OptiLog V2: Model, Solve, Tune and Run

Josep Alòs
Logic & Optimization Group (LOG), University of Lleida, Spain

Carlos Ansótegui
Logic & Optimization Group (LOG), University of Lleida, Spain

Josep M. Salvia
Logic & Optimization Group (LOG), University of Lleida, Spain

Eduard Torres
Logic & Optimization Group (LOG), University of Lleida, Spain

Abstract

We present an extension of the OptiLog Python framework. We fully redesign the solvers module to support the dynamic loading of incremental SAT solvers with support for external libraries. We introduce new modules for modelling problems into Non-CNF format with support for Pseudo Boolean constraints, for evaluating and parsing the results of applications, and we add support for constrained execution of blackbox programs and SAT-heritage integration. All these enhancements allow OptiLog to become a swiss knife for SAT-based applications in academic and industrial environments.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Tool framework, Satisfiability, Modelling, Solving

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.25

Supplementary Material *Software (OptiLog at PyPI)*: `https://pypi.org/project/optilog/`

Funding Supported by Grant PID2019-109137GB-C21 funded by MCIN/AEI/10.13039/501100011033, PANDEMIES 2020 by Agencia de Gestio d'Ajuts Universitaris i de Recerca (AGAUR), Departament d'Empresa i Coneixement de la Generalitat de Catalunya, FONDO SUPERA COVID-19 funded by Crue-CSIC-SANTANDER, and the MICNN FPU fellowship (FPU18/02929).

1 Introduction

In the last twenty years, the efficiency of SAT engines (solvers) has experimented a great success. Actually, they have become the core engines of other higher-level engines: #SAT (Sharp-SAT), MaxSAT (Maximum Satisfiability), QBF (Quantified Boolean Formulas), PBO (Pseudo-Boolean Optimization), SMT (Satisfiability Modulo Theories), Model finding, Theorem proving, ASP (Answer Set Programming), LCG (Lazy Clause Generation), CSP (Constraint Satisfaction Problems), etc.

Despite the tremendous success of SAT applications in several domains, the access to these resources by members of other research communities, industrial environments, and students of undergraduate courses has been rather limited due to the absence of friendly frameworks. The same story applies to other areas of computer science.

The Python programming language [40], thanks to its simplicity, has dramatically turned the situation around, becoming the middleware to interconnect many scientific libraries through Python bindings such as Numpy [24], Pandas [41], scikit-learn [38], Pytorch [37], Keras [13], etc. This interconnection has definitely allowed to develop more complex applications and to indirectly justify further the individual utility of each library.

In Constraint Programming we also find several Python applications or bindings such as CPLEX [25], Gurobi [23], OR-Tools [21], COIN-OR [14], SCIP [19], Z3 [15], PySMT [20], *cnfgen* [27], PySAT [26], PyPbLib [32], SAT Heritage [5], OptiLog [2], etc.

© Josep Alòs, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres;
licensed under Creative Commons License CC-BY 4.0

In this paper, we focus on a new release of the OptiLog Python framework [2] for SAT-based applications with significant contributions. The idea is to make of OptiLog the tool of choice to support researchers, practitioners, or students along all the development process that involves modelling the problem, implementing a solving approach, tuning the overall approach, and evaluating its effectiveness. OptiLog covers all these steps. Moreover, thanks to its versatility, it can be used to develop other systems not necessarily related to SAT problems.

The paper is structured as follows: in Section 2 we present the general architecture of the new OptiLog framework. In Section 3, we present the new *Modelling* module to define problems. In Section 4 we describe how the *Solvers* module has been completely redesigned. Next, in section 5 we introduce the new *BlackBox* module to execute external tools within OptiLog. In Section 6 we describe the new additions to the *Tuning* module. In Section 7 we describe the new *Running* module to execute experiments and parse the results. Finally, we conclude with Section 8, providing some closing thoughts and future work.

We provide as supplementary material (also available from the OptiLog's documentation [31]) a case study with full detail that covers all the steps of the development process.

2 OptiLog Framework Architecture

OptiLog [2] was designed as a Python library for rapid prototyping of SAT-based systems. However, we will see that it provides features (such as the running and tuning modules) that can be used in other scenarios not necessarily only for developing constraint programming systems. In particular, OptiLog provided four main modules for its end-user API: The Formula module, the PB Encoder module (renamed as *Encode*r Module), the SAT solver module, and the Automatic Configuration (AC) module (renamed as *Tuning* module).

In this paper, we extend OptiLog's end-user API to ease its usage in education and industry by providing three new modules: a higher-level modelling language within the *Modelling* module, a *Running* module that simplifies the execution of experiments and their analysis, and a *BlackBox* Module that eases the integration of third-party tools. Additionally, the original *Solvers* module, which allows executing within OptiLog the C++ libraries of incremental SAT solvers, has been completely redesigned. Also, the *Tuning* module that allows the interconnection with automatic configurators has been extended and integrated with the new *Running* module.

