OptiLog for Education

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 propose the integration of the OptiLog Python framework into undergraduate courses, mainly on courses that make use of SATisfiability-based applications, but also in courses where benchmarking and experimentation are relevant. We show a brief overview of the framework's features, and develop examples of cases where OptiLog would be suitable in educational environments. The student will find support to model problems, set up execution environments, and process the results in a friendly way. All the lessons learnt from the usage of OptiLog can be directly applied to solve industrial problems.

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

Keywords and phrases Constraint Programming, Satisfiability, Educational tools

Digital Object Identifier 10.4230/LIPIcs.WTCP.2023.6

1 Introduction

Combinatorial Optimization (CO) problems arise in many scientific and engineering disciplines since they tackle a very general and practical question, i.e., which is the optimal object from a finite set of objects. Therefore, it is natural that CO tools are used in many undergraduate courses.

In this paper, we focus on CO tools that use the power of SATisfiability technology. SAT technology [10] provides a highly competitive generic problem approach for solving a great variety of problems. In particular, the SAT problem is an NP-Complete problem which asks to determine whether there is an assignment to the Boolean variables in a propositional formula in Conjunctive Normal Form (CNF) (set of clauses) that *satisfies* the formula.

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 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 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 [36], thanks to its simplicity, has dramatically turned the situation around, becoming the middleware to interconnect many scientific libraries through Python bindings such as Numpy [22], Pandas [37], scikit-learn [33], Pytorch [32],

Workshop on Teaching Constraint Programming, WTCP 2023.
Editors: Tejas Santanam and Helmut Simonis; Article No. 6; pp. 6:1–6:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keras [11], etc. This interconnection has definitely allowed affording developing more complex applications and indirectly justifies further the individual utility of each library.

In Constraint Programming we also find several Python applications or bindings such as CPLEX [23], Gurobi [21], OR-Tools [20], COIN-OR [12], SCIP [19], Z3 [13], *cnfgen* [25], PySAT [24], PyPbLib [28], SAT Heritage [5], OptiLog [2, 1], etc.

In this paper, we present how the OptiLog Python framework can be used to introduce students in the CO field using a high-level programming language (i.e. Python), reducing the cognitive overhead that derives from the heterogeneous environment that is the SAT-related tools (solvers, encoders, modellers, etc.).

A typical issue when dealing with any CO tool is to effectively conduct comprehensive experimentation. This inherently adds overhead to any project including small projects coming from course assignments. In this sense, OptiLog provides the *Experiment* module that automatically manages several low-level details involved in any project. Launching experiments, parsing logs, and producing reports should not become a bottleneck issue in the project.

In summary, we can conclude that OptiLog, becomes a very accessible and friendly tool to support students on projects making use of SATisfiability technology while keeping all the power to develop industrial applications. The student is not *playing* anymore with a toy tool but with a powerful hammer to smash CO problems, yet light enough to be handled in an undergraduate course.

The paper is structured as follows: in Section 3 we present the general architecture of the OptiLog framework. In section 4, we present a guiding example on how to use the Modelling module. In particular, Sections 5 and Sections 6 show how the Sudoku and the Slitherlink problems, respectively, can be defined and solved using OptiLog. We also show in Section 7 the application of automatic configurators. Then we present the Experiment module and how experiments are conducted within OptiLog (Section 8), as well as how to process its results to produce meaningful data (Section 8.1). Finally, we end with Section 9 with some closing thoughts on the impact on the application of OptiLog in real courses, and with Section 10 providing future work.

2 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 unsatisfiabile.*

▶ **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* $A \land \neg B$ *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.*

3 OptiLog Framework Architecture

OptiLog [2, 1] is a Python library for rapid prototyping of SAT-based systems. OptiLog provides seven main modules for its end-user API: The *Formulas* module, the *Modelling* module, the *Encoders* module, the *Solvers* module, the *Tuning* module, the *Running* module, and the *BlackBox* module. Figure 1 shows the architecture of OptiLog, more information on the current architecture be found in the OptiLog manual [27].

In this paper, we focus on the usage of those modules in education, in particular, the *Modelling* module to define higher-level modelling features and the *Experiment* module that simplifies the execution of experiments and their analysis.

