Towards Modern and Modular SAT for LCG

Jip J. Dekker
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Alexey Ignatiev
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia

Peter J. Stuckey
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Allen Z. Zhong
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Abstract

Lazy Clause Generation (LCG) is an architecture for building Constraint Programming (CP) solvers using an underlying Boolean Satisfiability (SAT) engine. The CP propagation engine lazily creates clauses that define the integer variables and impose problem restrictions. The SAT engine uses the clausal model to reason and search, including, crucially, the generation of nogoods. However, while SAT solving has made significant advances recently, the underlying SAT technology in most LCG solvers has largely remained the same. Using a new interface to SAT engines, IPASIR-UP, we can construct an LCG solver which can swap out the underlying SAT engine with any that supports the interface. This new approach means we need to revisit many of the design and engineering decisions for LCG solvers, to take maximum advantage of a better underlying SAT engine while adhering to the restrictions of the interface. In this paper, we explore the possibilities and challenges of using IPASIR-UP for LCG, showing that it can be used to create a highly competitive solver.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Lazy Clause Generation, Boolean Satisfiability, IPASIR-UP

Digital Object Identifier 10.4230/LIPIcs.CP.2025.42

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/huub-solver/huub` [11] archived at `swh:1:dir:b28854946ae60e86a37051ea89465cce8b84b7ed`

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

Lazy Clause Generation [31] is an architecture for building Constraint Programming solvers by making use of an underlying SAT engine [27]. LCG solvers such as Chuffed [10] and CP-SAT [32] define the state-of-the-art for constraint programming and have consistently scored the most points in the Free search category in the MiniZinc challenge ever since 2010 when one was first entered.

However, the SAT infrastructure used in LCG solvers is mainly based on SAT technology from at least a decade ago. Meanwhile, SAT solvers have made significant advances, including inprocessing [24], chronological backtracking [29], as well as improvement to search strategies by including local search components [9].

With these developments, it is worthwhile to investigate how a more modern conflict-driven clause learning (CDCL) SAT solver impacts the design choices of Lazy Clause Generation solvers. In this paper, we make use of the IPASIR-UP interface [15, 16] for extending SAT solvers and developing a new LCG solver. While the interface limits the design decisions, it also directly benefits from all the improvements of modern SAT solvers. As a result, it becomes necessary to revisit and reassess key decisions in LCG to determine how they are affected by these improvements.

2 Background and Related Work

In this short paper, we assume that the readers have a reasonable understanding of how both CP and SAT solvers work. For more details, see e.g. CP [33] and SAT [19].

LCG solvers aim to combine the high-level reasoning of CP solvers, allowing, for example, integer variables and global constraints, with the powerful conflict analysis of SAT solvers. The integration is possible by creating CP propagators that make their inference available to the SAT solver in clausal form. A key challenge is mapping the representation of finite domain/integer variables from the CP model to Booleans. The usual approach is to represent variables domains using both equality $[\![x = d]\!]$ and bounds $[\![x \geq d]\!]$ Booleans. Each time the LCG solver propagates, which is equivalent to setting one of these Booleans to true or false, it must be able to explain the propagation, by computing a clause which is a consequence of the CP model which will cause the propagation in the current solver state.

The first LCG solver [30] was built on top of MiniSAT 2.0β. Representation of integer variables was eager, creating both the bounds $[\![x \geq d]\!]$ and equality $[\![x = d]\!]$ literals for each integer x before commencing search. The SAT solver controlled the search completely, and the CP engine was woken up once unit propagation ceased. This initial solver used *clausal explanation*, that is, when a propagator makes a new inference, it generated a clause that defined the propagation, and adds it to the SAT engine *permanently*. The new clauses then started more unit propagation in the SAT engine. This continued until failure was detected, and the SAT engine backjumped as usual, or a solution was found.

The next LCG architecture [17] reversed the control. Here the constraint programming engine was in control, performing search decisions. The unit propagation of clauses is treated as a (highest priority) propagator in the CP engine. The CP trail is extended with reason clauses for each propagation or failure, and conflict resolution works on the CP trail (in the usual CDCL manner). This architecture introduced *lazy encoding*, where equality literals were only created on demand, while bounds literals were created up front for integers x with small initial domains, or lazily during search for integers with large initial domains. In this architecture, the explanation strategy was *forward explanation*, where each propagator built an explanation clause for each new propagation, at the time of propagation, but these explanation clauses only lived on the trail, and are removed when backjumping.

