MaxSAT Evaluation 2026

Solver and Benchmark Descriptions

Dieter Vandesande, Christoph Jabs, Jeremias Berg, Johannes Fichte, George Katsirelos, Ruben Martins, Andre Schidler, and Dominik Schreiber *(editors)*

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2026-1

ISSN 1458-4786
Helsinki 2026

PREFACE

The MaxSAT Evaluations (`https://maxsat-evaluations.github.io`) constitute a series of events focusing on the evaluation of current state-of-the-art systems for solving optimization problems via the Boolean optimization paradigm of maximum satisfiability (MaxSAT). Organized yearly starting from 2006, with a break in 2025, the year 2026 brought on the 20th edition of the MaxSAT Evaluations, organized as a satellite event of the 29th International Conference on Theory and Applications of Satisfiability Testing (SAT 2026) and also associated with FLoC Olympics 2026. Some of the central motivations for the MaxSAT Evaluation series are to provide further incentives for further improving the empirical performance of the current state of the art in MaxSAT solving, to promote MaxSAT as a serious alternative approach to solving NP-hard optimization problems from the real world, and to provide the community at large with heterogeneous benchmark sets for solver development and research purposes.

The 2026 evaluation was run by an almost entirely new team of organizers. We would like to thank Matti Järvisalo, Andreas Niskanen and Tobias Paxian for their work in organizing the previous evaluations. On the whole, the evaluation consisted of a total of five tracks: two for exact solvers (one for solvers focusing on unweighted and one for solvers focusing on weighted MaxSAT instances), two for anytime MaxSAT solvers (using two short per-instance time limits, 60 and 300 seconds, and a randomly drawn time limit of 1707 seconds differentiating from the per-instance time limit of 1 hour imposed in the main exact tracks), and the incremental track.

In addition to the usual highlighting of well-performing solvers and benchmark submitters in a designated session of the SAT conference, this year the MaxSAT Evaluation awarded an FLoC medal dubbed the "MaxSAT Evaluation onion core award" to Ruben Martins for significant contributions to the benchmarks used in the evaluation and for developing the Open-WBO solver architecture that has been used and extended by many regular participants of the evaluation over the years.

Solvers were required to be made available in open source, and a 1-2 page solver description was expected to accompany each solver and benchmark submission, to provide some details on the search techniques implemented in the solvers and the domains the benchmarks are drawn from. The solver descriptions together with descriptions of new benchmarks for 2026 are collected together in this compilation.

We would like to thank everyone who contributed to MaxSAT Evaluation 2026 by submitting their solvers or new benchmarks. The resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government, as well as Finnish Computing Competence Infrastructure (FCCI). We are grateful for the support.

Dieter Vandesande, Christoph Jabs, Jeremias Berg, Johannes Fichte,
George Katsirelos, Ruben Martins, Andre Schidler, Dominik Schreiber
MaxSAT Evaluation 2026 Organizers

Contents

Preface 3

Exact and Incremental Solvers

CASHWMaxSAT in MaxSAT Evaluation 2026
Shiwei Pan, Yiyuan Wang, Shaowei Cai, Yiyang Dong, Minghao Yin 10

CPMpy submissions to the MSE26 competition *(also in the anytime track)*
Tias Guns, Thomas Sergeys, Ignace Bleukx, Hendrik Bierlee 11

COGS: An Ordered MaxSAT Solver
Josep Maria Salvia Hornos 13

CoreForge: An LLM-Generated Solver for Unweighted MaxSAT
Ruben Martins 15

CoreTrail and ShareTrail: Incremental Core-Guided MaxSAT Solvers
Josep Maria Salvia Hornos, Alexey Ignatiev, Joao Marques-Silva 17

EvalMaxSAT 2026
Florent Avellaneda 18

EvalMaxRes in MaxSAT Evaluation 2026
Shuolin Li, Sami Cherif, Chu-Min Li, Jialu Zhang 19

MaxCDCL in MaxSAT Evaluation 2026
Chu-Min Li, Jialu Zhang, Shuolin Li, Sami Cherif 21

MaxHS Revised
Edgar Justus, Jakob Berg, André Schidler, Mathias Fleury, Tobias Paxian, Armin Biere 23

UWrMaxSat 2.0 in the 2026 MaxSAT Evaluation (*also in the incremental track*)
Marek Piotróow 24

RustiMax: A New OLL-based MaxSAT Solver, Agentically Engineered
Stefan Szeider 26

UWrMaxSat-MaxCDCL: Solver Description
Mingming Jin, Yujie Cai, Jiongzhi Zheng, Kun He 28

WMaxCDCL in MaxSAT Evaluation 2026
Jialu Zhang, Shuolin Li, Chu-Min Li, Sami Cherif 29

Anytime Solvers

Aperture: an Anytime, Complete and Incremental MaxSAT Solver Entering MaxSAT Evaluation 2026 *(also in the incremental track)*
Alexander Nadel, Yam Slonimski 32
BouMS 3.0.1 in the MaxSAT Evaluation 2026
Ole Lübke 34
DeepDist-c: Solver Description
Menghua Jiang, Haokai Gao, Xiangui Kang 36
HWS: Solver Description
Weijie Yao, Menghua Jiang, Zhaojin Zhong, Yin Chen 38
Loandra
Jeremias Berg, Ole Lübke 39
Metaheuristic Solver PRINTEMPS for MaxSAT Evaluation 2026
Masahiro Sakai, Yuji Koguma 42
NuWLS-c-2026: Solver Description
Yi Chu, Yingyi Jiang, Yiwen Sun, Peng Lin, Chuan Luo, Shaowei Cai 44
NuWLS-c-2026-UB: Solver Description
Yingyi Jiang, Yi Chu, Yiwen Sun, Chuan Luo, Shaowei Cai 46
PASMaxSAT-c: an Anytime MaxSAT Solver Entering MSE'26
Mingming Jin, Yujie Cai, Jiongzhi Zheng, Kun He 48
SPB-NIM-c: Solver Description
Zhaojin Zhong, Yin Chen 49
SGAT-NuWLS-c for MaxSAT Evaluation 2026
Sota Moriyama, Katsumi Inoue 50

Benchmark Descriptions

A MaxSAT Benchmark for Multiple Optimization Domains based on OP-to-MaxSAT Reduction
Yuxin Zhao, Han Huang, Zhifeng Hao 53
Answer Set Optimization by MaxSAT: Benchmarks for MaxSAT Evaluation 2026
Masood Feyzbakhsh Rankooh, Matti Järvisalo 55
Benchmark Description: Network Modularity Optimization
Carlos Ansótegui, Vaidyanathan Peruvemba Ramaswamy, Stefan Szeider, Hai Xia 56
CPMpy dataset submissions to MSE26
Tias Guns, Thomas Sergeys, Ignace Bleukx, Hendrik Bierlee 58
Cost-Optimal Delete-Free Classical Planning as MaxSAT: Benchmarks for MaxSAT Evaluation 2026
Masood Feyzbakhsh Rankooh, Andreas Niskanen, Matti Järvisalo 60
Lights Out MaxSAT Benchmarks
André Schidler, Tobias Faller, Armin Biere 61

MaxSAT Benchmarks for Learning Optimal Oblique Decision Trees
Florent Avellaneda 62
MaxSAT Benchmarks: Finding Optimal Counterfactual Explanations for Random Forests
Alesya Raevskaya, Tuomo Lehtonen 63
MaxSAT encodings for learning MDL rules from noisy data: benchmark description
Andrew Cropper 65
Identifying Molecular Formulas from Mass Spectra via MaxSAT
Shuolin Li, Sami Cherif, Chu-Min Li, Ahlam Mentag, Rebecca Dauwe, Jialu Zhang 66
Overcommitted Cluster Scheduling Benchmarks
Jann Ehses, André Schidler, Armin Biere 68

Incrementat Benchmark Descriptions

Incremental MaxSAT Routing on a Physical Binary Sorter: A Deterministic Benchmark
Josep Maria Salvia Hornos 70
Incremental MaxSAT Benchmarks for Optimal TOHTN Planning via SibylSatOpt Trace Replay
Gaspard Quenard, Damien Pellier, Humbert Fiorino 71

Solver Index 72
Benchmark Index 73
Author Index 74

EXACT AND INCREMENTAL SOLVERS

CASHWMaxSAT in MaxSAT Evaluation 2026

Shiwei Pan[1,2], Yiyuan Wang[1,2,*], Shaowei Cai[3], Yiyang Dong[1,2] Minghao Yin[1,2]
[1]School of Computer Science and Information Technology, Northeast Normal University, China
[2]Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China
[3]School of Software, Beihang University, Beijing, China
[*]corrsponding author
pansw779@nenu.edu.cn, yiyuanwangjlu@126.com, caisw@buaa.edu.cn, dongyy@nenu.edu.cn, ymh@nenu.edu.cn

***Abstract*—This document describes the MaxSAT solver CASHWMaxSAT-2026, submitted to MaxSAT Evaluation 2026. CASHWMaxSAT is a complete MaxSAT solver for weighted and unweighted instances.**

I. Introduction

We developed an improved complete MaxSAT solver called CASHWMaxSAT-2026. The solver is based on our previous implementation developed from UWrMaxSAT [1], SCIP [2], and CASHWMaxSAT-DisjCad [3]. CASHWMaxSAT-2026 follows an unsatisfiable-core-based OLL procedure [4]–[7]. The solver also inherits the main strategies used in the previous version, including the extraction of multiple disjoint unsatisfiable cores and the weight-distribution-based assumption strategy. Based on these components, the 2026 version introduces two additional strategies.

- **Initial hard-clause satisfiability checking and upper-bound encoding.** Before the main MaxSAT solving procedure, the solver first calls the underlying SAT solver on the formula consisting only of hard clauses. If the hard clauses are satisfiable, the obtained assignment provides a feasible solution for the original MaxSAT instance. The cost of the falsified soft clauses under this assignment is then used as an initial upper bound. Based on this bound, we encode an objective constraint and add it to the hard part of the formula. In this way, assignments whose objective values are not better than the initial upper bound are excluded from the subsequent search.
- **Dynamic objective-bound tightening.** Inspired by solution-improving search (SIS), which repeatedly searches for a feasible solution with an objective value smaller than the current upper bound, CASHWMaxSAT-2026 dynamically tightens the objective bound whenever a better solution is found. Specifically, after the solver obtains a model with a smaller objective value, the current upper bound is tightened accordingly, and the corresponding objective constraint is updated in the SAT/PB encoding. This strategy allows the solver to exploit newly found high-quality solutions during the search and to prune assignments that cannot improve the current incumbent solution.

References

[1] M. Piotrów, "Uwrmaxsat in maxsat evaluation 2021," *MaxSAT Evaluation 2021*, p. 17, 2021.
[2] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner *et al.*, "The scip optimization suite 8.0," *arXiv preprint arXiv:2112.08872*, 2021.
[3] S. Pan, Y. Wang, S. Cai, J. Li, W. Zhu, and M. Yin, "Cashwmaxsat-disjcad: Solver description," *MaxSAT Evaluation 2024*, p. 25, 2024.
[4] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub, "Unsatisfiability-based optimization in clasp," in *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.
[5] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva, "Iterative and core-guided maxsat solving: A survey and assessment," *Constraints*, vol. 18, no. 4, pp. 478–534, 2013.
[6] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-guided maxsat with soft cardinality constraints," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2014, pp. 564–573.
[7] A. Ignatiev, A. Morgado, and J. Marques-Silva, "Rc2: an efficient maxsat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 11, no. 1, pp. 53–64, 2019.

CPMpy submissions to the MSE26 competition

Tias Guns
Department of Computer Science
KU Leuven
Leuven, Belgium
tias.guns@kuleuven.be

Thomas Sergeys
Department of Computer Science
KU Leuven
Leuven, Belgium
thomas.sergeys@kuleuven.be

Ignace Bleukx
Department of Computer Science
KU Leuven
Leuven, Belgium
ignace.bleukx@kuleuven.be

Hendrik Bierlee
Department of Computer Science
KU Leuven
Leuven, Belgium
henk.bierlee@kuleuven.be

***Abstract*—CPMpy [4] is an open-source Python library for modeling combinatorial (optimisation) problems. Solvers from different CO paradigms support different types of decision variables and constraints. Through a series of equivalence-preserving constraint reformulations, CPMpy can transform a constraint model into the target format, enabling systematic comparisons of solvers across paradigms. For this competition, we use CPMpy as a middleware that can translate *.wcnf* input to the solver APIs of varying technologies: SMT, CP, (M)ILP, PB, and (Max)SAT. Included in this submission are LCG CP solver OR-Tools CP-SAT and the ILP solver HiGHS. The purpose of our submission is to widen the range of solving technologies evaluated in the competition, while reusing and stress-testing the transformation capabilities of CPMpy.**

License : CPMpy is licensed under the Apache-2.0 license
Code : CPMpy's code is available on GitHub: https://github.com/CPMpy/cpmpy
Acknowledgement : This research received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant No. 101002802, CHAT-Opt) and Proof of Concept (POC) grant (Grant No.101158376, PO-CPMpy).

I. Format loaders

In the recent work of [8], we introduced a collection of file format readers and loaders to CPMpy. These tools convert well-known benchmarking instances into CPMpy models, giving access to our constraint reformulation stack; an extensive set of rewrite operations such as constraint decomposition, linearization, and Boolean encodings, that allow transforming between different constraint formalisms.

II. Constraint reformulations

Solvers from different CO paradigms support different types of decision variables and constraints. Constraint Programming (CP) has the most expressive language, supporting global constraints [10], potentially non-linear, over both integer and Boolean decision variables. Next are Integer Linear Programming (ILP) solvers supporting only linear inequality constraints, Pseudo-Boolean (PB) solvers supporting linear inequality constraints over Boolean variables, and (Max)SAT-solvers which support Boolean formulas in Conjunctive Normal Form (CNF).

Solvers from the more expressive paradigms can take less expressive languages as input, but the reverse direction requires rewriting a set of high-level expressions into a set of lower level expressions, often requiring the use of auxiliary variables. This is achieved through a series of equivalence-preserving constraint reformulations, transforming a constraint model into the target format. We identify five main reformulation stages: decomposition, flattening, linearization, encoding integer variables to Booleans, and encoding pseudo-Boolean constraints into CNF. Each of the transformations is implemented in CPMpy as a separate function.

A. Decomposition

When a CP solver does not support a particular CP *global constraint*, or when targeting a lower level paradigm, that global constraint needs to be rewritten into a collection of equivalent lower level expressions, i.e. decomposing the constraint. Examples of global constraints include AllDifferent, Element, Cumulative and Circuit.

We partition the constraints resulting from the decomposition into two groups; *value* and *defining*. The *value* constraints represent the logical value of the original constraint. When a global constraint occurs in a reified or nested context (e.g., under implication, disjunction, or negation), these constraints must themselves be reified or nested to preserve compositional semantics [1]. *Defining* constraints define the value of new auxiliary variables introduced by the decomposition, and should always be enforced at top-level. Returning both groups of constraints separately prevents unnecessary reification or nesting. In practice, CPMpy also uses the concept of global constraints to decompose non-linear functions such as Max, Min, Multiplication and Division.

B. Flattening

The second step in the transformation pipeline is to eliminate *nested* expressions; introducing auxiliary variables such

that each constraint is over decision variables rather than over other expressions. To improve the output of this process, we reuse auxiliary variables through common subexpression elimination (CSE). Whenever we introduce an auxiliary variable for a sub-expression, we first check if such a variable already exists in a cache we maintain.

C. Linearization

To transform the constraint model into a linear or Boolean model, we linearize the global constraints and functions. We employ standard decompositions for most global constraints available in the CPMpy library, and use a few alternative linear-friendly decompositions for some often-used global constraints. These decompositions encode the integer variables into auxiliary Boolean variables by using a direct encoding. For example, $\text{ALLDIFFERENT}(X)$ is decomposed into $\sum_{x_i \in X} [\![x_i = v]\!] \leq 1$ for each value in the domain of the variables in X. Similarly, $\text{ELEMENT}(A, idx, v)$ is decomposed to $\sum_{i=1..|A|} a_i [\![idx = i]\!] = v$.

Additionally, we detect the use of categorical variables and encode them once using a direct encoding, rather than linearizing each $b \Leftrightarrow (x = v)$ expression separately. This is done by checking whether comparison constraints such as $(x = v)$ are used as a subexpression. E.g., the constraint $(x = 3) \vee (y \neq 1)$ is linearized by first adding the direct encoding of both x and y and then linearizing the constraint as $[\![x = 3]\!] + \neg[\![y = 1]\!] \geq 1$, with $[\![x = v]\!]$ the Boolean variable from the direct Boolean encoding.

D. Boolean encoding of integer variables

To encode Constraint Programming models or ILP models for (Pseudo-)Boolean solvers, all integer variables must be encoded into Boolean variables. We currently support the most common encodings: direct encoding [11], order encoding [9] or log-encoding (also known as binary encoding) [3]. We reuse the direct encoding if it already exists, and otherwise heuristically decide an encoding based on the size of the domain.

CNF encoding of pseudo-Boolean constraints: When targeting a constraint model for (Max)SAT, the last stage in our transformation pipeline consists of encoding all pseudo-Boolean constraints into clausal form. For this final step, we connect to standard tools from the (Max)SAT community, namely PySAT (including pypblib) [6] and the pindakaas library [2], both of which contain a large number of state-of-the-art CNF encodings for pseudo-Boolean constraints.

All the above transformations include encoding choices, e.g. how to decompose the different global constraints, what integer encoding to use, what PB encoding to use etc. These choices are still under active study in the community today to determine different trade-offs related the size of the encoding and the overall runtime performance of the solver. Any library that aims to transform between different formalisms will have to set some reasonable default choices, these can always be changed or improved by expert users.

III. Solvers

CPMpy can translate to the Python API of many different solvers spanning CP, MIP, SMT, pseudo-Boolean and SAT technologies. Our CPMpy submissions bundle the following solvers:

- `cpmpy_ortools` [7] (version: 9.15.6755) The default solver in CPMpy, it is a lazy clause generation CP solver. It supports parallel search too.
- `cpmpy_highs` [5] (version: 1.14.0) An high performance serial and parallel linear optimisation solver, supporting LP, QP and MIP.

We did not significantly contribute to the development of any of these solvers and are grateful to the solver developers for making their technology available.

IV. Disclosure

Some code was written with the use of the Cursor agentic coding tool. All code was checked by the committer and by at least one pull request reviewer.

References

[1] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints An Int. J.*, 18(1):1–6, 2013.

[2] Hendrik Bierlee and Jip J. Dekker. Pindakaas (pyndakaas-v0.4.1), 2026. doi:10.5281/zenodo.10851855.

[3] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *IJCAI*, pages 1169–1177. Morgan Kaufmann, 1997.

[4] Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

[5] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: https://www.gurobi.com.

[6] Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. In *SAT*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018.

[7] Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

[8] Thomas Sergeys, Ignace Bleukx, and Tias Guns. Unified programmatic access to co benchmarks, to connect constraint solving communities. In *Proceedings of the 29th International Conference on Theory and Applications of Satisfiability Testing (SAT 2026)*, 2026. Accepted for publication.

[9] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009.

[10] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

[11] Toby Walsh. SAT v CSP. In *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.

COGS: An Ordered MaxSAT Solver

Josep Maria Salvia Hornos
University of Lleida
Lleida, Spain
josepmaria.salvia@udl.cat

***Abstract*—We submit the COGS MaxSAT solver to the MaxSAT Evaluation. COGS stands for Core Ordered Generalized Switch and is a CGSS2 augmented solver with SCIP preprocessing and UWrMaxSAT postprocessing with OLL ordered objective stratification and ordered objective core minimization.**

I. Solver Description

COGS main solving loop is based on CGSS2, a C++ implementation that extends the OLL ideas from RC2 [1] with refined core relaxation, weight-aware core extraction, and structure sharing for cardinality constraints [2], [3]. We incorporate ordered objective core simplification and stratification, motivated by recent work in MaxSAT solving [4]. Ordered objective core minimization keeps a graph of the soft implications and allows to quickly simplify cores by detecting redundancies from the implication graph. OLL ordered objective is also a generalization of the SimpleSIS algorithm baked into the OLL loop, we stratify based on ordered objective when the implication graph of the formula forms a chain.

We also tune CGSS2 with Glucose 4.1 instead of Glucose 3, which we believe to give better long term runtime. We configure a prerun with SCIP [5] that also contributes model and upper bound information. We postprocess with UWrMaxSAT [6], [7], which builds upon work done on kp-minisatp [8], MiniSAT+ [9] and COMiniSatPS [10].

This multi processing setup was inspired by EvalMaxSAT 2024 [11] and MaxCDCL 2024, which, used SCIP and OpenWBO processing techniques, respectively.

II. Ordered Objective Implication Graph

COGS augments the OLL loop with an implication. The purpose of the graph is twofold. First, it gives a cheap core simplification rule that is valid even when the full objective is not ordered. Second, when the graph induces a total order on the active objective assumptions, it is used as an ordered stratification. In that case the OLL loop follows the same ordered frontier as SimpleSIS for ordered objectives [4].

Let A be the set of active objective assumption literals, i.e., literals with positive residual weight that are not base assumptions. COGS constructs a directed graph $G_A = (A, E)$ where an edge $a \to c$ is inserted when unit propagation under assumption a propagates the active objective assumption c. If the objective variables are ordered as $b_1 \prec b_2 \prec \cdots \prec b_n$, under assumption literals $a_i = \neg b_i$, the contrapositive gives $a_i \to a_j \quad \text{for } i < j$ Thus the order on objective variables is reversed when viewed as implications.

A. Graph based core simplification

Let $C \subseteq A$ be a core returned by the SAT solver. The implication graph gives a direct redundancy rule.

Lemma 1. Let $p, q \in C$. If $F \models p \to q$, then $C \setminus \{q\}$ is also a core.

COGS applies this lemma using reachability in G_A. If $p \rightsquigarrow q$, then q is removed from the core. This simplification is independent of ordered stratification. It is valid on arbitrary instances because it uses only certified implications.

B. OLL stratification of ordered objective

COGS accepts ordered objective stratification only when this DAG is totally ordered by reachability: for every pair of components X, Y, either X reaches Y or Y reaches X. If this test succeeds, the components form a chain

$$K_0 \to K_1 \to \cdots \to K_m.$$

The implementation then unlocks the chain in reverse order:

$$K_m,\ K_{m-1},\ \ldots,\ K_0.$$

This is the SAT to UNSAT direction over assumption literals. This is the same direction as SimpleSIS over objective variables.

Therefore the SAT calls made by ordered OLL stratification have the same satisfiability outcomes as the SimpleSIS calls.

The first UNSAT frontier is A_t. Since A_{t+1} was satisfiable, any core contained in A_t must contain a_t. Moreover, because a_t implies every a_j for $j > t$, the graph simplification rule reduces any such core to the unit core $\{a_t\}$.

The remaining difference is only the certificate. This is a very conterintuive generalization of the OLL loop for ordered objectives, since OLL is usually thought of from an increasing lowerbound perspective.

III. Track submission

COGS is submitted to non-incremental track in unweighted categories. COGS successfully passes the regression test required in MSE2024.

The developers have found that weighted instances are not always sound and the solver is not recommended to solve weighted problems.

References

[1] A. Ignatiev, A. Morgado, and J. Marques-Silva, "RC2: an efficient MaxSAT solver," *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 53–64, 2019.

[2] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: a python toolkit for prototyping with SAT oracles," in *Proc. SAT*, 2018, pp. 428–437.

[3] H. Ihalainen, J. Berg, and M. Järvisalo, "Refined core relaxation for core-guided MaxSAT solving," in *Proc. CP*, ser. LIPIcs, vol. 210. Schloss Dagstuhl, 2021, pp. 28:1–28:19.
[4] J. Berg, A. Schidler, and M. Järvisalo, "Ordered objectives in maximum satisfiability," in *Proc. AAAI*, vol. 40, no. 17, 2026, pp. 14 166–14 174.
[5] C. Hojny, M. Besançon, K. Bestuzheva, S. Borst, J. Dionísio, J. Ehls, L. Eifler, M. Ghannam, A. Gleixner, A. Göß *et al.*, "The scip optimization suite 10.0," *arXiv preprint arXiv:2511.18580*, 2025.
[6] M. Piotrów, "Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems," in *Proc. ICTAI*. IEEE, 2020, pp. 132–136.
[7] M. Karpiński and M. Piotrów, "Encoding cardinality constraints using multiway merge selection networks," *Constraints*, vol. 24, no. 3, pp. 234–251, 2019.
[8] M. Karpinski, M. Piotrów, D. Berre, and M. Jarvisalo, "Competitive sorter-based encoding of pb-constraints into sat," *Proc. Pragmatics of SAT*, pp. 65–78, 2015.
[9] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into sat," *Journal on Satisfiability*, vol. 2, no. 1-4, pp. 1–26, 2006.
[10] C. Oh, "Cominisatps pulsar and ghackcomsps in the 2018 competition,"," *SAT Competition*, p. 19, 2018.
[11] J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian, "Maxsat evaluation 2024: Solver and benchmark descriptions," 2024.

MaxSAT Evaluation 2026: Solver and Benchmark Descriptions, volume B-2026-1 of Department of Computer Science Series of Publications B, University of Helsinki 2026.

CoreForge: An LLM-Generated Solver for Unweighted MaxSAT

Ruben Martins
Carnegie Mellon University
rubenm@cs.cmu.edu

***Abstract*—COREFORGE is an LLM-generated solver for unweighted MaxSAT, developed to explore whether modern LLMs can build a nontrivial optimization solver from research papers and high-level implementation guidance. The solver follows a core-guided MaxSAT architecture and is submitted in three configurations: a baseline OLL configuration, a configuration with ILP-based backend integration, and a configuration with a new core-sequence lookahead technique.**

This submission does not argue that building solvers directly from papers is the best path to competitive performance. Rather, COREFORGE is an experiment in LLM-assisted solver construction with limited manual engineering. The experience suggests that LLMs can help implement substantial solver functionality, while still requiring human guidance and validation.

I. MOTIVATION

Recent work has explored the use of LLMs in solver engineering, including heuristic optimization for SAT solvers and SMT solver construction [1], [2]. In parallel, paper-to-code benchmarks and systems have studied whether AI agents can implement research ideas from papers [3]. COREFORGE follows these research directions in the context of MaxSAT by asking whether an LLM-assisted workflow can construct a nontrivial optimization solver from research papers and high-level implementation guidance, without starting from an existing MaxSAT codebase.

II. LLM-ASSISTED DEVELOPMENT WORKFLOW

COREFORGE was developed through an iterative LLM-assisted workflow. Research papers were first discussed with ChatGPT to extract algorithmic ideas, identify useful solver components, and plan implementation steps. Most planning used GPT-5.4 in extended mode, with GPT-5.5 used near the end of the project. The resulting plans were then turned into Codex prompts, which generated or revised the C++ implementation using GPT-5.4 with medium reasoning. The code was then evaluated, audited with LLM assistance, and revised over many iterations.

A practical motivation for separating planning from implementation was cost control. ChatGPT was used for higher-level design and prompt construction, while Codex was reserved for implementation tasks. This made it possible to carry out the experiment using the Codex $20 plan over approximately four weeks. The experience suggests that a nontrivial MaxSAT solver can be developed with modest LLM usage when planning, implementation, and validation are separated.

The process was not fully autonomous. The human role was to select papers, define implementation goals, provide prompts, run experiments, inspect results, decide which generated changes to keep, and create commits. LLMs were used both to generate code and to audit the generated implementation, especially to look for mismatches between the intended algorithms and the implemented behavior. Thus, COREFORGE should be viewed as an LLM-generated solver developed under human guidance, rather than as the output of a one-shot code-generation prompt.

The difficulty of implementation varied across components. Incremental totalizer encodings [4] and partitioning-based techniques [5] required more iterations than simpler preprocessing or orchestration code. We also observed that the LLM often favored simpler implementations that were easier to validate, but not necessarily the most performance-oriented versions. This may reflect the way the prompts emphasized correctness and robustness during development.

The final codebase contains more than 25K lines of C++ code. Its size reflects the exploratory nature of the project: we considered more than 30 research papers and implemented several features that were not enabled in the final submitted configurations. As the codebase grew, refactoring became increasingly expensive because it required large generated edits followed by additional validation. Although some unused features were removed or reorganized, the codebase still contains components that should be cleaned in future versions.

III. SOLVER ARCHITECTURE AND CONFIGURATIONS

COREFORGE uses CaDiCaL [6] as the underlying SAT solver. Its main optimization engine is the OLL core-guided algorithm [7], implemented with incremental totalizer cardinality encodings [4]. We submitted three configurations to the MaxSAT Evaluation 2026. Table I summarizes the techniques enabled in each configuration.

The `baseline` configuration uses the OLL core-guided algorithm with AtMost1 preprocessing [8], disjoint-core preprocessing, core exhaustion, and core minimization [8].

The `ilp` configuration extends the baseline with bounded calls to external optimization frameworks. It first computes an initial upper bound by calling a SAT solver and then applying lightweight mutations to the resulting assignment to reduce the number of unsatisfied soft clauses. It then runs SCIP presolving with a 60 s limit, SCIP [9] search with a 90 s limit, and CP-SAT [10] with a 90 s limit. If these backend calls do

TABLE I
COREFORGE CONFIGURATIONS.

Component	baseline	ilp	lookahead
OLL core-guided search	✓	✓	✓
Intrinsic AtMost1 preprocessing	✓	✓	✓
Disjoint-core preprocessing	✓	✓	✓
Bounded core exhaustion	✓	✓	✓
Lightweight core minimization	✓	✓	✓
Initial upper bound	–	✓	✓
SCIP presolve/search	–	✓	✓
CP-SAT backend	–	✓	✓
SCIP presolved formula	–	✓	✓
Short MSU3 phase	–	✓	✓
Core-sequence lookahead	–	–	✓

not solve the instance, the solver continues with core-guided MaxSAT search. The configuration can also continue from the formula produced by SCIP presolving. To avoid excessive backend overhead, ILP backends are disabled on instances with more than 500,000 clauses. When the backend stages do not solve the instance, COREFORGE runs a short MSU3 phase [4] before returning to the main core-guided search.

The `lookahead` configuration extends `ilp` with a new technique that we call *core-sequence lookahead*. The motivation is that different early unsatisfiable-core sequences can lead to different reformulations and substantially different search behavior [11], [12]. Before committing to the main search, the solver runs bounded probes using different strategies for ordering assumptions and processing early cores. The observed prefixes are scored using lightweight statistics, and the solver then restarts the main search using the selected strategy.

Most techniques outside the main search are controlled by explicit resource limits. Preprocessing, backend calls, and lookahead probes are bounded by limits on SAT calls, conflicts per SAT call, and, in most cases, wall-clock time. These limits are intended to prevent auxiliary techniques from consuming too much of the total runtime. Since several components are time-bounded, some nondeterminism may be observed across runs of COREFORGE.