In the following sections, we will briefly describe each of OptiLog's main modules.

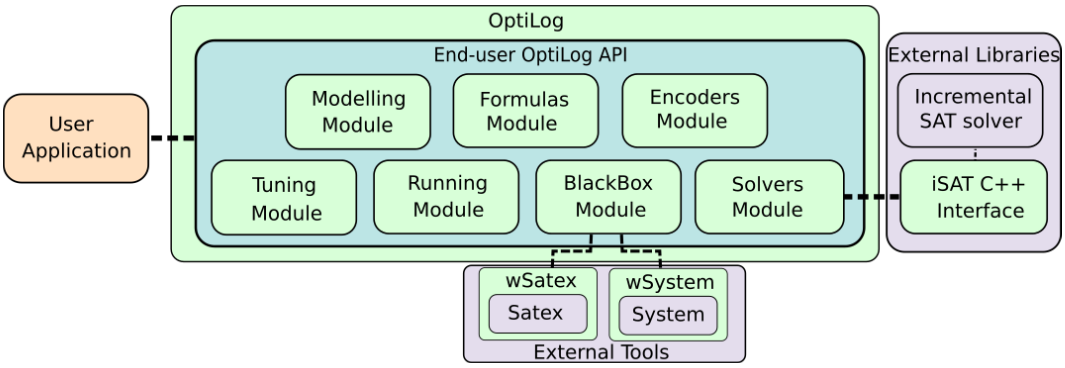

Figure 1 OptiLog's architecture.

3.1 Formulas Module

The *Formula* module allows the load and manipulation of several types of boolean formulas. In particular, it supports *CNF* for the typical Conjunctive Normal Form and *WCNF* formulas for the Weighed CNF version (see Definition 1).

3.2 Modelling module

The *Modelling* module allows for representing 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[35], while the Pseudo-Boolean relies on the Encoders module.

Additionally, this module allows the representation of the truth table of a formula (see Section 4) and the evaluation of each expression given a (partial) assignment, features that are interesting when teaching propositional logic concepts.

3.3 Encoders Module

Modelling problems into SAT usually involves the codification of Pseudo-Boolean (PB) constraints (see Definition 7). OptiLog provides access to several PB encoders that can efficiently translate these kinds of constraints into a CNF formula.

3.4 Solvers Module

OptiLog integrates several state-of-the-art SAT solvers that can be directly used in Python: Cadical [9], Glucose 4.1 and Glucose 3.0 [6], Picosat [7], Minisat [17] and Lingeling 18 [8].

Additionally, OptiLog uses the *iSAT C++* interface, which extends the basic SAT solving interface (add clauses, solve a formula, retrieve its model/unsatisfiable core) with other useful methods, such as setting and getting solver's parameters, setting and unsetting decision variables or obtaining learnt clauses from the solver.

3.4.0.1 The iSAT C++ Interface

allows to use any C/C++ SAT solver to the library by implementing the *iSAT C++* interface (for more details see OptiLog's official documentation [27]). OptiLog also provides a Plug&Play system for solvers that implement such interface, allowing the users to add their solvers without recompiling the entire OptiLog library.

3.5 Tuning Module

SAT solvers (as well as SAT-based systems) usually expose several configurable parameters that can potentially affect the system's performance, and whose value may not be known *a priori*. Automatic Configuration (AC) tools search for a proper setting of these configurable parameters by optimizing some objective function (e.g. run time) on a set of instances. The *Tuning* module abstracts the creation of the files required by different AC tools. This is ideal as an introduction to AC tools for students with no prior experience in the field.

3.6 Running Module

A common task that is performed to evaluate the performance of a SAT-based system is its execution over a set of instances. This can be tedious and error-prone work, especially if we have to do it manually. The *Running* module provides an automatic procedure to submit all these tasks to (potentially) any execution environment, as shown in Section 8.

3.7 BlackBox module

Some third-party tools are not directly integrable in a Python application (no bindings available, only the binary is available...). For such tools, OptiLog provides the *BlackBox* module, that allows the execution of arbitrary programs. It also allows to define limits for those executions (memory, CPU time...). This module avoids unnecessary boilerplate and lets users and students focus on critical code.