Chuffed [10] was built around a customized version of MiniSAT 2.2. It is extended to use a *spaghetti stack*, so it can insert into the stack for choicepoints earlier than the current choicepoint. It is much more similar to a SAT engine, with a propagation loop which runs when unit propagation quiesces. Again here explanation clauses only live in the trail, but propagators could implement forward explanation, or *backwards explanation* where

the explanation clause is computed during conflict analysis, and not at propagation time. Chuffed did not implement propagators for many constraints, and instead encodes a number of constraints, such as `element` and `table`, into clauses.

After Chuffed was made open source in 2016, there has been a surge in LCG solvers. This might have been championed by the high-profile CP-SAT solver [32], developed as part of Google's OR-Tools. This solver includes novel CP and SAT engines that implement some more modern SAT features, such as Glucose's Literal Block Distance [3]. Other notable enhancements include the integration of a simplex based linear programming propagation [2], and a parallel portfolio approach of different variations of the solver, which makes the solver robust to different problem classes. At the same time, alternative solvers, such as Geas [22] and Pumpkin [12], have been developed by academics and have introduced exciting new features, such as core-guided search [21] and proof logging [20] for LCG.

A common trend among these new solvers is their tight integration and customization of the SAT solver employed. In the remainder of this paper, we will evaluate the opposite approach, where we maintain a clear boundary between the SAT and CP components, and the SAT solving is used without any customization.

3 LCG Architecture with a Modular SAT Engine

The recent IPASIR-UP interface [15, 16] augments a conflict-driven clause learning (CDCL) SAT solver [27] with the capability to invoke and interact with an external propagation engine to improve the CDCL solver's performance. The interface was originally implemented in the CaDiCaL solver [8] and successfully showcased in the setting of Satisfiability Modulo Theories (SMT) solving [4].

Although IPASIR-UP can be used to construct an LCG solver, it requires a change of perspective from the solver designer. Most LCG solvers use a SAT solver as a component to perform specific actions, such as clausal propagation and conflict resolution, within their search process. However, when using the IPASIR-UP interface, it is the SAT solver that runs the solving process, and it will make different calls to the external propagation engine at certain points in the process, much like the original LCG solver [30]. In our case, the external propagation engine will behave similarly to the CP component of most LCG solvers. It tracks the state of non-Boolean decision variables, such as integer variables, and oversees the running of propagators, which will communicate using literals and clauses.

Figure 1 shows the expected interactions between the SAT solver and our LCG engine. Hereinafter, when explaining the interplay between the SAT solver and the LCG backend, the names of the IPASIR-UP methods corresponding to the functionality being described are given in `verbatim`. The most important functionality of the IPASIR-UP interface is requesting external propagation. When this is required, the SAT solver will first notify the LCG backend of all literals that have been assigned since the last propagation (`notify_assignments`), and then ask it to propagate. To perform propagation, we maintain a priority queue of propagators that, given the new assignments, perform further inference [33]. We then activate the propagators in order until one performs any inference or the queue becomes empty. Propagators are required to inform the SAT engine about the outcome of their latest inference by providing literals that the SAT engine should assert (`propagate`) and clauses that can be added to the solver (`add_external_clause`). If no propagator performs any inference, then the LCG backend will report to be at fix point.

If the SAT solver performs conflict analysis that involves a literal inferred by the LCG backend, then the backend is expected to explain it by providing the corresponding antecedent clause (`add_reason_clause`). In the case of *backward explanation*, the backend reacts by