OptiLog is the evolution of our PyPbLib [32] package, which is also used by PySAT [26]. OptiLog is now also available through PyPi [18]:

```
$ pip install optilog
```

Figure 1 shows the new architecture we propose for OptiLog, a full description on the current architecture can be found in the OptiLog manual [31]. This architecture supports the user along the development process:

1. **Model the problem** into a more richer and compact formalism (combining Non-CNF Boolean and Pseudo-Boolean expressions) through the *Modelling* module. And, translate the model into a formalism supported by constraint programming solvers, e.g. Boolean formulas in Conjunctive Normal Form (CNF) or Weighted CNF, thanks to the *Formula* and *Encoders* modules.
2. **Implement the solving algorithmic approach** through: (i) the *Solvers* module that allows using *External Libraries* such as libraries of incremental C++ SAT solvers and, (ii) the *BlackBox* module to execute *External Tools* such as generators, preprocessors, feature extractors, binaries of other solvers (such as the SAT solvers available from SAT Heritage [6]), or any other system command, etc.

3. **Tune both the encoding and solving choices** from a holistic point of view, through the *Tuning* module, to increase the effectiveness of their interconnection on a given training set.
4. **Evaluate the resulting system** on a test set thanks to the *Running* module.

In the following, we describe with further detail the new contributions.

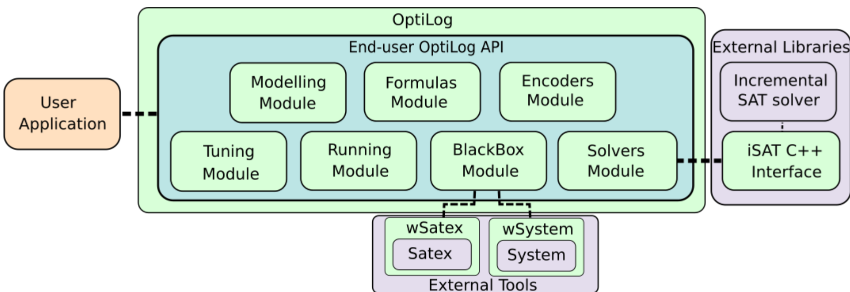


Figure 1 OptiLog's architecture.

3 Modelling Module

This module provides a rich and compact formalism to model problems. In particular, this module allows modelling problems with non-CNF Boolean and Pseudo Boolean expressions that can be automatically transformed into the SAT formula provided by the *Formulas* module. The non-CNF expressions are translated into SAT using the Tseitin transformation. The Pseudo-Boolean expressions are normalized [1] translated into SAT with the additional use of the *Encoders* Module. The goal is to come up with a richer formalism, that frees the user from reimplementing typical transformations to SAT, yet close enough to the formalism accepted by the solvers in OptiLog. We think that OptiLog can be further used by other Tools with higher formalism such as Minizinc [34], i.e, sort of *OptiLog FlatZinc*.

```
 1 a = Bool('a')
 2 b = Bool('b')
 3 c = Bool('c')
 4 e1 = ~a + ~b + ~c < 2
 5 e2 = ~(a & b & c)
 6 e3 = e1 & e2
 7 e4 = If(a, b ^ c)
 8 p1 = Problem(e1, name='p1')
 9 p2 = Problem(e2, name='p2')
10 p3 = Problem(e3, name='p3')
11 p4 = Problem(e4, name='p4')
12 t = TruthTable(p1, p2, p3, p4)
13 t.print()
```

Figure 2 Basic example of a problem definition.

a	b	c	p1	p2	p3	p4
0	0	0	0	1	0	1
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	1	1	1	1
1	0	0	0	1	0	0
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	0	0	0

Figure 3 Truth table representation for *p1*, *p2*, *p3* and *p4*.

In Figure 2, we can see a little example of the new modelling formalism. We first define the Boolean variables that will appear in the formula (lines 1-3). These variables have to be labelled with an identifier.

Then, in line 4 we create our first expression to encode the constraint $\neg a + \neg b + \neg c < 2$. In general, we can directly use the Python logic operators $(\sim, \&, |, ^\wedge)$ and their counterparts (`Not`, `And`, `Or`, `Xor`) to create Boolean expressions and the Python arithmetic operators $(+, -, ^*, <, <=, >=, >, ==)$ to create Pseudo-Boolean expressions. We can also use the `If`,

`Iff` classes to create implications and double implications[1]. Lines 5 and 7 encode the constraints $\neg(a \wedge b \wedge c)$ and $a \rightarrow (b \oplus c)$ respectively, whereas in line 6 we encode the conjunction of expressions `e1` and `e2`.

Finally, in lines 8-11 we transform the created expressions to instances of the class *Problem*. A *Problem* represents the conjunction of a set of expressions. In this case, we add a single expression to each *Problem*, and we name each of the problems to reference them later.

In line 12, as an example of how this package can be used for also for educational purposes, we create the truth table for our four problems and we print them in line 13 producing the output shown in Figure 3. This is very useful not only to teach any introductory course on propositional logic but also to double-check some small formulas.

