PySAT: A Python Toolkit for Prototyping with SAT Oracles *

Alexey Ignatiev[1,2], Antonio Morgado[1], and Joao Marques-Silva[1]

[1] LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{aignatiev,ajmorgado,jpms}@ciencias.ulisboa.pt
[2] ISDCT SB RAS, Irkutsk, Russia

Abstract. Boolean satisfiability (SAT) solvers are at the core of efficient approaches for solving a vast multitude of practical problems. Moreover, albeit targeting an NP-complete problem, SAT solvers are increasingly used for tackling problems beyond NP. Despite the success of SAT in practice, modeling with SAT and more importantly implementing SAT-based problem solving solutions is often a difficult and error-prone task. This paper proposes the PySAT toolkit, which enables fast Python-based prototyping using SAT oracles and SAT-related technology. PySAT provides a simple API for working with a few state-of-the-art SAT oracles and also integrates a number of cardinality constraint encodings, all aiming at simplifying the prototyping process. Experimental results presented in the paper show that PySAT-based implementations can be as efficient as those written in a low-level language.

1 Introduction

When compared with Satisfiability Modulo Theories (SMT), Answer Set Programming (ASP) or Constraint Programming (CP), a well-known drawback of Propositional Logic (concretely, its satisfiability (SAT) problem) is the low level at which the problem constraints are represented and the low-level programmatic interface that must be used. These limitations hinder a wider adoption of SAT solvers, but in part they are also one reason for the observed performance gains that SAT-based solutions often enable. Moreover, it is generally perceived that SAT-based modeling is difficult and also error-prone. Clearly, the aforementioned alternatives, SMT, ASP and CP, also enable some sort of direct encoding to SAT, and then invoking a SAT solver, but often key aspects of the problem formulation are lost. Other approaches that directly encode problems into SAT have been considered, including NP-SPEC [11].

This paper describes PySAT, a toolkit that simplifies prototyping problem solvers with SAT solvers as oracles. Similarly to existing solutions for SMT, the prototyping language is Python, with a simple interface to an abstract SAT solver that abstracts most details away, but also aims at compromising little in terms of performance. The paper illustrates the ease of modeling reasonably challenging problems, concretely MUS extraction, but also provides empirical evidence that the toolkit can achieve reasonably

* This work was supported by FCT funding of post-doctoral grants SFRH/BPD/103609/2014, SFRH/BPD/120315/2016, FCT grant ABSOLV (028986/02/SAICT/2017), and LASIGE Research Unit, ref. UID/CEC/00408/2013.

efficient implementations when compared with existing state-of-the-art tools. PySAT is open source, and it is publicly available on GitHub. Furthermore, PySAT is also readily installable as a Python package.

This paper is organized as follows. Basic definitions and notation are introduced in the next section. Section 3 describes the toolkit, its design and interface. Section 4 outlines the implementation of a deletion-based MUS extractor. Section 5 presents experimental results comparing a PySAT-based prototype of a MaxSAT algorithm compared to the state-of-the-art implementation. Section 6 overviews prior work related with PySAT. Finally, the paper concludes in Section 7.

2 Preliminaries

This section introduces the notation and definitions used throughout the paper. Standard propositional logic definitions apply (e.g. [10]). CNF formulas are defined over a set of propositional variables. A CNF formula F is a propositional formula represented as a conjunction of clauses, also interpreted as a set of clauses. A clause is a disjunction of literals, also interpreted as a set of literals. A literal is a variable or its complement. Throughout the paper, SAT solvers are viewed as oracles. Given a CNF formula F, a SAT oracle decides whether F is satisfiable, in which case it returns a satisfying assignment. A SAT oracle can also return an unsatisfiable core $U \subseteq F$, if F is unsatisfiable. Conflict-driven clause learning (CDCL) SAT solvers are summarized in [10].