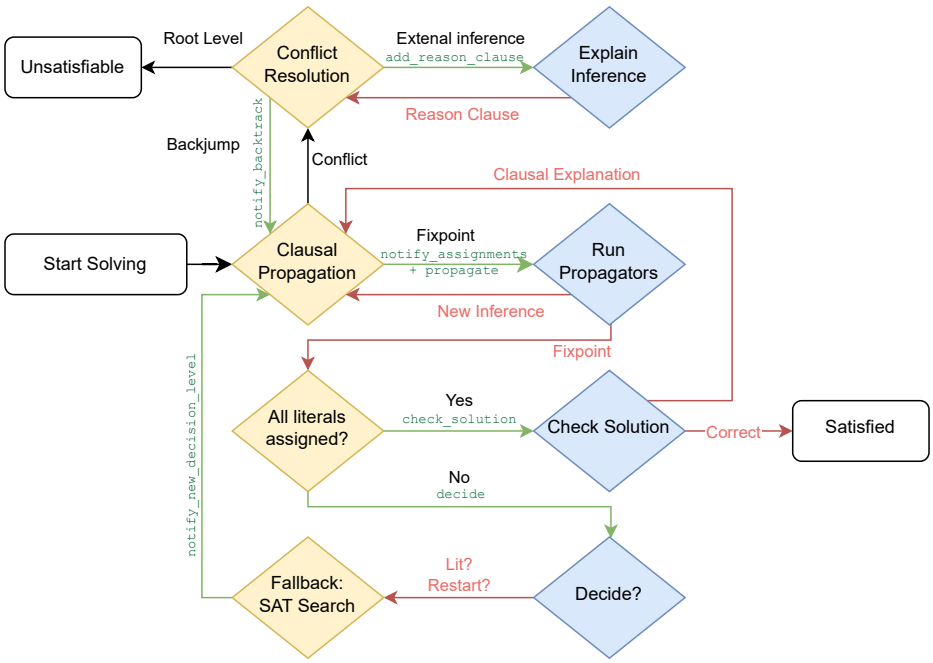

Figure 1 Depiction of the interaction between a SAT solver (yellow) and the LCG engine (blue) using IPASIR-UP interface. During green transitions, the SAT solver will make a call to the LCG. The red transitions represent the types of responses to these calls.

computing such a clause and passing it to the solver. Otherwise, in the case of *forward explanation*, it constructs and stores all antecedent clauses during propagation and then emits them to the SAT solver upon such a request.

Explanations passed by the LCG backend to the SAT engine through IPASIR-UP are allowed to contain only literals known to the solver during search. This represents an obstacle if propagators' literals are to be introduced lazily *during* backward explanation. This can, for example, prevent the strengthening of the explanation clauses through *lifting*, the process of generalizing an explanation. For example, given the constraint $2x + 2y + z < 5$, we might infer that $x \leq 1$ when $y \geq 1 \wedge z \geq 1$; however, we can infer the same when $z \geq 0$. So instead, the lifted explanation could be $(y \geq 1 \wedge z \geq 0) \rightarrow x \leq 1$. The above requirement prevents us from introducing (and defining) new literals at this point. So if $z \geq 0$ was an already existing literal, then this lifting optimization would be possible. However, if it is a *lazy* literal to be introduced for the explanation, then it is not.

For an LCG backend, `check_solution` behaves largely the same as `propagate`, since its task is to check that no constraints are violated to validate the solution. A complication is that any higher level decision variables that introduce literals lazily, such as integer variables, might not yet be fixed to a single value. The most straightforward solution to this problem is to assume that the variables take a value in their domain (e.g. the lower bound), and check the constraints using these values. If no conflicts are found under these assumed values, then a solution is found (with the assumed values). Otherwise, the conflict detected will result in a new clause that can be used by the SAT solver for further propagation, which will ensure that the combination of assumed values that caused the conflict cannot be taken.

It is well known that the search strategy determined by the branching heuristic used can have an enormous impact on solving performance. Using the IPASIR-UP's `decide` callback, the LCG backend can influence the search strategy. Most prominently, this method allows the backend to choose a literal to branch on. It can also be used to request the solver to restart, returning to a state where no decisions had been made. If no decision is made by the LCG backend, then the search strategy is left up to the SAT solver. Generally, it will make its decisions based on the variable state independent decaying sum (VSIDS) heuristic [28], and will use its own policy for restarts unless disabled.

Finally, the SAT solver is expected to inform the LCG backend of any changes to the decision level. It will inform the backend both when a new search decision level has been made (`notify_new_decision_level`) and when it backtracks to an earlier level (`notify_backtrack`). These notifications allow the LCG backend to maintain its own state that is reverted when backtracking. This can, for example, be used to maintain a state of which literals have been assigned and the current lower and upper bounds of integer variables.