Lines 14-19 in Figure 4 show how we can use a SAT solver to obtain a solution for our problem. First of all, we need to translate our formula into CNF DIMACS format [16] which is the input format for SAT solvers. In line 14, we create an instance of the SAT solver *Glucose41*. Then, in line 16, we add the clauses forming our CNF formula to the SAT solver and execute the solver in line 17. If the input instance is satisfiable we can obtain a model and decode that model according to the labels of our variables. The resulting model is finally printed in line 19 obtaining the output: `P3 solution: [a, b, ~c]`.

```
13 (...)
14 s = Glucose41()
15 p3_cnf = p3.to_cnf_dimacs()
16 s.add_clauses(p3_cnf.clauses)
17 s.solve()
18 solution = cnf.decode_dimacs(s.model())
19 print('P3 solution:', solution)
```

Figure 4 Example on how to solve *p3* and extract its model.

Now, we can also query whether problem $p4$ is a logic consequence of $p3$ ($p3$ entails $p4$), i.e., $\neg a + \neg b + \neg c < 2, \neg(a \wedge b \wedge c) \models a \rightarrow (b \oplus c)$ which is equivalent to ask whether the conjunction of all the premises and the negation of the consequence, i.e., $(\neg a + \neg b + \neg c < 2) \wedge \neg(a \wedge b \wedge c) \wedge \neg(a \rightarrow (b \oplus c))$ is unsatisfiable. Figure 5 shows how to do it in OptiLog.

```
19 (...)
20 s = Glucose41()
21 p5_cnf = Problem(e3 & ~e4).to_cnf_dimacs()
22 s.add_clauses(p5_cnf.clauses)
23 print('Is p5 Satisfiable:', s.solve())
```

Figure 5 Logic consequence example.

Since the logic consequence is valid the SAT solver reports the formula is unsatisfiable: `Is p5 Satisfiable: False`.

In the OptiLog documentation[31], we can find a more complex application of OptiLog to model, solve, tune and run the SlitherLink problem[2]. There exist other modelling python libraries that can be used to model problems. For example, CPMpy [22] is a library that allows the representation of matrix-related constraints using Numpy arrays [24]. pyAiger is a circuit-oriented modelling library to model combinatorial circuits. Although it shares some similarities with OptiLog's *Modelling* module, it lacks some of its higher-level features such as Pseudo-Boolean expressions support. As future work, we will integrate these libraries into OptiLog to be used within the *Modelling* module.

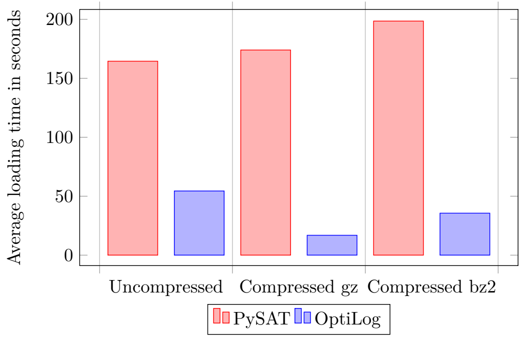
[1] These two classes do not naturally map to any Python operator.
[2] Also in the appendix attached to this submission.

4 Solvers Module

The solvers module has been completely redesigned with a more modular philosophy that allows dynamic solver loading and a more flexible compilation pipeline:

1. **Release of the iSAT interface:** Within this version the python framework OptiLog is able to use *External Libraries* provided they implement a given interface. In particular, OptiLog releases the iSAT C++ interface to the developer community [30] and welcomes solver developers to make their C++ SAT solvers compliant with this interface. Now, a SAT solver compiled with the interface as a shared object (.so file) can be dynamically loaded into OptiLog. At [30] GitHub repository several examples of solvers implementing the iSAT interface can be found.
2. **Dynamic Solver loading:** On import, OptiLog will automatically bind all its incorporated solvers and the user-provided ones. A user may include a new SAT solver by setting the `OPTILOG_SOLVERS` environment variable to the path where the .so files added by the user are located.
3. **Out of the box SAT solver integration:** OptiLog integrates several state-of-the-art SAT solvers as .so files that can be directly used in Python: Cadical 1.0.2 and Cadical 1.5.2 [12], Glucose 4.1 and Glucose 3.0 [7], Picosat [10], Minisat [17] and Lingeling 18 [11]. Unlike in PySAT these solvers can be also parameterized from OptiLog.
4. **Fast and Flexible Development:** This new way of managing External Libraries allows fast integration of the solvers by decoupling the solvers from the framework itself. Moreover, now solver developers are in control of the compilation pipeline, which allows them to link or include external libraries and modules. On the other hand, PySAT requires solver developers to implement their interface with a full binding through Python's C API and a corresponding Python wrapper class, forcing developers to have extensive knowledge of Python's C API. OptiLog's approach is simpler and less error-prone by removing code repetition and does not require solver developers to have Python knowledge.
   Since the last release, we have also expanded the capabilities of the interface. Now with automatic support for bz2 compression, solver cloning, SIGINT handling, and multithread support:
5. **Signal handling and multithread support:** OptiLog is able to handle SIGINT signals while providing multithread support by releasing the Global Interpreter Lock (GIL) before calling `solve` or `propagate`, while PySAT cannot handle signals when multithread support is enabled. This may be a bottleneck for complex SAT-systems where one thread is executing a SAT solver and other threads are dedicated to other tasks. In this case, PySAT can not manage signals sent to the SAT solver.
   Moreover, OptiLog handles SIGINT signals gracefully, allowing multiple SIGINT signals to be caught and handled, meanwhile, PySAT implementation blocks and does not throw exceptions after the first signal.
6. **Support for Solver cloning:** We have added support for solver cloning. Solver cloning allows solvers to copy their current state and create a new solver object that is equivalent to the original. Cloning can involve replicating the internal state of all the data structures of the solver. The goal is that we can guarantee that the search in the new solver will evolve exactly as in the original solver at the point where the cloning was performed.
7. **Efficient formula loading:** In contrast with PySAT, our formula loading methods are implemented directly in C++, which allows very efficient loading. These formulas may be loaded into Formula objects or directly into the solvers, avoiding the inefficient