4 Defining Problems

In this section, we present how we can use Non-CNF Boolean formulas augmented with PB constraints to encode problems.

```
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()
```

Listing 1 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  |
(...)
```

Listing 2 Truth table representation for *p1*, *p2*, *p3* and *p4*

As we can see in Listing 1, we first define the Boolean variables that will appear in the formula (lines 1-3). These variables have to be labeled with an identifier.

Then, in line 4 we create our first expression to encode the constraint $\neg a + \neg b + \neg c < 2$. Notice we can directly use the Python operators $(\sim, \&, |, ^\wedge, +, -, ^*, <, <=, >=, >, ==)$ to create a logical expression. 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 we create the truth table for our four problems and we print them in line 13 producing the output shown in Listing 2.

Listing 3 shows 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 [15] which is the input format for SAT solvers (line 14). In line 15, 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 cnf3 = p3.to_cnf_dimacs()
15 s = Glucose41()
16 s.add_clauses(cnf3.clauses)
17 s.solve()
18 solution = cnf3.decode_dimacs(s.model())
19 print('P3 solution:', solution)
```

Listing 3 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 $p4$),

```
(...)
s = Glucose41()
cnf5 = Problem(e3 & ~e4).to_cnf_dimacs()
s.add_clauses(cnf5.clauses)
print('Is p5 Satisfiable:', s.solve())
```

Listing 4 Logic consequence example.

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. The code in Listing 4 shows how to do it in OptiLog.

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

5 Modelling and solving the Sudoku problem

In this section, we present another well-known combinatorial problem: the Sudoku [18]. It consists of a grid that must be filled with numbers, according to some constraints. The classic version (9x9) divides the grid into (3x3 squared) subregions, and has the following constraints:

- All cells must have a number between 1 to 9.
- A number can only appear once in a column.
- A number can only appear once in a row.
- A number can only appear once in a subregion.

Other versions might specify additional constraints or subregions with a different shape.

Listing 5 shows how one can encode this constraints using OptiLog. The function `encode_sudoku` generates a `CNF` object with the constraints for the provided Sudoku. First, lines 10-13 encode the values that are known in the Sudoku (i.e. they are fixed). Lines 15-17 encode the constraint that each cell have assigned one value. Lines 19-22 encode the constraint that each value appears in a row, and similarly the constraint that each value appears in a column would be implemented in line 25. Finally, the constraint that each value appears once in a subregion is encoded in lines 27-30.

Despite being a simple encoding, composed mostly of At-Most-One constraints, the students must reason about which cells must be grouped together for those restrictions (implement `iter_rows`, `iter_cols`, `iter_subregions`), and more complex restrictions could be added in harder variants of the Sudoku problem.

To find a solution (if it exists) on the Sudoku, the clauses in the `CNF` object returned by the encoding function can be fed to a SAT solver using OptiLog, as seen in Listing 6. If it has a solution, `sol` (line 11) will be a list containing `Bool` (`Not`) objects if the corresponding variable was set to true (false).

6 Modelling and solving the Slitherlink problem

In this section, we show how to model a concrete problem in OptiLog. We focus on the Slitherlink problem, originally invented by Nikoli [30] which was shown to be NP-Compete in [38]. In this 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:

```python
from itertools import product
from optilog.modelling import *

def var(j, i, v):
  return Bool(f'Cell_{j}_{i}_{v}')

def encode_sudoku(s):
  p = Problem()

  for r, c in product(range(s.n_rows), range(s.n_cols)):
    v = s.cells[r][c]
    if v is not None:
      p.add_constr(var(r, c, v)))

  for r, c in product(range(s.n_rows), range(s.n_cols):
    vals = [var(r, c, v) for v in range(s.n_vals)]
    p.add_constr(Add(vals) == 1)

  for cells in s.iter_rows():
    for v in range(s.n_vals):
      vals = [var(r, c, v) for (r, c) in cells]
      p.add_constr(Add(vals) == 1)

  for cells in s.iter_cols():
    (...)

  for cells in s.iter_subregions():
    for v in range(s.n_vals):
      vals = [var(r, c, v) for (r, c) in cells]
      p.add_constr(Add(vals) == 1)

  return p.to_cnf_dimacs()
```