4 Experimental Evaluation

We now introduce **Huub** [11], an open-source LCG solver implemented using the IPASIR-UP interface. We will use it to evaluate the performance of using modern SAT features and explore several design and engineering decisions for LCG solvers. The core engine of Huub is implemented in Rust 1.81.0 using CaDiCal 2.1.3 [8] as the back-end SAT solver, although in theory any SAT solver supporting the IPASIR-UP interface could be used. In the following experiments, the solver exercises through its FlatZinc interface. This allows us to use the MiniZinc Challenge [35] 2024 benchmarks, which consist of 20 problems with 5 instances each. Huub has propagators for all FlatZinc primitive constraints, but only supports the `all_different` and `disjunctive` global constraints. All other constraints are decomposed into these constraints or clauses using MiniZinc's decompositions. All experiments were run on a single core of an Intel Xeon Gold 6338 processor with 16 GB and a 20-minute time limit.

4.1 Comparison with State-of-the-art Solvers

Table 1 Comparison using instances from MiniZinc Challenge 2024.

Search	Solver	PAR2	OPTIMAL	UNSAT	SAT	UNK
SAT-based	CP-SAT	163832.54	54 (165.35)	3 (9.09)	41	2
	Chuffed	184494.97	50 (231.49)	2 (7.55)	31	17
	Huub	157613.18	55 (173.98)	4 (125.89)	35	6
Fixed	CP-SAT	241410.42	31 (119.30)	3 (9.08)	53	13
	Chuffed	252075.25	28 (125.60)	3 (12.20)	56	13
	Huub	250824.30	29 (192.61)	3 (78.75)	56	12

We first compare the baseline of our solver against Chuffed and CP-SAT. We compare the solvers in two different scenarios: (1) fixed search, where solvers must follow the search strategy in a model, and (2) SAT-based search, where solvers use the SAT search heuristic.

Table 1 presents the PAR2 scores obtained by the competitors calculated as the sum of the runtime plus twice the timeout for unsolved instances, i.e., the smaller PAR2 the better. For each configuration, we report the number of instances that are proven optimal

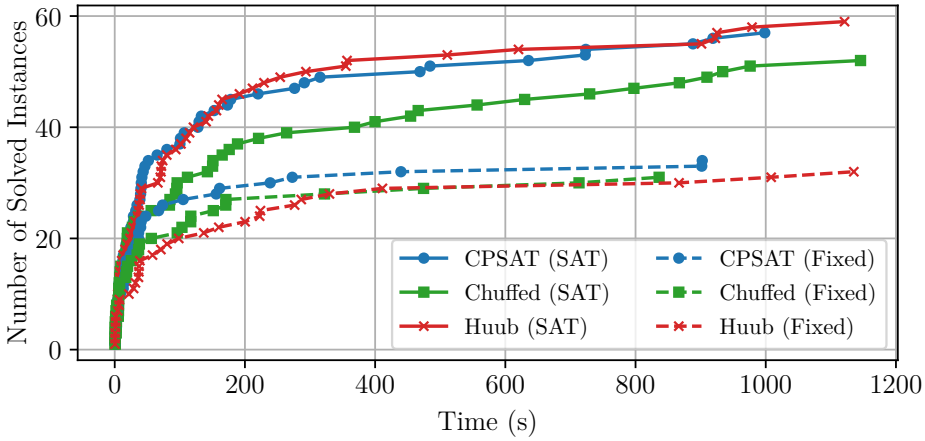

Figure 2 Cactus plot using instances from MiniZinc Challenge 2024.

(OPTIMAL), are proven unsatisfiable (UNSAT), where a solution without optimality proof is found (SAT), where no solutions are found in the time/memory limits (UNK). For the instance that are solved to completion, we also include the average solving time.