representation in Python. OptiLog provides support for .cnf and .wcnf files that may be compressed with gz or bz2. Here we can find a comparison in runtime speed between PySAT and OptiLog. The benchmark instantiates a Glucose 4.1 solver and loads the hard clauses of every WCNF instance in the Incomplete Weighted track of the MaxSat Evaluation 2021. The graph below shows the mean of the loading times of the 33 uncompressed instances that took more than 30 seconds to load on PySAT. OptiLog was able to load all the instances with less than 12GB of RAM, while PySAT required 36GB. Their average size of the instances in number of clauses is 34.5M, while their average size in MB for uncompressed, compressed in .gz, and in .bz2 is 1055.5 MB, 176.37 MB, and 102.7 MB respectively. Additionally, the largest instance that we used had 132.7M clauses, and a size of 3.7GB, 501MB, 200MB for its uncompressed version, compressed with .gz, and .bz2 respectively.



5 BlackBox Module

Executing *External tools* such as: generators, preprocessors, feature extractors, binaries of other solvers, or any other system command, usually becomes a very ad-hoc and contrived procedure. These are the main contributions:

1. **Execution and configuration of External Tools:** We have developed the *BlackBox* module that allows the execution and parsing of arbitrary programs while running in a memory and time-constrained environment with the help of the runsolver tool [39]. These black boxes are also integrated with the *Tuning* module so that they can be automatically configured (see section 6). We also provide utilities to parse the output of the execution of these programs with regular expressions.
2. **Integration with Satex:** We have further merged the *Blackbox* module with SAT Heritage [6]. This allows OptiLog to run any SAT solver developed in SAT Competitions [9] since 2000. We are in the process of collecting the configurable parameters for these solvers to enable the tuning of them. We have integrated the wrapper wSatex (see Figure 1) that acts as a bridge for the SAT Heritage Docker Images. With this wrapper, OptiLog can call solvers from all the SAT competitions and automatically parse their output. Solvers can be called with .cnf files or by using the CNF formula, which will transparently use memory files that do not write to disk for a more efficient resource usage.

6 Tuning Module

This module allows the user to define a configuration scenario to tune a given application (solver, algorithm, function, blackbox, etc). With this scenario OptiLog automatically generates all the files and resources to run independently an automatic configurator (tuner) [3] (see example in the supplementary material). These are the new features we have added:

1. **Automatic deployment of the winning configuration**. We have also simplified the user interaction with the tuning process. Now, the winning configuration reported by the tuner can be automatically recovered. The user can now get a new instance of the application properly set to the wining configuration and ready to be executed.
2. **Automatic configuration of BlackBoxes.** Thanks to the addition of the *BlackBox* module, we now allow to automatically configure any of the executables that the *BlackBox* module can handle (see section 5). We follow the same interface that we use to configure Python functions and iSAT solvers, as it can be seen in Figure 6. This addition opens the door to use OptiLog to ease the configuration process of any *External tool* and be applied in other research communities, education programs, or industrial environments.

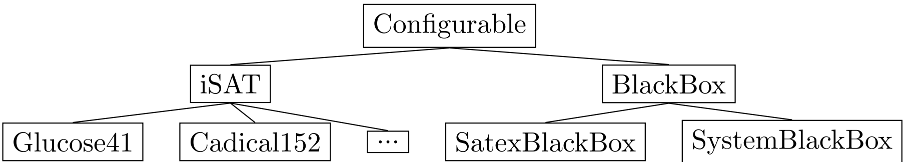


Figure 6 Diagram of the *Configurable* classes in OptiLog.

As we can see in the example on the Slitherlink problem in the documentation and the supplementary material, by applying the tuner GGA[3] we are able to solve 25% more instances and we decrease the PAR10 metric by 66%.