Listing 5 Encoding of the classical Sudoku constraints

```python
from optilog.solvers.sat import *

cnf = encode_sudoku(sudoku)

s = Glucose41()
s.add_clauses(cnf.clauses)
has_solution = s.solve()
print('Has solution?', has_solution)

if has_solution:
    sol = cnf.decode_dimacs(s.model())
    visualize(sol, sudoku)
```

Listing 6 Solving the classical Sudoku

- If a cell has a number k, then k of its edges have to be selected.
- The selected edges form exactly one cycle that does not cross itself.

In Figure 2 we can see an example of the Slitherlink problem and its only correct solution. For more implementation details you can check OptiLog's documentation: `http://ulog.udl.cat/static/doc/optilog/html/optilog/use-cases/slitherlink.html`

6.1 Modelling the Slitherlink problem

First, we present how to encode the problem using OptiLog.

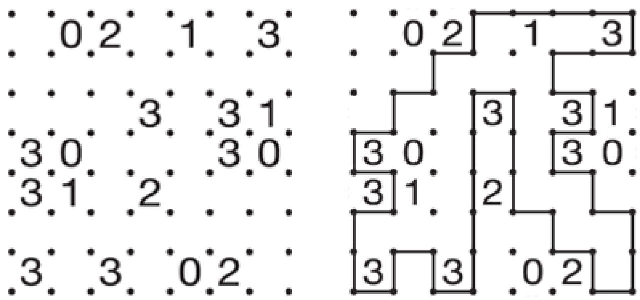

Figure 2 Problem representation (left) and solution (right)

Listing 7 shows the source code needed to model into SAT an instance of the problem. First, we generate an instance of the class *Problem* (line 2). Then, we encode the constraints of the problem. Lines 5-8 encode the vertex constraints that ensure that the path traced by the solution is contiguous. Line 7 calls the method `vertex_edges`, that returns a list of `Bool` objects representing edges that intersect at the vertex i, j. The encoded constraint is that a selected edge can be contiguous without crossing iff the number of selected edges that intersect at each vertex[1] is 0 or 2.

Lines 11-15 encode the cell constraint for each cell with a number. Line 14 calls the method `cell_edges`, which returns a list of `Bool` objects representing the edges that surround a cell. The added constraint imposes that the sum of incident edges is equal to the number in the cell. Then, we encode the problem to CNF DIMACS (line 17) and return the underlying CNF object.

```
1  def encode_slitherlink(sl):
2    p = Problem()
3
4    # Vertex Constraints
5    for i in range(sl.m + 1):
6      for j in range(sl.n + 1):
7        edges = sl.vertex_edges(i, j)
8        p.add_constr((Add(edges) == 0) | (Add(edges) == 2))
9
10   # Cell Constraints
11   for j, row in enumerate(sl.cells):
12     for i, cell in enumerate(row):
13       if cell is None: continue
14       edges = sl.cell_edges(i, j)
15       p.add_constr(Add(edges) == cell)
16
17   return p.to_cnf_dimacs()
```

Listing 7 Encoding to SAT for the Slitherlink problem

Notice that this model is not taking into account the fact that there has to be exactly one cycle.

[1] Computed by adding (`Add object`) all the edges that could intersect a vertex.

```
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)
```

Listing 8 Incremental SAT-based approach to solve the Slitherlink problem.

```
def manage_cycles(self, solver, cnf):
  model = solver.model()
  cycles = self.find_cycles(cnf.decode_dimacs(model))
  if len(cycles) > 1:
    for cycle in cycles:
      clause = [~edge for edge in cycle]
      solver.add_clause(cnf.to_dimacs(clause))
  return len(cycles)
```

Listing 9 Auxiliary function to manage cycles

6.2 Solving the Slitherlink problem

In this section, we describe an incremental SAT-based solving approach (implemented in function *solve_slitherlink* of Listing 8) 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 is selected and they form a contiguous path.