IV. PROVENANCE AND AI-USE DISCLOSURE

COREFORGE was not derived from an existing MaxSAT solver codebase. The submitted solver code was implemented entirely using Codex with GPT-5.4; no solver code was written manually. ChatGPT, using GPT-5.4 and GPT-5.5, was used for paper discussion, design exploration, prompt construction, and audits. Human contributions focused on selecting papers, defining goals, writing prompts, configuring experiments, validating results, and accepting or rejecting generated changes.

We did not perform a full manual code audit of the generated implementation. Therefore, the submitted configurations should be understood as externally validated LLM-generated implementations, described by their source-level configuration options and observed benchmark behavior, rather than as manually verified implementations of each referenced algorithm. In particular, we do not claim that every component faithfully realizes the corresponding paper algorithm.

V. VALIDATION, LESSONS LEARNED, AND LIMITATIONS

The submitted configurations were tested with more than 100K fuzzing runs [13], regression instances, and MaxSAT instances. No wrong answers were observed in these experiments. This evidence should be interpreted cautiously, since testing does not prove correctness or guarantee that every component implements the strategy that motivated it.

The main lesson from COREFORGE is that LLM-assisted paper-to-code workflows can produce a substantial and functional MaxSAT solver, but still require human guidance, benchmarking, and external validation. LLMs were useful for translating algorithmic ideas into code, integrating solver components, and auditing the implementation. However, COREFORGE should be viewed as an experimental LLM-generated solver developed with limited manual engineering, rather than as a fully tuned hand-engineered MaxSAT implementation.

These results suggest several directions for future work. The core-sequence lookahead technique should be refined. A more competitive solver could also start from an existing codebase and use agents to automate more of the planning, implementation, revision, evaluation, and data-analysis loop. Finally, future versions of COREFORGE should include a human audit of the generated code to check that the implementation matches the intended algorithms.

REFERENCES

[1] Y. Sun, F. Ye, Z. Chen, K. Wei, and S. Cai, "Automatically discovering heuristics in a complex SAT solver with large language models," *CoRR*, vol. abs/2507.22876, 2025.

[2] M. Janota and M. Olsák, "LLM2SMT: Building an SMT Solver with Zero Human-Written Code," *CoRR*, vol. abs/2603.06931, 2026.

[3] G. Starace, O. Jaffe, D. Sherburn, J. Aung, J. S. Chan, L. Maksin, R. Dias, E. Mays, B. Kinsella, W. Thompson, J. Heidecke, A. Glaese, and T. Patwardhan, "PaperBench: Evaluating AI's Ability to Replicate AI Research," *CoRR*, vol. abs/2504.01848, 2025.

[4] R. Martins, S. Joshi, V. Manquinho, and I. Lynce, "Incremental cardinality constraints for maxsat," in *CP*, ser. LNCS, vol. 8656. Springer, 2014, pp. 531–548.

[5] M. Neves, R. Martins, M. Janota, I. Lynce, and V. Manquinho, "Exploiting Resolution-Based Representations for MaxSAT Solving," in *SAT*, ser. LNCS, vol. 9340. Springer, 2015, pp. 272–286.

[6] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL 2.0," in *CAV*, ser. LNCS, vol. 14681. Springer, 2024, pp. 133–152.

[7] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-Guided MaxSAT with Soft Cardinality Constraints," in *CP*, ser. LNCS, vol. 8656. Springer, 2014, pp. 564–573.

[8] A. Ignatiev, A. Morgado, and J. Marques-Silva, "RC2: An Efficient MaxSAT Solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 11, no. 1, pp. 53–64, 2019.

[9] T. Achterberg, "SCIP: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[10] L. Perron and V. Furnon, "The CP-SAT-LP solver," in *CP*, ser. LIPIcs, vol. 280. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 3:1–3:2.

[11] J. Alòs, C. Ansótegui, and E. Torres, "Revisiting SAT-Based Solvers: MaxSAT Rules and Core Sequences," *Journal of Artificial Intelligence Research*, vol. 85, p. Article 16, 2026.

[12] A. Schidler and S. Szeider, "Analyzing Reformulation Performance in Core-Guided MaxSAT Solving," in *SAT*, ser. LIPIcs, vol. 341. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, pp. 26:1–26:18.

[13] T. Paxian and A. Biere, "Uncovering and Classifying Bugs in MaxSAT Solvers through Fuzzing and Delta Debugging," in *POS*, ser. CEUR Workshop Proceedings, vol. 3545. CEUR-WS.org, 2023, pp. 59–71.

MaxSAT Evaluation 2026: Solver and Benchmark Descriptions, volume B-2026-1 of Department of Computer Science Series of Publications B, University of Helsinki 2026.

CoreTrail and ShareTrail: Incremental Core-Guided MaxSAT Solvers

Josep Maria Salvia Hornos
University of Lleida
Lleida, Spain
josepmaria.salvia@udl.cat

Alexey Ignatiev
Monash University
Melbourne, Australia
alexey.ignatiev@monash.edu

Joao Marques-Silva
University of Lleida
Lleida, Spain
joaopaulo.marques@udl.cat

***Abstract*—We submit two incremental core-guided MaxSAT solvers to the MaxSAT Evaluation: *CoreTrail* and *ShareTrail*. Both solvers support sequences of related MaxSAT calls with added hard clauses, changing assumptions, and modified soft weights. CoreTrail is a C++ reimplementation of Python RC2 extended with incremental capabilities. ShareTrail extends CGSS2 with incremental solving and ordered-objective support.**

I. Overview

The IPAMIR interface introduces the concept of incremental MaxSAT queries through operations for adding hard clauses, changing assumptions and modifying soft weights [1]. Reusing information across incremental queries for core-guided solvers is an open problem that is not limited to the SAT solver state, for instance the MaxSAT state may have produced reusable core information.

CoreTrail and ShareTrail keep the underlying SAT solver alive across MaxSAT calls and also maintain part of the MaxSAT optimization state. When the active query changes, the solvers repair the optimization state based on the new IPAMIR query. The solvers then continue the ordinary core-guided optimization loop from this new state.

II. Solver Descriptions

CoreTrail. CoreTrail is a new C++ implementation of the Python RC2 solver distributed with PySAT [2]–[4]. RC2 is an OLL core-guided MaxSAT solver using soft cardinality relaxations [5], [6]. CoreTrail extends the OLL process with incremental support while maintaining AM1, core minimization, core trimming, exhaustion and stratification heuristics.

ShareTrail. ShareTrail is based on CGSS2, a C++ solver with refined core relaxation, weight-aware core extraction, and structure sharing for better cardinality constraints, among other improvements [7]. ShareTrail applies the same relaxation repair as CoreTrail, adapted to a larger set of CGSS2 heuristics. We also add ordered-objective processing to CGSS2, using stratification and core simplification techniques motivated by recent work in MaxSAT [8]. Although some heuristics were mantained, full compatibility with CGSS2 options was not done (e.g. Preprocessing or AbstCG).

III. Incremental Behaviour

Both solvers utilize a Glucose backend and natively process the three IPAMIR updates through localized state preservation. Hard clause additions are handled directly by the underlying OLL algorithm without requiring state invalidation. Conversely, shifts in assumptions or soft weights prune the invalidated optimization states reverting relaxations and bounding increments tied to inactive assumptions or modified literals.

After this repair phase, the solvers use a core reconstruction heuristic to evaluate the relevance of historical cores against the active query. Cores can be selectively reapplied in the formula if validated or may also be forgotten, mirroring SAT solver clause learning.

Once this reconstruction phase finishes, both solvers resume the standard OLL optimization loop. The subsequent SAT calls either extract new cores for the current query or certify optimality.

References

[1] A. Niskanen, J. Berg, and M. Järvisalo, "Incremental maximum satisfiability," in *Proc. SAT*, 2022, pp. 14:1–14:19.
[2] A. Ignatiev, A. Morgado, and J. Marques-Silva, "RC2: an efficient MaxSAT solver," *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 53–64, 2019.
[3] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: a python toolkit for prototyping with SAT oracles," in *Proc. SAT*, 2018, pp. 428–437.
[4] A. Ignatiev, Z. L. Tan, and C. Karamanos, "Towards universally accessible SAT technology," in *Proc. SAT*, 2024, pp. 4:1–4:11.
[5] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-guided MaxSAT with soft cardinality constraints," in *Proc. CP*, 2014, pp. 564–573.
[6] R. Martins, S. Joshi, V. Manquinho, and I. Lynce, "Incremental cardinality constraints for MaxSAT," in *Proc. CP*, 2014, pp. 531–548.
[7] H. Ihalainen, J. Berg, and M. Järvisalo, "Refined core relaxation for core-guided MaxSAT solving," in *Proc. CP*, ser. LIPIcs, vol. 210. Schloss Dagstuhl, 2021, pp. 28:1–28:19.
[8] J. Berg, A. Schidler, and M. Järvisalo, "Ordered objectives in maximum satisfiability," in *Proc. AAAI*, vol. 40, no. 17, 2026, pp. 14 166–14 174.

EvalMaxSAT 2026

Florent Avellaneda
Université du Québec à Montréal
Montréal, Canada
Email: avellaneda.florent@uqam.ca

EvalMaxSAT[1] is a MaxSAT solver written in modern C++ language mainly using the Standard Template Library (STL). The solver is built on top of the SAT solver CaDiCaL [1], but any other SAT solver can easily be used instead. EvalMaxSAT is based on the OLL algorithm [2] originally implemented in the MSCG MaxSAT solver [3], [4] and then reused in the RC2 solver [5]. For a general description of how EvalMaxSAT functions, please refer to [6].

The 2026 version of EvalMaxSAT introduces three main modifications compared to the previous version.

I. Internal Local Search

In recent versions, the initial upper bound used by the hardening mechanism was obtained by running an external incomplete solver (NuWLS [7]) for several minutes before launching the main core-guided search. While effective, this approach has several drawbacks: it relies on an external code base, it incurs a non-trivial startup overhead, and it cannot share data structures or propagation information with the rest of the solver.

In the 2026 version, this external dependency is removed and replaced by an internal local search procedure embedded directly in EvalMaxSAT.

The design philosophy of this new procedure is intentionally minimalist. Rather than implementing a sophisticated metaheuristic, the only goal of this local search is to reach a local minimum as quickly as possible. Once such a local minimum is found, control is returned to the core-guided algorithm, which is generally more effective at escaping local minima and at proving optimality.

II. Code Reorganization

The internal architecture of EvalMaxSAT has been significantly refactored in the 2026 version. The two main goals of this refactoring are robustness and extensibility.

On the robustness side, the new architecture isolates each algorithmic component behind clear interfaces. This reduces the risk of regressions when modifying one part of the solver and makes the overall behavior easier to reason about.

On the extensibility side, the new design makes it considerably easier to plug in additional strategies without having to modify the core of the solver. This refactoring also lays the groundwork for the more experimental feature described in the next section.

[1]https://github.com/FlorentAvellaneda/EvalMaxSAT

III. Cardinality Weight Reorganization via Linear Programming

The 2026 version also includes a preliminary implementation of a new technique aimed at reorganizing the weights of the cardinality constraints generated during the core-guided search inspired by [8].

In core-guided MaxSAT solving with the OLL algorithm, each discovered core is relaxed by introducing a new cardinality constraint whose weight is derived from the minimum weight of the soft clauses contained in the core. The remaining weight of these soft clauses is kept for subsequent iterations. The resulting distribution of weights across cardinality constraints is a direct consequence of the order in which cores are discovered, and it is not necessarily optimal from an algorithmic standpoint: redistributing the weights among the existing cardinality constraints can lead to smaller subsequent cores, earlier hardening opportunities, or a tighter lower bound.

To exploit this observation, we have started experimenting with a formulation of the weight redistribution problem as a linear program. The objective is to find an assignment of weights to the existing cardinality constraints that preserves the current cost lower bound while improving the structure exploited by the SAT solver in subsequent calls.

References

[1] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. University of Helsinki, 2020, pp. 51–53.

[2] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-guided MaxSAT with soft cardinality constraints," in *Principles and Practice of Constraint Programming – CP 2014*, ser. Lecture Notes in Computer Science, B. O'Sullivan, Ed., vol. 8656. Springer, 2014, pp. 564–573.

[3] A. Morgado, A. Ignatiev, and J. Marques-Silva, "MSCG: robust core-guided maxsat solving," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 129–134, 2014.

[4] A. Ignatiev, A. Morgado, V. M. Manquinho, I. Lynce, and J. Marques-Silva, "Progression in maximum satisfiability," in *ECAI 2014*, ser. Frontiers in Artificial Intelligence and Applications, vol. 263. IOS Press, 2014, pp. 453–458.

[5] A. Ignatiev, A. Morgado, and J. Marques-Silva, "RC2: an efficient maxsat solver," *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 53–64, 2019.

[6] F. Avellaneda, "A short description of the solver EvalMaxSAT," *MaxSAT Evaluation*, vol. 8, 2020.

[7] Y. Chu, S. Cai, and C. Luo, "NuWLS: Improving local search for (weighted) partial maxsat by new weighting techniques," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 3915–3923.

[8] G. Katsirelos: "Core-Guided Linear Programming-Based Maximum Satisfiability" in *International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, 2025.

EvalMaxRes in MaxSAT Evaluation 2026

1st Shuolin Li*
Aix Marseille Univ, *Université de Toulon*
CNRS, LIS, Marseille, France
shuolin.LI@univ-amu.fr

2nd Sami Cherif
Université de Picardie Jules Verne,
MIS, Amiens, France
sami.cherif@u-picardie.fr

3rd Chu-Min Li
Université de Picardie Jules Verne,
MIS, Amiens, France
Aix Marseille Univ, *Université de Toulon*
CNRS, LIS, Marseille, France
chu-min.li@u-picardie.fr

4th Jialu Zhang*
Université de Picardie Jules Verne,
MIS, Amiens, France
jialu.zhang@u-picardie.fr

I. Introduction

Core-guided approaches have been the dominant paradigm in practical MaxSAT solving for more than a decade [3], before the emergence and consolidation of modern Branch-and-Bound MaxSAT approaches [8]. In the former approaches, the optimization process is driven by the iterative detection of unsatisfiable cores, i.e., subsets of soft clauses that cannot be simultaneously satisfied. Each detected core is then processed by introducing additional constraints that enforce that at least one clause in the core must be falsified. This iterative refinement progressively strengthens the formula until an optimal solution is obtained.

A central aspect of these methods is the encoding of the added cardinality constraints into propositional logic. In practice, such constraints are typically represented using efficient encodings such as totalizers [4], [9], which allow for effective propagation and integration within CDCL-based SAT solving engines. The combination of core extraction and incremental constraint encoding has led to highly efficient solvers that have dominated previous MaxSAT competitions.

In parallel to these developments, an alternative line of research introduced MaxSAT resolution-based core handling [11], a proof-theoretic framework in which unsatisfiable cores are handled through resolution-based transformations applied directly to the formula, rather than through the introduction of auxiliary cardinality constraints. This approach provides a more explicit logical characterization of optimization reasoning and connects MaxSAT solving to classical proof systems [5].

However, despite its theoretical appeal, MaxSAT resolution has not been widely adopted in modern solver implementations, which predominantly rely on CDCL-based search and core-guided constraint strengthening. In particular, since the transition toward modern branch-and-bound MaxSAT frameworks, the use of MaxSAT resolution as a driving inference mechanism has been largely set aside. Earlier approaches preceding this paradigm shift did make use of resolution-based transformations [6], notably to transform disjoint unsatisfiable cores identified during the search. As a result, MaxSAT resolution-based reasoning has since been mostly relegated to preprocessing stages [1], and is no longer used as a primary inference mechanism in state-of-the-art MaxSAT solving.

In this work, we advocate revisiting this direction by proposing a solver architecture that integrates MaxSAT resolution as an internal reasoning component. The goal is to combine the efficiency of modern core-guided approaches with the structural and proof-theoretic advantages of MaxSAT resolution-based transformations, opening new perspectives for optimization, explainability, and the handling of highly structured instances.

II. EvalMaxRes: A Hybrid Core-Based Solver

We combine the traditional cardinality-constraint-based approach with the MaxSAT-resolution-based approach and propose a hybrid solver called EvalMaxRes. EvalMaxSAT [2] is one of the current state-of-the-art core-guided MaxSAT solvers, based on the OLL algorithm [7], [10]. In contrast, the PMRes algorithm [11] proposed a fundamentally different approach based on the MaxSAT resolution rule [5], where unsatisfiable cores are handled through resolution-based formula transformations rather than through auxiliary cardinality constraints.

Our solver EvalMaxRes is thus built on top EvalMaxSAT_2024 and further implements the PMRes core-transformation mechanism through the application of the MaxSAT resolution rule [11]. More specifically, whenever a core $C = \{l_1, \ldots, l_n\}$ is detected, it can be transformed as follows:

$$S = (S \setminus C) \cup \bigcup_{i=1}^{n-1} \{l_i \vee \overline{d_i}\}$$

$$H = H \cup \{(\overline{l_1} \vee \cdots \vee \overline{l_n})\} \cup \bigcup_{i=1}^{n-1} \{d_i \leftrightarrow \overline{l_{i+1}} \vee d_{i+1}\}$$

where S denotes the set of soft clauses, H the set of hard clauses and d_i a fresh variable. Note that the binary clauses introduced in S are simply reified into soft unit literals.

Furthermore, in the case of weighted MaxSAT, the minimal weight is typically extracted and used as in traditional core-guided approaches.

The main idea is thus to select the core-handling strategy according to the size of the detected core. When the core size is not greater than 10, the solver applies MaxSAT resolution-based transformation as in PMRes; otherwise, it uses the standard cardinality-constraint-based method applied in OLL and particularly EvalMaxSAT. We also note that, in the PMRes-based core transformation described above, we add the binary hard clause $\overline{l_1} \vee d_1$ instead of the long clause $\overline{l_1} \vee \cdots \vee \overline{l_n}$. Indeed, it is easy to see that d_1 corresponds to a rewriting of the disjunction $\overline{l_2} \vee \cdots \vee \overline{l_n}$ introduced through the sequential decomposition of the core transformation. Finally, we confirm that no AI-generated code was used in the implementation.

REFERENCES

[1] Josep Argelich, Chu Min Li, and Felip Manyà. A preprocessor for max-sat solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, pages 15–20, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[2] Florent Avellaneda. Evalmaxsat 2024. *MaxSAT Evaluation 2024 Solver and Benchmark Descriptions*, page 8, 2024.

[3] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In *Handbook of satisfiability*, pages 929–991. IOS Press, 2021.

[4] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *International conference on principles and practice of constraint programming*, pages 108–122. Springer, 2003.

[5] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for max-sat. *Artif. Intell.*, 171(8-9):606–618, 2007.

[6] Mohamed Sami Cherif, Djamal Habet, and André Abramé. Understanding the power of max-sat resolution through up-resilience. *Artificial Intelligence*, 289:103397, 2020.

[7] Alexey Ignatiev, António Morgado, and João Marques-Silva. Rc2: an efficient maxsat solver. *Journal on Satisfiability, Boolean Modelling and Computation*, 11(1):53–64, 2019.

[8] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining Clause Learning and Branch and Bound for MaxSAT. In Laurent D. Michel, editor, *CP 2021*, volume 210 of *LIPIcs*, pages 38:1–38:18, Dagstuhl, Germany, 2021.

[9] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. In *International conference on principles and practice of constraint programming*, pages 531–548. Springer, 2014.

[10] António Morgado, Carmine Dodaro, and Joao Marques-Silva. Core-guided maxsat with soft cardinality constraints. In *International conference on principles and practice of constraint programming*, pages 564–573. Springer, 2014.

[11] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

MaxCDCL in MaxSAT Evaluation 2026

1st Chu-Min Li
Université de Picardie Jules Verne,
MIS, Amiens, France
Aix Marseille Univ, *Université de Toulon*
CNRS, LIS, Marseille, France
chu-min.li@u-picardie.fr

2nd Jialu Zhang*
Université de Picardie Jules Verne,
MIS, Amiens, France
jialu.zhang@u-picardie.fr

3rd Shuolin Li*
Aix Marseille Univ, *Université de Toulon*
CNRS, LIS, Marseille, France
shuolin.LI@univ-amu.fr

4th Sami Cherif
Université de Picardie Jules Verne,
MIS, Amiens, France
sami.cherif@u-picardie.fr

I. Introduction

MaxCDCL is a Branch-and-Bound (BnB) MaxSAT solver for unweighted instances. Its main algorithm is based on the Conflict-Driven Clause Learning (CDCL) framework [2], [3]. This paper presents MaxCDCL2026 and the submitted versions to MSE2026.

II. New Techniques in MaxCDCL2026

MaxCDCL2026 implements three new techniques described in this section, which will also be extended into the weighted version WMaxCDCL2026 in the future.

A. Improved Incremental Lower Bounding

A local core κ is a subset of soft clauses that cannot be satisfied simultaneously without falsifying a hard clause under the current partial assignment α. It is associated with a weight $w(\kappa)$, meaning that at least $w(\kappa)$ soft clauses will be falsified in the core if α is extended into a complete one. The lower bound LB is equal to the sum of the weights of the local cores and the number of already falsified soft clauses under α. A soft conflict occurs if LB$\geq$UB. MaxCDCL uses a procedure called *lookahead* to compute LB, which consists in using unit propagation to detect local cores under α.

In MaxCDCL2026, the local cores are managed in a similar way as the trail vector storing the satisfied literals in a CDCL SAT solver. Recall that a variable is assigned a value at a decision level in a CDCL SAT solver. Each local core has a reason, which is a subset of falsified literals: It is because of the falsification of these literals that the soft clauses in the core cannot be satisfied simultaneously without falsifying a hard clause. The level of a core κ is equal to the maximum decision level in its reason. When MaxCDCL backtracks to decision level k, all cores with level greater than k are canceled and all cores with level smaller than or equal to k are kept.

When LB$\geq$UB, i.e., when a soft conflict occurs, the reasons of existing cores, together with the falsified soft clauses, constitute the conflicting clause. Analyzing this conflict clause using a learning scheme such as 1-UIP [8] allows to learn a hard clause explaining the soft conflict and guiding the backtracking in the same way as in a CDCL SAT solver. The incremental lower bounding has the following advantages:

- At decision level k, the existing cores with level smaller than or equal to k need not to be redetected, saving time for the lower bound computing.
- Different soft conflicts can share cores of smaller levels and their reasons. Consequently, the clauses learned from these soft conflicts are more tightly related than the case where all cores are independently detected for different soft conflicts.

B. Incremental Cardinality constraint Encoding

In MaxCDCL, a soft clause c is represented by a soft literal ℓ after adding the hard clauses encoding the equivalence $\ell \equiv c$. When there are n soft clauses, cardinality constraint $\neg\ell_1 + \cdots + \neg\ell_n \leq \mathrm{UB} - 1$ is encoded into CNF if $n \leq 200$ and $n \times (\mathrm{UB} - 1) \leq 10000$ using the sequential encoding of Sinz [6].

Although cardinality constraint encoding is essential for MaxCDCL to solve the instances in the families such as `atmostk` and `single_machine_scheduling`, it can substantially slow down the solving of other instances, probably because of two reasons:

- When n or $n \times (\mathrm{UB} - 1)$ is big, the number of clauses is high, which makes unit propagation slower.
- Cardinality constraint encoding can deviate the search into non-promising area for instances that do not need it.

MaxCDCL2026 implements incremental cardinality constraint encoding. Recall that a soft conflict occurs if LB$\geq$UB, and LB is equal to the sum of the weights of the local cores and the number of falsified soft literals. A soft conflict is said *pure* if there is no core. In other words, a soft conflict is pure if $\neg\ell_1 + \cdots + \neg\ell_n \geq \mathrm{UB}$ under the current partial assignment α.

We conjecture that cardinality constraint encoding is useful only when there are many pure soft conflicts and there are

* Corresponding authors.

many cases $\neg\ell_1 + \cdots + \neg\ell_n = \text{UB} - 1$ in which sequential cardinality constraint encoding should satisfy other soft literals. For this reason, MaxCDCL2026 does not add all clauses in the sequential encoding of Sinz, but only clauses for these special cases.

Example 1: Let UB=3 and ℓ_1, ℓ_2 and ℓ_3 be falsified soft literals. Using Sinz's variable s_{ij} encoding the constraint $\neg\ell_1 + \cdots + \neg\ell_i \geq j$, MaxCDCL2026 adds successively clauses $\ell_1 \vee s_{11}$, $\ell_2 \vee \neg s_{11} \vee s_{22}$ for $\neg\ell_1$ and $\neg\ell_2$, respectively as the reason to satisfy s_{11} and s_{22}, and then a conflict clause $\neg s_{22} \vee \ell_3$ which will be analyzed to learn a hard clause.
This approach has two major advantages :

- When there is no pure soft conflict or the case $\neg\ell_1 + \cdots + \neg\ell_n = \text{UB} - 1$ during the search, no clause is added for the cardinality constraint encoding.
- When there are some pure soft conflicts or some cases $\neg\ell_1 + \cdots + \neg\ell_n = \text{UB} - 1$ during the search, only a subset of clauses encoding the cardinality constraint is added. For instance, in Example 1, MaxCDCL2026 will not add any clause of the form $\neg s_{i2} \vee \ell_{i+1}$ with $i > 2$, while in normal cardinality constraint encoding, these clauses must be added in advance.

In a word, the incremental cardinality constraint encoding allows taking advantage of cardinality constraint encoding while avoiding its drawbacks. In fact, without cardinality constraint encoding, analyzing a pure soft conflict consists of analyzing a long conflict clause $\ell_{i_1} \vee \cdots \vee \ell_{i_{\text{UB}}}$ when UB is big, where each ℓ_{i_j} is falsified under the current partial assignment. In MaxCDCL2026, the conflict clause for a pure soft conflict is a binary or ternary clause from the cardinality constraint encoding, as shown in the above example.

C. Further Lookahead

A drawback in a branch-and-bound algorithm is that when the computed LB is not equal to or greater than UB, the current branch cannot be pruned and the time for the LB computation is lost. MaxCDCL2026 implements further lookahead (FLA) when normal lookahead terminates with LB<UB. The principle is stated as follows.

For each small core $\kappa = \{\ell_1, \ldots, \ell_\lambda\}$ detected by normal lookahead with $w(\kappa) = 1$, FLA tries to independently detect a new core κ_i for each $\ell_i \in \kappa$ after assuming $\ell_i = 0$. If FLA succeeds in detecting κ_i for each ℓ_i, then $\kappa \cup \kappa_1 \cup \cdots \cup \kappa_\lambda$ is a new core with weight 2, allowing to increase LB by 2. Note that κ only allows to increase LB by 1. For further details on this approach, we refer the reader to [7].

D. SCIP Integration

We integrate SCIP [1] (version 9.2.4) into the preprocessing phase of MaxCDCL. Specifically, after MaxCDCL simplifies the original MaxSAT instance, we evaluate the size of the reduced instance. If the reduced instance contains fewer than $100,000$ variables and $150,000$ hard clauses, we invoke SCIP with a given time limit to find an initial upper bound. This bound is subsequently utilized to facilitate pruning during the MaxCDCL search phase. Furthermore, if SCIP successfully finds and proves an optimal solution within the time limit, the solver terminates early.

III. Versions submitted to MSE2026

MaxCDCL2026 is developed on top of MaxCDCL2024. The main modifications are concentrated in the files `Solver.cc`, `ForInc.cc`, `Sinz.cc`, and `FurtherLA.cc`, corresponding to code improvements and the three functionalities introduced in the previous section. No AI-generated code is included in the implementation. We submitted three configurations of MaxCDCL2026 to MSE2026, designed to assess the impact of our proposed components:

- **MaxCDCL:** The standard version of the solver without SCIP integration.
- **MaxCDCL-S2:** A configuration that incorporates SCIP during the preprocessing phase, subject to a strict time limit of 200 seconds for SCIP.
- **MaxCDCL-S2-O3:** A portfolio combining Open-WBO [4], [5] and MaxCDCL-S2, where Open-WBO is executed for the initial 300 seconds, followed by MaxCDCL-S2 for the remaining 3300 seconds.

Acknowledgment

This work is partially supported by the French National Research Agency (ANR) under the Chair MASSAL'IA (ANR-19-CHIA-0013-01), co-funded by the French electricity distribution network operator Enedis, as well as by the project ANR-24-CE23-6126 (BforSAT). It was granted access to HPC resources of "Plateforme MatriCS" within University of Picardie Jules Verne, co-funded by the European Union with the European Regional Development Fund (FEDER) and the Hauts-De-France Regional Council.

References

[1] Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, July 2009.

[2] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining Clause Learning and Branch and Bound for MaxSAT. In Laurent D. Michel, editor, *CP 2021*, volume 210 of *LIPIcs*, pages 38:1–38:18, Dagstuhl, Germany, 2021.

[3] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound maxsat solvers with clause learning. *AI Communications*, 35(2):131–151, 2022.

[4] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Proceedings of SAT 2014*, pages 438–445, 2014.

[5] Ruben Martins, Norbert Manthey, Miguel Terra-Neves, Vasco Manquinho, and Inês Lynce. Open-wbo @ maxsat evaluation 2023. *MaxSAT Evaluation 2023 Solver and Benchmark Descriptions*, pages 18–19, 2023.

[6] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[7] Jialu Zhang, Chu-Min Li, Sami Cherif, and Shuolin Li. Enhanced lower bound computation in branch-and-bound for maxsat. 2026. To appear in the 32nd International Conference on Principles and Practice of Constraint Programming (CP 2026).

[8] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*, pages 279–285. IEEE, 2001.

MaxHS Revised

Edgar Justus, Jakob Berg, André Schidler, Mathias Fleury, Tobias Paxian, Armin Biere
University of Freiburg, Germany

I. Introduction

MaxHS [1]–[3] is an Implicit Hitting Set (IHS) solver that last participated in the MaxSAT Evaluation in 2022. We submit a revised version of the 2022 competition version that was manually revised in conjunction with WCNFuzz to detect bugs [4].

IHS is a portfolio approach that combines a core extractor (here the SAT solver CaDiCaL [5], [6]) and an optimizer (here the Mixed Integer Programming solver SCIP [7]). The core extractor extracts cores, i.e., sets of objectives where at least one objective has to incur cost (alternatively a set of soft clauses where not all clauses can be satisfied). The optimizer then computes a minimum cost hitting set over the known cores, which gives a lower bound on the optimal cost. This hitting set corresponds to objectives/soft clauses incurring cost. In case the hitting set corresponds to an optimal solution, the core extractor can find an assignment where no objective outside the hitting set incurs cost and the solver is done. Otherwise, the core extractor will find at least one previously unknown core not hit by the hitting set and the process repeats. While termination usually happens earlier, the process is guaranteed to terminate when all cores have been extracted, in which case the hitting set is guaranteed to correspond to an optimal solution.

II. Changes

We briefly state the difference between the 2022 and 2026 version of MaxHS. Our focus was making MaxHS more robust. Hence, we did not implement many novel features and mainly fixed bugs identified by WCNFuzz.

A key change is the integration of SCIP 10.0.2 [7] as an open source alternative to CPLEX. Furthermore, we offer an exact mode that leverages SCIP's numerically stable mode [8], [9]. Without this mode SCIP and CPLEX suffer from numerical issues on instances with large weights.