CNF formulas are often used to model overconstrained problems, for example, the maximum satisfiability (MaxSAT) problem and the minimal unsatisfiable subset (MUS) extraction problem. In general, clauses in a CNF formula are characterized as hard, meaning that these must be satisfied, or soft, meaning that these are to be satisfied, if at all possible. A weight can be associated with each soft clause, and the goal of MaxSAT is to find an assignment to the propositional variables such that the hard clauses are satisfied, and the sum of the satisfied soft clauses is maximized. Algorithms for MaxSAT have been overviewed in [1, 10, 31]. Recent algorithms based on implicit hitting sets have been described in [5]. In the analysis of unsatisfiable CNF formulas, consider a given unsatisfiable CNF formula F. An MUS of F is a set of clauses $M \subseteq F$ which is both unsatisfiable and irreducible. The goal of the MUS extraction problem is to determine an MUS of a given unsatisfiable CNF formula.

3 PySAT Toolkit Description

This section describes the design and implementation of the PySAT toolkit as well as its capabilities. The toolkit aims at simplifying the work with SAT oracles. It is to be used for fast prototyping solvers and tools that target tackling practical problems and exploit the power of the state-of-the-art SAT technology.

The choice of the Python programming language was done having the following in mind. First, the language is easy-to-use and proved itself a great language for fast prototyping. This enables users to focus on implementing and improving an algorithm rather than struggling with its low-level details. Also, Python programs are typically easy to debug. Second, Python is required for installation and, thus, ready for use on almost any operating system of the POSIX family including plenty of Linux distributions, BSD and MacOS among a multitude of others. Third, the use of Python enables a user to tightly

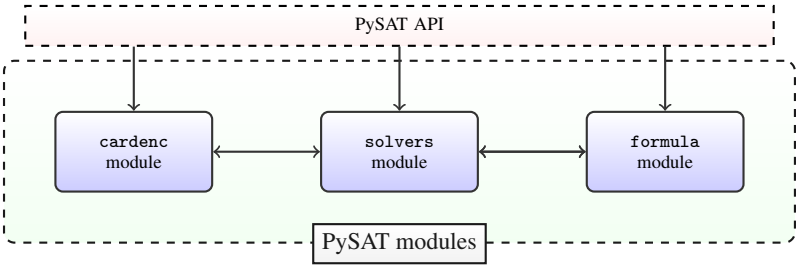


Fig. 1: PySAT toolkit and its modules.

and easily integrate his/her tools with the existing technology that provides Python API, e.g. such renowned packages for scientific computing as *NumPy* [33], *SciPy* [46] and *matplotlib* [24], ILP solvers (*ILOG CPLEX* [20], *Gurobi* [19]), graph and network related libraries (e.g. *networkX* [32] and *graphviz* [18]), state-of-the-art machine learning, data analysis and mining toolkits including *scikit-learn* [45], *PyTorch* [43] and *pandas* [35], among a number of other libraries and toolkits, which find myriads of practical use cases.

3.1 PySAT Design

As the PySAT toolkit targets fast prototyping with SAT oracles, it provides interface to a number of state-of-the-art CDCL SAT solvers including MiniSat 2.2 [13, 29] and its GitHub version [30], and also Glucose 3 and Glucose 4.1 [4, 17]. Additionally, it also includes a couple of SAT solvers augmented with extra reasoning capabilities, namely Lingeling [8, 9, 22] strengthened with *Gaussian elimination* and *cardinality-based reasoning* and MiniCard 1.2 [28], which besides clauses can natively work with a special kind of constraints called *cardinality constraints* [10], i.e. constraints of the form $\sum_{i=1}^{n} l_i \circ k$ where $i, n, k \in \mathbb{N}$, each l_i is either a positive or a negative literal of a Boolean variable and $\circ \in \{<, \leq, =, \neq, \geq, >\}$. The module of the PySAT toolkit responsible for providing an API to the SAT solvers is called `solvers`.

In many cases, SAT-based problem solving requires to efficiently deal with cardinality constraints. MiniCard can handle them natively but other solvers need them to be encoded into a CNF formula. There are multiple ways to encode cardinality constraints into a set of clauses and most state-of-the-art cardinality encodings are supported by PySAT including *pairwise* and *bitwise* encodings [38], *sequential counters* [48], *sorting networks* [7], *cardinality networks* [3], *ladder/regular* [2, 16], *totalizer* [6], *modulo totalizer* [34], and *iterative totalizer* [23]. This functionality is provided by the second module of the toolkit, namely by the `cardenc` module.