As shown in Table 1, Huub performs best when it uses the SAT search heuristic. When the fixed search strategy is used, Huub is outperformed by CP-SAT, but is better than Chuffed in terms of the PAR2 score. The cactus plot in Figure 2 also shows that Huub with the SAT-based search strategy outperforms the other solvers in terms of completed instances. Importantly, given that Huub currently implements only a limited set of propagators for global constraints, its performance has the potential for significant further improvement. A breakdown of the PAR2 scores for each problem, included as Section A, supports this observation. Specifically, CP-SAT and Chuffed outperform Huub substantially in the *accap*, *community-detection*, *network_50_cstr*, *tincy-cvrp*, *train-scheduling* and *yumi-dynamic* problems. For these problems, efficient propagators for global constraints, such as `diffn` [6], `cumulative` [34], and `global difference` [18], play crucial roles in solving performance. On the other hand, Huub demonstrates superior performance in problem categories such as *cable-tree-wiring*, *compression*, *fox-geese-corn*, *harmony*, and *word-equations*, where Huub has similar propagators to CP-SAT and Chuffed. This highlights the advantages of leveraging a modern and efficient SAT engine to enhance the performance of LCG solvers. We expect the overall performance of Huub can be improved with the integration of more efficient propagators for global constraints.

4.2 Inprocessing in Modern SAT Solvers

Using a SAT solver as modern as CaDiCaL enables us to use features that are not available in the SAT solvers used by other LCG solvers. Through the IPASIR-UP interface, Huub can easily leverage the latest advancements in modern SAT solvers. In particular, we will evaluate *inprocessing* techniques, which perform runtime simplification on the clause database, are widely recognized as essential for efficient SAT solving [14, 24]. Note that using the IPASIR-UP interface implicitly disables some pre/in-processing, which are not safe if the entire model is not known [16]. Additionally, our solver must sometimes reintroduce ternary clauses to ensure the consistency of integer literals.

Table 2 presents the results of an ablation study evaluating the impact of different inprocessing techniques including clause subsumption [5], variable elimination [13], failed literal probing [26], and globally blocked clause elimination [25]. The “Baseline” configuration does

Table 2 Comparison of different combinations of simplification techniques: clause subsumption (S), variable elimination (E), failed literal probing (P), and globally blocked clause elimination (C).

Configuration	PAR2	OPTIMAL	UNSAT	SAT	UNK
Baseline	157613.18	55 (173.98)	4 (125.89)	35	6
With All	154264.10	56 (174.73)	4 (117.36)	36	4
Without S	154259.11	56 (177.94)	4 (120.24)	35	5
Without E	157415.81	55 (172.44)	4 (124.77)	37	4
Without P	154514.60	56 (182.23)	4 (127.37)	36	4
Without C	154904.39	56 (189.04)	4 (128.99)	36	4

not perform any inprocessing, while "With All" enables all studied inprocessing techniques. The results indicate that enabling inprocessing techniques provides a slight, but consistent, improvement in solver performance compared to the baseline. Among the techniques studied, variable elimination has the most significant impact. Without it, Huub solves one fewer instance completely and has an increased PAR2 score. Removing any of the other inprocessing techniques results in an increased solving time for completed instances. This suggests that the performance benefits gained from inprocessing outweigh its computational overhead on these benchmarks. In the following, we enable all inprocessing techniques by default.

4.3 Revisit Explanation Mechanisms

In this section, we revisit a key decision in the construction LCG solvers, the way in which propagation is explained to the SAT engine. Chuffed uses both forward and backward explanation, and Huub implements the same strategy by default. However, with modern SAT solvers becoming increasingly efficient at handling clauses, pushing more clauses into the SAT engine can potentially enhance clause simplification and improve variable selection during search. To leverage these advantages, we implement a hybrid approach that combines clausal and backward explanation.

As a starting point, we experiment with using clausal explanations instead of forward explanation, and where the reasons were already created eagerly. We then introduce an additional threshold parameter L. Any propagation, using backward explanation, that is explained using a clause of size L or smaller will also use clausal explanation. This affects propagators handling global constraints with an undetermined number of variables, such as `linear`, `maximum` and `element`. Reason clauses are generated lazily when the number of involved variables exceeds the threshold; otherwise, they are generated eagerly.

Table 3 Comparison using different thresholds L for clausal explanation.