This version of MaxHS also contains updated dependencies: We updated the preprocessing to MaxPre 2.2 [10], [11] and CaDiCaL to 3.0 [5], [6] with deactivated ILB, Lucky Assumptions, and Factoring.

The most up-to-date version can be found under https://github.com/aschidler/MaxHS.

References

[1] J. Davies and F. Bacchus, "Exploiting the power of mip solvers in maxsat," in *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings* (M. Järvisalo and A. V. Gelder, eds.), Lecture Notes in Computer Science, pp. 166–181, Springer, 2013.

[2] J. Davies, *Solving MAXSAT by Decoupling Optimization and Satisfaction*. PhD thesis, University of Toronto, Canada, 2014.

[3] F. Bacchus, "Maxhs in the 2022 maxsat evaluation," *MaxSAT Evaluation 2022 : Solver and Benchmark Descriptions*, p. 17, 2022.

[4] T. Paxian and A. Biere, "Maxsat fuzzing and delta debugging," *J. Artif. Intell. Res.*, vol. 85, 2026.

[5] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "Cadical 2.0," in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I* (A. Gurfinkel and V. Ganesh, eds.), Lecture Notes in Computer Science, pp. 133–152, Springer, 2024.

[6] F. Pollitt, M. Fleury, K. Fazekas, N. Froleyks, A. Schidler, D. Schreiber, and A. Biere, "Cadical 3.0," in *29th International Conference on Theory and Applications of Satisfiability Testing, SAT 2026*, 2026. to appear.

[7] C. Hojny, M. Besançon, K. Bestuzheva, S. Borst, A. Chmiela, J. Dionísio, L. Eifler, M. Ghannam, A. Gleixner, A. Göß, A. Hoen, R. van der Hulst, D. Kamp, T. Koch, K. Kofler, J. Lentz, S. J. Maher, G. Mexi, E. Mühmer, M. E. Pfetsch, S. Pokutta, F. Serrano, Y. Shinano, M. Turner, S. Vigerske, M. Walter, D. Weninger, and L. Xu, "The SCIP Optimization Suite 10.0," technical report, Optimization Online, November 2025.

[8] W. J. Cook, T. Koch, D. E. Steffy, and K. Wolter, "A hybrid branch-and-bound approach for exact rational mixed-integer programming," *Math. Program. Comput.*, vol. 5, no. 3, pp. 305–344, 2013.

[9] L. Eifler and A. M. Gleixner, "A computational status update for exact rational mixed integer programming," *Math. Program.*, vol. 197, no. 2, pp. 793–812, 2023.

[10] H. Ihalainen, J. Berg, and M. Järvisalo, "Clause redundancy and preprocessing in maximum satisfiability," in *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings* (J. Blanchette, L. Kovács, and D. Pattinson, eds.), Lecture Notes in Computer Science, pp. 75–94, Springer, 2022.

[11] H. Ihalainen, A. Oertel, Y. K. Tan, J. Berg, M. Järvisalo, M. O. Myreen, and J. Nordström, "Certified MaxSAT preprocessing," in *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I* (C. Benzmüller, M. J. H. Heule, and R. A. Schmidt, eds.), Lecture Notes in Computer Science, pp. 396–418, Springer, 2024.

UWrMaxSat 2.0 in the 2026 MaxSAT Evaluation

Marek Piotrów
retired from the Institute of Computer Science,
University of Wrocław, Wrocław, Poland
marek.piotrow@cs.uni.wroc.pl

***Abstract*—UWrMaxSat 2.0 is an efficient solver for partial weighted MaxSAT instances, linear and non-linear pseudo-Boolean ones and weighted Boolean optimization formulas (WBO). It can be run as a stand-alone solver, but it is much more profitable to use it in pair with the SCIP Optimization Solver. Both solvers not only execute different solving strategies, but can also exchange information such as: lower and upper bounds, solutions found (not necessarily optimal ones), etc. Therefore, it is natural to run them in parallel, using different threads. This is a default configuration in UWrMaxSat, but a sequential run can be forced by a parameter, when a single thread is required. Several solving algorithms can be selected by parameters of UWrMaxSat, but the default solving method is based on the core-guided OLL procedure.**

The default SAT-solving oracle is the CaDiCaL solver by Armin Biere *et al.* . Clauses preprocessed by it are converted to linear constraints and input to the SCIP solver. This can significantly reduce the total size of linear constraints. The standard interface of CaDiCaL library was wrapped with MiniSAT-like set of functions. In this way, the solver can be easily replaced by a few other MiniSAT-based SAT solvers.

This paper describes the new elements of UWrMaxSat 2.0 compared to version 1.6, which participated in MaxSat Evaluation 2024.

***Index Terms*—MaxSAT-solver, UWrMaxSat, CaDiCaL, SCIP, sorter-based encoding, core-guided, complete solver**

I. Description

This is the sixth time when UWrMaxSat is participating in MaxSAT Evaluation (MSE). The progress in its development one can find in the series of publications [9], [10], [12]–[14] and a more complete description in [11]. We can summarizing the most important external components of UWrMaxSat as follows: CaDiCaL [3] by Armin Biere *et al.* is the default SAT solver since version 1.7. In this competition we use it in version 2.1.3. The two other components that can be used together with UWrMaxSat are: the SCIP solver, version 10.0.2 [5] and the MaxPre preprocessor [7] by Tuukka Korhonen *et al.* The code of UWrMaxSat was developed on the base of kp-minisatp by Marek Piotrów and Michał Karpiński, that, in turn, was an extension of the well-known pseudo-Boolean (PB) solver MiniSat+ by Niklas Eén and Niklas Sörensson [4]. In kp-minisatp a new encoding of PB linear constraints was proposed and implemented [6]. Git diff statistics shows that in UWrMaxSat 2.0 66 files were changed with 9471 insertions and 1007 deletions compared with MiniSat+ (git: d91742bc).

In this paper we present new elements of UWrMaxSat 2.0 compared to the version 1.6, which participated in the 2024 MaxSat Evaluation. The main extensions are: (1) new input file formats, (2) new default SAT solver, (3) parametrized core-minimization procedure and (4) an extended cooperation with SCIP.

(1) UWrMaxSat can now read and solve weighted Boolean optimization formulas (WBO) and non-linear pseudo-Boolean decision and optimization instances.
(2) CaDiCaL is now the default SAT solver. A clause iterator is used to get preprocessed clauses from CaDiCaL, convert them to linear constraints and input to the SCIP solver. In particular, UWrMaxSat is now compatible with the new API of CaDiCaL, required by the incremental version of BVA (bounded value addition).
(3) Each core returned by the SAT solver is minimized inside the MaxSAT solver by a sequence of additional calls to the SAT solver. The procedure is time consuming, therefore, it should be limited somehow. UWrMaxSat uses the number of conflicts to limit it. Now, this is parameterized.
(4) The generalized Boolean multi-level optimization (GBMO, see [8]) technique, which was used already in UWrMaxSat, is now extended to the SCIP solving procedure. GBMO splitting points computed inside the MaxSAT solver can be used to split a SCIP objective function into segments and to allow SCIP to optimize them one after another. Other two improvement are: information sharing about fixed variable values and a default delayed start of the SCIP solver in order to allow UWrMaxSat to precompute lower and upper bounds on the optimum value and to share them with SCIP. Fixed values of Boolean variables are also passes to SCIP.

Finally, we continue to use the MaxPre preprocessor [7] in the unweighted exact track with time limit of 60s for it. Generally, preprocessing techniques are an important part of an system for solving MaxSAT problems. I also strongly believe that MaxSAT solvers should be multi-threaded, because modern processors offer several cores and they can be used to run different solving strategy in parallel. In addition, partial results can be exchanged among threads.

References

[1] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving sat-based weighted maxsat solvers. In Michela Milano, editor, *Proc. CP*, pages 86–101. Springer, 2012.
[2] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf

van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. ZIB-Report 21-41, Zuse Institute Berlin, December 2021.

[3] 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.

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

[5] Christopher Hojny, Mathieu Besançon, Ksenia Bestuzheva, Sander Borst, Antonia Chmiela, João Dionísio, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Adrian Göß, Alexander Hoen, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Stephen J. Maher, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Sebastian Pokutta, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Matthias Walter, Dieter Weninger, and Liding Xu. The SCIP Optimization Suite 10.0. Technical report, Optimization Online, November 2025.

[6] Michał Karpiński and Marek Piotrów. Encoding cardinality constraints using multiway merge selection networks. *Constraints*, Apr 2019.

[7] Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. MaxPre: An extended MaxSAT preprocessor. In Serge Gaspers and Toby Walsh, editors, *Proc. SAT*, volume 10491 of *LNCS*, pages 449–456. Springer, 2017.

[8] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted maxsat. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 556–577, Cham, 2021. Springer International Publishing.

[9] Marek Piotrów. Uwrmaxsat - a new minisat+-based solver in maxsat evaluation 2019. In Fahiem Bacchus, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2019 : Solver and Benchmark Descriptions*, Department of Computer Science Report Series B-2019-2, pages 11–12, 2019.

[10] Marek Piotrów. Uwrmaxsat - an efficient solver in maxsat evaluation 2020. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2020 : Solver and Benchmark Descriptions*, Department of Computer Science Report Series B-2020-2, pages 34–35, 2020.

[11] Marek Piotrów. Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 132–136, 2020.

[12] Marek Piotrów. Uwrmaxsat in maxsat evaluation 2021. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2021 : Solver and Benchmark Descriptions*, Department of Computer Science Report Series B-2021-2, pages 17–18, 2021.

[13] Marek Piotrów. Uwrmaxsat in maxsat evaluation 2022. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen, editors, *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*, Department of Computer Science Series of Publications B, vol. B-2022-2, pages 21–22, 2022.

[14] Marek Piotrów. Uwrmaxsat entering the maxsat evaluation 2024. In Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian, editors, *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, Department of Computer Science Series of Publications B, vol. B-2024-2, pages 27–28, 2024.

RustiMax: A New OLL-based MaxSAT Solver, Agentically Engineered

Stefan Szeider
Algorithms and Complexity Group
TU Wien, Vienna, Austria
stefan@szeider.net

***Abstract*—RustiMax is a core-guided MaxSAT solver written from scratch in Rust. Its algorithmic core is the OLL algorithm with stratification and hardening, extended with several components novel to this submission. Two backend variants ship: `rustimax-c` uses a vendored CaDiCaL 1.9.5 SAT backend; `rustimax-r` uses RustiCal, a new pure-Rust CDCL SAT solver bundled in the source tree. Both are 100% AI-written code utilizing a multi-agent engineering architecture: the human author contributed direction, review, and research questions but did not write any code.**

I. Motivation

A Rust-native, Rust-only MaxSAT solver. A Rust MaxSAT solver linked against a C/C++ SAT backend is not end-to-end Rust. Related Rust efforts in the SAT/MaxSAT space include RustSAT [5] and Scuttle [6]. We built RustiCal, a pure-Rust CDCL SAT solver with an inprocessing stack (BVE, subsumption, vivification, failed-literal probing, ternary resolution, transitive reduction, ProbSAT walk), bundled as the backend of `rustimax-r`. Rust's safety guarantees rule out use-after-free, double-free, and data-race bugs by construction in both the MaxSAT kernel and the RustiCal backend.

Agentic software engineering as a test case. We wanted to see how far a supervised agentic coding setup could push a competitive solver implementation from scratch, with zero hand-written code, and minimal human supervision. Section IV documents the process.

II. Overview and Algorithm

RustiMax's core is the OLL algorithm [1], combined with the stratification and hardening refinements [3]. OLL iteratively extracts unsatisfiable cores under soft-literal assumptions and relaxes them through lazily-built totalizer [2] cardinality constraints.

The solver is organized as a seven-step pipeline:

1) **Fast path**. On very large instances (`num_clauses / timeout > 200,000`), skip preprocessing, run a single plain SAT call, and emit any discovered model as the anytime solution.
2) **SLS warmstart**. Run a stochastic local search once over a wall-clock budget and, on success, seed both the OLL upper bound and the SAT solver's preferred-phase vector before any clause learning. The local search uses BMS-sampled unit selection, decimation-based initialization, periodic *sw* + *paws* clause-weight smoothing, and a swap stack for incremental score updates. Default-on in the submission envelope.
3) **At-most-one (AM1) preprocessing**. Detect mutually exclusive soft literals using three Boolean constraint propagation (BCP)-driven methods (exact maximum-clique, fast probe, very-fast probe) and an at-most-two (AM2) rank-2 generalization; raise the lower bound and introduce a single residual soft per clique.
4) **Upper-bound (UB) seed**. Run an *isolated* SAT solver on only the hard clauses with a size-aware time budget and, on SAT, feed the solution's cost to the main solver as an initial UB. The isolated solver's learned clauses are never merged into the OLL solver, avoiding a contamination pathology we observed when a single solver was shared across preprocessing and OLL.
5) **Shared-solver fallback seed**. On very large instances where the isolated UB seed was refused by the memory pre-flight guard, run a single time-budgeted plain solve on the main solver as a last-resort UB.
6) **Family-specific primal**. Detect Minimum-Weight-Dominating-Set (MWDS) instance shape (hard clauses encoding dominance, soft clauses encoding vertex weights) and construct an initial UB by greedy SAT-verified minimum-weight covering. On detected MWDS instances the UB seeded here is, in our internal benchmarks, tighter than the SLS warmstart, because pure OLL extracts mostly weight-1 cores that carry no stratification signal.
7) **OLL main loop**. Iterate assumed solves: on UNSAT, extract a core, deduce the stratum-weight increment, and add a fresh lazy totalizer; on SAT, lower the stratum threshold or record an improving model. Across iterations, cache and re-inject the SAT solver's preferred-phase vector on warm calls, and run a bounded conflict-limited probe for a Hamming-neighborhood improvement after each UB update.

Three safety nets follow the main loop: a three-phase rescue for stalls; a random-feasibles fallback (default-on, since the anytime score is zero on no-solution runs) that emits the cheapest verified feasible; and a 150-second cap on the first outer solve to prevent starvation. Three further mechanisms govern time and bound use: an adaptive Lambert-W scheme apportions effort across cores; a UB-driven local optimizer

attempts a cheaper assignment between OLL iterations; and monotone lower-bound updates are emitted at each iteration alongside the upper-bound trace, per the MSE 2026 protocol.

Memory and input robustness. The run scripts pass `-mem-limit 28000` (MB) sized for the MSE 2026 32 GB runsolver allocation (override via `RUSTIMAX_MEM_CAP_MB`). The WCNF parser bounds literal indices at 2^{26} with graceful exit on out-of-range inputs.

Attribution. The SLS warmstart is a clean-room port of NuWLS-c-IBR [7], [8]; AM1 clique-based preprocessing is standard MaxSAT machinery. All other pipeline components (AM2 rank-2 generalization, isolated-solver UB seed, family-specific MWDS primal, SAT-phase cache, fast-path repair, and the vendored CaDiCaL BCP-API patch) are novel to this submission; full detail in the shipped `OLL-spec.md`.

III. SAT Backends

Both backends expose the same internal `Sat` trait (`src/sat.rs`, 715 LoC) so the MaxSAT kernel is identical across variants.

`rustimax-c` links against CaDiCaL 1.9.5 [4], vendored in-tree and extended with the BCP-propagate patch mentioned in Section II that exposes level-0 unit propagation as an explicit API (≈200 LoC).

`rustimax-r` links against RustiCal, a pure-Rust CDCL SAT solver developed alongside RustiMax and bundled in the submission source tree. RustiCal (≈17,800 LoC) is an algorithmic port of CaDiCaL covering the full CDCL search, the inprocessing stack enumerated in Section I, and incremental solving. It exposes an IPASIR C interface, DRAT proof logging, and a MaxSAT integration API used directly by RustiMax. RustiCal is submitted as part of the RustiMax source package; it is *not* a separate solver submission.

Both binaries are built static-musl with an x86-64-v3 codegen target and the mimalloc allocator. `rustimax-r` has no C or C++ runtime dependencies; `rustimax-c` embeds the vendored CaDiCaL statically. Single-file deployment.

IV. Autonomous Agentic Engineering

We used Claude Code (Anthropic's command-line coding harness) driving Claude Opus 4.6 and Opus 4.7 models (1M-token context window); occasional research cross-checks used other frontier models via MCP tools, with no code from those tools retained.

Custom dual-agent architecture. Each project (RustiMax and RustiCal) was developed by a supervised pair of agents in separate `tmux` sessions. An *outer (orchestrator) agent* handles strategy, cluster benchmarking, session-log maintenance, and cross-project coordination; it does not write code. An *inner (implementation) agent*, spawned by the outer agent per focused task (typically 1–8 hours, 100–2,000 LoC), reads its prompt, the relevant specification, and the dead-ends registry, then writes code, runs local tests and fuzzers, commits, pushes, and deploys built binaries. Continuity across inner sessions is achieved by an accumulating memory system following Xia et al. [9] that records design rationale, dead ends, and institutional knowledge. A rule file codifies deployment discipline (every binary corresponds to a specific git commit) and the independent-implementation rule (below). The inner agents accessed a Linux cluster server via an MCP server to carry out benchmark runs. Once both projects reached a working-code state, their outer agents began direct coordination via a push-based MCP intercom channel, used for cross-project release-candidate testing and debugging.

100% AI-authored code. All ≈36,500 lines of Rust source (RustiMax, ≈18,100 LoC; RustiCal, ≈18,400 LoC; including the ≈200-LoC BCP-propagate patch on vendored CaDiCaL) were written by the implementation agents. Human input consisted of research direction, veto and redirection, and review of the agentic trace across ≈104 RustiMax and ≈98 RustiCal outer-agent sessions over six weeks (each supervising 1–10 inner-agent sessions depending on task scale); instructions to the outer agent were task-level, not line-level.

Independent implementation: spec-only for RustiMax, open-reference for RustiCal. RustiMax's implementation agent had no read access to any MaxSAT solver source tree at any point; a 1000-line specification file in markdown format, written by a separate spec agent from published papers alone, was the sole algorithmic reference. RustiCal, by contrast, openly used CaDiCaL's source [4] as a structural port reference, function by function, with every component reimplemented from scratch in Rust. Both projects are released under the MIT license.

V. Availability

Each release candidate passes the MSE 2026 anytime regression suite on both backends and accumulates at least 600,000 MaxSAT-fuzzer iterations per backend.

The complete source for both projects is included in the submission tarball; we plan to release it publicly on GitHub after the MSE 2026 evaluation. Both projects are MIT-licensed; runtime dependencies are the vendored CaDiCaL 1.9.5 (MIT, `rustimax-c` only) and `signal-hook` (Apache-2.0/MIT). No closed-source components.

References

[1] A. Morgado, C. Dodaro, and J. Marques-Silva. "Core-Guided MaxSAT with Soft Cardinality Constraints." *CP 2014*.

[2] O. Bailleux and Y. Boufkhad. "Efficient CNF Encoding of Boolean Cardinality Constraints." *CP 2003*.

[3] J. Berg, E. Demirović, and P. J. Stuckey. "Core-Boosted Linear Search for Incomplete MaxSAT." *CPAIOR 2019*.

[4] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020." *SAT Competition 2020*.

[5] C. Jabs. "RustSAT: A Library for SAT Solving in Rust." *SAT 2025* (LIPIcs 341:15).

[6] C. Jabs. "Scuttle: A Multi-Objective MaxSAT Solver in Rust." https://github.com/chrjabs/scuttle.

[7] Z. Lei, S. Cai, et al. "NuWLS: Improving Local Search for (Weighted) Partial MaxSAT." *AAAI 2023*.

[8] Z. Lei, S. Cai, et al. "NuWLS-c-IBR Solver Description." *MaxSAT Evaluation 2024*.

[9] H. Xia, C. P. Gomes, B. Selman, and S. Szeider. "Agentic Neurosymbolic Collaboration for Mathematical Discovery: A Case Study in Combinatorial Design." *Neurosymbolic Learning and Reasoning (NeSy 2026)*. Preprint: arXiv:2603.08322.

UWrMaxSat-MaxCDCL: Solver Description

Mingming Jin*, Yujie Cai*, Jiongzhi Zheng*, Kun He*
*School of Computer Science and Technology, Huazhong University of Science and Technology
{mingmingk, yujie_cai, jzzheng, brooklet60}@hust.edu.cn,

***Abstract*—This document describes UWrMaxSat-MaxCDCL, submitted to the Exact-UW track of MaxSAT Evaluation 2026. The solver is based on the UWrMaxSat version submitted to MaxSAT Evaluation 2024. Compared with that version, our submission mainly adds lightweight engineering support for using MaxCDCL as a sequential fallback on unweighted instances.**

I. Description

UWrMaxSat-SCIP-MaxPre-MaxCDCL is a complete solver for unweighted partial MaxSAT. It is based on UWrMaxSat [1], more specifically on the UWrMaxSat-SCIP-MaxPre[1] version submitted to MaxSAT Evaluation 2024 [2]. We keep the original UWrMaxSat architecture, including MaxPre [3] and SCIP [4], and add bridge code and run-script support for an embedded MaxCDCL fallback and optional SPB upper-bound probing.

The submitted Exact-UW configuration first runs UWrMaxSat with MaxPre enabled and with SCIP delayed. MaxCDCL[2] [5] is called as a sequential fallback on the original unweighted instance when, after the delayed SCIP phase and continued UWrMaxSat search, the remaining lower–upper-bound gap is still large. The source package also contains optional support for using SPB-MaxSAT[3] [6] as an upper-bound provider for MaxCDCL.

II. Provenance

The bridge code for integrating MaxCDCL and SPB-MaxSAT into UWrMaxSat was developed with assistance from OpenAI ChatGPT/Codex.

References

[1] M. Piotrów, "Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems," in *2020 IEEE 32nd international conference on tools with artificial intelligence (ICTAI)*. IEEE, 2020, pp. 132–136.

[2] ——, "Uwrmaxsat entering the maxsat evaluation 2024," *MaxSAT Evaluation 2024*, p. 27, 2024.

[3] T. Korhonen, J. Berg, P. Saikko, and M. Järvisalo, "Maxpre: an extended maxsat preprocessor," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 449–456.

[4] C. Hojny, M. Besançon, K. Bestuzheva, S. Borst, J. Dionísio, J. Ehls, L. Eifler, M. Ghannam, A. Gleixner, A. Göß *et al.*, "The scip optimization suite 10.0," *arXiv preprint arXiv:2511.18580*, 2025.

[5] C.-M. Li, Z. Xu, J. Coll, F. Manyà, D. Habet, and K. He, "Combining clause learning and branch and bound for maxsat," in *27th International conference on principles and practice of constraint programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[6] J. Zheng, Z. Chen, C.-M. Li, and K. He, "Rethinking the soft conflict pseudo boolean constraint on maxsat local search solvers," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 2024, pp. 1989–1997.

[1] https://maxsat-evaluations.github.io/2024/mse24-solver-src/exact/weighted/UWrMaxSat-SCIP-MaxPre.zip

[2] http://home.mis.u-picardie.fr/~cli/MaxCDCL1.2.zip

[3] https://github.com/JHL-HUST/SPB-MaxSAT.git

WMaxCDCL in MaxSAT Evaluation 2026

1st Jialu Zhang
Université de Picardie Jules Verne,
MIS, Amiens, France
jialu.zhang@u-picardie.fr

2rd Shuolin Li*
Aix Marseille Univ, *Université de Toulon*
CNRS, LIS, Marseille, France
shuolin.LI@univ-amu.fr

3nd Chu-Min Li*
Université de Picardie Jules Verne,
MIS, Amiens, France
Aix Marseille Univ, *Université de Toulon*
CNRS, LIS, Marseille, France
chu-min.li@u-picardie.fr

4th Sami Cherif
Université de Picardie Jules Verne,
MIS, Amiens, France
sami.cherif@u-picardie.fr

I. Introduction

WMaxCDCL is a Branch-and-Bound (BnB) Weighted Partial MaxSAT (WPMS) solver. Its main algorithm is based on the Conflict-Driven Clause Learning (CDCL) framework [3], [4], integrating components tailored for WPMS and novel techniques to enhance overall efficiency [2], [5].

In this system description, we introduce WMaxCDCL2026, an updated version of WMaxCDCL for the MaxSAT Evaluation (MSE) 2026. This solver builds upon WMaxCDCL2024 [6], which secured first place in the unweighted category of MSE 2024 and placed 9th among 15 competitors in the weighted category. Compared to WMaxCDCL2024, WMaxCDCL2026 integrates new techniques to enhance its performance on WPMS.

II. New Techniques in WMaxCDCL2026

A. Stratified Hardening Strategy

Inspired by stratification techniques in core-guided algorithms, WMaxCDCL2026 implements a preprocessing strategy to exploit the hierarchical structure in WPMS instances.

The strategy consists of hardening high-weight soft clauses to simplify Boolean Multilevel Optimization (BMO [7]) instances. Specifically, for BMO instances, the hardening strategy iteratively hardens soft clauses in descending order of weights and invokes a SAT oracle to check the satisfiability of the extended hard clause set H. If H remains satisfiable, the hardened clauses are removed from the soft clause set S. Conversely, if H becomes unsatisfiable, the algorithm restores consistency by discarding the clauses hardened in the current iteration and terminates the hardening phase. The implementation details of this strategy are provided in [10].

This strategy allows WMaxCDCL2026 to solve more BMO-structured instances w.r.t WMaxCDCL2024, notably from the *frb* and *drmx-cryptogen* families.

* Corresponding authors.

B. Weight-Aware Lookahead

In WMaxCDCL2024, the lookahead framework does not prioritize soft literals based on their weights, potentially mixing literals of significantly disparate magnitudes into a local core. Since the weight of a local core is constrained by the minimum weight of the soft literals within it, this leads to the generation of low-weight local cores, diminishing the efficiency of the lookahead framework. As illustrated in Example 1, the lookahead procedure needs to detect two local cores to meet the pruning condition. This is because the first detected local core κ_1 contains the soft literal l_3 with weight $w(l_3) = 1$, which makes κ_1 low-weight and reduces the lookahead efficiency.

Example 1: Consider soft literals $S = \{l_1, l_2, l_3\}$ with weights $w(l_1) = w(l_2) = 10$, $w(l_3) = 1$, upper bound $\mathrm{UB} = 10$, and hard clauses $H = (x_1 \vee \neg l_1 \vee \neg l_3) \wedge (\neg l_1 \vee \neg l_2 \vee l_3)$. The remaining weight $rw(l)$ of each literal l is initialized to its weight $w(l)$. Given the partial assignment $\alpha = \{\neg x_1\}$, we illustrate a typical case of the classic lookahead procedure:

1) Propagating l_1 implies $\neg l_3$, yielding a local core $\kappa_1 = \{l_1, l_3\}$ with weight $w(\kappa_1) = \min(rw(l_1), rw(l_3)) = 1$. The remaining weights are updated to $rw(l_1) \leftarrow 9, rw(l_3) \leftarrow 0$ and $\mathrm{LB} = 1$.
2) Propagating l_1 makes l_3 falsified and the second hard clause unit, making $l_2 = 0$. This identifies a local core $\kappa_2 = \{l_1, l_2\}$ with weight $w(\kappa_2) = \min(rw(l_1), rw(l_2)) = 9$. Subsequently, the remaining weights are updated to $rw(l_1) \leftarrow 0$ and $rw(l_2) \leftarrow 1$, and $LB = 10$. Finally, the lookahead terminates since $LB \geq UB$.

To overcome the limitations of the classic lookahead, WMaxCDCL2026 implements a weight-aware lookahead. The key differences from the classic lookahead are as follows:

1) **Soft literal selection:** While classic lookahead selects soft literals for propagation based primarily on their recent conflict activity, weight-aware lookahead prioritizes soft literals with higher remaining weights, using activity scores strictly as a tie-breaker.

2) **Unit propagation:** Classic lookahead aborts propagation immediately upon falsifying a single soft literal. In contrast, weight-aware lookahead exhausts unit propagation, thereby allowing multiple soft literals to be falsified and evaluated.
3) **Core extraction:** Classic lookahead extracts a core from the first falsified soft literal, which frequently yields a low-weight core. Conversely, weight-aware lookahead evaluates all falsified soft literals to construct a high-weight core.

As illustrated in Example 2, the weight-aware lookahead procedure only needs to detect a high-weight core to prune the branch. We refer the reader to [10] for detailed implementation and advanced scenarios.

Example 2: Considering the same input as in Example 1, we illustrate the execution trace of the weight-aware lookahead: Propagating l_1 implies $\neg l_3$. In Example 1, the propagation stopped and a core $\kappa_1 = \{l_1, l_3\}$ with $w(\kappa_1) = 1$ is identified. However, in our new strategy, we do not stop unit propagation, but continue falsifying l_2. Then, we have two falsified soft literals $\{l_2, l_3\}$. Since $rw(l_2) > rw(l_3)$, we choose l_2 to identify the local core $\kappa_2 = \{l_1, l_2\}$ with a weight of $w(\kappa_2) = 10$, raising the lower bound to $LB = 10$ by itself. So, only one local core is sufficient to prune the current branch, because we already have $LB \geq UB$.

C. SCIP Integration

We integrate SCIP [1] (version 9.2.4) into the preprocessing phase of WMaxCDCL. Specifically, after WMaxCDCL simplifies the original MaxSAT instance, we evaluate the size of the reduced instance. If the reduced instance contains fewer than $100,000$ variables and $200,000$ hard clauses, we invoke SCIP with a given time limit to find an initial upper bound. This bound is subsequently utilized to facilitate pruning during the WMaxCDCL search phase. Furthermore, if SCIP successfully finds and proves an optimal solution within the time limit, the solver terminates early.

III. Versions submitted to MSE2026

WMaxCDCL2026 is developed on top of WMaxCDCL2024. The primary code modifications are localized within the files `BLO.h`, `lookahead.cc`, `Solver.cc`, and `scip_solver.cc`, which directly implement the novel techniques detailed in the preceding sections. No AI-generated code is included in the implementation. We submitted three configurations of WMaxCDCL2026 to MSE2026, designed to assess the impact of our proposed components:

- **WMaxCDCL:** The standard version of the WMaxCDCL2026 solver without SCIP integration.
- **WMaxCDCL-S6:** A configuration that incorporates SCIP during the preprocessing phase, subject to a strict time limit of 600 seconds for SCIP.
- **WMaxCDCL-S6-O3:** A portfolio combining Open-WBO [8], [9] and WMaxCDCL-S6, where Open-WBO is executed for the initial 300 seconds, followed by WMaxCDCL-S6 for the remaining 3300 seconds.

Acknowledgment

This work is partially supported by the French National Research Agency (ANR) under the Chair MASSAL'IA (ANR-19-CHIA-0013-01), co-funded by the French electricity distribution network operator Enedis, as well as by the project ANR-24-CE23-6126 (BforSAT). It was granted access to HPC resources of "Plateforme MatriCS" within University of Picardie Jules Verne, co-funded by the European Union with the European Regional Development Fund (FEDER) and the Hauts-De-France Regional Council.