Lines 4 and 5 instantiate the Cadical SAT solver and initialize it with a seed for the random number generator of the solver, and in line 6 we add to the solver the clauses of the formula. Then, we iteratively query the SAT solver (line 7) to provide a solution (a model). Notice that we can use any of the incremental SAT solvers included in OptiLog instead of Cadical, or even add other external incremental SAT solvers through the `iSAT` interface.

In line 8, we call function *manage_cycles* that checks the solution reported by the SAT solver (defined in Listing 9). If there is more than one cycle it adds to the SAT solver the clauses that forbid these cycles in the solution. To find the cycles it uses the function *find_cycles*. To discard a cycle, it just adds to the SAT solver as a clause the negation of all the edges that conform to the cycle.

If only one cycle was found, then 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 no solution with just one cycle and we report the problem has no solution.

To test our approach we generated a set of 100 random instances of size 101 $\times$ 101 (the generator can be found here: `http://ulog.udl.cat/static/doc/optilog/html/optilog/use-cases/slitherlink.html`). When transformed to CNF, these instances have an average of 149937 boolean variables, 308721 clauses for the first encoded formula and 395253 for the last one. The instances that we solve require an average of 120 iterations.

As incremental SAT solver we used Cadical in its default configuration with a timeout of

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

Listing 10 Modifications in *solve_slitherlink* to configure the Cadical SAT solver

```python
from optilog.blackbox import ExecutionConstraints, RunSolver
from optilog.tuning.configurators import GGAScenario
from slitherlink import solve_slitherlink

if __name__ == "__main__":
  time_limit = 300
  configurator = GGAScenario(
    solve_slitherlink,
    input_data="instances/training/*.txt", run_obj="runtime",
    data_kwarg="instance", seed_kwarg="seed",
    seed=1, cost_min=0, cost_max=10 * time_limit,
    tuner_rt_limit=60 * 60 * 4, instances_min=10, instances_gen_max=-10,
    constraints=ExecutionConstraints(
      s_real_memory="6G", s_wall_time=time_limit, enforcer=RunSolver()
    ),
  )

  configurator.generate_scenario("./gga_scenario")
```

Listing 11 Script to generate the AC scenario for GGA

5 minutes. We were able to find a solution for 51% of instances.

7 Tuning the Slitherlink problem

Since we used the default configuration for the Cadical SAT solver in our experiments in section 6.2, it would also be of interest to automatically configure (tune) Cadical to find a solution for more instances within the same timelimit.

Cadical 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 constructor of an automatically configured SAT solver, as shown in line 2 of Listing 10.

Then, we can proceed to create an automatic configuration scenario as shown in Listing 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 runtime penalization and a time limit of 4 hours. The configurator will be trained on a new set of 100 instances generated with different seeds than those used to test our approach. Finally, we generate the scenario at the directory `gga_scenario`.

We configured Cadical with PyDGGA 1.6.0 on a computer cluster with Intel Xeon Silver 4110 CPUs at 2.1GHz cores with 4 parallel processes each. When the optimization was completed, we extracted the best configuration found by GGA for each solver and reexecuted the experiments on our original set of instances. In our analysis of the experimental results, thanks to the new configuration found by GGA, we solve 89% of the instances and we decrease the PAR10 metric by a factor of 4.45. The largest instance that we were able to solve had a size of 101 $\times$ 101.

8 Running the *Pac-Man* project

Setting up properly the experimentation environment required to evaluate a solving approach can result in a time-consuming task also source of bugs conducting to wrong evaluations. This increases the frustration of the student since it has to employ energy that otherwise he could invest in improving the solving approach.

OptiLog provides support in this sense, automating as much as possible some parts of the process. In this section, we present an example on how OptiLog can be used to evaluate the performance of different search algorithms implemented for *Pac-Man* [14].