Configuration	PAR2	OPTIMAL	UNSAT	SAT	UNK
Forward	154264.10	56 (174.73)	4 (117.36)	36	4
$L = 0$	151590.61	57 (194.07)	4 (113.47)	36	3
$L = 3$	149572.99	58 (222.72)	4 (109.54)	34	4
$L = 5$	149860.45	58 (227.55)	4 (108.07)	36	2
$L = \infty$	157834.48	55 (174.20)	4 (116.44)	36	5

Table 3 presents the results of using different values of L. We note that when $L = 0$ only forward explanation is replaced by clausal propagation, and conversely when $L = \infty$ all propagation is through clausal explanation. The results indicate that increasing the number

of clauses pushed to the SAT solver generally improves the number of optimally solved instances compared to the baseline. Meanwhile, pushing more clauses into the SAT engine can introduce overhead, as we observe that there are more unknown instances with $L = 3$ compared to $L = 0$. $L = 5$ achieves the highest number of optimal instances and also reduces the number of unknown instances. However, the $L = \infty$ does not improve performance overall. Its computational overhead leads to a lower number of (completely) solved instances. This suggests that a moderate threshold allows the SAT solver to benefit from additional clauses, without introducing excessive overhead.

4.4 Disabling High-Level Propagation

Table 4 Comparison of different mechanisms for adaptive propagations.

Configuration	PAR2	OPTIMAL	UNSAT	SAT	UNK
Always Enabled	154264.10	56 (174.73)	4 (117.36)	36	4
Adaptive Engine	280496.22	20 (308.80)	4 (122.18)	72	4
$T = 10^{-2}$	154064.75	56 (174.82)	4 (118.25)	37	3
$T = 10^{-4}$	153367.16	56 (163.04)	4 (116.16)	35	5
$T = 10^{-6}$	151331.5	57 (188.60)	4 (118.82)	35	4

Our final experiment explores adaptive propagation. Previous work using IPASIR-UP [23] introduces an *adaptive* propagation engine for pseudo-Boolean constraints. This engine conditionally disables itself based on the number of calls to the propagation engine, the number of literals it propagates, and the proportion of assignments that satisfies its constraints. We reproduce the experiments in Huub using the same set of instances, where the goal is to enumerate all solutions. When the LCG engine is always enabled, all instances are solved in 2095 seconds. While disabling the LCG engine using the same conditions reduces the solving time to 1759 seconds. This reaffirms the claims in [23] and suggests that disabling higher-level propagation can lead to improvements in solver performance.

Since Huub supports a wider variety of constraint types, each of which may have different impacts on solving, we refined the adaptive mechanism to be more fine-grained. We introduce an adaptive scoring mechanism, similar to NVSIDS [7], that conditionally disables individual propagators based on their activity. The score for each propagator is initialized as 1. The score s is updated to $s' = f \cdot s$ if the propagator is called but neither propagates literals nor detects a conflict; otherwise, the score is updated as $s' = f \cdot s + (1 - f)$. The key difference with NVSIDS is that f takes the value $(1 - 0.5^c)$, where c is the number of conflicts encountered. The intuition behind this change is that, initially, the SAT-based search heuristic is random, and CP propagation may not yield useful information. As the search progresses, however, the propagator should become more active for proving optimality. If the score falls below a predefined threshold, the execution of that propagator is postponed until a solution is found, effectively transforming it into a checker that verifies whether assigned values remain consistent when all variables are fixed, which is done in `check_solution`.

Table 4 presents the results of applying the adaptive engine and adaptive propagators with different thresholds T to instances from the MiniZinc Challenge 2024. As shown, directly applying the adaptive engine mechanism reduces the number of instances solved to optimality, whereas adaptive propagators with an appropriate threshold slightly improve performance. Although the overall performance remains similar to the baseline, the adaptive

propagators approach demonstrates positive results for specific problems. In the *cable-tree-wiring*, *community-detection*, and *train-scheduling* problems, it either reduced solving times or improved solutions. Further details can be found in Appendix B. These results suggest that adaptive propagation, while promising, requires further investigation.

5 Conclusion

We present Huub, an LCG solver built based on the IPASIR-UP interface to SAT engines. This modular design allows us to easily swap in any SAT solver that supports the interface, enabling the solver to benefit from the latest advancements in SAT solving technology. Our implementation demonstrates competitive performance compared to state-of-the-art LCG solvers, and we expect further improvements with more global constraint propagators.

As SAT solvers become more and more efficient in handling clauses, it is timely to revisit the design and engineering of LCG solvers. In this work, we investigate various explanation strategies and adaptive propagation techniques. In the future, we plan to explore other options, such as dynamically decomposing constraints and encoding them into the SAT solver on the fly [1].

