The SAT4J library, Release 2.2, System Description

Daniel Le Berre, Anne Parrain

Journal on Satisfiability, Boolean Modeling and Computation 7 (2010) 59-64

The Sat4j library, release 2.2

SYSTEM DESCRIPTION

Daniel Le Berre* leberre@cril.univ-artois.fr
Anne Parrain parrain@cril.univ-artois.fr
Université Lille - Nord de France
CRIL-CNRS UMR 8188
Université d'Artois, Lens, France

Abstract

Sat4j is a mature, open source library of SAT-based solvers in Java. It provides a modular SAT solver architecture designed to work with generic constraints. Such architecture is used to provide SAT, MaxSat and pseudo-boolean and solvers for lightweight constraint programming. Those solvers have been evaluated regularly in the corresponding international competitive events. The library has been adopted by several academic softwares and the widely used Eclipse platform, which relies on a pseudo-boolean solver from Sat4j for its plugins dependencies management since June 2008.

KEYWORDS: *resolution, cutting-planes, SAT-solver, MAXSAT*

Submitted March 2010; revised May 2010; published July 2010

1. Introduction

Sat4j (http://www.sat4j.org/) is an open source library of SAT solvers which aims at allowing Java programmers to access cross-platform SAT-based solvers. The Sat4j library started in 2004 as an implementation in Java of the Minisat specification[10]. It has been developed since then with the spirit to allow testing various combinations of features developed in new SAT solvers while keeping the technology easily accessible to a newcomer. For instance, it allows the Java programmer to express constraints on objects and hides all the mapping to the various research community input formats from the user. Sat4j allows to solve a range of decision and optimization problems using pseudo-boolean solving as a universal engine for which the problems are translated. See Figure 1 for an overview of the features available in the library.

Sat4j is developed using both Java and open source standards: the project is supported by the OW2 consortium infrastructure and is released under both the EPL and the GNU LGPL licenses. It has been adopted by various Java-based academic software, in the area of software engineering[3]; bioinformatics[7]; or formal verification[11], but also by the popular Eclipse open platform. Each time an Eclipse's user installs a plugin or updates its system, such request is converted into a pseudo-boolean optimization problem modeling the plugins dependencies and the plugins selection policy[13]. Such application in Eclipse makes it one of the most widely used pseudo-boolean solver around the world (more than 13M downloads

* Part of this work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013'.

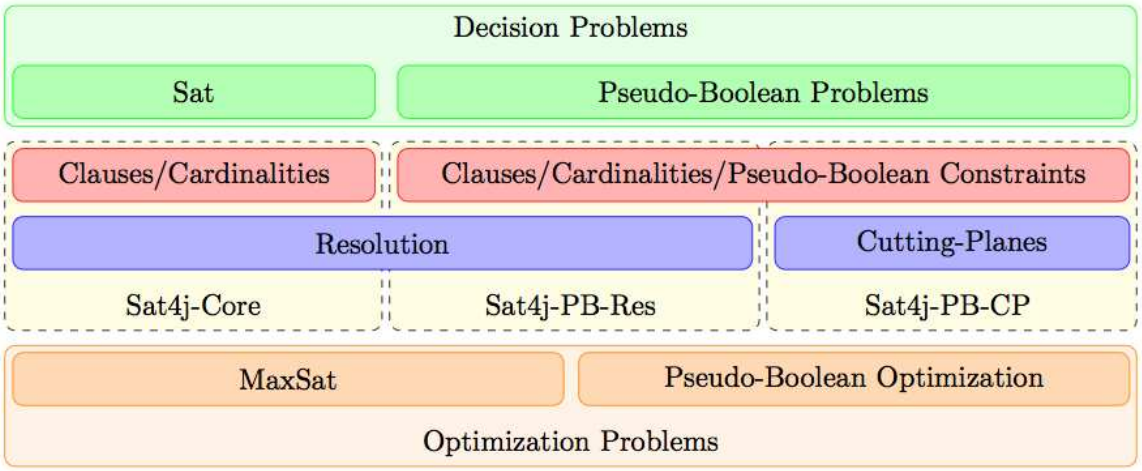

Figure 1. Overview of the features available in the SAT4J library

of Eclipse 3.4 between June 2008 and June 2009 and since then more than 12M downloads of Eclipse 3.5 as of May 2010[9]).

2. The conflict driven clause learning SAT solver

The underlying SAT solver (Sat4j-Core) is based on the original Minisat 1.x implementation [10]: the generic conflict driven clause learning engine and the variable activity scheme have not changed. Most of the key components of the solver have been made configurable. Here are the settings used in the default SAT solver available in Sat4j 2.2: the *rapid restarts* strategy is the in/out one proposed by Armin Biere in Picosat[4]; the *conflict clause minimization* of Minisat 1.14 (so called Expensive Simplification)[15] is used at the end of the conflict analysis; the phase selection strategy uses the *lightweight caching scheme* of RSAT[14] ; finally, the solver keeps derived clauses with literals from few different decision levels as proposed in 2009 award winner Glucose [2] using the settings "start cleanup at 5000 conflicts and increase that bound by 1000 conflicts when reached" provided by Armin Biere.