Additionally, PySAT provides a user with an input/output interface for simplified reading and writing formulas in the DIMACS format including plain CNF, partial CNF and weighted partial CNF formulas (WCNF). This is covered by the third module of the toolkit, which is referred to as `formula`.

As a result, the toolkit has three modules, two of which are implemented as C/C++ extensions (i.e. `solvers` and `cardenc`) and one module (`formula`) is a pure Python module. The structure of the PySAT toolkit can be seen as shown in Figure 1.

3.2 Provided Interface

Boolean variables in PySAT are represented as natural identifiers, e.g. numbers from $\mathbb{N}$. A positive (negative, resp.) literal in PySAT is assumed to be a positive (negative, resp.) integer, e.g. `-1` represents a literal $\neg x_1$ while `5` represents a literal x_5. A clause is a list of literals, e.g. `[-3, -2]` is a clause $(\neg x_3 \vee \neg x_2)$.

The `pysat.solvers` module provides an interface to SAT solvers directly as well as the abstract `Solver` class. Each SAT solver can be used in the MiniSat-like *incremental* fashion [14], i.e. with the use of *assumption literals*, and exhibits methods `add_clause()`, `solve()`, `get_model()`, and `get_core()`. [1] Using a solver incrementally can be helpful when multiple calls to the solver are needed in order to solve a problem, e.g. in MaxSAT solving or in MUS/MCS extraction and enumeration. In this case, a user needs to create a solver and feed it with a CNF formula only once while calling it multiple times with different sets of assumption literals. Observe that instead of using a solver incrementally, one can opt to create a new solver from scratch at every invocation.

```
>>> from pysat.solvers import Glucose3
>>> g = Glucose3()
>>> g.add_clause([-1, 2])
>>> g.add_clause([-2, 3])
>>>
>>> print g.solve()
True
>>> print g.get_model()
[-1, -2, 3]
>>> g.delete()
```

The `pysat.formula` module can be used for performing input/output operations when working with DIMACS formulas. This can be done using classes `CNF` and `WCNF` of this module. `CNF` and `WCNF` objects have a list of clauses, which can be added to a SAT oracle directly. The `cardenc` module operates through the `pysat.card` interface and provides access to the `atmost()`, `atleast()`, and `equals()` methods (they return an object of class `CNF`) of the abstract class `CardEnc`, e.g. in the following way:

```
>>> from pysat.card import *
>>> am1 = CardEnc.atmost(lits=[1, -2, 3], encoding=EncType.pairwise)
>>> print am1.clauses
[[-1, 2], [-1, -3], [2, -3]]
>>>
>>> from pysat.solvers import Solver
>>> with Solver(name='m22', bootstrap_with=am1.clauses) as s:
...     if s.solve(assumptions=[1, 2, 3]) == False:
...         print s.get_core()
[3, 1]
```

[1] The method `get_model()` (`get_core()`, resp.) can be used if a prior SAT call was made and returned `True` (`False`, resp.). The `get_core()` method additionally assumes the SAT call was provided with a list of assumptions.

input : Unsatisfiable CNF $\mathcal{F}$
output : MUS $\mathcal{M}$

$\mathcal{M} \leftarrow \mathcal{F}$

foreach $c_i \in \mathcal{M}$ **do**
  if not SAT($\mathcal{M} \setminus \{c_i\}$) **then**
    $\mathcal{M} \leftarrow \mathcal{M} \setminus \{c_i\}$
  end
end
return $\mathcal{M}$

(a) Pseudo-code of deletion-based MUS extraction.

```
# oracle: SAT solver (initialized)
# as:     full set of assumptions
i = 0
while i < len(as):
    ts = as[:i] + as[(i + 1):]
    if oracle.solve(assumptions=ts):
        i += 1
    else:
        as = ts
return as
```

(b) Its possible implementation with PySAT.