7 Running Module

To evaluate the performance of an application (e.g. SAT-based system) we typically evaluate it on test data-sets (e.g. set of SAT instances) and analyze the results according to some metric. In this context, a task is the evaluation of the application on a particular data-set.

In this work, we have also extended OptiLog to tackle this tedious and error-prone step by providing an automatic procedure. This procedure submits all these tasks to (potentially) any execution environment, collects and parses the output of these tasks, and aggregates the results. These are the main features of the new *Running* module:

1. **Execution scenario.** An execution scenario is defined by: (i) the application(s), (ii) the data-sets (e.g. SAT instances) that will be used, (iii) the execution limits (CPU, memory…), and (iv) a submitter script.
2. **Running the scenario.** In order to execute the scenario, OptiLog will call a *submitter script* supplied by the user. OptiLog provides example scripts for executing locally (using Task Spooler [29]) and in a High Performance Cluster (using Sun Grid Engine [33]), and potentially on the cloud (currently under development). Note that OptiLog remains agnostic against execution platforms, thus allowing the user to provide ad-hoc scripts for custom execution environments without hassle. Upon execution, the logs for each task will be stored in the scenario directory.

[3] We currently support GGA [3] and SMAC [28].

3. **Parsing and aggregating results.** Optilog also provides tools to extract information of the logs, once the experiment has finished. It will read the raw output of each task and will parse the information that the user specifies using *filters*. The information is then presented to the user as a *Pandas* [41] dataframe. We decided to use this data structure as, over the years, it has become the *de facto* standard by data scientists. Its flexibility, as well as the large support from third-party tools (such as visualization tools), allows the user to extract insights from the results of the experiment painlessly.
   A *filter* is defined by:
   - The regular expression for the value to be extracted.
   - If we want to retrieve all the matches of the regular expression.
   - If we want to store the timestamp from when the value was reported.
   - A name for the value. This will correspond to a column in the dataframe.

   In particular, for SAT-based solvers that follow a standardized output format, we provide some templates. Currently, we support SAT solvers and MaxSAT solvers that conform to the DIMACS output format used by the SAT competition [9] and MaxSAT evaluation [8]. For example, the SAT template comes with filters to detect the satisfiability of a formula and its model. The MaxSAT filter also detects the cost.

Competition organizers, research groups, etc. have already their own tools, less or more automatized, to carry out their experiments. This is a very time-consuming part, error-prone of the research process and a non-negligible task of software engineering. Even in the same research group or community different individuals reinvent the wheel systematically when it has to do with managing experiments.

OptiLog aims to provide a common baseline so that we can all build on top and mitigate the impact of this step in the development process. Moreover, having this common base we enforce the reproducibility and trust on experimental results.

8 Conclusions and Future Work

We have presented new extensions for the framework OptiLog which opens a new range of applications. These new applications range from supporting researchers, educators, and practitioners to create more ambitious end-to-end applications cases where SAT plays a key role. OptiLog allows to focus our energy on modelling and solving problem issues while still being able to carry out comprehensive experimental studies involving also tuning steps.

As future work, we plan to enrich the OptiLog framework. From a more technical perspective, we plan to extend it with support for distributed algorithms (including cloud computing) and a new module for metaheuristic algorithms. From a more educational point of view, we will generate a database of course assignments with instructions for educators and students including autograders.

References

1. Amir Aavani. Translating pseudo-boolean constraints into cnf. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 357–359, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
2. Carlos Ansótegui, Jesus Ojeda, António Pacheco, Josep Pon, Josep M. Salvia, and Eduard Torres. Optilog: A framework for sat-based systems. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2021. doi:10.1007/978-3-030-80223-3_1.

3 Carlos Ansótegui, Josep Pon, and Meinolf Sellmann. Boosting evolutionary algorithm configuration. *Annals of Mathematics and Artificial Intelligence*, 2021. doi:10.1007/s10472-020-09726-y.

4 Carlos Ansótegui, Josep Pon, Meinolf Sellmann, and Kevin Tierney. Pydgga: Distributed gga for automatic configuration. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 11–20, Cham, 2021. Springer International Publishing.

5 Gilles Audemard, Loïc Paulevé, and Laurent Simon. SAT heritage: A community-driven effort for archiving, building and running more than thousand SAT solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 107–113. Springer, 2020.

6 Gilles Audemard, Loïc Paulevé, and Laurent Simon. Sat heritage: A community-driven effort for archiving, building and running more than thousand sat solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 107–113, Cham, 2020. Springer International Publishing.

7 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

8 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2021.

9 Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2021.

10 Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

11 Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.

12 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

13 Francois Chollet et al. Keras, 2015. URL: https://github.com/fchollet/keras.

14 COIN-OR Foundation. Computational infrastructure for operations research. https://www.coin-or.org/, 2016.

15 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

16 dimacs.rutgers.edu. Dimacs cnf suggested format, 2021. URL: http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps.