References

[1] Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, July 2009.

[2] Jordi Coll, Chu-Min Li, Shuolin Li, Djamal Habet, and Felip Manyà. Solving weighted maximum satisfiability with branch and bound and clause learning. *Computers & Operations Research*, 183:107195, 2025.

[3] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining Clause Learning and Branch and Bound for MaxSAT. In Laurent D. Michel, editor, *CP 2021*, volume 210 of *LIPIcs*, pages 38:1–38:18, Dagstuhl, Germany, 2021.

[4] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound maxsat solvers with clause learning. *AI Communications*, 35(2):131–151, 2022.

[5] Shuolin Li, Chu-Min Li, Jordi Coll, Djamal Habet, and Felip Manyà. Improving the lower bound in branch-and-bound algorithms for maxsat. AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025.

[6] Shuolin Li, Chu Min Li, Jordi Coll, Djamal Habet, Felip Manyà, and Kun He. Wmaxcdcl in maxsat evaluation 2024. *MaxSAT Evaluation 2024 Solver and Benchmark Descriptions*, pages 17–18, 2024.

[7] Joao Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: Algorithms & applications. 62(3–4):317–343, 2011.

[8] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Proceedings of SAT 2014*, pages 438–445, 2014.

[9] Ruben Martins, Norbert Manthey, Miguel Terra-Neves, Vasco Manquinho, and Inês Lynce. Open-wbo @ maxsat evaluation 2023. *MaxSAT Evaluation 2023 Solver and Benchmark Descriptions*, pages 18–19, 2023.

[10] Jialu Zhang, Chu-Min Li, Sami Cherif, and Shuolin Li. Weight-aware branch-and-bound for weighted maximum satisfiability. 2026. To appear in proceedings of IJCAI-26.

ANYTIME SOLVERS

Aperture: an Anytime, Complete and Incremental MaxSAT Solver Entering MaxSAT Evaluation 2026

Alexander Nadel, Yam Slonimski
Faculty of Data and Decision Sciences, Technion, Haifa, Israel
alexandernad@technion.ac.il, yamslonimski@campus.technion.ac.il

***Abstract*—This document describes `Aperture`, an anytime, complete and incremental MaxSAT solver submitted to the MaxSAT Evaluation 2026. `Aperture` is a new solver written from scratch in C++ that follows the high-level anytime optimization paradigm introduced in previous anytime solvers, based on `TT-Open-WBO-Inc`, while addressing their architectural limitations. In particular, `Aperture` provides a unified solving flow for weighted and unweighted MaxSAT, incrementality, guarantees completeness and exposes native in-memory APIs. The solver combines modern local-search engines, SAT-based optimization via MrsBeaver and Polosat, and a complete LSU-based optimization stage.**

I. Introduction

`Aperture` [1] is an anytime MaxSAT solver designed to combine strong anytime performance with completeness and incrementality. It follows the general anytime MaxSAT paradigm implemented in `TT-Open-WBO-Inc` [2], [3] and its descendant solvers, namely: obtaining an initial model with SAT, improving it using classical local search, refining it via SAT-based optimization (MrsBeaver [4], [5]), and finally switching to a complete Linear SAT–UNSAT (LSU) [6] stage.

Unlike `TT-Open-WBO-Inc`, `Aperture` is written entirely from scratch and provides a unified control flow for weighted and unweighted MaxSAT, native incremental solving, and in-memory APIs in C++ and Python.

II. Algorithmic Components

`Aperture` combines the following algorithmic components:

- Modern local-search solvers, default: `SPB-Band` [7] but also supporting `USW` [8] and `DeepDist` [9].
- The incomplete MrsBeaver SAT-based optimization stage for both weighted and unweighted MaxSAT.
- The Polosat [10] SAT-based local search algorithm as a SAT oracle inside MrsBeaver.
- A complete LSU optimization stage using either totalizer [11], [12], [13] (unary-sum) or adder [14], [15] (binary-sum) encodings for upper bound enforcement.
- The Target-Optimum-Rest-Conservative (TORC) polarity heuristic and Target-Score-Bump (TSB) [5] variable selection heuristic in all SAT stages.

A key difference from `TT-Open-WBO-Inc` lies in the control of the non-incremental stage of MrsBeaver. `Aperture` bounds the number of consecutive non-improving iterations and, once this bound is exceeded, forces a transition to the complete LSU stage. This prevents the solver from spending an impractical amount of time in the approximation phase waiting to reach the global iteration threshold, and ensures timely escape from local optima through a guaranteed handover to complete optimization. Additionally, weighted and unweighted MaxSAT are handled within the same unified control flow.

III. Submitted Configurations

We submitted four configurations of `Aperture` to the anytime tracks, differing in the underlying SAT solver and the SAT solver used for the initial SAT query:

- `Aperture-G-B-G`: ANYTIME-UW & ANYTIME-W
- `Aperture-I-B-I`: ANYTIME-UW.
- `Aperture-K-B-G`: ANYTIME-W & ANYTIME-UW
- `Aperture-K-B-I`: ANYTIME-W.

Where each letter refers to the following configuration, in the respective order:

1) SAT solver used for the initial SAT query.
2) Local Search solver.
3) Main SAT backend.

`K` = `AE-Kissat-MAB` [16], `G` = `Glucose` [17], `I` = `IntelSAT` [18], `B` = `SPB-Band` [7].

We also submitted `Aperture` to the incremental track, in `Aperture-G-B-G` configuration.

IV. Provenance Information

Development of `Aperture` began in late 2025 and is ongoing. (i) Code base origin: The solver was written entirely from scratch in C++. No code was copied or reused from `TT-Open-WBO-Inc` or any other MaxSAT solver. The architecture, data structures, and control flow are original. (ii) Algorithmic inspiration: The high-level solving paradigm is inspired by the anytime MaxSAT approach implemented in `TT-Open-WBO-Inc` and by the algorithms described in prior work on MrsBeaver, Polosat, and LSU. However, all implementations are original and adapted to a unified and incremental architecture. (iii) Human modifications: All algorithmic design and implementation were carried out manually by the authors. (iv) Use of generative AI tools: GitHub Copilot was used as a coding assistant, primarily for code autocompletion during implementation. Additionally, Copilot's agent-based features were used in auxiliary scripts for testing and evaluation.

REFERENCES

[1] A. Nadel, Y. Slonimski, and O. Strichman, "Aperture: an anytime, complete and incremental maxsat solver," 2026, unpublished, submitted for review.

[2] A. Nadel, "Polarity and variable selection heuristics for sat-based anytime maxsat," *J. Satisf. Boolean Model. Comput.*, vol. 12, no. 1, pp. 17–22, 2020. [Online]. Available: https://doi.org/10.3233/sat-200126

[3] ——, "Tt-open-wbo-inc: an efficient anytime maxsat solver," *Journal on Satisfiability, Boolean Modelling and Computation*, vol. 15, no. 1, pp. 1–7, 2024.

[4] ——, "Solving maxsat with bit-vector optimization," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2018, pp. 54–72.

[5] ——, "Anytime weighted maxsat with improved polarity selection and bit-vector optimization," in *2019 Formal methods in computer aided design (FMCAD)*. IEEE, 2019, pp. 193–202.

[6] D. L. Berre and A. Parrain, "The sat4j library, release 2.2," *JSAT*, vol. 7, no. 2-3, pp. 59–6, 2010. [Online]. Available: http://jsat.ewi.tudelft.nl/content/volume7/JSAT7$_4$$_L$$eBerre.pdf$

[7] M. Jin, K. He, J. Zheng, J. Xue, and Z. Chen, "Combining bandmaxsat and fps with spb-maxsat-c," *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, pp. 23–24, 2024.

[8] Y. Chu, C. Li, F. Ye, and S. Cai, "Enhancing maxsat local search via a unified soft clause weighting scheme," in *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, Pune, India, August 21-24, 2024*, ser. LIPIcs, S. Chakraborty and J. R. Jiang, Eds. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 8:1–8:18. [Online]. Available: https://doi.org/10.4230/LIPIcs.SAT.2024.8

[9] M. Jiang, H. Gao, S. Chen, and Y. Chen, "Enhancing local search for maxsat with deep differentiation clause weighting," *arXiv preprint arXiv:2512.05619*, 2025.

[10] A. Nadel, "On optimizing a generic function in sat," in *2020 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2020, pp. 205–213.

[11] O. Bailleux and Y. Boufkhad, "Efficient CNF encoding of boolean cardinality constraints," in *CP 2003*, 2003, pp. 108–122. [Online]. Available: https://doi.org/10.1007/978-3-540-45193-8$_8$

[12] M. Büttner and J. Rintanen, "Satisfiability planning with constraints on the number of actions," in *ICAPS 2005*, 2005, pp. 292–299. [Online]. Available: http://www.aaai.org/Library/ICAPS/2005/icaps05-030.php

[13] S. Joshi, R. Martins, and V. M. Manquinho, "Generalized totalizer encoding for pseudo-boolean constraints," in *CP 2015*, 2015, pp. 200–209. [Online]. Available: https://doi.org/10.1007/978-3-319-23219-5_15

[14] J. P. Warners, "A linear-time transformation of linear inequalities into conjunctive normal form," *Information Processing Letters*, vol. 68, no. 2, pp. 63–69, 1998.

[15] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," *J. Satisf. Boolean Model. Comput.*, vol. 2, no. 1-4, pp. 1–26, 2006. [Online]. Available: https://doi.org/10.3233/sat190014

[16] H. Ding, M. Luo, C. Li, S. Li, R. Chen, C. Xiong, and X. Wu, "AE-Kissat-MAB: Automated evolution of kissat with multi-armed bandits," in *Proceedings of SAT Competition 2025: Solver and Benchmark Descriptions*, C. Codel, K. Fazekas, M. J. H. Heule, and M. Iser, Eds. TU Wien, 2025. [Online]. Available: https://repositum.tuwien.at/handle/20.500.12708/218424

[17] G. Audemard and L. Simon, "On the glucose SAT solver," *Int. J. Artif. Intell. Tools*, vol. 27, no. 1, pp. 1 840 001:1–1 840 001:25, 2018. [Online]. Available: https://doi.org/10.1142/S0218213018400018

[18] A. Nadel, "Introducing intel (r) sat solver," in *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022, pp. 8–1.

BouMS 3.0.1 in the MaxSAT Evaluation 2026

Ole Lübke
Hamburg University of Technology
Institute for Software Systems
Hamburg, Germany
ole.luebke@tuhh.de

***Abstract*—In recent years, some real-time applications for MaxSAT solving have emerged. State-of-the-art MaxSAT solvers are complex programs built for more powerful general purpose computing systems. It is impossible to deploy them in embedded systems or to subject them to safe timing analysis, due to dynamic memory usage and potentially infinite execution paths. *BouMS* (known as noSAT-MaxSAT in previous MaxSAT Evaluations) is a comparatively simple solver based on the NuWLS-2.0 local search algorithm. It is developed according to best practices for embedded real-time systems. Since the previous MaxSAT Evaluation, the application programming interface of BouMS has been extended for better interoperability. It is now closer to that of established solvers.**

I. Introduction

In recent years, some applications for MaxSAT solving in embedded and/or real-time systems have emerged. Examples include plan minimization in online planning [1], routing and scheduling in wireless mesh networks [2], taxi sharing [3], error correction in quantum computers [4], and train dispatching [5]. In practice, such embedded and real-time systems differ significantly from the powerful general purpose computing systems that state-of-the-art MaxSAT solvers are built for. Key restrictions are that often no dynamic memory allocation can be used, and programs need to finish in predictable time. However, state-of-the-art MaxSAT solvers, as well as the SAT solvers they commonly include as subroutines, allocate and free memory highly dynamically, and use programming techniques, such as unbounded (at compile time) loops or recursion, making their timing behavior impossible to analyze statically.

Our goal is to develop an efficient MaxSAT solver suitable for deployment in embedded real-time systems. Consequently, *BouMS* (for Bounded MaxSAT) is developed under a set of constraints commonly found in embedded systems programming, outlined in Section II.

BouMS is the successor of *noSAT-MaxSAT* [6] and based on the NuWLS-2.0 local search algorithm [7], but includes a custom heuristic for escaping local optima called least-frequently flipped (LFF), and a fixed-precision integer arithmetic scheme to replace floating point calculations we call instance-dependent precision (IDP) [6]. Previous experiments showed that LFF slightly improves the performance of BouMS on MaxSAT Evaluation (MSE) benchmarks, and especially helps with solving hard constraints without a dedicated SAT solver. IDP can be slightly detrimental to performance, but allows the solver to run on a wider range of hardware. We elaborate on the algorithm in Section III.

Compared to its predecessor, BouMS features two notable technical updates:

- The application programming interface (API) has been extended for improved interoperability. The extensions are described in Section IV.
- The dependency on `rand` from the C standard library has been removed and replaced by a simple linear congruential random number generator implementation.

For insight into the cost of not using a dedicated SAT solver, we pair BouMS with the *CaDiCaL 2.1.3* SAT solver [8], resulting in the solver *BouMS-CaDiCaL*. It executes CaDiCaL once to obtain a feasible initial assignment for local search. The most recent version of BouMS is published on Zenodo following semantic versioning [9].

Section V provides provenance information on the source code of BouMS.

II. Requirements & Architecture

BouMS is developed under the following requirements commonly found in programming embedded real-time systems: It 1) is programmed in C; 2) is self-contained, i.e., it has no external dependencies (other than the C standard library); 3) does not allocate memory dynamically; 4) does not use floating-point operations; 5) does not contain unbounded loops, i.e., the maximum number of iterations of any loop can be upper-bounded at compile time; 6) does not use recursion.

BouMS is split into a library that fulfills these requirements, and an application that uses the library but is not bound by the above-mentioned restrictions to facilitate, e.g., parsing input instances of arbitrary size in the MSE. Given the number of variables and clauses of a formula, the library can compute (an upper bound on) the amount of memory required for running the algorithm.

III. Algorithm

Algorithm 1 shows how NuWLS-2.0 is implemented in BouMS. In the first iteration, INITVARS (line 6) executes Algorithm 1 (non-recursively) only on the hard constraints of the MaxSAT instance to obtain a feasible initial assignment (instead of using a dedicated SAT solver). Then, decimation [10] is used to improve on that assignment[1], and the

[1]Decimation is disabled in BouMS-CaDiCaL to keep the feasible assignment intact.

Algorithm 1 BouMS

```
1: for try ← 1 ... maxTries do
2:     if No improvement since maxNoImpr flips then
3:         break
4:     end if
5:     if No improvement in the previous try then
6:         INITVARS; DECIMATION; INIT
7:     end if
8:     for flip ← 1 ... maxFlips do
9:         if No improvement since maxNoImpr flips then
10:            break
11:        end if
12:        v ← SELECTVARIABLE
13:        FLIPVARIABLE(v)
14:    end for
15: end for
```

internal data structures of the algorithm are initialized according to the NuWLS-2.0 weighting scheme. After these initializations, local search is executed according to the NuWLS-2.0 weighting scheme and LFF (lines 8 to 14). When local search finds an improved variable assignment, the initialization procedures are skipped in the next try to continue searching in the same direction (line 5). Otherwise, INITVARS sets the current variable assignment to the best assignment found so far.

$maxNoImpr$ flips is part of the extended API of BouMS. It allows terminating the search process early if no improvement has been found for the specified number of variable flips.

IV. API Extensions

Because of the restrictions regarding dynamic memory allocation, before the extension of the API, users had to manually construct the various `struct` types of BouMS to create MaxSAT instances. This made it unnecessarily difficult to integrate BouMS into systems that actually can use dynamic memory. Users that have access to the functions `realloc` and `free` from the C standard library (or equivalent custom functions), can now use the functions `newFormula`, `deleteFormula`, and `addClause`, to conveniently create instances using an interface very similar to interfaces found in established C++ solvers. Since adding many clauses one-by-one through this interface can be costly because it may require reallocations, a mechanism for batch adding is provided as well.

The input file parser in the BouMS application has been rewritten to use this new interface.

V. Code Provenance

BouMS is the successor of noSAT-MaxSAT [6] developed from scratch by the same author. NuWLS-2.0 [7] and decimation [10] have been implemented based on the respective published papers and corresponding source code, but no code has been copied from these sources. No generative artificial intelligence has been used to generate source code for BouMS.

References

[1] Tomáš Balyo, Lukáš Chrpa, and Asma Kilani. "On Different Strategies for Eliminating Redundant Actions from Plans". In: *Proceedings of the International Symposium on Combinatorial Search*. Ed. by Stefan Edelkamp and Roman Barták. Vol. 5. Prague, Czech Republic: AAAI, 2014, pp. 10–18. DOI: 10.1609/socs.v5i1.18318.

[2] Takuya Yoshihiro and Taiki Nishimae. "Practical Fast Scheduling and Routing over Slotted CSMA for Wireless Mesh Networks". In: *2016 IEEE/ACM 24th International Symposium on Quality of Service*. Beijing, China: IEEE, 2016. DOI: 10.1109/IWQoS.2016.7590389.

[3] Aolong Zha, Qiong Chang, and Itsuki Noda. "An Incremental SAT-based Approach for Solving the Real-Time Taxi-Sharing Service Problem". In: *Discrete Applied Mathematics* 335 (2023), pp. 131–145. DOI: 10.1016/j.dam.2022.08.008.

[4] Lucas Berent, Lukas Burgholzer, Peter-Jan H. S. Derks, Jens Eisert, and Robert Wille. "Decoding Quantum Color Codes with MaxSAT". In: *Quantum* 8.1506 (2024). DOI: 10.22331/q-2024-10-23-1506.

[5] Anna Livia Croella, Bjørnar Luteberget, Carlo Mannino, and Paolo Ventura. "A MaxSAT Approach for Solving a New Dynamic Discretization Discovery Model for Train Rescheduling Problems". In: *Computers & Operations Research* 167.106679 (2024). DOI: 10.1016/j.cor.2024.106679.

[6] Ole Lübke. "noSAT-MaxSATv3". In: *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Ed. by Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian. Department of Computer Science, University of Helsinki, 2024, pp. 19–20.

[7] Yi Chu, Shaowei Cai, and Chuan Luo. "NuWLS-c-2023: Solver Description". In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Ed. by Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen. University of Helsinki, Department of Computer Science, 2023, pp. 23–24.

[8] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. "CaDiCaL 2.0". In: Computer Aided Verification. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14681. LNCS. Montreal, Canada: Springer, 2024, pp. 133–152. DOI: 10.1007/978-3-031-65627-9_7.

[9] Ole Lübke. *BouMS*. Version 3.0.1. Zenodo, 2025. DOI: 10.5281/zenodo.17733801.

[10] Shaowei Cai, Chuan Luo, and Haochen Zhang. "From Decimation to Local Search and Back: A New Approach to MaxSAT". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. Ed. by Carles Sierra. Melbourne, Australia: IJCAI, 2017, pp. 571–577. DOI: 10.24963/ijcai.2017/80.

DeepDist-c: Solver Description

Menghua Jiang[1], Haokai Gao[2], and Xiangui Kang[1]
[1] School of Computer Science and Engineering, Sun Yat-sen University, China
[2] School of Computer Science, South China Normal University, China

***Abstract*—This document describes the solver DeepDist-c, which was submitted to both the unweighted and weighted anytime tracks of MaxSAT Evaluation 2026.**

I. Introduction

Our DeepDist-c solver is built upon NuWLS-c-2023 [1] and SPB-MaxSAT-c [2]. Similar to these solvers, DeepDist-c consists of two engines. The first is our proposed stochastic local search (SLS) solver, DeepDist [3], which incorporates a novel deep differentiation clause weighting (Deep-Weighting) scheme and a unit-and-hard-clause decimation (UnH-Decimation) method. Note that only the Deep-Weighting scheme is integrated into the hybrid solver DeepDist-c. The second engine is the SAT-based solver TT-Open-WBO-Inc [4].

II. DeepDist

A. Preliminaries

For each clause c, we use $w(c)$ to denote the dynamic weight of clause c, which is updated during the search process. For each soft clause c, $w_{\text{org}}(c)$ denotes the original weight of c given in the instance. Furthermore, avg_{soft} denotes the average original weight of the soft clauses.

Given a solution α, α is called feasible if it satisfies all hard clauses. The cost of a feasible solution α, denoted by $cost(\alpha)$, is defined as the number of falsified soft clauses for PMS instances and the total original weight of falsified soft clauses for WPMS instances. If α is infeasible, we define $cost(\alpha) = +\infty$. A solution α_1 is considered better than another solution α_2 if $cost(\alpha_1) < cost(\alpha_2)$.

At the beginning of each round of local search, $w_{init}(c)$ denotes the initial weight assigned to clause c.

B. Deep-Weighting

We propose a novel weighting scheme, called Deep-Weighting, which operates as follows:

Initialization of Clause Weights: At the start of each round of local search, Deep-Weighting initializes the weight of each clause as follows:

- For each clause c in **PMS** instances: If the solution found in the previous round of local search is infeasible, set $w_{init}(c) := 1$ for hard clauses and $w_{init}(c) := 0$ for soft clauses. Otherwise, set $w_{init}(c) := 0$ for hard clauses and $w_{init}(c) := 1$ for soft clauses.
- For each clause c in **WPMS** instances: If the solution found in the previous round of local search is infeasible, set $w_{init}(c) := 1$ for hard clauses and $w_{init}(c) := 0$ for soft clauses. Otherwise, set $w_{init}(c) := 1$ for hard clauses and $w_{init}(c) := \frac{w_{org}(c)}{avg_{soft}}$ for soft clauses.

Update of Clause Weights: When the search encounters a local optimum α, the clause weights are updated as follows:

- For each hard clause c in **(W)PMS** instances: If α is infeasible, set $w(c) := w(c) + h_{inc}$.
- For each soft clause c in **PMS** instances: If the first feasible solution has been found and the *cost* of the current solution α isn't smaller than that of the best solution found so far α^* (i.e., $cost(\alpha) \geq cost(\alpha^*)$), set $w(c) := \delta \times (w(c) + \frac{w_{org}(c)}{avg_{soft}})$.
- For each soft clause c in **WPMS** instances: If α is feasible, set $w(c) := \delta \times (w(c) + \frac{w_{org}(c)}{avg_{soft}})$.

Our clause weighting scheme updates the weights of soft clauses in PMS and WPMS instances according to different conditions, and further introduces a novel initialization strategy that sets the initial weights of hard clauses to zero once the first feasible solution is found for PMS instances.

Based on the Deep-Weighting scheme and the UnH-Decimation method, we develop a new SLS solver named DeepDist. More details can be found in [3].

III. The Hybrid Solver: DeepDist-c

Following previous hybrid MaxSAT solvers [1], [2], [5]–[7], we integrate DeepDist with the state-of-the-art SAT-based solver TT-Open-WBO-Inc [4], resulting in the hybrid solver DeepDist-c.

IV. Provenance and AI Usage Statement.

DeepDist-c is built upon the previous hybrid MaxSAT solvers NuWLS-c-2023 [1] and SPB-MaxSAT-c [2], and further integrates our local search solver DeepDist [3] with the SAT-based solver TT-Open-WBO-Inc [4]. The modifications mainly include the integration of the Deep-Weighting scheme into the hybrid framework and support for the MaxSAT Evaluation 2026 requirements.

All modifications to the solver were implemented manually by the authors. No generative AI tools were used to generate or modify the source code of DeepDist-c.

References

[1] Y. Chu, S. Cai, and C. Luo, "Nuwls-c-2023: Solver description," *MaxSAT Evaluation*, vol. 2023, p. 23, 2023.
[2] M. Jin, K. He, J. Zheng, J. Xue, and Z. Chen, "Combining bandmaxsat and fps with spb-maxsat-c," *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, pp. 23–24, 2024.
[3] M. Jiang, H. Gao, S. Chen, and Y. Chen, "Enhancing local search for maxsat with deep differentiation clause weighting," in *ECAI*, 2025.

[4] A. Nadel, "Anytime weighted maxsat with improved polarity selection and bit-vector optimization," in *FMCAD*, 2019, pp. 193–202.
[5] Z. Lei, S. Cai, F. Geng, D. Wang, Y. Peng, D. Wan, Y. Deng, and P. Lu, "Satlike-c: Solver description," *MaxSAT Evaluation*, vol. 2021, p. 19, 2021.
[6] J. Zheng, K. He, Z. Chen, J. Zhou, and C.-M. Li, "Decision tree based hybrid walking strategies," *MaxSAT Evaluation 2022*, p. 24, 2022.
[7] Y. Chu, S. Cai, Z. Lei, and X. He, "Nuwls-c: Solver description," *MaxSAT Evaluation*, vol. 2022, p. 28, 2022.

HWS: Solver Description

Weijie Yao[1], Menghua Jiang[1], Zhaojin Zhong[1], and Yin Chen[1,2]
[1]School of Computer Science, South China Normal University, Guangzhou, China
{2024023471, jiangmenghua, 2024023473}@m.scnu.edu.cn
[2]School of Artificial Intelligence, South China Normal University, Guangzhou, China
ychen@scnu.edu.cn

***Abstract*—This document describes HWS, an incomplete stochastic local search solver submitted to the unweighted anytime track of the MaxSAT Evaluation 2026. HWS is designed for Partial MaxSAT (PMS). Its main idea is to improve local search by thoroughly exploiting historical best states. The solver combines a history-aware restart strategy with a sliding-window controlled clause weighting scheme.**

I. Introduction

HWS is a local search solver for PMS, where all hard clauses must be satisfied and the objective is to minimize the number of unsatisfied soft clauses. Since HWS is incomplete and does not prove optimality, it is submitted to the unweighted anytime track.

The solver maintains a complete assignment and repeatedly flips variables according to scores computed from dynamic clause weights. During search, HWS prints every improved objective value. At normal termination, or when receiving a termination signal, it outputs the best verified assignment found so far.

II. HWS

HWS is based on the local search framework used in modern MaxSAT solvers such as USW-LS [1], SPB-MaxSAT [2], and DeepDist [3]. The solver maintains stacks of falsified hard clauses, falsified soft clauses, and good variables with positive score. If good variables exist, HWS selects a variable by a best-of-multiple-samples rule with timestamp tie breaking. Otherwise, the search is considered to be at a local optimum and clause weights are updated.

A. History-Aware Search

The main feature of HWS is the use of historical best states. When a new global-best feasible solution is found in one search round, HWS records it and uses this historical state to guide the following round. This allows later rounds to continue the exploration around high-quality regions instead of treating every round as independent.

B. Sliding-Window Clause Weighting

HWS also uses a sliding-window controlled clause weighting scheme. Let t be the current step and let t_{best} be the step at which the most recent improved feasible solution was found. Clause weights are updated only when

$$t - t_{\text{best}} \geq \delta.$$

Algorithm 1: SWC

Input: Current step t, last local optimum step t^{best}, window size δ, hard increment h_{inc}, soft increment s_{inc}, current assignment α, global-best assignment $\alpha^{\star}$
Output: Updated clause weights $w(\cdot)$

```
Δ := t − t^best;;
if Δ ≥ δ then
    if α is infeasible then
        for each falsified hard clause c do
            w(c) := w(c) + h_inc;;
    if the first feasible solution has been found and
     cost(α) ≥ cost(α*) then
        for each soft clause c do
            w(c) := w(c) + s_inc;;
return;
```

This delays perturbation while recent improvements are still observed. When the condition is met, falsified hard clauses are increased by h_{inc}. For PMS instances, after a feasible solution has been found, soft clauses can also be increased by s_{inc} when the current feasible solution is not better than the global-best one. In our submitted configuration, $h_{\text{inc}} = 1$, $s_{\text{inc}} = 1$, and $\delta = 5$.

References

[1] Y. Chu, C. M. Li, F. Ye, and S. Cai, "Enhancing MaxSAT local search via a unified soft clause weighting scheme," in *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing*, 2024.

[2] J. Zheng, Z. Chen, C.-M. Li, and K. He, "Rethinking the soft conflict pseudo Boolean constraint on MaxSAT local search solvers," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1989–1997, 2024.

[3] M. Jiang, H. Gao, S. Chen, and Y. Chen, "Enhancing local search for MaxSAT with deep differentiation clause weighting," in *Proceedings of the 28th European Conference on Artificial Intelligence*, 2025.

Loandra in the 2026 MaxSAT Evaluation

Jeremias Berg*, Ole Lübke†
*Department of Computer Science, University of Helsinki, Finland
†Hamburg University of Technology, Institute for Software Systems

***Abstract*—We briefly overview the any-time Loandra MaxSAT solver as it participated in the anytime tracks of the 2026 MSE, focusing especially on the differences between earlier versions. In short the 2026 submission focuses on the techniques for tight integration between SLS and core-boosted search, using CaDiCaL as the backend SAT solver and BouMS as the SLS component. While these techniques most likely will not result in the best possible performance of Loandra, we still felt them to be interesting enough to warrant submission. In short, the submitted version corresponds to the `boums-cadical-integration` branch of the Loandra repository (https://github.com/jezberg/loandra) while the main branch contains our best approximation of the best performing variant of Loandra.**

***Index Terms*—MaxSAT, core-guided search, linear search, local search, preprocessing, pseudo-Boolean constraints, DPW, Open-WBO, RustSAT, BouMS**

I. Introduction and Preliminaries

The base algorithm implemented in Loandra is core-boosted linear search, first proposed for MaxSAT [1] and later extended to PBO, CP and multi-objective optimization [2]–[4].

The 2026 submission of Loandra focuses on the techniques related to tight integration of SLS with core-boosted search as presented in [5]. Specifically, and in contrast to previous years, the backend SAT solver is CaDiCaL 2.0 [6] and BouMS [7] the SLS component. The other external libraries of Loandra remains the same. Loandra is built on top of Open-WBO [8] and uses the RustSAT [9] implementation of the Dynamic Polynomial Watchdog (DPW) [10] for encoding PB constraints.

We assume familiarity with conjunctive normal form (CNF) formulas and weighted partial maximum satisfiability (MaxSAT). Treating a CNF formula as a set of clauses, a MaxSAT instance $\mathcal{I} = (F_H, F_S, w)$ consists of two CNF formulas, the hard clauses F_H and the soft clauses F_S, as well as a weight w associated with each $C \in F_S$. A solution to F is an assignment τ that satisfies F_H. The cost $\text{COST}(\mathcal{I}, \tau)$ of a solution τ is the sum of weights of the soft clauses falsified by τ. A solution is optimal if it has minimum cost. The cost $\text{COST}(\mathcal{I})$ of an instance is the cost of its optimal solutions. An unsatisfiable core κ of $\mathcal{I}$ is a subset of soft clauses such that $F^h \wedge \kappa$ is unsatisfiable.

II. Structure of Loandra

Figure 1 overviews the structure of Loandra. The solver implements core-boosted linear search [1] augmented with tightly integrated MaxSAT preprocessing [11]–[14] and local search [7]. More specifically, Loandra consists of four main components:

(a) Preprocessing,
(b) Core-Guided search (CG),
(c) Solution improving search (SIS), and
(d) Stochastic Local Search (SLS).