Fig. 2: An example of a PySAT-based algorithm implementation.

3.3 Installation

The PySAT library can be installed from the PyPI repository [41] simply by executing the following command:

```
$ pip install python-sat
```

Alternatively, one can manually clone the library's GitHub repository [42] and compile all of its modules following the instructions of the README file.

4 Usage Example

Let us show how one can implement prototypes with the use of PySAT. Here we consider a simple deletion-based algorithm for MUS extraction [47]. Its main procedure is shown in Figure 2a. The idea is to try to remove clauses of the formula one by one while checking the formula for unsatisfiability. Clauses that are necessary for preserving unsatisfiability comprise an MUS of the input formula and are reported as a result of the procedure. Figure 2b shows a possible PySAT-based implementation. The implementation assumes that a SAT oracle denoted by variable `oracle` is already initialized, and contains all clauses of the input formula $\mathcal{F}$. Another assumption is that each clause $c_i \in \mathcal{F}$ is augmented with a selector literal $\neg s_i$, i.e. considering clause $c_i \vee \neg s_i$. This facilitates simple activation/deactivation of clause c_i depending on the value of variable s_i. Finally, a list of assumptions `as` is assumed to contain all clause selectors, i.e. `as` $= \{s_i \,|\, c_i \in \mathcal{F}\}$. Observe that the implementation of the MUS extraction algorithm is as simple as its pseudo-code. This simplicity is intrinsic to Python programs, and enables users to think on algorithms rather than implementation details.

5 Experimenting with MaxSAT

One of the benefits provided by the PySAT toolkit is that it enables users to prototype quickly and sacrifice just a little in terms of performance. In order to confirm this claim in practice, we developed a simple (non-optimized) PySAT-based implementation of the Fu&Malik algorithm [15] for MaxSAT. The implementation is referred to as *fm.py*. The idea is to compare this implementation to the state-of-the-art MaxSAT solver *MiFuMaX* [21], which can be seen as a well thought and efficient implementation of the

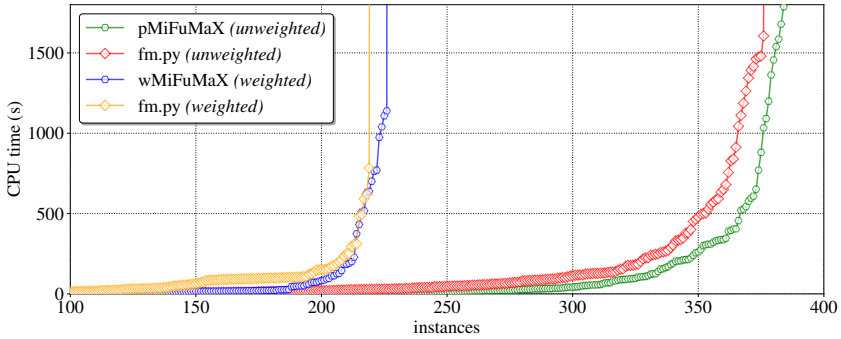


Fig. 3: Performance of fm.py and MiFuMaX on the MSE17 benchmarks.

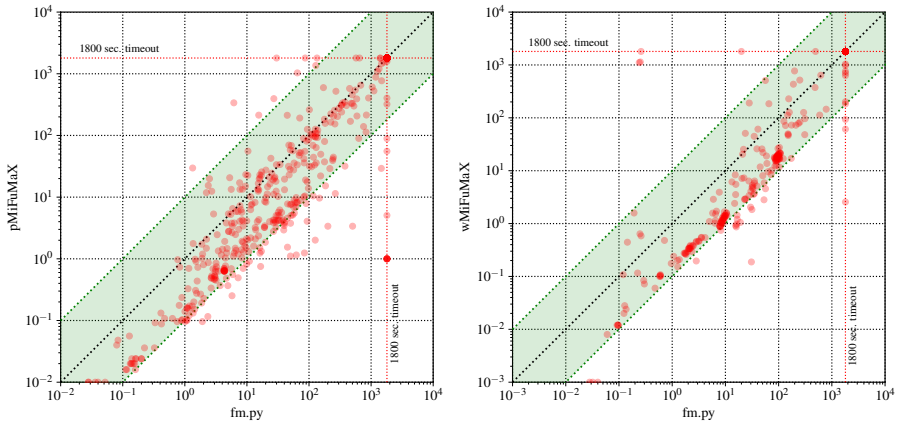
Fu&Malik algorithm written in C++ and available online [27]. MiFuMaX has proven its efficiency by winning the *unweighted* category in the MAX-SAT evaluation 2013 [25].