The *Pac-Man* project provides a foundation where the students can implement different search algorithms and heuristics. For simplicity, we will focus on the basic search algorithms applied to the mazes (where the objective is to find the single food in the map), but this approach could be extended to all the problems presented in the framework. Using OptiLog, we can perform a batch execution of all the implementations with all the provided layouts (as well as other layouts generated randomly). The basic setup for the experiment is shown in Listing 12.

```
1  from optilog.running import RunningScenario
2  from optilog.blackbox import ExecutionConstraints, RunSolver
3
4  if __name__ == "__main__":
5    solvers = {
6      "bfs": "./wrappers/bfs.sh", "dfs": "./wrappers/dfs.sh", ... }
7    runner = RunningScenario(
8      solvers=solvers,
9      tasks="layouts/searchLayouts/*.lay",
10     submit_file="submit.sh", unbuffer=True,
11     constraints=ExecutionConstraints(
12       s_wall_time=300, s_real_memory="1G", enforcer=RunSolver()),
13   )
14   runner.generate_scenario(scenario_dir="./scenario")
```

Listing 12 Execution scenario for the *Pac-Man* project

First, we describe the settings of our scenario. We assume a wrapper has been provided that runs *Pac-Man* with the appropiate values to execute each algorithm (`bfs.sh`, `dfs.sh`...). We declare them as solvers (line 8), which will be run against the list of tasks (e.g. the layouts we want to solve) (line 9). It is also possible to specify other options such as the CPU time limit or the maximum memory available (`ExecutionConstraints`).

By default, OptiLog incorporates compatibility for two optional tools, `unbuffer` [16], to automatically flush to the log files and `runsolver` [34], to constraint the number of resources (time and memory) available to the process. In order to use these tools, they have to be available in the `PATH`.

OptiLog provides a backend-agnostic running environment. This means that the underlying tasks need to be delegated to a Job Scheduler like `SGE` [29] or `Task Spooler` [26] to get executed. The `submit_file` parameter points to the script in charge of submitting each task. In the example we assume `Task Spooler` in a local machine.

Finally, the method `generate_scenario()` in line 14 generates an scenario directory (`./scenario`) containing all the necessary files to run the experiments.

Then, the user can interact with this scenario directly from a terminal by launching a command of the form `optilog-running /path/to/scenario/ ACTION` where action can be `{list,submit,clean}`, which will list all the information of the scenario (tasks, solvers and seeds); launch the experiments and collect the logs; and clean up the logs of previous executions respectively.

By default, the logs of the experiment are stored inside the scenario folder, separated into directories for each solver. For more information about OptiLog's Running module you can check the official documentation: `http://ulog.udl.cat/static/doc/optilog/html/optilog/running.html`.

8.1 Processing Experimental Results

To process the results, we can use OptiLog to parse the logs and extract information. Listing 13 shows the code used to parse the logs for the *Pac-Man* experiment, and Listing 14 shows the output of this parsing.

First we have to define in a `ParsingInfo` object which information we want to extract. Suppose we are interested in evaluating which algorithm expands more nodes during the exploration, as well as assessing which ones can find optimal solutions. In lines 4 and 7 we add filters based on regular expressions to the parser to extract this information from the output of each execution. The `parse_scenario` function call (line 10) parses the result of the experiments and returns a Pandas dataframe [31] with the parsed data. Based on the students experience with Pandas, we can either provide them with sample code to analyze the results or let them to explore the dataframe by themselves. Listing 14 shows the result of the execution.

```
1  from optilog.running import *
2
3  pi = ParsingInfo()
4  pi.add_filter(name="cost", cast_to=int,
5    expression=r"Path found with total cost of (\d+)")
6
7  pi.add_filter(name="expand", cast_to=int,
8    expression=r"Search nodes expanded: (\d+)")
9
10 df = parse_scenario("./scenario", pi)
11 df = df.drop(["seed"], axis=1, level=1)
12
13 print("Cost of the path:")
14 print(df.xs("cost", level=1, axis=1))
15 print("===========================")
16 print("Expanded nodes:")
17 print(df.xs("expand", level=1, axis=1))
```

Listing 13 Log processing for *Pac-Man*

9 Feedback from Using OptiLog in Education

OptiLog is ready to be used by practitioners offering a simple use with lots of functionality to support many industrial tasks. With the aim of closing the gap between academic lectures and real-world development of SAT-based applications, we introduced OptiLog last year in an undergraduate course on Computational Logic (1st year, first semester) and Artificial Intelligence (3rd year).

```
Cost of the path:
                    bfs  dfs ...
tinyMaze.lay          8   10 ...
smallMaze.lay        19   49 ...
...
============================
Expanded nodes:
                    bfs  dfs ...
tinyMaze.lay         15   14 ...
smallMaze.lay        90   59 ...
...
```

Listing 14 Output of script in Listing 13

In the Computational Logic subject students have a very basic programming background (i.e. loops and functions) at the time the lab exercise is introduced. The assignments are evaluated using automatic verification tools, which are also provided to the students to validate their implementations.