All phases of the algorithm can find solution to the instance that provide upper bounds. The best-known solution α_{best} and the corresponding upper bound is maintained and made available to all phases

A. Preprocessing

On input $\mathcal{I} = (F_H, F_S, w)$, the execution starts by invoking the MaxPre 2.0 [12], [14] preprocessor. MaxPre is run with the technique string `[u]#[uvsrgVGc]`, enforcing a 30 s time limit and a skip technique value of 20.

In more detail, the preprocessor runs unit propagation, bounded variable elimination, subsumption elimination, self-subsuming resolution, group subsumed label elimination, and binary core removal, as well as intrinsic at-most-one and TrimMaxSAT techniques [15], [16]. TrimMaxSAT is extended to all literals rather than only literals appearing in soft clauses. The preprocessing step also converts the instance into an *objective-centric* view [5] in which an instance is represented by a set F^P of clauses and a linear objective O to be minimized.

In addition to simplifying the instance, preprocessing can compute an optimal solution to the instance, terminating search. Otherwise, the next core-guided search phase is invoked on the simplified instance (F^P, O). The core-guided phase reuses the assumption variables introduced during preprocessing [11].

When using CaDiCaL as the backend SAT solver, we observed less benefits of preprocessing, thus the evaluation submission of Loandra simply converts the soft clauses into an objective and performs basic simplifications .

B. Core-Guided Search (CG)

The CG phase is unchanged from previous versions of Loandra. Loandra uses a from-scratch reimplementation of PMRES [17] extended with weight-aware core extraction (WCE) [18], stratification, and clause hardening. The phase returns either an optimal solution to the preprocessed instance (from which an optimal solution to the input instance is reconstructed), or the final reformulated instance which is given as input to the next phase.

C. Solution Improving Search (SIS)

The SIS phase of Loandra is an implementation of the SAT/UNSAT linear search algorithm [19] extended with solution-guided phase saving and varying resolution in the style of LinSBPS [20]. The pseudo-Boolean constraint is encoded

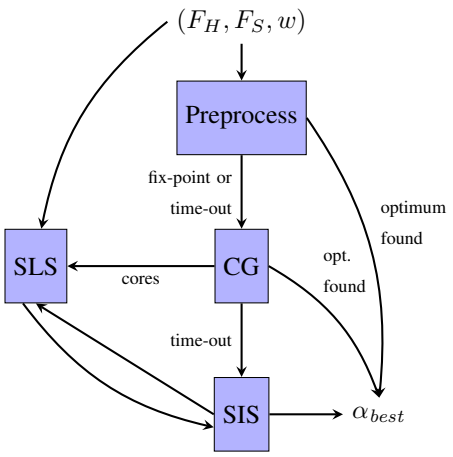


Fig. 1. The structure of Loandra. High-level flow: preprocessing P(F); core-guided search; SIS with PB encoding (DPW) initialized by ub and τ^*; periodic local search; reconstruction upon optimum or resource limits.

with the dynamic polynomial watchdog (DPW) encoding [10], using the implementation from RustSAT [9] via its C API. As detailed in the next section, SIS periodically invokes a local search solver.

SIS runs until either finding an optimal solution or running out of time at which point the best-known solution α_{best} is returned. Notice that the reconstruction of a solution to the preprocessed instance into an solution to the original instance; we use the standard, linear-time reconstruction algorithm as implemented in MaxPre.

D. *Local Search* SLS

The SLS component of Loandra is invoked on the initial instance (before preprocessing, for weighted instances only) as well as after preprocessing on the reformulated instance (where all soft clauses are unit), and the instance obtained after core-guided search. SLS is invoked at beginning of each new resolution (i.e., reinitialization of the SIS search with a larger set of soft clauses) for weighted instances. In general, Loandra also implements introducing unsatisfiable cores into the SLS search, but this is disabled in the evaluation submission. The goal of the local search is twofold. On the one hand, we seek a good upper bound in order to decrease the size of the PB constraint encoded by SIS. On the other hand, as detailed in [13], we know that preprocessing in the context of anytime solving can result in misinterpretation of costs and the computation of solutions whose costs are fairly easy to improve by local changes to the variable assignments.

III. Implementation Details

All algorithms are implemented on top of the publicly available Open-WBO system [8]. To minimize I/O overhead, we make direct use of the preprocessor interface offered by MaxPre.

In the evaluation, we set a 30 s time limit for the preprocessing phase and a 30 s time limit for the core-guided phase. These limits were chosen based on preliminary experiments. The core-guided phase is also terminated on weighted instances when the stratification bound is lowered to 1. On unweighted instances, the phase is terminated at the latest after extracting one set of disjoint cores.

IV. Compilation and Usage

Building and using Loandra resembles building and using Open-WBO. A statically linked version of Loandra in release mode can be built by running `MAKE RS` in the base folder. Note that you need to have an installation of Rust and `cargo` as well. Loandra is tested with Rust version 1.73.0. Additionally, building BouMS requires at least CMake version 3.21.

After building, Loandra can be invoked from the terminal. Except for the formula file, Loandra accepts several command line arguments. The flag `-pmreslin-cglim` sets the maximum time that the core-guided phase can run for (in seconds). The rest of the flags resemble the flags accepted by Open-WBO and MaxPre and BouMS; invoke `./loandra_static -help-verb` for more information.

V. Code Provenance

The code of BouMS is fully developed by the second contributor. The first contributor wrote the PMRES and SIS algorithms from scratch in the Open-WBO architecture. Formula handling and other supporting code (I/O, parameters, etc.) are slightly modified variants of code originally written by the developers of Open-WBO. MaxPRE and the RustSAT implementation of DPW (and other PB encodings) are used without modifications.

Acknowledgments

The algorithmic techniques underlying Loandra have been developed in collaboration with several different people. The solver's initial work was done with Emir Demirović and Peter Stuckey. Other contributors to Loandra include Marcus Leivo, Tuukka Korhonen, Christoph Jabs, Hannes Ihalainen, and Matti Järvisalo. The Academy of Finland supports the primary developer under grant 362987. Our sincerest thanks to everyone who has contributed to the algorithmic ideas underlying Loandra. We also thank the developers of Open-WBO and RustSAT.

References

[1] J. Berg, E. Demirović, and P. J. Stuckey, "Core-boosted linear search for incomplete maxsat," in *CPAIOR*, ser. Lecture Notes in Computer Science, vol. 11494. Springer, 2019, pp. 39–56.

[2] G. Gange, J. Berg, E. Demirović, and P. J. Stuckey, "Core-guided and core-boosted search for CP," in *Proc. CPAIOR*, ser. LNCS, vol. 12296. Springer, 2020, pp. 205–221.

[3] 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 *Proc. AAAI*. AAAI Press, 2021, pp. 3750–3758.

[4] C. Jabs, J. Berg, and M. Järvisalo, "Core boosting in sat-based multi-objective optimization," in *CPAIOR (2)*, ser. Lecture Notes in Computer Science. Springer, 2024, pp. 1–19.

[5] O. Lübke and J. Berg, "Sls-enhanced core-boosted linear search for anytime maximum satisfiability," in *CP*, ser. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 28:1–28:20.

[6] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL 2.0," in *Computer Aided Verification 36th International Conference*, ser. LNCS, vol. 14681. Springer, 2024, pp. 133–152.
[7] O. Lübke. (2025) BouMS. [Online]. Available: https://doi.org/10.5281/zenodo.14191232
[8] R. Martins, V. Manquinho, and I. Lynce, "Open-wbo: A modular maxsat solver," in *Proc. SAT*, ser. Lecture Notes in Computer Science, vol. 8561. Springer, 2014, pp. 438–445.
[9] C. Jabs, "Rustsat: A library for SAT solving in rust," in *SAT*, ser. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 15:1–15:13.
[10] T. Paxian, S. Reimer, and B. Becker, "Dynamic polynomial watchdog encoding for solving weighted maxsat," in *Proc. SAT*, ser. LNCS, vol. 10929. Springer, 2018, pp. 37–53.
[11] J. Berg, P. Saikko, and M. Järvisalo, "Improving the effectiveness of sat-based preprocessing for maxsat," in *Proc. IJCAI*. AAAI Press, 2015, pp. 239–245.
[12] H. Ihalainen, J. Berg, and M. Järvisalo, "Clause redundancy and preprocessing in maximum satisfiability," in *Proc. IJCAR*, ser. LNCS, vol. 13385. Springer, 2022, pp. 75–94.
[13] M. Leivo, J. Berg, and M. Järvisalo, "Preprocessing in incomplete maxsat solving," in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, vol. 325. IOS Press, 2020, pp. 347–354.
[14] T. Korhonen, J. Berg, P. Saikko, and M. Järvisalo, "Maxpre: An extended maxsat preprocessor," in *SAT*, ser. Lecture Notes in Computer Science, vol. 10491. Springer, 2017, pp. 449–456.
[15] A. Ignatiev, A. Morgado, and J. Marques-Silva, "Rc2: an efficient maxsat solver," *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 53–64, 2019.
[16] T. Paxian, P. Raiola, and B. Becker, "On preprocessing for weighted maxsat," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 12597. Springer, 2021, pp. 556–577.
[17] N. Narodytska and F. Bacchus, "Maximum satisfiability using core-guided maxsat resolution," in *Proc. AAAI*. AAAI Press, 2014, pp. 2717–2723.
[18] J. Berg and M. Järvisalo, "Weight-aware core extraction in sat-based maxsat solving," in *Proc. CP*, ser. Lecture Notes in Computer Science, 2017.
[19] D. L. Berre and A. Parrain, "The sat4j library, release 2.2," *J. Satisf. Boolean Model. Comput.*, vol. 7, no. 2-3, pp. 59–6, 2010. [Online]. Available: https://satassociation.org/jsat/index.php/jsat/article/view/82
[20] E. Demirović and P. J. Stuckey, "Techniques inspired by local search for incomplete maxsat and the linear algorithm: Varying resolution and solution-guided search," in *Proc. CP*, ser. LNCS, vol. 11802. Springer, 2019, pp. 177–194.

Metaheuristic Solver PRINTEMPS for MaxSAT Evaluation 2026

Masahiro Sakai
Email: masahiro.sakai@gmail.com

Yuji Koguma
Email: yuji.koguma@gmail.com

***Abstract*—PRINTEMPS is a metaheuristic solver originally developed for general integer linear programming (ILP) problems. For MaxSAT Evaluation 2026, it has been extended to handle MaxSAT problems by translating each WCNF clause into a 0–1 inequality and translating soft clauses into a weighted objective penalty.**

I. Introduction

PRINTEMPS (PoRtable INTEger Mathematical Programming Solver) is a metaheuristic solver developed by the second author for general integer linear programming (ILP) problems [1]. For the Pseudo-Boolean Competition 2025 [2], it was extended to handle Pseudo-Boolean (PB) problems, including those with nonlinear terms and soft constraints [3]. The MaxSAT extension submitted here reuses the PB encoding machinery for soft constraints and adds a WCNF front-end together with an anytime-style output protocol. This document describes the supported tracks, the computational overview of PRINTEMPS, and the MaxSAT-specific extension.

II. Supported Tracks

Since PRINTEMPS is based on heuristics, it can prove neither the optimality of a solution nor the unsatisfiability of an instance in general. Therefore, we submitted to the **Main Tracks for Anytime Solvers**, namely **ANYTIME-W** (weighted) and **ANYTIME-UW** (unweighted). The same binary handles both tracks, since the only difference between them is the weight distribution of the soft clauses in the input WCNF file [4].

The solver consumes the new WCNF format (post-2022): hard clauses are prefixed with `h`, soft clauses begin with an integer weight, and each clause is terminated with `0`. Upon termination—either at the wall-clock limit or after receiving `SIGTERM` from the judging harness—the solver outputs the best assignment found so far as an `o`/`v` pair followed by a single status line `s SATISFIABLE` (exit code 10) or `s UNKNOWN` (exit code 0). The wrapper also reports `s UNSATISFIABLE` (exit code 20) when bound propagation during preprocess detects that the hard clauses are infeasible.

III. Computational Overview of PRINTEMPS

An ILP problem can be formulated as follows:

$$(\mathrm{ILP}): \underset{\boldsymbol{x}\in\mathbb{Z}^N,\, \boldsymbol{l}\le\boldsymbol{x}\le\boldsymbol{u}}{\text{minimize}} \ \boldsymbol{c}^\top\boldsymbol{x} \quad \text{subject to} \ \ \boldsymbol{A}_1^\top\boldsymbol{x}=\boldsymbol{b}_1,\ \ \boldsymbol{A}_2^\top\boldsymbol{x}\le\boldsymbol{b}_2,$$

where N is the number of variables, $\boldsymbol{c}\in\mathbb{R}^N$ is the cost vector, and $\boldsymbol{l},\boldsymbol{u}\in\mathbb{Z}^N$ $(\boldsymbol{l}\le\boldsymbol{u})$ are the lower and upper bounds of $\boldsymbol{x}$, respectively. The matrices $\boldsymbol{A}_1,\boldsymbol{A}_2$ and the vectors $\boldsymbol{b}_1,\boldsymbol{b}_2$ have appropriate dimensions consistent with $\boldsymbol{x}$.

PRINTEMPS searches for solutions to an instance of (ILP) using Weighted Tabu Search that minimizes an objective function penalized by constraint violations [5]. It also incorporates performance-enhancing techniques tailored for (ILP). They include instance size reduction, dependent variable extraction, neighborhood filtering, and incremental penalty evaluation [6].

IV. Extension for MaxSAT Problems

A. Encoding of clauses

Let $x_i\in\{0,1\}$ denote the propositional variable indexed by i, and let a clause be a disjunction of literals $C=(\ell_1\vee\ell_2\vee\cdots\vee\ell_k)$. We split the literals into positive ones $P(C)=\{i\mid x_i\in C\}$ and negative ones $N(C)=\{i\mid\neg x_i\in C\}$. The clause is satisfied if and only if at least one positive literal evaluates to 1 or at least one negated variable evaluates to 0, which is equivalent to

$$\sum_{i\in P(C)} x_i-\sum_{j\in N(C)} x_j\ \ge\ 1-|N(C)|.$$

Each hard clause is added to the model directly as such a linear constraint.

B. Soft clauses

A soft clause with weight $w\ge 1$ is handled as a special case of the soft inequality treatment introduced for PB problems [3]. Since a clause is a one-sided $\ge$ constraint, only a single lower-side slack variable $y^-\in\{0,1\}$ is required:

$$\sum_{i\in P(C)} x_i-\sum_{j\in N(C)} x_j+y^-\ \ge\ 1-|N(C)|,$$

together with the penalty $w\cdot y^-$ added to the objective function. Because the unrelaxed left-hand side falls below the right-hand side by exactly one when the clause is falsified, y^- acts as the indicator of clause falsification, and the total objective penalty equals the sum of weights of unsatisfied soft clauses.

Soft-clause weights are read from the WCNF file as 64-bit unsigned integers and converted to double-precision floating-point coefficients in the objective penalty. Weight sums up to 2^{53} are represented exactly; beyond that, rounding may occur in the double-typed objective. To avoid mis-reporting, the value printed on the `o`-line is therefore recomputed in

64-bit unsigned integer arithmetic from the original WCNF weights and the y^- indicators of the incumbent assignment.

C. Anytime behavior

The MaxSAT Evaluation anytime tracks score solvers based on the quality of every reported intermediate solution [4]. The wrapper subscribes to the per-iteration callback of PRINTEMPS' Tabu Search loop and emits a fresh `o`/`v` pair whenever the current feasible incumbent strictly improves over the previously reported one. The tracks also mandate single-threaded execution; PRINTEMPS is therefore configured to use a single OpenMP thread.

V. Provenance

The submitted system extends PRINTEMPS [1], an open-source C++ metaheuristic ILP solver originally authored by the second author. PRINTEMPS is distributed as a header-only library and has no proprietary runtime dependencies.

The PRINTEMPS core (the ILP modeler/solver library and its Weighted Tabu Search algorithm) and the Pseudo-Boolean Competition 2025 extension were both developed without generative AI assistance. The structural layout of the competition wrapper and the soft-constraint encoding pattern are inherited from our Pseudo-Boolean Competition 2025 submission [3].

The components newly added for this submission, with approximate line counts, are:

- the WCNF parser module `printemps/wcnf/` (∼330 lines);
- the `import_wcnf` method on the IP model in `printemps/model/model.h` (∼135 lines);
- the competition wrapper under `extra/maxsat_evaluation/` (∼340 lines);
- unit tests for the parser and an end-to-end test for the wrapper (∼420 lines).

This new code, together with the corresponding build-system updates and the present description, was developed with the assistance of Claude Code [7] (Anthropic, invoked with the Claude Opus 4.7 model) used in a conversational pair-programming style. Every generated change was reviewed by the first author before being committed; the first author also directed the design, including the WCNF-to-IP encoding scheme, and the error-handling protocol for the anytime tracks. The competition's regression test suite [4] was used during development to validate parser correctness and the `s UNSATISFIABLE` exit path triggered by bound-propagation infeasibility.

The submitted Linux binary is built statically inside the project's Docker image (Ubuntu 22.04) and contains no proprietary or closed-source components.

VI. Conclusion

The metaheuristic ILP solver PRINTEMPS has been extended to handle MaxSAT problems by reusing the PB soft-constraint encoding and adding a WCNF front-end and an anytime output protocol.

References

[1] PRINTEMPS C++ metaheuristics modeler/solver for general integer optimization problems, https://snowberryfield.github.io/printemps/.

[2] Pseudo-Boolean Competition 2025, https://www.cril.univ-artois.fr/PB25/.

[3] Y. Koguma and M. Sakai, "An Incomplete Approach to Pseudo-Boolean Problems with Metaheuristic Solver PRINTEMPS," Solver description, Pseudo-Boolean Competition 2025, https://www.cril.univ-artois.fr/PB25/descr/PRINTEMPS-Pseudo_Boolean_Competition_2025.pdf.

[4] MaxSAT Evaluation 2026, https://maxsat-evaluations.github.io/2026/.

[5] K. Nonobe and T. Ibaraki, "A Tabu Search Approach to the Constraint Satisfaction Problem as a General Problem Solver," *Eur. J. Oper. Res.*, Vol. 106, pp. 599–623, 1998, doi:10.1016/S0377-2217(97)00294-4.

[6] Y. Koguma, "Tabu Search-Based Heuristic Solver for General Integer Linear Programming Problems," *IEEE Access*, Vol. 12, pp. 19059–19076, 2024, doi: 10.1109/ACCESS.2024.3361323.

[7] Anthropic, "Claude Code," an AI-assisted coding agent, https://www.anthropic.com/claude-code.

NuWLS-c-2026: Solver Description

1st Yi Chu
Key Laboratory of System Software,
Institute of Software, Chinese Academy of Sciences
Beijing, China
chuyi2020@iscas.ac.cn

2nd Yingyi Jiang
School of Software, Beihang University
Beijing, China
jiangyy25@buaa.edu.cn

3rd Yiwen Sun
School of Data Science, Fudan University
Shanghai, China
ywsun22@m.fudan.edu.cn

4th Peng Lin
Key Laboratory of System Software,
Institute of Software, Chinese Academy of Sciences
Beijing, China
linpeng@ios.ac.cn

5th Chuan Luo
School of Software, Beihang University
Beijing, China
chuanluo@buaa.edu.cn

6th Shaowei Cai†
School of Software, Beihang University
Key Laboratory of System Software,
Institute of Software, Chinese Academy of Sciences
Beijing, China
shaoweicai.cs@gmail.com

***Abstract*—This document describes NuWLS-c-2026, a hybrid MaxSAT solver submitted to all anytime tracks of the MaxSAT Evaluation 2026. NuWLS-c-2026 is built upon the NuWLS-c solver line, which combines a stochastic local search component with a SAT-based MaxSAT solver.**

I. Introduction

NuWLS-c-2026 is an improved version of NuWLS-c-2023 [1]. The solver contains two complementary components: a stochastic local search solver and an SAT-based MaxSAT solver. The local search component is effective in quickly finding high-quality assignments, while the SAT-based component provides strong robustness between different types of MaxSAT instances.

Allocating an appropriate amount of computational budget to the local search component is a challenging task. To address this issue, we employ a lightweight machine learning technique to learn an offline prediction model based on general structural features of MaxSAT instances.

Given a MaxSAT instance, the solver extracts a set of general formula-level features and uses the learned model to guide the allocation of computational budget to the local search component. The prediction is only used to adjust the amount of computational budget assigned to the local search component, while the underlying solving algorithms remain unchanged.

II. Base Solver

NuWLS-c-2026 follows the hybrid framework of the SATLike-c [2] and NuWLS-c solver [1], [3] lines.

The local search component is based on the Unified Soft Clause Weighting (Unified-SW) framework proposed in our previous work [3], [4], which provides a unified weighting mechanism for (weighted) partial MaxSAT. In particular, NuWLS-2.0 uses a global coefficient to dynamically adjust soft-clause weights while preserving the relative importance of original soft weights in weighted instances. This design allows the local search procedure to better balance feasibility and objective optimization during the search process.

The second component is the SAT-based MaxSAT solver TT-Open-WBO-Inc [5], [6]. The two components are combined within a hybrid anytime framework.

III. Instance-Aware Local Search Budget Allocation

The main change in NuWLS-c-2026 is an instance-aware mechanism for allocating the computational budget of the local search component.

Before solving an instance, the solver extracts a set of general formula-level features and uses a lightweight regression model to estimate the expected usefulness of additional local search effort. The predicted score, denoted by p_{LS}.

According to p_{LS}, NuWLS-c-2026 uses three scheduling modes:

- **Conservative mode**: disable the local search component and directly use the SAT-based component.
- **Default mode**: use the default NuWLS-c-2023 local search budget.
- **Enhanced mode**: allocate additional computational budget to the local search component by increasing the local search time limit and the number of restart attempts. The corresponding scaling factors and restart parameters are statically configured in the solver source code before the evaluation. The same scaling factors and restart parame-

†Corresponding author.

ters are used across weighted and unweighted tracks, as well as across different timeout settings.

This mechanism is intended to reduce ineffective local search effort and to spend more time on instances where local search is expected to contribute useful bounds or high-quality solutions.

All thresholds and budget factors are fixed before the evaluation. No benchmark-name information, manually designed family identifiers, or narrowly targeted triggers are used.

IV. Training Methodology

The prediction model was trained using benchmark instances from the anytime (incomplete) tracks of MaxSAT Evaluations 2018–2022.

For each instance, NuWLS-c-2026 extracts a set of features before solving. These features include statistics such as the number of variables, the number of clauses, the number of hard clauses, the number of soft clauses, clause-to-variable ratios, clause length statistics, weight-related statistics, and other structural properties of the input formula. The feature design is partially inspired by feature extraction methodologies in SATZilla [7]. All extracted features are implemented explicitly in the solver source code.

For each training instance, we measured the usefulness of allocating additional computational budget to the local search component within the hybrid solving framework. Based on these measurements, a LightGBM regression model was trained offline.

The trained model is embedded into the solver as a static artifact. During the evaluation, the solver only performs feature extraction and lightweight model inference before solving.

V. Conclusion

NuWLS-c-2026 keeps the main solving framework of NuWLS-c-2023 and introduces a general instance-aware scheduling mechanism for local search. The goal is to improve robustness by allocating local search effort according to general structural properties of the input instance.

References

[1] Yi Chu, Shaowei Cai, and Chuan Luo. “Nuwls-c-2023: Solver description.” MaxSAT Evaluation 2023 (2023): 23.

[2] Zhendong Lei, Shaowei Cai, SATLike-c: Solver description. MSE’18

[3] Yi Chu, Shaowei Cai, Zhendong Lei, and Xiang He. “Nuwls-c: Solver description.” MaxSAT Evaluation 2022 (2022): 28.

[4] Yi Chu, Chu-Min Li, Furong Ye, Shaowei Cai: Enhancing MaxSAT Local Search via a Unified Soft Clause Weighting Scheme. SAT 2024: 8:1-8:18

[5] Alexander Nadel.“Anytime weighted maxsat with improved polarity selection and bit-vector optimization” InClark W. Barrett and Jin Yang, editors, 2019 Formal Methodsin Computer Aided Design, FMCAD 2019,San Jose, CA, USA,October 22-25,2019,pages 193-202. IEEE, 2019

[6] Alexander Nadel. ”TT-Open-WBO-Inc-21: an Anytime MaxSAT Solver Entering MSE’21.” MaxSAT Evaluation 2021

[7] Hadar Shavit, Holger H. Hoos: Revisiting SATZilla Features in 2024. SAT 2024: 27:1-27:26

NuWLS-c-2026-UB: Solver Description

1st Yingyi Jiang
School of Software, Beihang University
Beijing, China
jiangyy25@buaa.edu.cn

2nd Yi Chu
Key Laboratory of System Software,
Institute of Software, Chinese Academy of Sciences
Beijing, China
chuyi2020@iscas.ac.cn

3rd Yiwen Sun
School of Data Science, Fudan University
Shanghai, China
ywsun22@m.fudan.edu.cn

4th Chuan Luo
School of Software, Beihang University
Beijing, China
chuanluo@buaa.edu.cn

5th Shaowei Cai†
School of Software, Beihang University
Key Laboratory of System Software,
Institute of Software, Chinese Academy of Sciences
Beijing, China
shaoweicai.cs@gmail.com

***Abstract*—This document describes NuWLS-c-2026-UB, a hybrid MaxSAT solver submitted to all anytime tracks of the MaxSAT Evaluation 2026. NuWLS-c-2026-UB is built upon the NuWLS-c solver line, which combines a stochastic local search component with a SAT-based MaxSAT solver.**

I. Introduction

NuWLS-c-2026-UB is an improved version of NuWLS-c-2023 [1]. The solver contains two complementary components: a stochastic local search solver and an SAT-based MaxSAT solver. The local search component is effective in quickly finding high-quality assignments, while the SAT-based component provides strong robustness between different types of MaxSAT instances.

Compared with its predecessor, NuWLS-c-2023, the current version incorporates a lightweight unit propagation routine for formula simplification and enforces a more conservative conflict budget within the Linear Search SAT-UNSAT with Weight Clustering framework. These enhancements aim to optimize computational resource allocation, specifically preventing the backend SAT subsolvers from stalling during the clustered approximation phase. Meanwhile, the default stochastic local search and SAT-based optimization procedures inherited from NuWLS-c-2023 remain fully intact.

II. Base Solver

NuWLS-c-2026-UB follows the hybrid framework of the SATLike-c [2] and NuWLS-c solver [1], [3] lines.

The local search component is based on the Unified Soft Clause Weighting (Unified-SW) framework proposed in our previous work [3], [4], which provides a unified weighting mechanism for (weighted) partial MaxSAT. In particular, NuWLS-2.0 uses a global coefficient to dynamically adjust soft-clause weights while preserving the relative importance of original soft weights in weighted instances. This design allows the local search procedure to better balance feasibility and objective optimization during the search process.

The second component is the SAT-based MaxSAT solver TT-Open-WBO-Inc [5], [6]. The two components are combined within a hybrid anytime framework.

III. Unit Preprocessing

Before solving the instance, NuWLS-c-2026-UB employs a lightweight preprocessing routine based on standard unit propagation. After an initial literal deduplication step, the preprocessor checks for the presence of unit hard clauses. If any are detected, it triggers an iterative propagation process: forced assignments derived from unit clauses are repeatedly propagated to simplify the entire formula. This procedure continuously removes satisfied clauses and strips falsified literals from the remaining ones, cascading until a logical fixed point is reached or a contradiction is detected.

IV. SAT Conflict Budgeting in Weighted LinearSU Clustering

Within the LinearSU clustering framework for weighted MaxSAT instances, the incremental solving engine iteratively alternates between evaluating SAT queries under assumption sets and updating the current objective bound through the *core-guided exact refinement* procedure upon identification of satisfiable assignments.

To control computational overhead during the *approximation layer*, we enforce a bounded conflict policy on each SAT solver invocation. This mechanism restricts excessive exploration in subproblems of the approximation layer while preserving solution quality.

†Corresponding author.

During the *core-guided exact refinement*, the conflict bound is fully disabled, ensuring that bound updates are precise and that the search converges to an exact solution, in accordance with the default behavior of NuWLS-c-2023.

V. Conclusion

NuWLS-c-2026-UB retains the core hybrid framework of NuWLS-c-2023 while introducing unit propagation and conflict budgeting in LinearSU clustering to improve efficiency and robustness on weighted MaxSAT instances.

References

[1] Yi Chu, Shaowei Cai, and Chuan Luo. "Nuwls-c-2023: Solver description." MaxSAT Evaluation 2023 (2023): 23.

[2] Zhendong Lei, Shaowei Cai, SATLike-c: Solver description. MSE'18

[3] Yi Chu, Shaowei Cai, Zhendong Lei, and Xiang He. "Nuwls-c: Solver description." MaxSAT Evaluation 2022 (2022): 28.

[4] Yi Chu, Chu-Min Li, Furong Ye, Shaowei Cai: Enhancing MaxSAT Local Search via a Unified Soft Clause Weighting Scheme. SAT 2024: 8:1-8:18

[5] Alexander Nadel."Anytime weighted maxsat with improved polarity selection and bit-vector optimization" InClark W. Barrett and Jin Yang, editors, 2019 Formal Methodsin Computer Aided Design, FMCAD 2019,San Jose, CA, USA,October 22-25,2019,pages 193-202. IEEE, 2019

[6] Alexander Nadel. "TT-Open-WBO-Inc-21: an Anytime MaxSAT Solver Entering MSE'21." MaxSAT Evaluation 2021

PASMaxSAT-c: an Anytime MaxSAT Solver Entering MSE'26

Mingming Jin*, Yujie Cai*, Jiongzhi Zheng*, Kun He*
*School of Computer Science and Technology, Huazhong University of Science and Technology
{mingmingk, yujie_cai, jzzheng, brooklet60}@hust.edu.cn,

I. Solver Description

PASMaxSAT-c is an anytime MaxSAT solver developed by modifying tt-open-wbo-inc [1], a solver submitted to the MaxSAT Evaluation 2024. The solver builds on the original tt-open-wbo-inc framework and introduces a new PASMaxSAT component.

The initial design of PASMaxSAT consisted of two main ideas: gradual subproblem expansion and conflict repair based on SPB-MaxSAT [2]. In this design, only a subset of clauses is activated at first, and the active subproblem is gradually expanded during the search. The SPB-MaxSAT component is then used to repair conflicts and improve the incumbent solution on the active subproblem.

However, tt-open-wbo-inc already integrates several solving strategies. In our experiments, the gradual expansion of subproblems did not consistently improve the solver on multiple types of instances. Instead, the expansion process sometimes introduced additional overhead and increased the total solving time. Therefore, in the submitted version of PASMaxSAT-c, all clauses are activated directly, and SPB-MaxSAT is applied to the full MaxSAT instance.

The main difference between PASMaxSAT-c and the SPB-MaxSAT-c solver submitted in 2024 is the use of restarts inside SPB-MaxSAT. Within the time budget assigned to SPB-MaxSAT, PASMaxSAT-c allows multiple restart rounds. The initial number of restart rounds is set to 3. At the end of each round, if the best solution is improved during that round, the allowed number of restart rounds is increased by one. The number of restart rounds is restricted to the range from 3 to 10.