For the comparison, we chose all (i.e. unweighted and weighted) benchmarks from MaxSAT Evaluation 2017 [26]. The benchmarks suite contains 880 unweighted and 767 weighted MaxSAT instances. The experiments were performed in Ubuntu Linux on an Intel Xeon E5-2630 2.60GHz processor with 64GByte of memory. The time limit was set to 1800s and the memory limit to 10GByte for each individual process to run.

The cactus plot depicting the performance of MiFuMaX and fm.py is shown in Figure 3. As to be expected, our simple implementation of the Fu&Malik algorithm is outperformed by MiFuMaX. However, one could expect a larger performance gap between the two implementations given the optimizations used in MiFuMaX. Observe that MiFuMaX solves 384 unweighted and 226 weighted instances while fm.py can solve 376 and 219 unweighted and weighted formulas, respectively. The performance of the two implemetations is detailed in Figure 4. In both cases (unweighted and weighted benchmarks) MiFuMaX tends to be at most a few times faster than fm.py. Also note that even though surprising, there are instances, which are solved by fm.py more efficiently than by MiFuMaX. Overall, the performance of fm.py demonstrates that a PySAT-based implementation of a problem solving algorithm can compete with a low-level implementation of the same algorithm, provided that most of the computing work is done by the underlying SAT solver, which is often the case in practice.

6 Related Work

A number of Python APIs for specific SAT solvers have been developed in the recent past. These include *PyMiniSolvers* [40] providing an interface to MiniSat and MiniCard, *satispy* [44] providing an API for MiniSat and lingeling, *pylgl* [39] for working with lingeling, and the Python API for CryptoMiniSat [12, 49, 50]. Compared to these solutions, PySAT offers a wider range of SAT solvers accessed through a unified interface, more functionality provided (e.g. unsatisfiable core and proof extraction), as well as a number of encodings of cardinality constraints. Cardinality constraints (as well as pseudo-Boolean constraints) can be alternatively manipulated using encodings provided by some other libraries. One such example is the *PBLib* library [36,37]. However, PBLib currently does not expose a Python API.



(a) MSE17 unweighted benchmarks

(b) MSE17 weighted benchmarks

Fig. 4: Detailed comparison of fm.py and MiFuMaX.

7 Conclusions

Despite the remarkable progress observed in SAT solvers for over two decades, in many settings the option of choice is often not a SAT solver, even when this might actually be the ideal solution. One reason for overlooking SAT solvers is the apparent difficulty of modeling with SAT, and of implementing basic prototypes. This paper describes PySAT, a Python toolkit that enables fast prototyping with SAT solvers. The Python interface offers (incremental) access to a blackbox exposing the basic interface of a SAT solver, but which compromises little in terms of performance. The toolkit also offers access to a number of often-used implementations of cardinality constraints. A simple implementation of a MaxSAT solver shows performance comparable with a state-of-the-art C++ implementation. The PySAT tookit is publicly available as open source from GitHub, and also as a Python package on most POSIX-compliant OSes. It is expected that the community will contribute to improving the toolkit further, with additional features, but also with proposals for improvements. Several extensions are planned. These include the integration of more SAT solvers (e.g. CryptoMiniSat and other MiniSat- and Glucose-based solvers), lower level access to the SAT solver's parameters and policies when necessary (e.g. setting preferred "polarities" of the variables), high-level support for arbitrary Boolean formulas (e.g. by Tseitin-encoding them [51] internally), and encodings of pseudo-Boolean constraints.