References

1 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J Stuckey. To encode or to propagate? the best choice for each constraint in SAT. In *International Conference on Principles and Practice of Constraint Programming*, pages 97–106. Springer, 2013. doi:10.1007/978-3-642-40627-0_10.

2 T Achterberg. Constraint integer programming. *Ph. D. Thesis, Technische Universitat Berlin*, 2007.

3 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

4 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. doi:10.3233/FAIA201017.

5 Roberto J Bayardo and Biswanath Panda. Fast algorithms for finding extremal sets. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 25–34. SIAM, 2011. doi:10.1137/1.9781611972818.3.

6 Nicolas Beldiceanu, Qi Guo, and Sven Thiel. Non-overlapping constraints between convex polytopes. In *International Conference on Principles and Practice of Constraint Programming*, pages 392–407. Springer, 2001. doi:10.1007/3-540-45578-7_27.

7 Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008. doi:10.1007/978-3-540-79719-7_4.

8 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53, 2020.

9 Shaowei Cai and Xindi Zhang. Deep cooperation of CDCL and local search for sat. In *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24*, pages 64–81. Springer, 2021. doi:10.1007/978-3-030-80223-3_6.

10 Geoffrey Chu. *Improving combinatorial optimization.* PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

11 Jip J. Dekker, Alexey Ignatiev, Peter J. Stuckey, and Allen Z. Zhong. Huub: Lazy Clause Generation Solver. Software, swhId: `swh:1:dir:b28854946ae60e86a37051ea89465cce8b84b7ed` (visited on 2025-07-24). URL: `https://github.com/huub-solver/huub`, `doi:10.4230/artifacts.24103`.

12 Emir Demirović, Maarten Flippo, Imko Marijnissen, Jeff Smits, and Konstantin Sidorov. Pumpkin Solver. URL: `https://github.com/ConSol-Lab/Pumpkin`.

13 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *International conference on theory and applications of satisfiability testing*, pages 61–75. Springer, 2005. `doi:10.1007/11499107_5`.

14 Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*, pages 136–154. Springer, 2019. `doi:10.1007/978-3-030-24258-9_9`.

15 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: user propagators for CDCL. In *SAT*, volume 271 of *LIPIcs*, pages 8:1–8:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPICS.SAT.2023.8`.

16 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. Satisfiability modulo user propagators. *Journal of Artificial Intelligence Research*, 81:989–1017, 2024. `doi:10.1613/JAIR.1.16163`.

17 T. Feydy and P.J. Stuckey. Lazy clause generation reengineered. In I. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *LNCS*, pages 352–366. Springer-Verlag, 2009. `doi:10.1007/978-3-642-04244-7_29`.

18 Thibaut Feydy, Andreas Schutt, and Peter J Stuckey. Global difference constraint propagation for finite domain solvers. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 226–235, 2008. `doi:10.1145/1389449.1389478`.

19 Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of sat. *Communication of the ACM*, 66(6):64–72, May 2023. `doi:10.1145/3560469`.

20 Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirovic. A multi-stage proof logging framework to certify the correctness of CP solvers. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.CP.2024.11`.

21 Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J Stuckey. Core-guided and core-boosted search for CP. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*, pages 205–221. Springer, 2020. `doi:10.1007/978-3-030-58942-4_14`.

22 Graeme Gange, Daniel Harabor, and Peter J. Stuckey. Lazy CBS: Implicit conflict-based search using lazy clause generation. In Nir Lipovetzky, Eva Onaindia, and David Smith, editors, *Proceedings of the 29th International Conference on Automated Planning and Scheduling*, pages 155–162. AAAI Press, 2019. URL: `https://www.aaai.org/ojs/index.php/ICAPS/article/view/3471`.

23 Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible sat technology. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 16–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

24 Matti Järvisalo, Marijn JH Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning*, pages 355–370. Springer, 2012.

25 Benjamin Kiesl, Marijn J. H. Heule, and Armin Biere. Truth assignments as conditional autarkies. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2019. doi:10.1007/978-3-030-31784-3_3.

26 Inês Lynce and Joao Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 105–110. IEEE, 2003.

27 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

28 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

29 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 111–121, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94144-8_7.

30 O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation = lazy clause generation. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 544–558. Springer-Verlag, 2007. doi:10.1007/978-3-540-74970-7_39.