17 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

18 Python Software Fundation. Python package index - pypi. URL: https://pypi.org/.

19 Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin, March 2020.

20 Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*, 2015.

21 Google. Google OR-Tools. `https://developers.google.com/optimization`, 2021.

22 T. Guns. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *The 18th workshop on Constraint Modelling and Reformulation at CP (ModRef 2019)*, 2019. URL: `https://github.com/CPMpy/cpmpy`.

23 Gurobi Optimization. Gurobi. `https://www.gurobi.com/`, 2021.

24 Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. `doi:10.1038/s41586-020-2649-2`.

25 IBM. IBM ILOG CPLEX. `https://www.ibm.com/products/ilog-cplex-optimization-studio`, 2021.

26 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

27 Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. Cnfgen: A generator of crafted benchmarks. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2017.

28 Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization, 2021. `arXiv:2109.09831`.

29 Lluis Batlle i Rossell. Task spooler man page, 2021. URL: `http://manpages.ubuntu.com/manpages/xenial/man1/tsp.1.html`.

30 Logic and Optimization Group. OptiLog C++ Interface for Python bindings. `https://github.com/optilog/OptiLogFrameworkInterface`. Accessed: 2022-02-26.

31 Logic and Optimization Group. OptiLog official documentation, 2022. URL: `http://ulog.udl.cat/static/doc/optilog/html/index.html`.

32 Logic Optimization Group. PyPBLib: PBLib Python3 bindings. `https://pypi.org/project/pypblib/`, 2018. Described in OptiLog [2].

33 W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, page 35, USA, 2001. IEEE Computer Society.

34 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

35 Nikoli. Nikoli's slitherlink webpage, 2021. URL: `https://www.nikoli.co.jp/en/puzzles/slitherlink.html`.

36 The pandas development team. pandas-dev/pandas: Pandas, February 2020. `doi:10.5281/zenodo.3509134`.

37 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

38 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

39 Olivier Roussel. Controlling a solver execution: the runsolver tool. *JSAT*, 7:139–144, November 2011. doi:10.3233/SAT190083.

40 Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

41 Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010. doi:10.25080/Majora-92bf1922-00a.

42 Takayuki Yato. On the np-completeness of the slither link puzzle. *IPSJ SIGNotes ALgorithms*, 74:25–32, 2000.

A Preliminaries

▶ **Definition 1.** *A literal is a propositional variable x or a negated propositional variable $\neg x$. A clause is a disjunction of literals. A formula in Conjunctive Normal Form (CNF) is a conjunction of clauses.*

▶ **Definition 2.** *A truth assignment for an instance ϕ is a mapping that assigns to each propositional variable in ϕ either 0 (False) or 1 (True). A truth assignment is* partial *if the mapping is not defined for all the propositional variables in ϕ.*

▶ **Definition 3.** *A truth assignment I satisfies a literal x ($\neg x$) if I maps x to 1 (0); otherwise, it is falsified. A truth assignment I satisfies a clause if I satisfies at least one of its literals; otherwise, it is violated or falsified. A truth assignment that satisfies all the clauses of a CNF formula is a model.*

▶ **Definition 4.** *The SAT problem asks whether there exists a model for a CNF formula. If that is the case, the formula is said to be satisfiable, otherwise it is unsatisfiable.*

▶ **Definition 5.** *An unsatisfiable core is a subset of clauses of a SAT instance that is unsatisfiable.*

▶ **Definition 6.** *Let A and B be SAT instances. $A \models B$ denotes that A entails B, i.e. all assignments satisfying A also satisfy B. It holds that $A \models B$ iff $A \wedge \neg B$ is unsatisfiable.*

▶ **Definition 7.** *A* pseudo-Boolean *(PB) constraint is a Boolean function of the form $\sum_{i=1}^{n} q_i l_i \diamond k$, where k and the q_i are integer constants, l_i are literals, and $\diamond \in \{<, \leq, =, \geq, >\}$. A Cardinality (Card) constraint is a PB constraint where all q_i are equal to 1.*

B Example: Modelling, Solving, Tuning and Running the Slitherlink Problem using OptiLog

In these sections, we show how to model, solve, tune and run a concrete problem using OptiLog. We focus on the Slitherlink problem, originally invented by Nikoli [35] which was shown to be NP-Compete in [42]. We assume that OptiLog is installed, and *runsolver* [39] exists in the `PATH`. For more implementation details see the official documentation [31].

Modelling the Slitherlink problem

In the Slitherlink problem, we are given an $n \times m$ grid. A cell in the grid can be empty or contain a number between 0 and 3. Each cell has 4 associated edges (its borders). The goal is to select a set of edges among all cells such that: (1) if a cell has a number k, then k of its edges have to be selected; and (2) the selected edges form exactly one cycle that does not cross itself.

Figure 7 Problem representation (left) and solution (right).