The initial solution of each restart round is selected according to the result of the previous round. If a feasible solution was found in the previous round, the best solution from that round is used as the initial solution for the next round. Otherwise, when no feasible solution is available, the initial solution is generated randomly. This mechanism allows the solver to exploit high-quality incumbent solutions while still benefiting from diversification through restarts.

To adapt the solver to longer anytime time limits, PASMaxSAT-c also relaxes the previous restriction on the number of invocations of this local-search component. In previous MSE settings, this type of local search was used only once. In PASMaxSAT-c, the solver is allowed to enter PASMaxSAT again when the elapsed time since the previous PASMaxSAT invocation exceeds 200 seconds and the current best solution is better than 95% of the best solution at the end of the previous local-search phase. This condition is designed to invoke PASMaxSAT again only when sufficient progress has been made and enough time remains for further improvement.

II. Implementation and Provenance

PASMaxSAT-c is implemented on top of tt-open-wbo-inc, which participated in the MaxSAT Evaluation 2024. The original framework and the integrated solving components of tt-open-wbo-inc are retained. The newly added code is mainly located in:algorithm/pas/. Additional minor modifications were made in:MaxSAT.cc and Alg_LinearSU_Clustering.cc. The new code implements the PASMaxSAT module, the direct activation of all clauses for SPB-MaxSAT, the restart mechanism of SPB-MaxSAT, and the condition for re-entering PASMaxSAT under longer anytime time limits.

All newly added code was written manually. No generative artificial intelligence tools were used to generate the new code.

References

[1] J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian, "Maxsat evaluation 2024: Solver and benchmark descriptions," 2024.

[2] J. Zheng, Z. Chen, C.-M. Li, and K. He, "Rethinking the soft conflict pseudo boolean constraint on maxsat local search solvers," in *Thirty-Third International Joint Conference on Artificial Intelligence {IJCAI-24}*. International Joint Conferences on Artificial Intelligence Organization, 2023, pp. 1989–1997.

SPB-NIM-c: Solver Description

Zhaojin Zhong[1] and Yin Chen[2]
[1]School of Computer Science, South China Normal University, Guangzhou, China
[2]School of Artificial Intelligence, South China Normal University, Guangzhou, China
{zhongzhaojin}@m.scnu.edu.cn, ychen@scnu.edu.cn

***Abstract*—This document describes the solver SPB-NIM-c, submitted to the weighted anytime tracks of MaxSAT Evaluation 2026.**

I. Introduction

Recently, several efficient local search algorithms for MaxSAT have been proposed, such as USW-LS [1], SPB-MaxSAT[2]. Although these algorithms differ in their specific weight-updating strategies, they share the idea of unified weighting. In this framework, all soft clauses are treated as a single global constraint, and their weights are updated in a unified manner, regardless of whether they are currently satisfied. This weighting mechanism preserves the priority relationships among soft clauses during the search, thereby guiding the algorithm toward satisfying clauses with larger weights first.

However, under this mechanism, soft clauses with large original weights tend to remain dominant throughout the search, while clauses with smaller weights may be overlooked for a long time. Nevertheless, these low-weight clauses may contain useful information for escaping the current search region and discovering new promising directions. Motivated by this observation, we propose an improved weighting mechanism that introduces controlled perturbations into the unified weighting process while largely preserving the original priority structure among soft clauses. This mechanism enhances search diversification and guides the algorithm toward potentially better solution regions.

The anytime solver SPB-MaxSAT-c and its variants [3], combine SPB-MaxSAT [2] with the well-known SAT-based solver TT-Open-WBO-Inc [4], and have achieved excellent performance in previous MaxSAT Evaluations. SPB-NIM-c is developed on top of SPB-MaxSAT-c-Band by integrating the proposed new weighting mechanism into its clause weighting framework.

II. SPB-NIM-MaxSAT

The proposed weighting mechanism serves as a complement and refinement to the original SPB-Weighting scheme. It is not activated at every search round; instead, it is triggered only when the previous round of local search successfully finds a better solution. The main idea is to introduce a moderate weight adjustment on top of SPB-Weighting, while preserving its original advantages, so as to further guide the search toward potentially promising regions. The detailed weighting strategy is described as follows.

- For **WPMS** instances:
  - If a better solution was found in the previous round of local search, the initial weight of each hard clause is set to 1, and the initial weight of each soft clause is
$$w_{init}(c) = \frac{avg_{soft}}{w_{org}(c)},$$
where $w_{org}(c)$ is the original weight of clause c, avg_{soft} is the average original weight of all soft clauses.

This initialization assigns higher initial search priority to soft clauses with smaller original weights, encouraging the algorithm to explore regions that may be overlooked by the original weighting mechanism at the early stage of search. To avoid deviating from the original weighted objective for a prolonged period, once the search reaches a local optimum for the first time, clause weights are restored to an objective-guided form: $w(c) = 2 * \frac{w_{org}(c)}{avg_{soft}}$. This form of the weights is exactly the same as the one used without the initialization perturbation. Afterwards, the algorithm continues to update the clause weights based on these restored weights.

References

[1] Y. Chu, C.-M. Li, F. Ye, and S. Cai, "Enhancing MaxSAT local search via a unified soft clause weighting scheme," SAT 2024: 8:1–8:19.
[2] J. Zheng, J. Chen, Z. Li, and C.-M. Li, "Rethinking the soft conflict pseudo-Boolean constraint on MaxSAT local search solvers," IJCAI 2024: 1989–1997.
[3] J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian, "MaxSAT Evaluation 2024: Solver and benchmark descriptions," Department of Computer Science, University of Helsinki, 2024.
[4] N. Alexander, "Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization," FMCAD 2019: 193–202.

SGAT-NuWLS-c for MaxSAT Evaluation 2026

Sota Moriyama[1,2] and Katsumi Inoue[2]
[1]The Graduate University for Advanced Studies, SOKENDAI
[2]National Institute of Informatics
{sotam, inoue}@nii.ac.jp

***Abstract*—SGAT-NuWLS-c is a MaxSAT solver submitted to the Anytime Track of the 2026 MaxSAT Evaluation. It combines NuWLS-c-2023 with SAT-based Graph Attention Networks (SGAT), using SGAT to construct initial assignments for the NuWLS local-search phase.**

I. Introduction

SGAT-NuWLS-c is a MaxSAT solver that extends NuWLS-c-2023 [1] using initialization heuristics based on machine learning techniques. NuWLS-c-2023 was the best-performing anytime solver in the 2023 MaxSAT Evaluation, and combines the local search algorithm NuWLS [2] with a SAT-based solver TT-Open-WBO-Inc [3]. In this solver, NuWLS uses a solution given by a SAT solver as an initial assignment and performs local search to find better solutions. SGAT-NuWLS-c replaces this SAT-based initialization with a Graph Neural Network (GNN)-based initialization, where the GNN is trained to predict assignments that maximize the weighted clause satisfaction objective. Specifically, we use SAT-based Graph Attention Networks (SGAT) [4], a recent GNN architecture shown to perform well on MaxSAT competition benchmarks.

II. SGAT for NuWLS Initialization

A. Preliminaries

For non-partial MaxSAT instances, the solver first converts the input instance into a variable–clause graph $G = (V_X \cup V_C, E)$, where V_X contains one node for each variable and V_C contains one node for each clause. An edge $(x_i, C_j) \in E$ is added whenever variable x_i appears in clause C_j. Let N_j^+ and N_j^- denote the sets of variables that occur positively and negatively in C_j, respectively, and let w_j^{norm} be the normalized weight of C_j computed as $w_j / \max_{C_k \in V_C} w_k$. Following the SGAT representation, the edge feature for (x_i, C_j) is

$$e_{ij} = \begin{cases} \left(\frac{1}{|N_j^+|}, \frac{1}{|N_j^-|}, 0, 0\right) w_j^{\mathrm{norm}}, & x_i \in N_j^+, \\ \left(0, 0, \frac{1}{|N_j^+|}, \frac{1}{|N_j^-|}\right) w_j^{\mathrm{norm}}, & x_i \in N_j^-. \end{cases}$$

When a denominator is zero, the corresponding entry is set to zero. This feature encodes polarity, clause length information, and clause weight in a single edge vector.

For partial MaxSAT instances, we encode soft and hard clauses using the same graph format. If C_j is a soft clause with weight w_j, we set $w_j^{\mathrm{norm}} = w_j / W_{\mathrm{top}}$, where W_{top} is the top weight used in the input instance, i.e., the total weight of all soft clauses plus one. If C_j is a hard clause, we set $w_j^{\mathrm{norm}} = 1$.

Given G with initial variable valuations $v(x_i) = \mathrm{rand}(0,1)$ for all $x_i \in V_X$, SGAT maintains continuous variable valuations $v(x_i) \in [0,1]$ and computes continuous clause valuations $v(C_j) \in [0,1]$ using a t-norm layer. For a clause $C_j = \ell_{j1} \vee \cdots \vee \ell_{jk}$, define

$$\tilde{v}(\ell) = \begin{cases} 1 - v(x), & \ell = x, \\ v(x), & \ell = \neg x. \end{cases}$$

Then the clause valuation is computed as

$$v(C_j) = 1 - \min(\tilde{v}(\ell_{j1}), \ldots, \tilde{v}(\ell_{jk})),$$

which corresponds to the Gödel t-conorm for disjunction. Intuitively, $v(C_j)$ represents the satisfaction degree of C_j, so lower values indicate clauses that are closer to being unsatisfied.

The SGAT layer then sends update messages from clauses to variables. Intuitively, clauses with low $v(C_j)$ (low satisfaction) send stronger requests to their neighboring variables, and the attention mechanism assigns coefficients α_{ij} to decide which requests should be prioritized. Each variable valuation is updated by aggregating the corresponding update messages m_{ij}^U:

$$v'(x_i) = \sum_{C_j \in N(x_i)} \alpha_{ij} m_{ij}^U.$$

Repeating this t-norm and attention procedure produces one prediction score $p_i \in [0,1]$ for each variable x_i. When a Boolean value is needed, we interpret p_i as the probability that x_i should be assigned true; that is, we set $x_i \mapsto 1$ with probability p_i and $x_i \mapsto 0$ with probability $1 - p_i$.

The original SGAT implementation is written in Python. For SGAT-NuWLS-c, to integrate SGAT inference into the C++ codebase of NuWLS-c-2023, we reimplemented the SGAT inference procedure in C++ and load the trained model weights from a file at runtime. The resulting SGAT inference code is self-contained and does not depend on external libraries. The SGAT model is pretrained on instances from the 2018–2020 MaxSAT Evaluations, excluding partial instances.

B. Initialization with SGAT

Using predictions from a pretrained SGAT model, we construct an initial assignment for NuWLS using an algorithm built on the Unit Propagation (UP)-based decimation algorithm used in prior works [5].

We use the following functions in Algorithm 1. $\mathrm{HardUnit}(A)$ and $\mathrm{SoftUnit}(A)$ return whether a hard

Algorithm 1 SGAT-Guided Initialization

Require: MaxSAT instance F, SGAT scores p
Ensure: Complete assignment α

1: $\alpha \leftarrow \emptyset$, $A \leftarrow F$
2: **while** α is incomplete **do**
3: **if** $\mathrm{HardUnit}(A)$ exists **then**
4: $(x, b) \leftarrow \underline{\mathrm{HardUnitAssign}(A, p)}$
5: **else if** $\mathrm{SoftUnit}(A)$ exists **then**
6: $(x, b) \leftarrow \underline{\mathrm{SoftUnitAssign}(A, p)}$
7: **else**
8: Select a random unassigned variable x
9: $u \leftarrow \mathrm{rand}(0, 1)$
10: $b \leftarrow \underline{\mathbb{I}[p_x > u]}$
11: **end if**
12: $\alpha \leftarrow \alpha \cup \{x \mapsto b\}$
13: $A \leftarrow \mathrm{Simplify}(A, x, b)$
14: **end while**
15:
16: **return** α

or soft unit clause exists in the simplified formula A, respectively. The procedures $\mathrm{HardUnitAssign}(A, p)$ and $\mathrm{SoftUnitAssign}(A, p)$ are identical to the greedy UP-based Decimation proposed in prior work [5]. The only difference is that, when a choice is not uniquely determined, SGAT scores are used to resolve conflicts or break ties in the same way as in Lines 9–10. $\mathrm{Simplify}(A, x, b)$ removes clauses satisfied by $x \mapsto b$ and removes falsified literals from the remaining clauses, creating new unit clauses.

In the pseudocode, underlined operations indicate the points where the assignments provided by SGAT may be used (primarily to resolve conflicts or break ties, and in the final fallback assignment step).

C. Integration into NuWLS-c-2023

SGAT-NuWLS-c uses the SGAT-guided UP-based decimation algorithm to produce an initial solution only for the NuWLS local-search phase. After parsing the input MaxSAT instance, the solver builds the NuWLS local-search instance and runs SGAT inference on the corresponding graph representation. The prediction scores are converted into a complete assignment α by Algorithm 1, and NuWLS starts local search from α.

For partial MaxSAT instances, if this NuWLS run finds a feasible solution (a solution that satisfies all hard clauses), TT-Open-WBO-Inc then continues its standard search procedure. If no feasible solution is found, the solver falls back to the SAT solver-based initialization before continuing with the standard TT-Open-WBO-Inc procedure.

Additionally, SGAT is given a predefined time limit (default 5 seconds) for inference. If SGAT does not finish within this limit, the solver also falls back to the SAT solver-based initialization. This keeps the solver from spending too much time on SGAT inference for large instances, which can result in suboptimal performance or even no solution at all within the overall time limit. Furthermore, the SGAT procedure itself is skipped when the problem size is too large (default of over 5 million literal occurrences), since SGAT inference consistently exceeds the time limit in such cases.

III. Provenance Information

This solver is based on the NuWLS-c-2023 solver, which was the best-performing anytime solver in the 2023 MaxSAT Evaluation, with approximately 2,200 lines of code modifications in total. We note that we used generative AI coding assistants (Codex and GitHub Copilot) during development, primarily to help translate the SGAT inference implementation from Python into C++ and to refactor surrounding code for integration into the NuWLS-c-2023 codebase. All such AI-suggested changes were reviewed and edited by the authors before inclusion.

Acknowledgements

This work has been supported by JSPS KAKENHI Grant Number JP25K03190, JST CREST Grant Number JPMJCR22D3 and Grant-in-Aid for JSPS Fellows Grant Number JP26KJ1236, Japan.

References

[1] Yi Chu and Shaowei Cai, "NuWLS-c-2023: Solver description," MSE'23.
[2] Yi Chu, Shaowei Cai, and Chuan Luo, "NuWLS: Improving Local Search for (Weighted) Partial MaxSAT by New Weighting Techniques," in *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*, Brian Williams, Yiling Chen, and Jennifer Neville, Eds. AAAI Press, 2023, pp. 3915–3923.
[3] Alexander Nadel, "Anytime Weighted MaxSAT with Improved Polarity Selection and Bit-Vector Optimization," in *2019 Formal Methods in Computer Aided Design, FMCAD 2019*, Clark W. Barrett and Jin Yang, Eds. IEEE, 2019, pp. 193–202.
[4] Sota Moriyama and Katsumi Inoue, "Graph-based attention for differentiable maxSAT solving," in *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026.
[5] Shaowei Cai and Zhendong Lei, "Old techniques in new ways: Clause weighting, unit propagation and hybridization for maximum satisfiability," *Artif. Intell.*, vol. 287, p. 103354, 2020.

BENCHMARKS

MaxSAT Evaluation 2026: Solver and Benchmark Descriptions, volume B-2026-1 of Department of Computer Science Series of Publications B, University of Helsinki 2026.

A MaxSAT Benchmark for Multiple Optimization Domains based on OP-to-MaxSAT Reduction

1st Yuxin Zhao
School of Software Engineering
South China University of Technology
Guangzhou, China
yuxinzhaozyx@163.com

2nd Han Huang
School of Software Engineering
Sun Yat-sen University
Zhuhai, China
huangh985@mail.sysu.edu.cn

3rd Zhifeng Hao
Department of Mathematics, College of Science
Shantou University
Shantou, China
haozhifeng@stu.edu.cn

***Abstract*—Optimization problems are ubiquitous across a wide range of domains, including logistics, manufacturing, telecommunications, and scientific computing, and they exhibit a great variety of forms. Despite this diversity, a large number of optimization problems can be reduced to Maximum Satisfiability (MaxSAT), a unified formalism for optimization in Boolean space. In existing research, reductions from optimization problems to MaxSAT have predominantly relied on manual, problem-specific encodings, which require substantial domain expertise and are difficult to generalize. In our recent work, we proposed the OP-to-MaxSAT reduction, a unified and automated method for transforming various optimization problems into MaxSAT instances, thereby enabling MaxSAT solvers to be applied to a broad spectrum of optimization problems. To promote the application of MaxSAT solvers across multiple classical optimization domains, in this paper, we present a benchmark for MaxSAT solvers based on the OP-to-MaxSAT reduction. The benchmark contains 136 instances across 11 diverse optimization problem types, including routing problems (Traveling Salesman Problem, Capacitated Vehicle Routing Problem, Two-Echelon Vehicle Routing Problem), scheduling problems (Job-Shop, Open-Shop), combinatorial optimization (Multidimensional Knapsack, Graph Coloring, Quadratic Assignment), and continuous optimization (Shifted Sphere, Schwefel's, Rosenbrock's functions). All instances are derived from well-established benchmark libraries such as TSPLIB, CVRPLIB, OR-Library, COLOR, QAPLIB, and IEEE CEC 2005. The benchmark and generation scripts are publicly available. This benchmark enables standardized evaluation of MaxSAT solvers on real-world optimization problems.**

***Index Terms*—Optimization Problem, MaxSAT, Reduction**

I. Introduction

Optimization problems are ubiquitous across a wide range of scientific and industrial domains, including logistics, manufacturing, telecommunications, finance, and scientific computing. These problems exhibit a great variety of forms, ranging from discrete combinatorial structures such as routing and scheduling to continuous landscapes encountered in engineering design. Despite this diversity, they share a common mathematical essence: finding the best solution from a vast space of feasible candidates under given constraints. Despite decades of research, solving such problems efficiently remains a central challenge, as many of them are NP-hard and resist exact solution at scale.

Maximum Satisfiability (MaxSAT) is a fundamental optimization problem that generalizes Boolean satisfiability (SAT) by seeking assignments that satisfy the maximum number of clauses. Over the past two decades, MaxSAT has evolved into a practically powerful formalism, and modern solvers can now tackle instances with millions of variables and clauses, finding applications in hardware verification, software debugging, and bioinformatics. A key insight is that a remarkably broad class of optimization problems can be reduced to MaxSAT, providing a unified formalism for optimization in Boolean space. However, existing reductions have predominantly relied on manual, problem-specific encodings, which require substantial domain expertise and are difficult to generalize. This fragmentation limits the broader adoption of MaxSAT solvers as a general optimization tool.

In our recent work [1], we addressed this limitation by proposing the OP-to-MaxSAT reduction, a unified and automated method for transforming various optimization problems into MaxSAT instances within a single framework. Existing MaxSAT benchmarks are predominantly drawn from traditional application domains and do not systematically cover the breadth of classical optimization problem types now accessible through the OP-to-MaxSAT reduction. The OP-to-MaxSAT reduction opens the door to leveraging the full power of modern MaxSAT solvers across a much wider spectrum of optimization domains.

To extend the application range of MaxSAT solvers to real-world optimization problems, in this paper, we propose a comprehensive MaxSAT benchmark for evaluating MaxSAT solvers on classical optimization problems via the OP-to-MaxSAT reduction. Our benchmark contains 136 instances across 11 problem types, all derived using the OP-to-MaxSAT reduction from well-established benchmark libraries, ensuring that the benchmark is built upon widely recognized and reproducible problem settings. Both the generated MaxSAT instances and the scripts for reproducing them are publicly available, facilitating standardized evaluation and future extensions. We hope that this benchmark will contribute to advancing MaxSAT solvers toward more general optimization capabilities.

II. OP-to-MaxSAT Reduction

The OP-to-MaxSAT reduction transforms optimization problems into MaxSAT instances. Given an optimization prob-

TABLE I
OVERVIEW OF THE PROPOSED MAXSAT BENCHMARK.

Problem Type	Data Source	#Instance
Traveling Salesman Problem (TSP)	TSPLIB	11
Capacitated Vehicle Routing (CVRP)	CVRPLIB	8
Two-Echelon VRP (2E-VRP)	OR-Library	20
Multidimensional Knapsack (MDKP)	OR-Library	20
Graph Coloring (GCP)	COLOR	13
Job-Shop Scheduling (JSSP)	Taillard	10
Open-Shop Scheduling (OSSP)	Taillard	20
Quadratic Assignment (QAP)	QAPLIB	8
Shifted Sphere Function	CEC 2005	10
Shifted Schwefel's Problem	CEC 2005	6
Shifted Rosenbrock's Function	CEC 2005	10
Total		**136**

lem formulated in the mathematical modeling language, the reduction encodes the problem constraints as hard clauses and the objective function as some weighted soft clauses and hard clauses. The resulting MaxSAT instance encodes the original optimization problem such that finding a satisfying assignment with maximum weight corresponds to finding the optimal solution of the original problem.

The reduction works by first manually modeling the optimization problem with the constraints and the objective. For each constraint, the reduction generates a CNF formula that enforces the constraint. The objective function is then encoded as a collection of weighted soft clauses, where each soft clause corresponds to a bit of the objective. By assigning appropriate weights, the MaxSAT solver is guided to prefer solutions with better objective values.

More details on the OP-to-MaxSAT reduction can be found in [1], which presents GORED, a general optimization solver based on this reduction approach.

III. BENCHMARK DESCRIPTION

The benchmark contains 136 instances derived from 11 optimization problem types using the OP-to-MaxSAT reduction. Table I provides an overview of the benchmarks.

The benchmarks covers four major categories of optimization problems:

Routing Problems. This category includes the Traveling Salesman Problem (TSP), Capacitated Vehicle Routing Problem (CVRP), and Two-Echelon Vehicle Routing Problem (2E-VRP). TSP instances are sourced from TSPLIB [2], a standard library for TSP and related routing problems. CVRP instances come from CVRPLIB [3], and 2E-VRP instances are from OR-Library [4]. These problems are fundamental in logistics and transportation.

Scheduling Problems. We include Job-Shop Scheduling Problem (JSSP) and Open-Shop Scheduling Problem (OSSP) instances from Taillard's benchmark [5]. JSSP and OSSP are classic scheduling problems with extensive applications in manufacturing and production planning.

Combinatorial Optimization. This category covers the Multidimensional Knapsack Problem (MDKP), Graph Coloring Problem (GCP), and Quadratic Assignment Problem (QAP). MDKP instances are from OR-Library [4], GCP instances from the COLOR library [6], and QAP instances from QAPLIB [7]. These problems represent different aspects of combinatorial optimization.

Continuous Optimization. We include three benchmark functions from IEEE CEC 2005 [8]: Shifted Sphere, Shifted Schwefel, and Shifted Rosenbrock functions. These are standard test functions for evolutionary algorithms and continuous optimization.

All instances are encoded in the standard WCNF format, following the standard MaxSAT Evaluation format. The benchmark can be used to evaluate both exact and anytime MaxSAT solvers on the weighted tracks.

IV. CONCLUSION

We presented a comprehensive MaxSAT benchmark containing 136 instances across 11 diverse optimization problem types. The benchmark is derived from established benchmark libraries using the OP-to-MaxSAT reduction. This benchmark enables standardized evaluation of MaxSAT solvers on real-world optimization problems and facilitates fair comparison across different solver approaches. The generation scripts are publicly available at https://github.com/YuxinZhaozyx/OP-to-MaxSAT-reduction-benchmark to support future research in MaxSAT solvers and optimization.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (92570116, 62276103), Innovation Team Project of General Colleges and Universities in Guangdong Province (2023KCXTD002), the Research and Development Project on Key Technologies for Intelligent Sensing and Analysis of Urban Events Based on Low-Altitude Drones (2024BQ010011), Guangdong Basic and Applied Basic Research Foundation (2023B1515120020) and National Key R&D Program of China (2025YFC3410000).

REFERENCES

[1] Y. Zhao, H. Huang, and Z. Hao, "A general optimization solver based on OP-to-MaxSAT reduction," *arXiv preprint arXiv:2604.1961*, 2026. [Online]. Available: https://arxiv.org/abs/2604.1961

[2] G. Reinelt, "TSPLIB - a traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.

[3] E. Uchoa, D. Pecin, A. Pessoa, M. Poggi, T. Vidal, and A. Subramanian, "New benchmark instances for the capacitated vehicle routing problem," *European Journal of Operational Research*, vol. 257, no. 3, pp. 845–858, 2017.

[4] J. E. Beasley, "Or-library: distributing test problems by electronic mail," *Journal of the Operational Research Society*, vol. 41, no. 11, pp. 1069–1072, 1990.

[5] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[6] M. Trick, "The COLOR library," *Online Resource*, 2024. [Online]. Available: https://mat.tepper.cmu.edu/COLOR/instances.html

[7] R. Burkard, S. Karisch, and F. Rendl, "QAPLIB-a quadratic assignment problem library," *European Journal of Operational Research*, vol. 55, no. 1, pp. 115–119, 1991.

[8] P. Suganthan, N. Hansen, J. Liang, K. Deb, Y.-p. Chen, A. Auger, and S. Tiwari, "Problem definitions and evaluation criteria for the cec 2005 special session on real-parameter optimization," *Natural Computing*, vol. 341-357, 2005.

Answer Set Optimization by MaxSAT: Benchmarks for MaxSAT Evaluation 2026

Masood Feyzbakhsh Rankooh and Matti Järvisalo
HIIT, Department of Computer Science, University of Helsinki, Finland
{masood.feyzbakhshrankooh,matti.jarvisalo}@helsinki.fi

***Abstract*—We describe a benchmark set for the weighted track of the MaxSAT Evaluation 2026 based on answer set programming (ASP) optimization. The instances are obtained from non-tight ASP optimization benchmarks and translated into weighted MaxSAT instances using a SAT-based ASP solving framework based on generalized transitive closure encodings of acyclicity. The underlying approach captures stable model semantics by enforcing acyclicity of the dependency graph induced by selected dependencies. The benchmark set consists of 50 WCNF instances from two ASP Competition benchmark domains: TRAVELING SALESMAN PROBLEM and CONNECTED MAXIMUM-DENSITY STILL LIFE.**

I. INTRODUCTION

We describe a benchmark set submitted to the weighted track of the MaxSAT Evaluation 2026 based on translating ASP optimization benchmarks into weighted MaxSAT. The benchmark set was generated using the MaxSAT-based ASP solving framework introduced in [1] and consists of 50 WCNF instances generated from non-tight ASP optimization benchmarks. We refer the reader to [1] for more details and here provide only a high-level description.

The underlying approach translates ASP programs into propositional formulas together with acyclicity constraints over the positive dependency graph. For optimization problems, the resulting formulas are translated into weighted MaxSAT instances, allowing MaxSAT solvers to search for optimal stable models. Since the benchmarks are non-tight, enforcing acyclicity is essential for capturing the stable model semantics.

The benchmark set is intended to provide structured optimization instances originating from ASP encodings for evaluating modern MaxSAT solvers. The instances are drawn from ASP Competition benchmark domains previously used for evaluating ASP optimization systems.

II. ASP OPTIMIZATION AS MAXSAT

The submitted benchmark instances are generated using the translation framework introduced in [1] for solving ASP decision and optimization problems via SAT and MaxSAT. The framework targets non-tight ASP programs, for which Clark's completion alone is not sufficient to capture stable models. In such programs, positive dependencies between atoms may form cycles, and these cycles have to be ruled out in order to enforce the minimality condition of stable model semantics.

The translation first instruments the ASP program so that relevant positive dependencies are represented explicitly. The program is then normalized and translated into a propositional formula using completion. The remaining semantic condition is acyclicity of the dependency graph induced by the selected dependencies. For optimization problems, weak constraints and optimization statements are represented in the objective of the resulting weighted MaxSAT instance.

Acyclicity is enforced using the generalized transitive closure framework of [1]. Among the concrete instantiations introduced in the framework, the submitted benchmark instances employ the cycle-elimination-based encoding derived from the cycle elimination method introduced in [2]. The framework captures several alternative acyclicity encodings, including full transitive closure, alternative transitive closure, ordered transitive closure, feedback-vertex-set-based encodings, vertex elimination, and cycle elimination.

The resulting weighted MaxSAT instances encode the search for optimal stable models of the original ASP programs. Hard clauses encode the completion and acyclicity constraints needed for stable model semantics, while weighted soft clauses encode the optimization objective. The instances are provided in the standard WCNF format used in the MaxSAT Evaluation.

III. BENCHMARK DESCRIPTION

The benchmark set consists of 50 weighted MaxSAT instances generated from ASP optimization benchmarks. The submitted instances are derived from two benchmark domains used in the ASP Competitions: TRAVELING SALESMAN PROBLEM (TSP), used in the 2017 ASP Competition, and CONNECTED MAXIMUM-DENSITY STILL LIFE (Connect), used in the 2015 ASP Competition.

For the TRAVELING SALESMAN PROBLEM domain, all 30 available instances were used. For the CONNECTED MAXIMUM-DENSITY STILL LIFE domain, instances `0101`–`0120` were used, yielding 20 instances. Together, these form a benchmark set of 50 weighted MaxSAT instances.

The generated benchmark files use the naming convention `domainName-instanceNumber.wcnf`. Example benchmark names include `TSP-0001.wcnf`, `TSP-0030.wcnf`, `Connect-0101.wcnf`, and `Connect-0120.wcnf`.

The benchmark instances were generated using the toolchain described in [1]. The ASP programs were first processed using the ASPTOOLS library [3], which is available at [1]. The resulting SAT modulo acyclicity instances were

[1]https://github.com/asptools/software

Benchmark Description: Network Modularity Optimization

Carlos Ansótegui*, Vaidyanathan Peruvemba Ramaswamy†, Stefan Szeider†, Hai Xia†
*Logic and Optimization Group, University of Lleida, Lleida, Spain
carlos.ansotegui@udl.cat
†Algorithms and Complexity Group, TU Wien, Vienna, Austria
{vaidyanathan,sz,hxia}@ac.tuwien.ac.at

***Abstract*—We present a benchmark set for exact network modularity optimization via weighted partial MaxSAT. The set contains 155 sparse MaxSAT encodings derived from undirected graphs used in our recent study on exact and verifiable modularity computation. The instances cover both synthetic and real-world networks and range from 10 to 1461 vertices and from 13 to 2812 edges. For each graph, we provide the original graph file, a weighted partial MaxSAT encoding, and an auxiliary decoding file. This description summarizes the optimization problem, the benchmark composition, the sparse encoding used in the submission, and the file organization of the archive.**

I. Problem Setting

Given an undirected graph $G = (V, E)$ with $m = |E|$, the goal is to partition the vertices into communities so as to maximize the Newman–Girvan modularity [1]

$$Q(\mathcal{P}) = \sum_{C \in \mathcal{P}} \left(\frac{|E(C)|}{m} - \left(\frac{\sum_{v \in C} d(v)}{2m} \right)^2 \right), \qquad (1)$$

where $\mathcal{P}$ is a partition of V, $E(C)$ is the set of edges with both endpoints in C, and $d(v)$ is the degree of vertex v. Exact modularity maximization is NP-hard [2], [3], which makes it a natural target for MaxSAT-based optimization.