For the Artificial Intelligence subject, students already have an advanced programming knowledge and they are also provided with automatic validation tools. In contrast to Computational Logic students, we do also evaluate the quality of their implementations.

This initiative has resulted in great success. Students immediately got fully motivated since they were able to develop and touch real applications that they never imagined from a subject that results quite abstract at first glance. Moreover, they were introduced to good practices in setting up a proper experimental environment. This is out of reach in many subjects since it requires a non-negligible amount of time unless you have the support of tools like OptiLog.

In contrast, small groups of Computational Logic students found the programming level a bit higher compared to other subjects in the degree. This is expected since first-year students typically have very different learning curves.

Instructors can also focus now their energy on providing additional support. For example, for the sudoku example, we created, an assignment auto-grader, similar to those available in the Pac-Man project [14], that gives concrete feedback on the mistakes of the students and allows instructors to evaluate the task's deliverables easily.

While OptiLog was born as a solution for developing SAT-based applications, its additional transversal modules (blackbox, tuning, running) make of it a good travel companion in many subjects and undergraduate courses.

10 Future Work

We plan on generating a database of course assignments, auto-graders and algorithm examples for educators and students. We also intend to deploy this framework in more advanced educational master courses.

References

1 Josep Alòs, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. OptiLog V2: Model, Solve, Tune and Run. In Kuldeep S. Meel and Ofer Strichman, editors, *SAT 2022*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16699, doi:10.4230/LIPIcs.SAT.2022.25.

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, *SAT 2021*, 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, *SAT 2020*, volume 12178 of *Lecture Notes in Computer Science*, pages 107–113. Springer, 2020.

6 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI 09*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

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

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

9 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.

10 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

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

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

13 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.

14 John DeNero and Dan Klein. Teaching introductory artificial intelligence with pac-man. In *First AAAI Symposium on Educational Advances in Artificial Intelligence*, 2010.

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

16 Don Libes. Unbuffer man page, 2021. URL: https://linux.die.net/man/1/unbuffer.

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 Bertram Felgenhauer and Frazer Jarvis. Mathematics of sudoku i. *Mathematical Spectrum*, 39(1):15–22, 2006.

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 Google. Google OR-Tools. https://developers.google.com/optimization, 2021.
21 Gurobi Optimization. Gurobi. https://www.gurobi.com/, 2021.
22 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.
23 IBM. IBM ILOG CPLEX. https://www.ibm.com/products/ilog-cplex-optimization-studio, 2021.
24 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.
25 Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. Cnfgen: A generator of crafted benchmarks. In Serge Gaspers and Toby Walsh, editors, *SAT 2017*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2017.
26 Lluis Batlle i Rossell. Task spooler man page, 2021. URL: http://manpages.ubuntu.com/manpages/xenial/man1/tsp.1.html.
27 Logic and Optimization Group. Optilog official documentation, 2021. URL: http://ulog.udl.cat/static/doc/optilog/html/index.html.
28 Logic Optimization Group. PyPBLib: PBLib Python3 bindings. https://pypi.org/project/pypblib/, 2018. Described in OptiLog [2].
29 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.
30 Nikoli. Nikoli's slitherlink webpage, 2021. URL: https://www.nikoli.co.jp/en/puzzles/slitherlink.html.
31 The pandas development team. pandas-dev/pandas: Pandas, February 2020. doi:10.5281/zenodo.3509134.
32 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.
33 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.
34 Olivier Roussel. Controlling a solver execution: the runsolver tool. *JSAT*, 7:139–144, 11 2011. doi:10.3233/SAT190083.
35 G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.
36 Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.
37 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.
38 T. Yato. On the np-completeness of the slither link puzzle. 2003.