In Figure 7 we can see an example of the Slitherlink problem and its only correct solution. Figure 8 shows the code needed to model the problem into a SAT instance. We generate a *Problem* (line 2) instance where the constraints are added (lines 7, 13). Then, we convert this *Problem* to a *CNF* formula (line 14). Notice that this model is not taking into account the fact that there has to be exactly one cycle.

```
1  def encode_slitherlink(sl):
2    p = Problem()
3    # Vertex Constraints: ensure the path is contiguous
4    for i in range(sl.m + 1):
5      for j in range(sl.n + 1):
6        edges = sl.vertex_edges(i, j)  # edges that intersect at vertex i,j
7        p.add_constr((Add(edges) == 0) | (Add(edges) == 2))
8    # Cell Constraints
9    for j, row in enumerate(sl.cells):
10     for i, cell in enumerate(row):
11       if cell == None: continue
12       edges = sl.cell_edges(i, j)
13       p.add_constr(Add(edges) == cell)
14   return p.to_cnf_dimacs()
```

Figure 8 Encoding to SAT for the Slitherlink problem.

Solving the Slitherlink problem

In this section, we describe an incremental SAT-based solving approach (implemented in function *solve_slitherlink* of Figure 9) for the Slitherlink problem. We use the encoding described in the previous section to obtain a solution to the CNF formula generated in line 3 that guarantees that for each cell exactly the amount of edges described by the number associated with the cell are selected and they form a continuous path.

```
1  def solve_slitherlink(instance, seed):
2    sl = SlitherLink(instance)
3    cnf = encode_slitherlink(sl)
4    s = Cadical()
5    s.set('seed', seed)
6    s.add_clauses(cnf.clauses)
7    while s.solve() is True:
8      n_cycles = sl.manage_cycles(s, cnf)
9      if n_cycles > 1: continue
10     print('s YES', flush=True)
11     return cnf.decode_dimacs(s.model())
12   print('s NO', flush=True)
13
14 def manage_cycles(self, solver, cnf):
15   model = solver.model()
16   cycles = self.find_cycles(cnf.decode_dimacs(model))
17   if len(cycles) > 1:
18     for cycle in cycles:
19       clause = [~edge for edge in cycle]
20       solver.add_clause(cnf.to_dimacs(clause))
21   return len(cycles)
```

Figure 9 Incremental SAT-based approach to solve the Slitherlink problem.

Lines 4 to 6 instantiate the SAT solver with a seed, where the clauses generated in Figure 8 are added. Then, we iteratively query the SAT solver (line 7) to provide a solution (a model). In line 8, we call *manage_cycles* that checks the number of cycles reported by the SAT solver solution and adds to the SAT solver the clauses that forbid these cycles in the solution. Then, if only one cycle was found we are done and we have found a solution. We return the solution once decoded the model provided by the SAT solver (line 11). Otherwise, we will exit the main loop (line 7) if there is not a solution with just one cycle and we report the problem has no solution.

Tuning with OptiLog

Now that we have concluded the *modelling* and *solving* part for the Slitherlink problem, we can try to improve its performance by *tuning* our algorithm. In particular, we found that the Cadical SAT solver has a total of 146 discrete finite domain parameters that would be of interest to configure. In order to do so, we will use OptiLog's *Tuning* module.

The first thing we need to do is to update the `solve_slitherlink` function to receive a Cadical SAT solver with its parameters already set, which we achieve by using *CfgCls* from the *Tuning* module, as shown in line 2 of Figure 10.

```
1 @ac
2 def solve_slitherlink(instance, seed, Solver: CfgCls(Cadical)):
3     sl = SlitherLink(instance)
4     cnf = encode_slitherlink(sl)
5     s = Solver()
6     s.set("seed", seed)
7     s.add_clauses(cnf.clauses)
8     (...)
```

Figure 10 Modifications in *solve_slitherlink* to configure the Cadical SAT solver.

Introducing this change to the function signature does not produce any change to how it was called before[4], so there is no need to modify any of the calls to `solve_slitherlink`.

Then, we can proceed to create an automatic configuration scenario as shown in Figure 11. In this example, we will use the `GGAConfigurator` class to generate the scenario files for PyDGGA [3, 4]. The following configuration describes a GGA scenario with a PAR10 score[5], a memory and time limits for execution of 6GB and 300s respectively, and a total configuration time limit of 4 hours. The configurator will be trained on a set of 100 random instances different to the ones used to evaluate its performance. Finally, we generate the scenario at the directory `gga_scenario`.

```
1  from optilog.tuning.configurators import GGAConfigurator
2  from optilog.blackbox import ExecutionConstraints
3  from slitherlink import solve_slitherlink
4
5  time_limit = 300
6  configurator = GGAConfigurator(
7      solve_slitherlink, global_cfgcalls=[solve_slitherlink],
8      input_data="instances/training/*.txt", run_obj="runtime",
9      data_kwarg="instance", seed_kwarg="seed", seed=1,
10     cost_min=0, cost_max=10 * time_limit,
11     tuner_rt_limit=60 * 60 * 4,
12     constraints=ExecutionConstraints(memory=f'6G', wall_time=time_limit),
13 )
14 configurator.generate_scenario("./gga_scenario")
```

Figure 11 Script to generate the *tuning* scenario for GGA.

