComp24-cg Mixed-Bag & PB-OLL-RS. https://bitbucket.org/coreo-group/pbcomp24-cg.

31 RoundingSat. The pseudo-Boolean solver powered by proof complexity! https://gitlab.com/MIAOresearch/software/roundingsat.

32 Paul Saikko. *Implicit Hitting Set Algorithms for Constraint Optimization*. PhD thesis, University of Helsinki, Finland, 2019.

33 Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Trans. Inf. Syst.*, 98-D(6):1121–1127, 2015. doi:10.1587/TRANSINF.2014FOP0007.

34 Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-boolean optimization by implicit hitting sets. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 51:1–51:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.51.

35 Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Improvements to the implicit hitting set approach to pseudo-boolean optimization. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.13.

36 VeriPB Prooflogger, a library for proof logging in the VeriPB proof format. https://github.com/DietVds/VeriPB_Prooflogger.

37 Dag Wedelin. An algorithm for large scale 0–1 integer programming with application to airline crew scheduling. *Annals of Operations Research*, 57(1):283–301, 1995. doi:10.1007/BF02099703.

38 Yujiao Zhao, Yiyuan Wang, Yi Chu, Wenbo Zhou, Shaowei Cai, and Minghao Yin. Improving local search algorithm for pseudo boolean optimization. *J. Artif. Intell. Res.*, 83, 2025. doi:10.1613/JAIR.1.16626.

A Core Processing and Simplification Rules

We detail here the techniques that the constraint simplification method of HitPBO implements (Section 3). All techniques preserve the equivalence of the constraints. Given a core $C = \sum_{i=1}^{n} a_i \ell_i \geq b$, and assume w.l.o.g. that $a_i \leq a_{i+1}$ for all i, the following techniques are applied until fixpoint.

- **Saturation.** Let k be the minimum index for which $a_i > b$ for each $i \geq k$. Change C to $\sum_{i=1}^{k-1} a_i \ell_i + \sum_{i=k}^{n} b\ell_i \geq b$
- **Detection of clauses.** Let k be the maximum index for which $\sum_{i=1}^{k} a_i < b$. If $a_{k+1} \geq b$ change C to $\sum_{i=k+1}^{n} \ell_i \geq 1$.
- **At-least-k detection.** Select an index $j \leq t$. Let k_1 be the minimal value for which $\sum_{i=j}^{j+k_1} a_i \geq b$ and k_2 the minimal value for which $\sum_{i=n-k_2}^{n} a_i \geq b$. If k_1 and k_2 both exist, if $k_1 = k_2$, and if $\sum_{i=1}^{j+k_1-1} a_i < b$, change C to $\sum_{i=j}^{n} \ell_i \geq k$.
- **Extension to the at-least-k detection.** Let t be the maximum index for which $a_t < b$. Select an index $j \leq t$. Let k_1 be the minimal value for which $\sum_{i=j}^{j+k_1} a_i \geq b$ and k_2 the minimal value for which $\sum_{i=t-k_2}^{t} a_i \geq b$. If k_1 and k_2 both exist, if $k_1 = k_2$, and if $\sum_{i=1}^{j+k_1-1} a_i < b$, change C to $\sum_{i=j}^{t} \ell_i + \sum_{i=t+1}^{n} k\ell_i \geq k$.

- **Division by the GCD of the coefficients.** Change C to $\sum_{i=1}^{n} \frac{a_i}{gcd}\ell_i \geq \left\lceil \frac{b}{gcd} \right\rceil$ where gcd is the greatest common divisor of the coefficients a_i. Only applied if $gcd > 1$.
- **Division by the GCD of all but one coefficient.** Let $gcd_{\bar{j}}$ be the greatest common divisor of all coefficients except a_j and assume $gcd_{\bar{j}} > 1$. Change C to $\sum_{i|i\neq j} \frac{a_i}{gcd_{\bar{j}}}\ell_i + A'_j\ell_j \geq \left\lceil \frac{b}{gcd_{\bar{j}}} \right\rceil$. Here $A'_j = \left\lfloor \frac{a_j}{gcd_{\bar{j}}} \right\rfloor$ if $(a_j \mod gcd_{\bar{j}}) < (b \mod gcd_{\bar{j}})$, or $(b \mod gcd_{\bar{j}}) = 0$. Otherwise $A'_j = \left\lceil \frac{a_j}{gcd_{\bar{j}}} \right\rceil$.
- **Search for a divisor.** Try the following rules for different integers d. The implementation tests values from 2 to $\min\{1000, b\}$ for 1–3 and from $d = 2$ to $\min\{100, b\}$ for 4
  1. If $\frac{a_i \mod d}{b \mod d} < \frac{\lfloor a_i/d \rfloor}{\lfloor b/d \rfloor}$ for all i, change C to $\sum_i \lfloor \frac{a_i}{d} \rfloor \ell_i \geq \lceil \frac{b}{d} \rceil$.
  2. If $\frac{d-(a_i \mod d)}{a_i} \leq \frac{d-(b \mod d)}{b}$ for all i, change C to $\sum_i \left\lceil \frac{a_i}{d} \right\rceil \ell_i \geq \lceil \frac{b}{d} \rceil$. (a special case of Chvátal-Gomory cuts)
  3. Let $r = d$ if $(b \mod d) = 0$ and $r = (b \mod d)$ otherwise
     a. If $\sum_i (a_i \mod d) < r$, change C to $\sum_i \lfloor \frac{a_i}{d} \rfloor \ell_i \geq \lceil \frac{b}{d} \rceil$.
     b. If $r \leq \sum_i (a_i \mod d) < d + r$, and $\sum_i (a_i \mod d) x_i \geq r$ is equivalent to $\sum_{i|i\in S} x_i \geq k$ for a subset S of indices change C to

$$\sum_{i|i\notin S} t \left\lfloor \frac{a_i}{d} \right\rfloor + \sum_{i|i\in S} \left(t \left\lfloor \frac{a_i}{d} \right\rfloor + 1 \right) x_i \geq t \left(\left\lceil \frac{b}{d} \right\rceil - 1 \right) + k,$$

     where $t = \max\{k+1, |S| - k + 1\}$. Only applied if $t < d$.
  4. If for all subsets S of indices $\sum_{j\in S} (a_j \mod d) \geq (b \mod d)$ implies $\sum_{j\in S} \lfloor \frac{a_j}{d} \rfloor \geq \lceil \frac{b}{d} \rceil$, change C to $\sum_i \lfloor \frac{a_i}{d} \rfloor \ell \geq \lceil \frac{b}{d} \rceil$. Given C and d, the implementation checks the condition with a dynamic programming algorithm.
- **Chvátal-Gomory cuts [11, 2].** Find integers c and d such that $0 < \frac{c}{d} < 1$ and $\frac{\lceil \frac{c}{d} a_i \rceil}{a_i} \leq \frac{\lceil \frac{c}{d} b \rceil}{b}$ for all i. Change C to $\sum_i \lceil \frac{c}{d} a_i \rceil \ell_i \geq \lceil \frac{c}{d} b \rceil$. The implementation tries values 1 to 5 for c combined with minimum coefficient and maximum coefficient in C as a value for d.