Lazy data structure The library provides two implementations of a lazy data structure for SAT: the classical watched literals as found in Minisat, using *literals move to front* (from Picosat) and an implementation of the head/tail lazy data structure where we swap the literals in the head and tail of the clause, instead of moving the pointers. The two data structures have the same property to be backtrack cost free. The former is easier and more elegant to implement. The latter has not been patented. In our experience, the head/tail data structure is slightly less efficient than the watched literals data structure.

Generic clause minimization The simplification of clauses derived by the conflict analysis procedure presented in [15] has been made generic to work with arbitrary constraints and data structures. This was achieved by relaxing two assumptions: i) that the first literal of the reason is satisfied and ii) that all the other literals in the reason are falsified. Those

assumptions do not hold with pseudo-boolean constraints because several literals may be propagated in such constraint, while the remaining literals are either falsified or undefined. The first one does not hold with the head/tail data structure for instance. The simplification procedure has been made generic by applying the recursive procedure to the falsified literals of the constraint only, using a specific test to gather the truth value of a literal on all the literals of the constraint. That procedure is thus strictly slower than the original one in case of pure clausal constraints using the watched literals data structure.

3. The Pseudo-Boolean solvers

Pseudo-boolean constraints are more expressive and more compact than clauses to represent some boolean formulas. They are for instance convenient to express that an optimization function should have a value greater or lower than a specific value. Two different categories of pseudo-boolean solvers are available in Sat4j:

Sat4j PB Res The solver is exactly the core SAT engine with the ability to process cardinality constraints and pseudo-boolean constraints. Such constraints are considered as simple clauses during conflict analysis (i.e. only their falsified literals are taken into account). The positive side of using resolution is the ability to use the generic clause minimization procedure defined in the previous section. The negative side is that such a solver cannot solve efficiently benchmarks such as pigeon hole for instance. This solver is similar in spirit to PBS or SATZOO, but using most recent techniques in SAT solving, arbitrary size coefficients and the pseudo-boolean competition input format.

Sat4j PB CP This solver uses exactly the same settings as Sat4j PB Res but relies on cutting planes instead of resolution during conflict analysis as described in [5, 8]. The proof system of the solver is thus more powerful than resolution and it allows to solve crafted benchmarks such as pigeon hole. However, the conflict analysis procedure is much more complex to implement than plain resolution and uses arbitrary precision arithmetic to avoid overflow. Note that the solver is two orders of magnitude slower than the resolution-based one in terms of number of assignments per second on many benchmarks but it can solve some crafted benchmarks or prove the optimality of some benchmarks out of reach for the resolution-based solver thanks to its powerful proof system.

Both PB solvers get slower when dealing with pseudo-boolean constraints because we have not yet found an efficient lazy data structure similar to the head-tails or watched literals for those constraints. This is especially the case for the cutting-planes-based solver because the number of pseudo-boolean constraints grows during the search. For that reason, we always represent inside the solver a constraint in its simplest form (clause or cardinality constraint, if possible), independently of the way it is represented originally.

4. From a decision to an optimization procedure

The optimization part is solved by strengthening (cf. algorithm 1). Once a solution M is found, the value of the objective function $objFct = \sum a_i x_i$ for such solution is computed ($y = objFct(M)$). We add a new pseudo-boolean constraint in the solver to prevent so-

lutions with value equal or greater than y to be found: $objFct < y$. Since all the added constraints are of the form $objFct < y'$ with $y' < y$, we simply keep one strengthening constraint per problem by replacing $objFct < y$ by $objFct < y'$ while keeping all learned constraints. Once the solver cannot find a new solution, the latest one is proved optimal. Such an approach is often referred to as "linear search", in contrast with the more classical binary search using both a lower bound and an upper bound to locate the optimal value. In our case, Sat4j solvers usually have a hard time to prove unsatisfiability (i.e. lower bounds) of pseudo-boolean problems. This is the reason why we use only upper bounds.

```
input  : A set of clauses, cardinalities and pseudo-boolean constraints
         setOfConstraints and an objective function objFct to minimize
output: a model of setOfConstraints, or UNSAT if the problem is unsatisfiable.

answer ← isSatisfiable (setOfConstraints);
if answer is UNSAT then
  return UNSAT
end
repeat
  model ← answer;
  answer ← isSatisfiable (setOfConstraints ∪ {objFct < objFct (model)});
until (answer is UNSAT);
return model;
```

Algorithm 1: Optimization using strengthening (linear search)

Over the years, we succeeded in reducing the differences between the SAT and pseudo-boolean solvers. The main noticeable changes in case of optimization problems are:

- the heuristics takes into account the objective function in the phase selection strategy: literals that appear with a negative weight will be satisfied first, while literals appearing with a positive weight will be falsified first.
- release 2.2 also provides a better integration of the restarts strategy within the optimization procedure: it takes into account the context of the search, i.e. it is not reset at each call to the SAT solver in the algorithm 1.