We configured Cadical with PyDGGA 1.5.8 on a computer cluster with Intel Xeon Silver 4110 CPUs at 2.1GHz cores with 4 parallel processes each. Once the automatic configuration process has finished, we can automatically extract the best configuration found by GGA by using the `GGAParser` functionality of OptiLog's *Tuner* module, as shown in Figure 12.

```
1 from optilog.tuning.configurators.utils import GGAParser
2
3 parser = GGAParser("<GGA output log>")
4 parser.save_configs("./configs", "./gga_scenario", name="best-conf")
```

Figure 12 Script to parse the best configuration found by the GGA tuner.

This function extracts the best configuration found by GGA and creates the executable `./configs/best-conf.sh` with the function `solve_slitherlink` properly set to the best configuration. With this final step we have all the required pieces to evaluate and compare the final performance of our approach by using the OptiLog's *Running* module.

Running Experiments

We are interested on comparing the performance of the default configuration with the *tuned* configuration of our `solve_slitherlink` function. To achieve that we will create an execution scenario by using OptiLog's *Running* module, as shown in Figure 13.

First, we describe the settings of our scenario. In particular, we will run the *default* configuration by executing the `slitherlink.py` file, and the *tuned* configuration found by GGA (with the wrapper generated by the *Tuning* module). The problem instances are

[4] Although we added a new parameter, *CfgCls* will automatically add a default value.

[5] PAR10 score for a given time limit T is a hybrid measure, defined as the average of the runtimes for solved instances and of $10 \times T$ for unsolved instances.

```
1  from optilog.running import SolverRunner
2  from optilog.blackbox import ExecutionConstraints
3
4  runner = SolverRunner(
5      solvers={"default": "slitherlink.py", "gga": "./configs/best-conf.sh"},
6      tasks="./instances/test/*.txt", scenario_dir=f"./default_running",
7      submit_file="./enque.sh",
8      constraints=ExecutionConstraints(memory=f'6G', wall_time=300),
9      slots=1, seeds=[1, 2, 3], unbuffer=False, runsolver=False,
10 )
11 runner.generate_scenario()
```

Figure 13 Execution scenario for the Slitherlink problem.

located by expanding the glob `./instances/test/*.txt`. Other execution constraints such as the time and memory limits can be set, as well as the number of `slots`. A list of seeds is provided to the `seeds` parameter such that each experiment will be executed for each of the seeds.

OptiLog provides a computing-agnostic running environment. The `submit_file` parameter points to the script that launches each task[6] to the computing backend.

Finally, the method `generate_scenario()` in line 11 of Figure 13 creates an scenario directory containing all the necessary settings to run the experiments (by default it will create a directory called `default_running`). Then, the user can easily run these experiments by using the `optilog-scenario` command provided by OptiLog. Unless a specific directory for storing the logs is indicated using the `logs` parameter, the directory `./default_running/logs` will be automatically created.

Processing Experimental Results

We include in OptiLog a new functionality within the *Running* module to automatically process the logs of the experiments. Figure 14 shows the code to parse the logs for the Slitherlink problem and its output.

```
1  >>> from optilog.running import ParsingInfo, parse_scenario
2
3  >>> pi = ParsingInfo()
4  >>> pi.add_filter("res", r"^s (.+)", timestamp=True)
5  >>> df = parse_scenario("./default_running", parsing_info=pi)
6
7  >>> def solved(col): return (col == "YES").sum() / col.shape[0]
8  >>> def parK(col, k=10): return col.fillna(1000 * 300 * k).mean()
9
10 >>> def stats(df, solver):
11 ...     print("* Pctg solved:", df[(solver, "res")].agg(solved))
12 ...     print("* PAR10:", df[(solver, "time_res")].agg(parK), end="\n\n")
13
14 >>> stats(df, "default")
15 * Pctg solved: 0.51
16 * PAR10: 1541310.3933333333
17 >>> stats(df, "gga")
18 * Pctg solved: 0.87
19 * PAR10: 520862.7966666667
```

Figure 14 Log processing for the Slitherlink experiment in Figure 13.

[6] We provide some example scripts for submitting jobs on OptiLog's official documentation [31].

The experiments are parsed line by line following a set of filters that are described with a `ParsingInfo` object. A `ParsingInfo` object is instantiated (line 3), where we add filters (line 4) to parse the ouptut (specifically `'s YES'` or `'s NO'` lines), and also records the time at which the result is reported in milliseconds. The `parse_scenario` function (line 5) parses the result of the experiments and returns a Pandas [36] dataframe with the parsed data.

Lines 7-12 process the experiment results by using some basic Pandas functions. Finally, lines 14-19 show the final results of our experiments. In particular, we found that the automatic configuration of the `solve_slitherlink` function allows us to solve about 25% more instances than the default configuration, and improves the PAR10 score by about 66%.