31 O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009. doi:10.1007/S10601-008-9064-X.

32 Laurent Perron and Frédéric Didier. CP-SAT. URL: https://developers.google.com/optimization/cp/cp_solver/.

33 Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, 2008. doi:10.1145/1452044.1452046.

34 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Explaining the cumulative propagator. *Constraints*, 16:250–282, 2011. doi:10.1007/S10601-010-9103-2.

35 Peter J Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15:307–316, 2010. doi:10.1007/S10601-010-9093-0.

A Additional Results for Solver Comparison

Table 5 compares the PAR2 scores for different solver configurations on individual problems from the MiniZinc Challenge 2024.

B Additional Results for Adaptive Propagation

Table 6 presents a comparison of solving times for different adaptive propagation mechanisms across instances from the *cable-tree-wiring*, *community-detection*, and *train-scheduling* problems. Only instances solved by at least one configuration are included. The entry "t.o." denotes cases where the instance exceeded the time limit.

Table 5 Breakdown of PAR2 scores, with the smallest PAR2 score for each row highlighted.

Problem	SAT			Fixed		
	CP-SAT	Chuffed	Huub	CP-SAT	Chuffed	Huub
accap	**4382.25**	8452.66	7559.29	18002.19	18005.07	18003.38
aircraft-disassembly	10830.06	14419.04	11061.13	**10827.55**	10837.57	11050.72
cable-tree-wiring	8103.01	4317.21	**858.01**	18010.37	18011.34	18009.67
community-detection	**346.26**	1920.21	10950.04	7350.97	7827.96	11702.54
compression	4328.04	7213.36	**3719.28**	4042.27	7202.23	4741.58
concert-hall-cap	**14487.89**	14504.51	14761.80	18006.69	18009.45	18008.94
fox-geese-corn	14694.75	11625.72	**3843.77**	18001.89	18002.43	18001.61
graph-clear	7349.13	7841.57	7316.19	7250.30	**7224.61**	7337.01
harmony	7265.72	3796.52	**1048.57**	11143.52	7497.17	7795.56
hoist-benchmark	7619.59	4435.85	**4054.75**	18013.18	18017.04	18014.37
monitor-placement	5601.80	**5582.54**	5649.02	5601.50	11014.31	11003.24
neighbours	11802.65	**11068.51**	11781.09	14402.04	11638.80	14403.29
network_50_cstr	182.20	18017.16	424.07	**177.71**	7379.87	577.85
peacable_queens	14507.04	14412.62	14229.03	14476.76	**14411.15**	14484.60
portal	**7542.36**	8384.32	7585.00	14427.55	8089.08	8623.93
tiny-cvrp	**11024.93**	11259.59	14406.99	14406.67	14725.86	18003.81
train-scheduling	4713.67	**4241.32**	5377.87	14412.48	10910.69	14414.13
triangular	**14431.54**	14487.15	14614.11	14438.36	14440.20	14603.16
word-equations	3642.49	3639.17	**274.30**	3771.83	10813.68	3943.58
yumi-dynamic	**10977.15**	14875.93	18098.88	14646.59	18016.73	18101.33

Table 6 Solving times for instances in *cable-tree-wiring*, *community-detection*, and *train-scheduling*.

Problem	Instance	With All	Adaptive Engine	$T = 10^{-2}$	$T = 10^{-4}$	$T = 10^{-6}$
cable-tree-wiring	A022	100.59	t.o.	65.83	53.17	**47.88**
	A033	181.37	t.o.	188.59	240.81	**150.98**
	A041	11.56	t.o.	13.25	**7.21**	7.98
	A046	60.05	t.o.	56.81	51.47	**35.90**
	R193	451.48	t.o.	385.63	**258.37**	579.40
community-detection	adjourn.s200.k3	65.24	t.o.	55.67	**50.53**	51.24
	Polblogs.s2480.k2	77.45	t.o.	41.91	**37.56**	37.78
	Zakhary.s12.k3	**806.09**	t.o.	1071.65	1036.39	903.26
train-scheduling	trains09	932.35	t.o.	865.73	**547.81**	893.91
	trains12	29.83	21.37	24.31	**13.34**	20.44
	trains15	**7.02**	57.23	9.79	7.33	7.39
	trains18	665.75	t.o.	**541.34**	622.26	569.59