5. The MaxSat solvers

Sat4j translates Partial Weighted MaxSat (PWMS) problems into pseudo-boolean optimization ones. Since all the other variants (MaxSat, Partial MaxSat and Weighted MaxSat) can be considered as specific cases of PWMS, such approach can be used for all categories of the MaxSat evaluations.

The idea is to add a new variable per weighted soft clause that represents that such clause has been violated, and to translate the maximization problem on those weighted soft clauses into a minimization problem on a linear function over those variables. Formally, suppose $T = \{C_1^{w1}, C_2^{w2}, \ldots, C_n^{wn}\}$ is the original set of weighted clauses of the WPMS problem. We translate that problem into $T' = \{s_1 \vee C_1, s_2 \vee C_2, \ldots, s_n \vee C_n\}$ plus the objective function $min : \Sigma_{i=1}^{n} w_i s_i$. That approach may look unapplicable in practice because one needs to

add as many new selector variables as clauses in the original problem. However, there are several cases for which no new variable is necessary:

hard clauses ($w_i = \infty$) there is no need for new variables for hard clauses since they must be satisfied. They can be treated "as is" by the SAT solver.

unit soft clauses those constraints are violated when its literal is falsified. As such, it is sufficient to consider that literal only in the optimization function, so no new selector variable is needed. In that case, the optimization function should minimize $\Sigma_{i=1}^{n} w_i \overline{l_i}$ where l_i is the literal in the unit clause C_i.

Thus, on Partial [Weighted] MaxSAT, depending on the proportion of soft clauses compared to the hard clauses, the number of additional variables can be negligible compared to the original number of variables. This is especially true for instances of the binate covering problem [6], a specific case of the partial weighted MaxSAT problem, whose soft clauses are all unit, because we do not need to add any new selector variable in that case. The pseudo-boolean solver used in Sat4j MaxSAT is Sat4j PB Res. Since the underlying SAT solver is tailored to solve application benchmarks, our approach performed poorly on randomly generated PWMS problems but provided good results on some "industrial [weighted] partial MaxSAT" classes of application benchmarks during the MaxSAT evaluation 2009.

6. Assumption-based unsat core

One of the most demanded features when Sat4j was integrated within Eclipse was to be able to gather an explanation in case of failure. This is often referred to as minimal UNSAT core or Minimal Unsatisfiable Subformula (MUS) in the SAT community. There are many existing approaches to compute those UNSAT cores. But they usually require to keep track of the resolution steps performed in the solver, or require some information from a local search solver. We did not want to implement an intrusive solution in our solver. Furthermore, we needed a solution that works with generic constraints (clauses and cardinality constraints for the specific case of Eclipse[13]) because it allows a one-to-one mapping between the constraints and the explanation. We decided to use the idea of assumption-based satisfiability from the original Minisat and the ability to derive for the final top-level conflict a clause only made of assumption literals (similarly to the procedure implemented in Minisat 1.14) as described in [1]. As for MaxSAT, we append a new selector variable per constraint. The set of assumptions is the negation of the selector variables. If the formula is inconsistent, the specific conflict analysis procedure applied on the top level conflict will return a subset of the assumptions that corresponds to an unsat core. That unsat core is then minimized using a tailored QuickXplain algorithm [12], using also selector variables to activate/deactivate clauses.

7. Conclusion

Sat4j is a mature, open-source library providing access to SAT-related technologies to Java programmers. While the core SAT engine is not really competitive with state-of-the-art SAT solvers (last during the SAT Race 2008, not qualified for the second stage of the SAT competition 2007 and 2009), the results of the library on pseudo-boolean problems

or on MaxSAT problems are reasonable, sometimes even good on some specific classes of problems. The library is designed to be reusable and robust. Sat4j has been adopted by several academic or commercial softwares. Its inclusion into the open platform Eclipse brings SAT technology to millions of desktop computers worldwide since June 2008.

References

[1] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in sat. In *Proc. of LPAR'08*, pages 16–30, 2008.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solver. In *Proc. of IJCAI'09*, pages 399–404, jul 2009.

[3] Don S. Batory. Feature models, grammars, and propositional formulas. In *Proc. of SPLC'05*, pages 7–20, 2005.

[4] Armin Biere. Picosat essentials. *JSAT*, **4**(2-4):75–97, 2008.

[5] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. of DAC'03*, pages 830–835, Anaheim, CA, 2003.

[6] O. Coudert. On solving covering problems. In *Proc. of DAC'96*, pages 197–202, 1996.

[7] Hidde de Jong and Michel Page. Search for steady states of piecewise-linear differential equation models of genetic regulatory networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, **5**(2):208–222, 2008.

[8] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proc. of AAAI'02*, pages 635–640, 2002.

[9] Eclipse Foundation. http://www.eclipse.org/.

[10] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proc. of SAT'03*, pages 502–518, 2003.

[11] Daniel Jackson and Felix Chang. http://alloy.mit.edu/.

[12] Ulrich Junker. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *Proc. of AAAI'04*, pages 167–172, 2004.

[13] Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem. In *Proc. of IWOCE2009*, August 2009.

[14] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proc. of SAT'07*, pages 294–299, 2007.

[15] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proc. of SAT'09*, pages 237–243, 2009.