II. Benchmark Set

The submitted benchmark family consists of 155 instances, where the data set combines two groups of graphs considered in our paper [5]. First, it contains 103 benchmark graphs used in previous work on exact modularity computation [4], including 50 synthetic LFR/ABCD networks and 53 real-world networks. Second, it adds 52 further real-world graphs. The resulting family spans a wider size range than the earlier benchmark collections.

Across the 155 graphs, the number of vertices ranges from 10 to 1461 and the number of edges from 13 to 2812. The instances are named after the underlying graph, for example `ambassador`, `dom`, `baseball`, and `netscience`. The benchmark set includes small social networks, biological interaction graphs, geographical and economic networks, edit networks from Wikibooks, and synthetic community-structured graphs. This heterogeneity is useful for MaxSAT evaluation because it produces encodings with rather different variable interactions, clause densities, and objective-weight distributions. Table I summarizes the main properties of the submitted archive.

TABLE I
Summary of the submitted benchmark archive

Total instances	155
Vertex range	10–1461
Edge range	13–2812
Instances with unique optimum	137
Instances with multiple optima	18

Each benchmark instance is represented by three files sharing the same base name:

- `<name>.gr`: the original graph in DIMACS graph format;
- `<name>.wcnf`: the weighted partial MaxSAT encoding in the revised MSE22/MSE26 WCNF format;
- `<name>.wcnf.aux`: an auxiliary decoding map used to reconstruct communities from solver models.

The file names are fixed graph names rather than parameter strings, reflecting that the submission is a curated finite benchmark set rather than a generator-based family. The additional real-world graphs were obtained from the official Netzschleuder repository[1]. Detailed per-instance statistics, including numbers of optimal clusterings and optimal modularity values, are included in the Figshare[2] supplementary material in `results.csv` and `results.pdf`. In particular, the archive records that 137 instances have a unique optimal clustering and 18 have multiple optimal clusterings, with 11 of those exhibiting essentially different optimal solutions. These metadata are not needed by MaxSAT solvers, but they provide additional context on the structural diversity of the benchmark family.

III. Instance Generation and Archive Contents

The benchmark files contain the sparse encodings actually used in the experiments [5]. The supplementary file consists of the concrete `.wcnf` instances together with the original graph files and the encoder implementation used to reproduce them. The supplementary material also contains the Python

Corresponding author: Hai Xia.

[1]https://networks.skewed.de
[2]https://figshare.com/s/caec9f3c34f5c31488c2?file=52609217

script `encoding.py`, helper utilities, a file of separator sets used by the sparse encoding, and result summaries describing the observed empirical solving behavior.

The `.wcnf` files are weighted partial MaxSAT instances with integer weights only. The first comment line identifies the original graph and reports the number of vertices and edges. Further comment lines provide additional encoding statistics such as the number of variables and the number of hard and soft clauses. The companion `.wcnf.aux` file stores the mapping between MaxSAT variables and graph vertex pairs. This mapping is used by our decoder to reconstruct the clustering represented by a satisfying assignment and to recompute the corresponding modularity value on the original graph.

IV. MaxSAT Encoding

Our full encoding is based on a pairwise representation of community membership. For each unordered pair $\{u, v\} \subseteq V$ with $u \neq v$, we introduce a Boolean variable $c_{u,v}$, intended to mean that u and v belong to the same community. In the full encoding, feasible assignments encode equivalence relations: reflexivity and symmetry are implicit in the representation, while transitivity is enforced by hard clauses.

For any distinct vertices u, v, w, the full encoding uses ternary transitivity clauses of the form

$$\neg c_{u,v} \vee \neg c_{v,w} \vee c_{u,w}. \tag{2}$$

Using all such clauses leads to a cubic number of hard clauses. The submitted instances use the sparse encoding from our paper [5]. This encoding keeps only selected transitivity clauses. For each unordered pair $\{u, w\}$, we compute a separator set $K_G(u, w)$, derived from a minimum vertex separator in the input graph, and introduce only clauses mediated by vertices $k \in K_G(u, w)$, namely clauses of the form

$$\neg c_{u,k} \vee \neg c_{k,w} \vee c_{u,w}. \tag{3}$$

Its purpose is to preserve the optimal solutions of the modularity objective while reducing the number of hard clauses substantially.

The soft clauses encode the modularity objective. Define the gain of a pair u, v by

$$g(u, v) = \begin{cases} 2m - d(u)d(v) & \text{if } uv \in E, \\ -d(u)d(v) & \text{otherwise.} \end{cases} \tag{4}$$

If $g(u, v) > 0$, we add the soft unit clause $c_{u,v}$ with weight $g(u, v)$; if $g(u, v) < 0$, we add the soft unit clause $\neg c_{u,v}$ with weight $|g(u, v)|$. Pairs with zero gain are omitted. Maximizing the total weight of satisfied soft clauses is equivalent, up to a fixed additive and multiplicative transformation, to maximizing the modularity value of the represented clustering [5]. All submitted weights are integers.

An important practical point is that the sparse encoding preserves optimal solutions while often greatly decreasing the number of hard clauses compared to the full cubic transitivity encoding. This was the main reason for choosing it for submission. In our experiments, sparse encodings led to substantially faster exact solving on many instances, especially for larger graphs. For simplicity, we chose to submit only the sparse encodings for all 155 graphs rather than larger full encodings that would increase archive size and reduce solver throughput without changing the underlying optimization task. But `encoding.py` could also generate the full encodings.

V. Remarks on Difficulty and Use

The benchmark family was curated to include a spread of difficulty. In our experiments, larger and denser graphs tend to be harder, although structure matters substantially as well. Some real-world networks with moderate size still yield challenging MaxSAT instances due to their less regular separator structure and more ambiguous community organization, while some synthetic instances are solved comparatively quickly because their planted communities induce clearer objective preferences.

The provided archive also contains the encoder implementation (`encoding.py`) so that the submited sparse encodings can be reproduced and decoded. The `.wcnf.aux` files are not needed by MaxSAT solvers themselves, but they are useful for translating satisfying assignments back into community partitions and modularity values. We expect the benchmark family to be relevant primarily for complete weighted MaxSAT solving, but the instances may also be useful for evaluating approximate methods when the recovery of high-quality clusterings is of interest.

References

[1] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, 2004.

[2] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[3] T. N. Dinh and M. T. Thai, "Toward optimal community detection: From trees to general weighted networks," *Internet Mathematics*, vol. 11, no. 3, pp. 181–200, 2015.

[4] S. Aref, M. Mostajabdaveh, and H. Chheda, "Heuristic modularity maximization algorithms for community detection rarely return an optimal partition or anything similar," in *Computational Science – ICCS 2023, Part IV*, LNCS 14076, pp. 612–626, 2023.

[5] C. Ansótegui, V. P. Ramaswamy, S. Szeider, and H. Xia, "Uncovering and verifying optimal community structure in complex networks: A MaxSAT approach," in *Computational Science – ICCS 2025*, LNCS 15904, pp. 35–49, 2025.

CPMpy dataset submissions to MSE26

Tias Guns
Department of Computer Science
KU Leuven
Leuven, Belgium
tias.guns@kuleuven.be

Thomas Sergeys
Department of Computer Science
KU Leuven
Leuven, Belgium
thomas.sergeys@kuleuven.be

Ignace Bleukx
Department of Computer Science
KU Leuven
Leuven, Belgium
ignace.bleukx@kuleuven.be

Hendrik Bierlee
Department of Computer Science
KU Leuven
Leuven, Belgium
henk.bierlee@kuleuven.be

***Abstract*—CPMpy [5] is an open-source Python library for modeling combinatorial (optimisation) problems. It provides a high-level modeling interface and connects to solver backends from various CO paradigms. Solvers don't all support the same types of decision variables and constraints. Through a series of equivalence-preserving constraint reformulations, CPMpy can transform a constraint model into the target format, enabling systematic comparisons of solvers across paradigms. Thanks to a collection of file format readers and loaders, well-known CO benchmarking and application datasets can be brought into CPMpy and solved with one of its backends. Combined with our recent work on format writers, CPMpy can serve as a translation tool, converting datasets to different formats and making them accessible to other communities [11]. For the MSE26 competition we translated CP models for the JSPLib and PSPLib RCPSP datasets to *.wcnf* files.**

License : CPMpy is licensed under the Apache-2.0 license
Code : CPMpy's code is available on GitHub: https://github.com/CPMpy/cpmpy
Acknowledgement : This research received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant No. 101002802, CHAT-Opt) and Proof of Concept (POC) grant (Grant No.101158376, PO-CPMpy).

I. Format loaders

In the recent work of [11], we introduced a collection of file format readers and loaders to CPMpy. These tools can convert well-known benchmarking instances into CPMpy models, giving access to our constraint reformulation stack; an extensive set of rewrite operations such as constraint decomposition, linearization, and Boolean encodings, that allow transforming between different constraint formalisms.

Application datasets like JSPLib and PSPLib only contain problem data. Since they do not explicitly define any constraints, there are many design choices to make regarding the model. The provided modelling contains one set of choices, and can be exchanged for alternatives.

II. Constraint reformulations

Solvers from different CO paradigms support different types of decision variables and constraints. Constraint Programming (CP) has the most expressive language we consider, supporting global constraints [12], potentially non-linear, over both integer and Boolean decision variables. Next are Integer Linear Programming (ILP) solvers supporting only linear inequality constraints, Pseudo-Boolean (PB) solvers supporting linear inequality constraints over Boolean variables, and (Max)SAT-solvers which support Boolean formulas in Conjunctive Normal Form (CNF).

Solvers from the more expressive paradigms can take less expressive languages as input, but the reverse direction requires rewriting a set of high-level expressions into a set of lower level expressions, often requiring the use of auxiliary variables. This is achieved through a series of equivalence-preserving constraint reformulations, transforming a constraint model into the target format. We identify five main reformulation stages: decomposition, flattening, linearization, encoding integer variables to Booleans, and encoding pseudo-Boolean constraints into CNF. Each of the transformations is implemented in CPMpy as a separate function.

III. File writing

To convert any CPMpy model back to file, we provide a collection of file format writers. The currently supported formats are MPS, OPB and DIMACS; an overview is given in Table I. Through the SCIP Python interface we additionally support LP, CIP, GMS, PIP, and MiniZinc files. The file writers make use of the same set of transformations to translate any generic CPMpy model into expressions matching the required output language.

During reformulation of the input model, some variables are encoded and auxiliary variables are introduced. For example, integer decision variables are encoded to Boolean literals when the target format is DIMACS or OPB. To allow for checking validity of the solution found by the solver, we augment the output files with the *semantics* of an auxiliary variable. More specifically, we adopt the VeriPB-style annotations to indicate a literal is part of the encoding of an integer variable. Such annotations allow to reconstruct the value of the original

```
p cnf
c 1 x_eq_0
```

```
p cnf
c 2 x_ge_1
```

```
p cnf
c 3 x_bit0
```

Listing 1: Different annotations used in DIMACS comments to indicate the meaning of encoding literals for x: direct encoding (left), order encoding (center) and binary encoding (right).

TABLE I: Initial set of supported datasets and I/O capabilities

Format	Dataset	Read	Write
jsplib	JSPLib [7]	✓	
psplib	PSPLib RCPSP [8]	✓	
nurserostering	Nurse rostering (1–24) [10]	✓	
XCSP3	XCSP3 competition [1]	✓	
OPB	PB Competition [9]	✓	✓
MSE (wcnf)	MaxSAT Evaluation [2]	✓	✓
MPS	MIPLib [4]	✓	✓
DIMACS (cnf)	SAT competition [6]	✓	✓

integer variables after solving, and check the solution with the specification of the high-level model.

IV. Datasets

Our recent work also added dataset classes to CPMpy, with a PyTorch compatible access pattern, for a collection of well-known CO benchmarks [11]. Such a dataset object can be used to download, transform, and iterate over problem instances. Combined with CPMpy's constraint reformulations, expressions can be translated to the Python API of many different solvers spanning CP, MIP, SMT, pseudo-Boolean and SAT technologies, enabling cross-paradigm benchmarking. When combined with the format writers, we can translate datasets to other file formats from potentially different CO solving paradigms.

We submitted instances from the following translated application datasets to MSE26:

- `JSPLib` [7] Benchmark of job-shop scheduling problems
- `PSPLib RCPSP` [8] Benchmark of Resource-Constrained Project Scheduling Problems

A. JSPLib

JSPLib [7] is a collection of benchmark instances for the classical Job Shop Scheduling Problem (JSSP), in which a set of jobs must be processed on a set of machines subject to precedence and machine-capacity constraints. Each job consists of an ordered sequence of operations, where every operation requires exclusive use of a specified machine for a fixed processing time. The standard objective for these instances is the minimisation of the makespan. The collection aggregates several well-known benchmark families from the scheduling literature. Instance names typically encode the originating benchmark family and an instance identifier, for example `ft06`, `la21`, `orb01`, `swv01`, or `ta80`.

B. PSPLib RCPSP

PSPLib [8] is a benchmark library for project scheduling problems, including the single-mode Resource-Constrained Project Scheduling Problem (RCPSP). In the RCPSP, project activities must be scheduled subject to precedence constraints and limited renewable resources, with the objective of minimising the total project duration. The benchmark instances were generated systematically using the ProGen project generator [3] and are characterized by input parameters controlling structural and resource-related properties of the projects, such as resource factor, resource strength, and network complexity. The standard single-mode RCPSP benchmark sets are commonly denoted `j30`, `j60`, `j90`, and `j120`, where the number indicates the number of non-dummy project activities.

Due to the large number of instances included in these datasets, we restricted our submission to a subset of the problem families and instances. We solved each instance using CPMpy with the default OR-Tools CP-SAT as solver, and selected a subset with varying solve-time requirements; both easy and hard instances to evaluate the scaling rules of different solvers.

References

[1] Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2024 xcsp3 competition, 2024. URL: https://arxiv.org/abs/2412.00117, arXiv:2412.00117.

[2] Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian. Maxsat evaluation 2024: Solver and benchmark descriptions, 2024.

[3] Andreas Drexl, Rüdiger Nissen, James H. Patterson, and Frank Salewski. Progen/πx – an instance generator for resource-constrained project scheduling problems with partially renewable resources and further extensions. *European Journal of Operational Research*, 125(1):59–72, 2000. URL: https://www.sciencedirect.com/science/article/pii/S0377221799002052, doi:10.1016/S0377-2217(99)00205-2.

[4] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 2021. doi:10.1007/s12532-020-00194-3.

[5] Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

[6] Marijn JH Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of sat competition 2024: Solver, benchmark and proof checker descriptions, 2024.

[7] Taemyung Kim. JSPLIB: Job shop scheduling problem library. https://github.com/tamy0612/JSPLIB, 2016.

[8] Rainer Kolisch and Arno Sprecher. PSPLIB-a project scheduling problem library: OR software-ORSEP operations research software exchange program. *European journal of operational research*, 96(1):205–216, 1997.

[9] Pseudo-Boolean Competition 2024 (PB24). https://www.cril.univ-artois.fr/PB24/, 2024.

[10] Shift Scheduling Benchmark Instances — schedulingbenchmarks.org. https://schedulingbenchmarks.org/. [Accessed 14-05-2026].

[11] Thomas Sergeys, Ignace Bleukx, and Tias Guns. Unified programmatic access to co benchmarks, to connect constraint solving communities. In *Proceedings of the 29th International Conference on Theory and Applications of Satisfiability Testing (SAT 2026)*, 2026. Accepted for publication.

[12] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

Cost-Optimal Delete-Free Classical Planning as MaxSAT: Benchmarks for MaxSAT Evaluation 2026

Masood Feyzbakhsh Rankooh, Andreas Niskanen, and Matti Järvisalo
HIIT, Department of Computer Science, University of Helsinki, Finland
{masood.feyzbakhshrankooh,andreas.niskanen,matti.jarvisalo}@helsinki.fi

***Abstract*—We describe a benchmark set for the weighted track of the MaxSAT Evaluation 2026 based on cost-optimal delete-free classical planning. The instances are obtained from standard IPC/PDDL planning domains and translated into weighted MaxSAT instances. The underlying approach encodes planning tasks as MaxSAT formulas that guarantee achievement of the planning goals while maintaining feasible causal relations through acyclicity constraints. The benchmark set consists of 50 WCNF instances from three planning problem sets: PIPESWORLD-TANKAGE, TRANSPORT-SAT14, and VISITALL-SAT14.**

I. Introduction

We describe a benchmark set submitted to the weighted track of the MaxSAT Evaluation 2026 based on cost-optimal delete-free classical planning. The benchmark instances were generated with he the MaxSAT encoding introduced in [1] and consist of 50 WCNF instances generated from standard IPC/PDDL planning domains.

The underlying approach translates delete-free planning tasks into weighted MaxSAT instances by representing actions, propositions, and causal dependencies using propositional variables and constraints. The encoding guarantees achievement of the planning goals while maintaining feasible causal relations through acyclicity constraints over the dependency structure induced by the selected actions. Action costs are represented directly in the optimization objective of the resulting weighted MaxSAT instances. For a detailed description on the problem and the MaxSAT encoding, we refer the reader to [1]; we provide here only a high-level intuitive description. The benchmark set is intended to provide structured optimization instances originating from AI planning for evaluating modern MaxSAT solvers.

II. Delete-Free Planning as MaxSAT

The submitted benchmark instances are generated using the weighted MaxSAT encoding introduced in [1] for computing the h^+ heuristic for classical planning tasks. The h^+ heuristic is the optimal delete-relaxation heuristic: it gives the minimum cost of a plan in the delete-free relaxation of a planning task. Since h^+ provides an admissible and informative estimate of the cost of reaching the goal, computing it exactly is an important optimization problem in classical planning.

The MaxSAT encoding represents the delete-free planning task as a cost-minimization problem over selected actions and their causal support relations. A solution corresponds to a set of actions that achieves the goal propositions from the initial state while respecting the causal dependencies between actions and propositions. The total cost of the selected actions forms the optimization objective.

To ensure that the selected support structure corresponds to a valid relaxed plan, the encoding rules out cyclic causal dependencies. The submitted instances use the vertex-elimination-based acyclicity encoding of Rankooh and Rintanen [2]. The resulting weighted MaxSAT instances therefore encode exact computation of h^+ using a compact acyclicity-aware formulation.

III. Benchmark Description

The benchmark set consists of 50 weighted MaxSAT instances generated from standard IPC/PDDL planning benchmarks obtained from the classical planning benchmark repository.[1] The submitted instances are derived from three planning problem sets: PIPESWORLD-TANKAGE (20 instances), TRANSPORT-SAT14 (10 instances), and VISITALL-SAT14 (20 instances).

For the TRANSPORT-SAT14 benchmarks, instances `p01`–`p10` were used. For the VISITALL-SAT14 benchmarks, all 20 instances provided in the repository were used. For the PIPESWORLD-TANKAGE benchmarks, instances `p21`–`p40` were used. The generated benchmark files use the naming convention `domainname-instancename.wcnf`, where `domainname` denotes the planning domain and `instancename` corresponds to the original planning instance. Example benchmark names include `transport-sat14-strips-p01.wcnf`, `visitall-sat14-strips-pfile30.wcnf`, and `pipesworld-tankage-p21-net3-b12-g2-t60.wcnf`.

The benchmark instances were generated using the software described in [1]. The implementation is publicly available at [2]. Each planning instance was translated into a single weighted MaxSAT instance using the encoding described in Section II together with the vertex-elimination-based acyclicity encoding of [2]. All benchmark instances are provided in the standard WCNF format and are intended for the weighted track of the MaxSAT Evaluation 2026.

Acknowledgment

The work is financially supported by Research Council of Finland under grants 347588 (AN) and 356046 (MFR, MJ),

[1] https://github.com/AI-Planning/classical-domains/
[2] https://bitbucket.org/coreo-group/opth

Lights Out MaxSAT Benchmarks

André Schidler, Tobias Faller and Armin Biere
University of Freiburg, Germany

I. Introduction

Lights Out is a game played on a 5×5 grid, where each cell is an LED button. Buttons can be either *on* or *off* and at the start of the game, the state of each button is randomized. Whenever the player presses a button, the state of the button's closed (including the button itself) 4-neighborhood is flipped. The player wins when all buttons are off. We consider the game's generalization to $n \times n$ grids. Although the game can be generalized to general graphs, we focus on grids.

These benchmarks are hard for SAT solvers, although a polynomial time algorithm is known. A possible explanation is the problem's similarity to Tseitin Formulas. While Tseitin Formulas are defined over graphs, we use a hypergraph: the hypergraph's vertices are the grid's buttons and each closed 4-neighborhood is represented by a hyperedge. As with Tseitin Formulas, the game's rules can now be expressed using parity constraints over the neighborhoods in the hypergraph. Furthermore, these instances are interesting for core-based MaxSAT solvers: preliminary testing showed that core minimization is generally detrimental to good running times, particularly when using a core-guided approach.

II. Encoding

Given a sequence of button presses, the state of a button only depends on the number of button presses in its closed 4-neighborhood. In case the button is initially off, the button remains off whenever this number is even, otherwise, the number must be odd. This implies that the order of button presses does not matter, and we only require a Boolean variable $p_{x,y}$ which is true iff the button at coordinates (x, y) is pressed by the player (double presses are ignored). Further, we use a constant $s_{x,y}$ for each button, to indicate the button's initial state. Using $\oplus$ to denote the XOR operator and N[x, y] to denote the closed 4-neighborhood of the button at coordinates (x, y), the state of a button at coordinates (x, y) is described by

$$s_{x,y} \oplus \bigoplus_{(x',y') \in N[x,y]} p_{x',y'} \tag{1}$$

The SAT encoding then enforces that the state of each button must be 0. Hence, the encoded instance is satisfiable iff there exists a winning sequence of button presses. The soft clauses are simply the variables, thereby minimizing the size of the solution. The benchmark set only contains satisfiable instances.

We vary three parameters in our benchmarks: the size of the grid (n), how the grid is initialized (initialization), and how the XOR constraints are encoded (xor encoding). Benchmarks are named using the following convention: *lightsout_sat_⟨n⟩_unbounded_⟨xor encoding⟩_⟨initialization⟩_⟨i⟩_⟨satisfiable⟩.cnf*. Here, i is an index to distinguish benchmarks created with the same parameters and *satisfiable* (sat/unsat/unknown) indicates whether the instance is satisfiable.

A. Initialization

We initialize the grid using three different methods.

corner: Only the button at position $(0, 0)$ is on. Solving this configuration is generally hard, the size of the solution is unknown and a solution does not always exist.

randomk: We turn k many randomly selected buttons on. These instances are mostly hard, the size of the solution is unknown, and a solution does not always exist.

plantedk: Starting from a fully deactivated grid, we simulate press k many random button presses, which defines a winning strategy. Instances range from very hard to very easy, the solution length is at most k, and all instances are satisfiable.

B. Encoding the XOR

We consider two methods of encoding Equation 1.

direct: In case $s_{x,y}$ is true, we explicitly exclude each assignment that presses exactly 0, 2 or 4 buttons in $N[x, y]$. In case $s_{x,y}$ is false, we exclude those assignments pressing an odd number of buttons. For a vertex with four adjacent vertices, this requires 16 clauses, each of them containing five literals.

totalizer: We perform the XOR pairwise using auxiliary variables until only one auxiliary variables with the result remains. Conceptually, we construct a binary tree with the variables $p_{x',y'}$ as leaves, where each internal node represents an auxiliary variable that stores the parity over the variables represented by that node's children. For a vertex with four adjacent vertices, this requires 3 auxiliary variables, each defined by four ternary clauses. However, the auxiliary variables can be reused in the constraints for the neighbors.

MaxSAT Benchmarks for Learning Optimal Oblique Decision Trees

Florent Avellaneda
Université du Québec à Montréal (UQAM)
Montréal, Canada
avellaneda.florent@uqam.ca

***Abstract*—We submit a collection of MaxSAT benchmark instances arising from the problem of learning optimal sparse oblique decision trees from real-world classification datasets. The instances are produced by the MaxSAT encoding introduced in [1], which formulates the search for a depth-bounded oblique decision tree that minimizes the training classification error on a given dataset. The benchmark suite covers 13 datasets from the UCI Machine Learning Repository [2] and three tree depths ($d \in \{1, 2, 3\}$), yielding 39 WCNF instances that span a wide range of size and hardness.**

I. Problem Domain

An *oblique decision tree* is a binary tree in which each internal (branching) node splits the data according to a linear inequality over the feature space, and each leaf is labelled with a class. Given a set of labelled examples E and a maximum depth d, the *optimal oblique decision tree* problem consists in computing an oblique tree of depth at most d that maximizes the training accuracy on E. In the sparse variant considered here, each split is restricted to a linear combination of at most two features, a setting that preserves the interpretability of the resulting model.

II. Encoding

For each example $e \in E$, each branching node $n \in \mathcal{N}_B$, each leaf $n \in \mathcal{N}_L$, each class c, and each pair of features (f_1, f_2), the encoding introduces Boolean variables that respectively describe (i) the path followed by e through the tree, (ii) the leaf at which e ends, (iii) the class assigned to each leaf, and (iv) the pair of features used to split at each branching node.

Hard clauses enforce the structural constraints of the tree: each leaf is assigned a single class, each example reaches exactly one leaf, and each branching node selects exactly one pair of features. Additional hard clauses encode the geometric condition characterizing the existence of a linear separator in the plane of the two selected features. This geometric encoding, based on triangles involving two points at infinity in the projective plane, is the key contribution that makes the SAT formulation tractable; we refer the reader to [1] for a complete description.

Soft clauses (each of unit weight) state that whenever an example e reaches a leaf n, the class label of n must match the true label of e. Minimizing the total weight of falsified soft clauses is therefore equivalent to minimizing the number of training misclassifications, i.e. maximizing training accuracy.

III. Datasets and Parameters

The 13 datasets used to generate the instances are taken from the UCI Machine Learning Repository [2]. Their characteristics are summarized in Table I. All datasets correspond to standard classification tasks and span a variety of sizes (from 70 to 1727 examples), number of features (from 3 to 16) and number of classes (from 2 to 7).

TABLE I
Datasets used to generate the benchmark instances.

Dataset	#Examples	#Features	#Classes
Balance-scale	625	4	3
BreastCancer	116	9	3
Car	1727	6	2
Cryptotherapy	90	6	2
Heart 60+	309	13	2
Heart	1025	13	2
Immunotherapy	90	7	2
Iris	150	4	3
Mouse	70	5	2
Person Index	500	3	6
SIRTUIN6	100	6	2
Wine	178	13	3
Zoo	101	16	7

For each dataset, three instances are generated, one for each maximum tree depth $d \in \{1, 2, 3\}$. This results in a total of $13 \times 3 = 39$ WCNF instances.

IV. File Naming Convention

Each instance is stored as a single file in the revised WCNF submission format of MSE 2026. File names follow the pattern:

`oblique_dt_<dataset>_d<depth>.wcnf`

where `<dataset>` is a short identifier of the source dataset (e.g. `iris`, `wine`, `breastcancer`, `balance-scale`) and `<depth>` $\in \{1, 2, 3\}$ is the maximum depth of the oblique decision tree. For instance, the file `oblique_dt_iris_d2.wcnf` contains the MaxSAT encoding of the optimal sparse oblique decision tree of depth at most 2 for the Iris dataset.

References

[1] F. Avellaneda, "Learning optimal oblique decision trees with (Max)SAT," in *Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI)*, 2025.
[2] M. Kelly, R. Longjohn, and K. Nottingham, "The UCI machine learning repository," https://archive.ics.uci.edu, 2023.

MaxSAT Benchmarks: Finding Optimal Counterfactual Explanations for Random Forests

Alesya Raevskaya
Aalto University
Finland
alesya.raevskaya@aalto.fi

Tuomo Lehtonen
Aalto University
Finland
tuomo.lehtonen@aalto.fi

We present MaxSAT benchmarks from the domain of finding an optimal (minimum distance/cost) counterfactual explanation for random forest classifiers [1]. The underlying random forest classifiers are trained on various datasets from the UCI machine learning library [2]. The hard clauses encode the paths of each decision tree in the forest as well as the votes of the trees, while the soft clauses encode the distance from a (fixed) input sample to another sample that is classified differently to the input. The weights are originally real-valued, but to obtain this competition benchmark set, the weights are multiplied by 10^{12} and rounded to the nearest integer.

I. Optimal Counterfactual Explanations for Random Forests

A random forest is a machine learning classifier consisting of a collection of decision trees, which separately classify an input sample, and the overall classification is selected by aggregating the outputs of the trees. Our encoding is for *hard voting* random forests, where each tree votes for a single class and the class with the most votes is the output for the forest. Given a random forest classifier C, an input sample $\mathbf{v}$ (i.e. a vector over the features that C operates on), and a distance function d, a counterfactual explanation is a sample $\mathbf{x}$ such that C classifies $\mathbf{v}$ and $\mathbf{x}$ differently, and there is no counterfactual $\mathbf{x}'$ closer to $\mathbf{v}$ in terms of d. We now give a high-level overview of the MaxSAT encoding for finding optimal counterfactual explanations for random forests [1].

The hard clauses mostly follow the encoding by [3]. The hard clauses enforce 1) domain consistency over variables for features of the dataset 2) the path that a sample takes in each tree, 3) the votes of each tree, and 4) finally, via cardinality constraints that the votes of the trees are such that the desired class is selected by the whole forest. A key point is that for each numerical feature, the trees have a set of known thresholds; nodes in the decision trees have conditions of the form: "if the value for a feature f is less than $c \in \mathbb{R}$, then the left child is selected, and otherwise the right". A Boolean variable is introduced for each of these thresholds, and they are encoded to be ordered according to the numerical values of the corresponding thresholds. In particular, for each feature i, we have the variables t^i_j for $1 \leq j \leq h$ where h is the number of conditions on feature i in any of the trees.

The soft clauses capture the distance from an input sample to other possible samples. In these benchmarks, the one norm is used as the distance function. This results in two types of soft clauses. For categorical features, soft clauses are added to enforce that the category changing incurs a cost of 1. For real-valued features, soft clauses enforce that for each threshold occurring in any tree in the classifier, if the counterfactual lies on the other side of threshold compared to the input sample, then a cost corresponding to the distance between that and the previous threshold is incurred. That is, if for a real-valued feature i, the value in the input sample is between the j-th and $(j+1)$-th thresholds for i, we have the soft clauses

$$\begin{array}{ll} (\neg t^i_k) & \text{for } 1 \leq k < j, \\ (\neg t^i_j) & \\ (t^i_{j+1}) & \\ (t^i_k) & \text{for } j+1 < k \leq T_i, \end{array}$$

where T_i is the total number of thresholds for feature i. The cost is normalized so that crossing all thresholds equals a cost of 1. Finally, the weights are multiplied by 10^{12} and rounded to the nearest integer; a smaller multiplier is possible but risks losing precision in the optimal cost. The weights for the soft clauses tend to be highly varied, since there can be a large number of different thresholds for each real-valued feature.