References

1. C. Ansótegui, M. L. Bonet, and J. Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013.
2. C. Ansótegui and F. Manyà. Mapping problems with finite-domain variables to problems with Boolean variables. In *SAT*, pages 1–15, 2004.
3. R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In *SAT*, pages 167–180, 2009.
4. G. Audemard, J. Lagniez, and L. Simon. Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT*, pages 309–317, 2013.

5. F. Bacchus, A. Hyttinen, M. Järvisalo, and P. Saikko. Reduced cost fixing in maxsat. In *CP*, pages 641–651, 2017.
6. O. Bailleux and Y. Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *CP*, pages 108–122, 2003.
7. K. E. Batcher. Sorting networks and their applications. In *AFIPS Conference*, pages 307–314, 1968.
8. A. Biere. Lingeling, plingeling and treengeling entering the SAT competition 2013. In A. Balint, A. Belov, M. Heule, and M. Järvisalo, editors, *Proceedings of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 51–52. University of Helsinki, 2013.
9. A. Biere. Lingeling essentials, A tutorial on design and implementation aspects of the SAT solver lingeling. In *Pragmatics of SAT workshop*, page 88, 2014.
10. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
11. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artif. Intell.*, 162(1-2):89–120, 2005.
12. CryptoMiniSat. https://github.com/msoos/cryptominisat/.
13. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
14. N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
15. Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *SAT*, pages 252–265, 2006.
16. I. Gent and P. Nightingale. A new encoding of alldifferent into SAT. In *Intl. Workshop on Modelling and Reformulating Constraint Satisfaction Problem*, pages 95–110, 2004.
17. Glucose 3 and Glucose 4.1. http://www.labri.fr/perso/lsimon/glucose/.
18. graphviz. https://www.graphviz.org/.
19. Gurobi. http://www.gurobi.com/.
20. IBM ILOG: CPLEX optimizer 12.7.0. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer, 2016.
21. M. Janota. MiFuMax — a literate MaxSAT solver. *JSAT*, 9:83–88, 2015.
22. Lingeling bbc-9230380-160707. http://fmv.jku.at/lingeling/.
23. R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. Incremental cardinality constraints for MaxSAT. In *CP*, pages 531–548, 2014.
24. matplotlib. https://matplotlib.org/.
25. Eighth Max-SAT Evaluation. http://www.maxsat.udl.cat/13/.
26. MaxSAT Evaluation 2017. http://mse17.cs.helsinki.fi/.
27. MiFuMax. http://sat.inesc-id.pt/~mikolas/.
28. MiniCard 1.2. https://github.com/liffiton/minicard/.
29. MiniSat 2.2. http://minisat.se/MiniSat.html.
30. MiniSat GitHub. https://github.com/niklasso/minisat/.
31. A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.
32. networkx. https://networkx.github.io/.
33. NumPy. http://www.numpy.org/.
34. T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, and H. Fujita. Modulo based CNF encoding of cardinality constraints and its application to maxsat solvers. In *ICTAI*, pages 9–17, 2013.
35. pandas. https://pandas.pydata.org/.
36. PBLib. http://tools.computational-logic.org/content/pblib.php.
37. T. Philipp and P. Steinke. PBLib - A library for encoding pseudo-boolean constraints into CNF. In *SAT*, pages 9–16, 2015.

38. S. D. Prestwich. CNF encodings. In *Handbook of Satisfiability*, pages 75–97. 2009.
39. pylgl. https://github.com/abfeldman/pylgl/.
40. PyMiniSolvers. https://github.com/liffiton/PyMiniSolvers/.
41. PyPI. https://pypi.python.org/.
42. PySAT. https://pysathq.github.io/.
43. Pytorch. http://pytorch.org/.
44. satispy. https://github.com/netom/satispy/.
45. scikit-learn. http://scikit-learn.org/.
46. SciPy. https://scipy.org/.
47. J. M. Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL*, pages 9–14, 2010.
48. C. Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *CP*, pages 827–831, 2005.
49. M. Soos. Enhanced gaussian elimination in dpll-based SAT solvers. In *POS@SAT*, pages 2–14, 2010.
50. M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, pages 244–257, 2009.
51. G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.