II. Benchmarks

The underlying random forests are trained on a subset of the datasets considered in [1]: Adult, Online News, and Spambase[1]. Each of these datasets include multiple numerical features, and most have categorical features: Adult has 5 numerical and 6 categorical, Online News has 43 numerical and 4 categorical, and Spambase has 57 numerical and no categorical features.

The benchmark set includes random forests of increasing size[2], in particular in terms of the maximum allowed depth d for the decision trees and the number of decision trees in the classifier N. We selected the possible values for these

[1]Some other datasets were observed to result in very easy MaxSAT instances and were thus excluded.

[2]In terms of the application, scaling up the size of the random forests make sense, because in general the larger the forest, the better classification accuracy it has (up to a point, but performance does not tend to decrease).

parameters separately for each dataset in order to generate difficult instances, based on preliminary testing. The parameters are $N \in \{300, 400, 500\}$ and $d \in \{8, 10, 12\}$ for ADULT, $N \in \{100, 150, 200\}$ and $d \in \{7, 8, 9\}$ for ONLINE NEWS, and $N \in \{80, 100, 120\}$ and $d \in \{4, 5, 6\}$ for SPAMBASE. For each dataset, 10 samples from the dataset were drawn as input samples to find a counterfactual explanation for. This results in a total of 270 instances.

Note that a different input sample results in different soft clauses, since the soft clauses encode distance from the input sample to other samples. The hard clauses are the same for samples that are classified to the same class. Thus instances with a different forest (with different maximum depth or number of trees) differ from each other more than instances with the same forest but a different sample. Nonetheless, instances with the same forest but different sample can also differ significantly from each other in terms of the optimal cost and hardness of solving.

Given $dataset \in \{\text{ADULT}, \text{NEWS}, \text{SPAMBASE}\}$, number of trees N and maximum depth of trees d, we have a WCNF file with the following naming for each of the 10 input samples numbered by $0 \leq i \leq 9$:

maxorc_$dataset$_trees-N_depth-d_i.wcnf.

(MaxORC is the name of the algorithm, short for MaxSAT for optimal random forest counterfactuals.)

ACKNOWLEDGMENTS

This work was financially supported by Helsinki Institute for Information Technology HIIT.

REFERENCES

[1] A. Raevskaya and T. Lehtonen, "Optimal counterfactual explanations for random forests with maxsat," in *Proceedings of the 28th European Conference on Artificial Intelligence*, ser. Frontiers in Artificial Intelligence and Applications, I. Lynce, N. Murano, M. Vallati, S. Villata, F. Chesani, M. Milano, A. Omicini, and M. Dastani, Eds. IOS Press, 2025, pp. 903–910.

[2] K. N. Markelle Kelly, Rachel Longjohn, "The UCI machine learning repository," available at https://archive.ics.uci.edu.

[3] Y. Izza and J. Marques-Silva, "On explaining random forests with SAT," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, Z. Zhou, Ed. ijcai.org, 2021, pp. 2584–2591.

MaxSAT encodings for learning MDL rules from noisy data: benchmark description

Andrew Cropper
University of Helsinki
andrew.cropper@helsinki.fi

I. Inductive Logic Programming

Inductive logic programming (ILP) [1], [2] is a form of logic-based machine learning. The goal is to find a hypothesis (a set of rules) that generalises given training examples and background knowledge. As with all forms of machine learning, handling noisy data is a challenge in ILP. POPPER [3], [4] is an ILP approach that learns from noisy data using the minimum description length (MDL) principle [5]. MDL balances how well a hypothesis fits the training examples against hypothesis complexity.

II. MDL Problem

For an ILP learning task, let E^+ and E^- be positive and negative training examples respectively, B be background knowledge, and $R = \{r_1, \ldots, r_m\}$ be a set of candidate definite rules. The goal is to find a hypothesis $h \subseteq R$ that minimises the MDL cost function:

$$cost(h) = \left(\sum_{r \in h} s(r)\right) + fn(h) + fp(h) \tag{1}$$

where $s(r)$ is the number of literals in rule r and the exceptions are:

$$fp(h) = |\{e \in E^- \mid \exists r \in h : B \cup \{r\} \models e\}| \quad \text{(false positives)}$$

$$fn(h) = |\{e \in E^+ \mid \forall r \in h : B \cup \{r\} \not\models e\}| \quad \text{(false negatives)}$$

where $\models$ denotes logical entailment.

III. MDL Encoding

POPPER formulates the search for an MDL hypothesis as a weighted partial MaxSAT problem. We use a variable x_i for each rule $r_i \in R$ and an indicator variable c_e for each example $e \in E^+ \cup E^-$. For each positive example $e \in E^+$, we include the hard clause:

$$c_e \rightarrow \bigvee_{\substack{r_i \in R \\ B \cup \{r_i\} \models e}} x_i \tag{2}$$

to ensure that if the hypothesis h covers e, then at least one selected rule covers e. For each negative example $e \in E^-$, we include the hard clause:

$$\neg c_e \rightarrow \bigwedge_{\substack{r_i \in R \\ B \cup \{r_i\} \models e}} \neg x_i \tag{3}$$

to ensure that if the hypothesis h does not cover e, then none of the selected rules cover e. We encode the cost function using soft clauses:

- $(\neg x_i)$ with weight $s(r_i)$, for each rule $r_i \in R$, penalising rule complexity
- (c_e) with weight 1, for each $e \in E^+$, penalising false negatives
- $(\neg c_e)$ with weight 1, for each $e \in E^-$, penalising false positives

IV. Benchmark Set

The benchmark set comprises 7,512 WCNF instances derived from 916 tasks across 23 domains. Each instance filename follows the format `domain_task_increment.wcnf`. Each increment corresponds to a version of the problem with a growing candidate rule set R: the instance at increment $k+1$ contains all the candidate rules of increment k plus additional ones. The domains include knowledge base completion [6], relational reasoning [7], string transformations [8], list processing [9], molecular property prediction [10], game playing [11], and visual reasoning [12].

References

[1] S. Muggleton, "Inductive logic programming," *New Generation Computing*, vol. 8, no. 4, pp. 295–318, 1991.
[2] A. Cropper and S. Dumancic, "Inductive logic programming at 30: A new introduction," *J. Artif. Intell. Res.*, vol. 74, pp. 765–850, 2022.
[3] A. Cropper and R. Morel, "Learning programs by learning from failures," *Mach. Learn.*, vol. 110, no. 4, pp. 801–856, 2021.
[4] C. Hocquette, A. Niskanen, M. Järvisalo, and A. Cropper, "Learning MDL logic programs from noisy data," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI*. AAAI Press, 2024, pp. 10 553–10 561.
[5] J. Rissanen, "Modeling by shortest data description," *Autom.*, vol. 14, no. 5, pp. 465–471, 1978.
[6] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 26, 2013.
[7] J. Robinson, R. Ranjan, W. Hu, K. Huang, J. Han, A. Dobles, M. Fey, J. E. Lenssen, Y. Yuan, Z. Zhang, X. He, and J. Leskovec, "Relbench: A benchmark for deep learning on relational databases," in *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024.
[8] A. Cropper, "Playgol: Learning programs through play," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, 2019, pp. 6074–6080.
[9] J. S. Rule, S. T. Piantadosi, A. Cropper, K. Ellis, M. Nye, and J. B. Tenenbaum, "Symbolic metaprogram search improves learning efficiency and explains rule learning in humans," *Nature Communications*, vol. 15, no. 1, p. 6847, 2024.

Identifying Molecular Formulas from Mass Spectra via MaxSAT

Shuolin Li*, Sami Cherif†, Chu-Min Li†, Ahlam Mentag†, Rebecca Dauwe† and Jialu Zhang†
*Aix Marseille University
shuolin.li@univ-amu.fr
†Université de Picardie Jules Verne
sami.cherif, chu-min.li, ahlam.mentag, rebecca.dauwe, jialu.zhang@u-picardie.fr

***Abstract*—This document describes a set of benchmarks submitted to the MaxSAT Evaluation 2026, for identifying molecular formulas from mass spectra.**

I. Background

Mass spectrometry (MS), combined with fragmentation techniques, is widely used in metabolomics, proteomics, and small-molecule analysis to identify and characterize chemical compounds. In MS, an unknown molecule is fragmented (e.g., by collision-induced dissociation), producing a set of fragment ions. Each fragment corresponds to a portion of the original molecule, and its mass-to-charge ratio (m/z) and intensity are recorded in the spectrum. In a mass spectrum such as Figure 1, the x-axis shows the mass normalized by charge (m/z), and the y-axis shows relative intensity, which reflects the abundance of each detected ion.

However, inferring the molecular formula directly from a spectrum remains difficult: a given m/z value (a peak) can correspond to multiple chemical formulas, and fragments with the same m/z may originate from different parts of the molecule. To address this ambiguity, in 2008, Böcker and Rasche [3] proposed identifying the most likely fragmentation process and using it to determine the molecular formula. This process can be represented as computing an optimal fragmentation tree, a problem which is classically formulated as a *Maximum Colorful Subtree problem*.

II. Maximum Colorful Subtree Problem

We consider a directed acyclic graph $G = (V, E)$ where each vertex $v \in V$ represents a candidate fragment from a mass spectrum, we assume a distinguished vertex representing the original molecule, which serves as the root. Each edge $(u, v) \in E$ represents a fragmentation step: fragment v can be derived from fragment u by losing a particular ion; each edge has a real-valued weight given by a function $w : E \to \mathbb{R}$. The edge weight reflects the chemical plausibility of the corresponding loss, where common losses are assigned higher weights, while rare or implausible losses receive lower weights or penalties [2]. A coloring function $c : V \to C$ assigns each vertex a color, vertices corresponding to the same peak share the same color; different peaks have different colors. We call G a fragmentation graph and the task is to find a rooted subtree $G_T = (V_T, E_T)$ that is *colorful*, meaning that no two vertices in the subtree share the same color, and that maximizes $W(G_T)$, where $W(G_T) = \sum_{e \in E_T} w(e)$.

For a given mass spectrum, there may be multiple candidate fragmentation graphs. Figure 2 shows one such fragmentation graph containing 9 vertices and 11 edges. The root node $C_{17}H_{22}BFO_6$ fragments into 6 ion candidates, each connected to the root by a directed edge. Two nodes are colored green and two are colored brown, indicating that they correspond to the same observed peak (i.e., they have similar mass). Each edge is associated with a Boolean selection variable x_i and a weight that may be negative; for clarity, we use integer weights in this example, although the model allows real-valued weights. In this graph, the three solid line edges correspond to the optimal solution, which has objective value 3 and selects edges x_2, x_8, and x_{10}, yielding a fragmentation tree with 4 nodes and total weight $3 (= -3 + 1 + 5)$.

The Maximum Colorful Subtree problem is NP-hard [4]. In practice, integer-linear-programming-based methods [6] are widely used to solve it. Here, we succinctly describe the

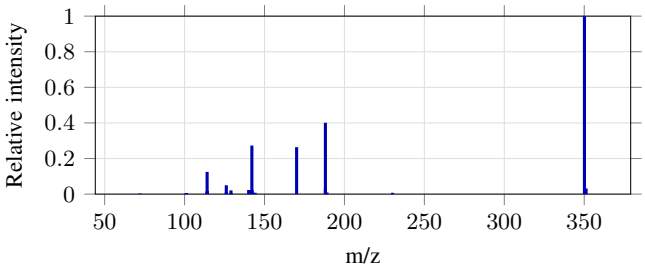


Fig. 1. Mass Spectrum

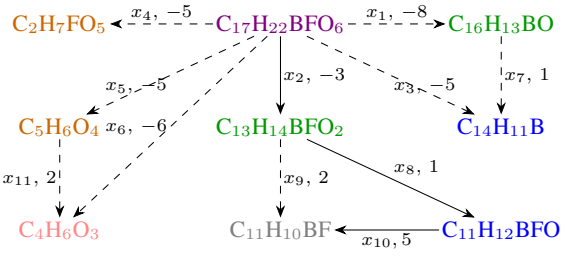


Fig. 2. A Fragmentation Graph

MaxSAT encoding for this problem and our benchmark submitted to the MaxSAT Evaluation 2026. For further details, we refer the reader to [5].

III. MaxSAT Encoding

The objective of the problem can be captured by introducing a unit soft clause for each edge $uv \in E$, whose satisfaction corresponds to selecting (or rejecting) that edge in the final subtree. For edges with positive weight $w_{uv} > 0$, we add the unit soft clause (x_{uv}) with weight w_{uv}, so selecting uv increases the MaxSAT objective by w_{uv}. Standard MaxSAT formats, however, do not allow negative weights. Therefore, for edges with negative weight $w_{uv} < 0$, we instead add the unit soft clause $(\bar{x}_{uv})$ with weight $|w_{uv}|$. Intuitively, this transformation rewards *excluding* a negative-weight edge, which is equivalent to penalizing its inclusion in the original objective.

The following hard constraints ensure connectivity, tree structure and the colorfulness constraint. In particular, Eq. (1) enforces that each selected vertex must have an incoming edge thus ensuring connectivity. Eq. (2) is a cardinality constraint ensuring that two nodes cannot have the same color. Importantly, we note that this constraint necessarily implies the tree structure since for each vertex $v \in V \setminus \{r\}$, all incoming edges uv share the same color (namely, the color of v), so enforcing that at most one vertex of each color is selected also enforces that v has at most one selected incoming edge.

$$\neg x_{vw} \vee \bigvee_{\substack{u \in V \\ uv \in E}} x_{uv}, \qquad \forall v \in V \setminus \{r\}, vw \in E \tag{1}$$

$$\sum_{uv \in E} x_{uv} \leq 1, \qquad \forall c \in C,\ \forall v \in V(c) \tag{2}$$

IV. MaxSAT Evaluation 2026 Benchmark

We submit 50 benchmark instances derived from MaxSAT encoding of the Maximum Colorful Subtree problem. The underlying fragmentation graphs contain 900–1000 vertices and 50,000–78,000 edges. The chosen CNF encoding for constraint (2) is Cardinality network [1].

References

[1] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 167–180. Springer, 2009.

[2] Sebastian Böcker and Kai Dührkop. Fragmentation trees reloaded. *Journal of cheminformatics*, 8(1):5, 2016.

[3] Sebastian Böcker and Florian Rasche. Towards de novo identification of metabolites by analyzing tandem mass spectra. *Bioinformatics*, 24(16):i49–i55, 2008.

[4] Michael R Fellows, Jens Gramm, and Rolf Niedermeier. On the parameterized intractability of motif search problems. *Combinatorica*, 26(2):141–167, 2006.

[5] Shuolin Li, Sami Cherif, Chu-Min Li, Ahlam Mentag, Rebecca Dauwe, and Jialu Zhang. Sur le calcul des sous-arbres colorés maximaux. In *Journées Francophones de Programmation par Contraintes (JFPC 2026)*, 2026.

[6] Imran Rauf, Florian Rasche, François Nicolas, and Sebastian Böcker. Finding maximum colorful subtrees in practice. *Journal of Computational Biology*, 20(4):311–321, 2013.

MaxSAT Evaluation 2026: Solver and Benchmark Descriptions, volume B-2026-1 of Department of Computer Science Series of Publications B, University of Helsinki 2026.

Overcommitted Cluster Scheduling Benchmarks

Jann Ehses, André Schidler and Armin Biere
University of Freiburg, Germany

I. Introduction

The motivation for this problem comes from efficiently benchmarking SAT solvers, where efficiently means using the cluster resources the shortest time possible. These solver runs are usually forced to stay within a memory limit and the number of parallel jobs on one cluster node is then limited such that the nodes' memory would not be exceeded even if each parallel solver run reaches the memory limit at the same time. However, in practice a solver rarely reaches (or gets even close to reaching) this memory limit.

This blocks unnecessary resources when the number of permitted parallel runs is lower than the number of cores. However, over the years the number of cores per CPU increased, and CPUs may now have more than 100 cores, while memory availability has not increased at the same rate.

We give as an example the Helix cluster we use for our experiments.[1] A node has 64 cores and 236 GB of memory which equals 3.7 GB per core. Using the SAT competition memory limit[2] of 32GB, we would use only 7 of the 64 cores, prolonging overall benchmarking time and blocking resources for other users by a factor of nine.

The submitted benchmarks compute a better schedule, where the worst-case memory usage may exceed the nodes' memory, but the expected memory usage based on past data is sufficiently low. The optimization goal is minimizing how long it takes to run the solver on all benchmark instances.

II. Definition

An instance consists of a tuple (B, t, m, s, ℓ), where B is a set of benchmark instances, $t : B \to \mathbb{R}^+$ is the expected runtime, $m : B \to \mathbb{R}^+$ is the expected memory consumption, s is the maximum number of parallel threads and ℓ is the memory limit. We use $s = 32$ and $\ell = 118$ which corresponds to one CPU on a Helix node.

The goal is to partition B into classes $\{B_1, \ldots, B_n\}$ such that for each $i \in [n]$ holds that $\Sigma_{b \in B_i} m(b) \leq m$. Further, the *runtime* of a class is $r(B_i) = \max_{b \in B_i} t(b)$, as the CPU is only released once all threads/solver runs in the class finish. The optimization goal is then to minimize the total runtime $\Sigma_i r(B_i)$.

III. MaxSAT Encoding

The encoding was created using PySAT [1]. The domain for runtimes and memory are reduced by dividing by a constant and then rounding to the next integer. The maximum number of classes can be determined by $\lceil \frac{|B|}{s} \rceil$. The assignment of benchmark instances to classes is then a simple case of using at-least-one and at-most-one constraints.

Memory constraints are enforced using pseudo-Boolean constraint implementations offered by PySAT. Further, let $T = \max_{b \in B} r(b)$ be the maximum runtime of any instance, then, we introduce for each class B_i weight one objective variables $r_{i,t}$ with $t \in [T]$. Then, for each benchmark instance $b \in B$ and class B_i, with $a_{b,i}$ being the variable indicating whether $b \in B_i$, we add the constraint

$$a_{b,i} \to r_{j,t}, \quad \text{for all } t \in [t(b)]$$

Running a MaxSAT solver on the instance will then minimize the overall runtime.

IV. Instances

We provide six instances, three are for the benchmark sets from the 2021, 2022 and 2023 SAT Competition respectively, all use minute precise runtime estimations and MB precise memory estimations. The other three instances are for the anniversary track. Since the number of benchmark instances in the anniversary track is much higher, the three instances use coarser granularity for runtimes and memory. The naming convention for these three instances uses *_tgX_* to denote the constant X (in seconds) by which all times are divided by before rounding, and *_rgX_* to denote the same for memory (in MB).

[1] https://wiki.bwhpc.de/e/Helix/Hardware
[2] https://satcompetition.github.io/2026/tracks.html

References

[1] A. Ignatiev, A. Morgado, and J. Marques-Silva, "Pysat: A python toolkit for prototyping with SAT oracles," in *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings* (O. Beyersdorff and C. M. Wintersteiger, eds.), Lecture Notes in Computer Science, pp. 428–437, Springer, 2018.

INCREMENTAL BENCHMARKS

Incremental MaxSAT Routing on a Physical Binary Sorter: A Deterministic Benchmark

Josep Maria Salvia Hornos
Department of Computer Engineering and Digital Design
University of Lleida, Spain
Department of IT and Systems, AI Division
Corporación Alimentaria Guissona S.A., Spain

***Abstract*—We present an incremental weighted CNF (WCNFI) file format for incremental MaxSAT solving and benchmarking, and showcase it with a routing optimization problem. This benchmark is derived from a real-world industrial logistics application of a physical binary sorter. The model translates rolling horizon item routing into incremental MaxSAT solving.**

I. Routing Optimization Problem

This benchmark models a logistics routing process on a physical binary sorter [1]. The system continuously receives a sequence of items, requiring real-time decisions to route each item to one of two physical outputs. The objective is to minimize mechanical route switches over time.

This minimization is subject to feasibility quotas defined at the article type level. Because the system operates online, decisions must be made under a rolling horizon where only partial future information is known. The benchmark is compiled into a deterministically generated sequence of optimization queries based on industrial traces published in IEEE DataPort [2].

II. Incremental solving

The incremental benchmark is generated from the rolling model of [1]. Each generated instance is parameterized by:

- prefix length j (how many items are included),
- optimization window size i (local rolling horizon),
- soft-weight profile parameters (α, k).

The generator emits WCNFI as follows:

1) Build a static MaxSAT encoding j items.
2) Emit all static hard clauses once as `h ... 0`.
3) Run rolling steps $t = 1, \ldots, j - i + 1$. At each step:
   - Update soft clause weights for positions in the active window $[t, t + i - 1]$ following (α, k).
   - Fix the (right-side) suffix positions $t + i, \ldots, j$ to their original policy target using query assumptions.
   - After the query, commit one decision literal for position t via added hard clause.

The commit literals are precomputed once from a fixed rolling non-incremental run and then replayed; this guarantees satisfiability. The benchmark is a deterministic replay log of incremental operations.

III. The WCNFI Format

The benchmark utilizes a deterministic incremental weighted CNF (WCNFI) format.

Valid lines are:

- `h <lit> ... 0`: Appends a permanent hard clause.
- `<weight> <lit> ... 0`: Defines or updates the weight of a soft clause.
- `r <lit> ... 0`: Executes a solve query under the list of assumption literals.

IV. Determinism and API Integration

The WCNFI format guarantees cost sequence determinism. That is, because the sequence of queries performed on the solver does not depend on random number generators or on the results of previous queries, sound solvers produce the same sequence of optimal costs. Moreover, if the problem is guaranteed to have a unique optimal model, the sequence of models is also guaranteed to be the same.

WCNFI allows all the operations of the IPAMIR API [3]: persistent hard constraints, incrementally mutable soft weights, and query assumptions. It is designed as a natural extension of the WCNF format popularized by the MaxSAT Evaluation [4]. WCNFI makes incremental MaxSAT benchmarking easier to debug than driver-dependent benchmarks (e.g. arbitrary code execution) at the expense of less flexibility.

Acknowledgments

Funding provided by Generalitat de Catalunya AGAUR DI-2024-00020.

References

[1] J. M. Salvia Hornos, C. Fernández Camón, C. Mateu Piñol, J. Marques-Silva, I. Rafegas Fonoll, and J. Cirera Balcells, "Modelling and optimising switching on a physical binary sorter," *Annals of Operations Research*, pp. 1–22, 2026.

[2] J. M. S. Hornos, J. P. Riera, R. P. Figueras, and R. T. Amigo, "Logistics routing of a box sequence on a physical binary sorter," 2024. [Online]. Available: https://dx.doi.org/10.21227/63x4-ta24

[3] A. Niskanen, J. Berg, and M. Järvisalo, "Incremental maximum satisfiability," in *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022, pp. 14–1.

[4] F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen, Eds., *MaxSAT Evaluation 2022 : Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B. Helsinki: Department of Computer Science, University of Helsinki, 2022, vol. B-2022-2, open access. [Online]. Available: http://hdl.handle.net/10138/347396

Incremental MaxSAT Benchmarks for Optimal TOHTN Planning via SibylSatOpt Trace Replay

Gaspard Quenard, Damien Pellier, Humbert Fiorino
Université Grenoble Alpes, CNRS, Grenoble INP, LIG, Grenoble, France
{gaspard.quenard, damien.pellier, humbert.fiorino}@univ-grenoble-alpes.fr

***Abstract*—This submission provides incremental MaxSAT benchmarks for optimal total-order hierarchical task network (TOHTN) planning. In automated planning, the goal is to compute a sequence of actions that transforms an initial state into a desired goal state. Hierarchical task network (HTN) planning extends this setting with high-level tasks and decomposition methods, which recursively refine tasks into smaller subtasks until an executable action sequence is obtained. The submitted benchmarks are generated from HDDL instances [2] using SibylSatOpt [1] and replayed through a lightweight IPAMIR benchmark application. Each input file records the sequence of MaxSAT calls produced during the SibylSatOpt search, allowing MaxSAT solvers to be evaluated on a realistic incremental optimization workload arising from hierarchical planning.**

I. Problem Description

The benchmarks target optimal TOHTN planning. In HTN planning, a problem is defined by an initial state, a set of high-level tasks to accomplish, and decomposition methods describing how a task can be refined into smaller subtasks. In TOHTN planning, the subtasks produced by each decomposition are totally ordered, so the final plan must follow a single action sequence. Given an HDDL domain and problem, the objective is to select decompositions until only primitive actions remain and the resulting action sequence is executable, while minimizing the total action cost.

The benchmarks considered in this submission follow the optimal TOHTN planning approach implemented in SibylSatOpt [1]. The solver explores the decomposition space layer by layer and encodes each layer as a MaxSAT formula. At each layer, it solves:

- a relaxed solution of this layer providing a lower bound on the plan cost of any solution that might exist in this problem, and
- the best concrete solution of this layer, with the cost of this solution

Optimality is established when both calls return the same objective value. The incremental benchmarks correspond exactly to this sequence of MaxSAT calls.

II. Benchmark Construction

For each HDDL instance, SibylSatOpt is executed with the following options:

- `-optimal`, to search for an optimal solution;
- `-wf`, to write the generated MaxSAT formula to a WCNF file;
- `-wim`, to record in WCNF comments the assumptions and soft clauses defining the intermediate MaxSAT calls; and
- `-sibylsat=0`, to enforce breadth-first exploration instead of the default greedy strategy, so that benchmark generation is deterministic and reproducible across runs.

The resulting trace contains a single incremental WCNF stream together with comment blocks specifying, for each search layer:

- the prefix of hard clauses active at that layer,
- the soft clauses at that layer, and
- the assumptions used in the corresponding primitive feasibility and optimality check.

The submitted benchmark application, `ipamirsibyltrace`, parses this trace and replays it layer by layer. For each layer, the application:

1) loads the newly activated hard clauses,
2) installs the layer-specific soft clauses,
3) solves the abstract MaxSAT call without assumptions,
4) solves the primitive MaxSAT call with the recorded assumptions, and
5) stops as soon as both objective values coincide.

This reproduces the optimization logic of SibylSatOpt while exposing it as a pure incremental MaxSAT benchmark. The application itself is identical for all TOHTN benchmark instances; only the trace input file changes.

III. Input Families

The traces can be generated from any TOHTN benchmark encoded in HDDL [2]. In this submission, the source instances are taken from the total-order HTN benchmarks of the 2023 International Planning Competition (IPC 2023) [3]. Only instances where a solution has been found by SibylSatOpt are included.

IV. File Naming Convention

Trace files follow the naming scheme

`inputs/<domain>/<domain>_<problem>.wcnf`

where:

- `<domain>` is a short identifier for the HDDL domain, and
- `<problem>` is the original problem identifier.

Solver Index

Aperture, 32

BouMS 3.0.1, 34

CASHWMaxSAT, 10
COGS, 13
CoreForge, 15
CoreTrail and ShareTrail, 17
CPMpy, 11

DeepDist-c, 36

EvalMaxRes, 19
EvalMaxSAT, 18

HWS: Solver Description, 38

Loandra, 39

MaxCDCL, 21
MaxHS Revised, 23
Metaheuristic Solver PRINTEMPS for MaxSAT Evaluation 2026, 42

NuWLS-c-2026, 44
NuWLS-c-2026-UB, 46

PASMaxSAT-c, 48

RustiMax, 26

SGAT-NuWLS-c, 50
SPB-NIM-c, 49

UWrMaxSat 2.0, 24
UWrMaxSat-MaxCDCL, 28

WMaxCDCL, 29

Benchmark Index

Answer Set Optimization, 55

Cost-Optimal Delete-Free Classical Planning, 60
CPMpy dataset, 58

Incremental MaxSAT Routing on a Physical Binary Sorter, 70

Learning MDL rules from noisy data, 65
Learning Optimal Oblique Decision Trees, 62
Lights Out MaxSAT Benchmarks, 61

Molecular Formulas from Mass Spectra, 66
Multiple Optimization Domains based on OP-to-MaxSAT Reduction, 53

Network Modularity Optimization, 56

Optimal Counterfactual Explanations for Random Forests, 63
Optimal TOHTN Planning via SibylSatOpt, 71
Overcommitted Cluster Scheduling Benchmarks, 68

Author Index

Ansótegui, Carlos, 56
Avellaneda, Florent, 18, 62

Berg, Jakob, 23
Berg, Jeremias, 39
Biere, Armin, 23, 61, 68
Bierlee, Hendrik, 11, 58
Bleukx, Ignace, 11, 58

Cai, Shaowei, 10, 44, 46
Cai, Yujie, 28, 48
Chen, Yin, 38, 49
Cherif, Sami, 19, 21, 29, 66
Chu, Yi, 44, 46
Cropper, Andrew, 65

Dauwe, Rebecca, 66
Dong, Yiyang, 10

Ehses, Jann, 68

Faller, Tobias, 61
Feyzbakhsh Rankooh, Masood, 55, 60
Fiorino, Humbert, 71
Fleury, Mathias, 23

Gao, Haokai, 36
Guns, Tias, 11, 58

Hao, Zhifeng, 53
He, Kun, 28, 48
Huang, Han, 53

Ignatiev, Alexey, 17
Inoue, Katsumi, 50

Jiang, Menghua, 36, 38
Jiang, Yingyi, 44, 46
Jin, Mingming, 28, 48
Justus, Edgar, 23
Järvisalo, Matti, 55, 60

Kang, Xiangui, 36
Koguma, Yuji, 42

Lehtonen, Tuomo, 63
Li, Chu-Min, 19, 21, 29, 66
Li, Shuolin, 19, 21, 29, 66
Lin, Peng, 44
Luo, Chuan, 44, 46
Lübke, Ole, 34, 39

Marques-Silva, Joao, 17
Martins, Ruben, 15
Mentag, Ahlam, 66
Moriyama, Sota, 50

Nadel, Alexander, 32
Niskanen, Andreas, 60

Pan, Shiwei, 10
Paxian, Tobias, 23
Pellier, Damien, 71
Piotrów, Marek, 24

Quenard, Gaspard, 71

Raevskaya, Alesya, 63
Ramaswamy, Vaidyanathan Peruvemba, 56

Sakai, Masahiro, 42
Salvia Hornos, Josep Maria, 13, 17, 70
Schidler, André, 23, 61, 68
Sergeys, Thomas, 11, 58
Slonimski, Yam, 32
Sun, Yiwen, 44, 46
Szeider, Stefan, 26, 56

Wang, Yiyuan, 10

Xia, Hai, 56

Yao, Weijie, 38
Yin, Minghao, 10

Zhang, Jialu, 19, 21, 29, 66
Zhao, Yuxin, 53
Zheng, Jiongzhi, 28, 48
Zhong, Zhaojin, 38, 49