Argumental: an abstract argumentation solver based on SAT solvers

Comlan Amouwotor

Lidia Sadi

Djaodoh Ossara

December 30, 2022

Abstract

PySAT is a Python module bringing the SAT technology to Python. It offers a Python interface to many state-of-the-art SAT solvers compiled in lower-level languages. We use this module to compute the stable and complete extensions based on the logical encodings proposed in Besnard and Dourtre 2004 article[BD04] referred to in class.

1 Introduction

Given an argumentation framework $F =< A, R >$ the problem of finding its complete and stable extensions are known to NP-complete problems (cf cours). For the definitions,

- a subset $C(C \subset A)$ of arguments is an an extension for the complete semantics if its elements are conflict-free (i.e. there exist no $(a, b) \in R$ such that $a \in C$and$b \in C$ and such that all attackers of the elements in the subset are attacked by some other element in the subset and no argument outside of the subset is defended by an element of the subset without another element of the subset attacking it.
- stable extensions are special kinds of complete extensions which attack all arguments outside of their subset.

Our task for this project was to build a program that is able to compute extensions for the complete and stable semantics and which, depending on some parameters, would print one extension for the given semantic or decide the credulous or skeptical acceptability of a certain argument with respect to that semantic.

There are many ways to compute extensions but as was said in our course (cf cours) the most efficient techniques currently known are based on SAT solvers.

1.1 SAT solvers

SAT solvers (short for Boolean Satisfiability Solvers) are algorithms that can be used to determine whether a given Boolean formula (a set of logical constraints) can be satisfied, or in other words, whether there exists an assignment of truth values to the variables of the formula that makes the formula evaluate to true.

SAT solvers are widely used in various fields, including computer science, artificial intelligence, and engineering, to solve a wide range of problems. For example, SAT solvers can be used to automatically find solutions to puzzles, optimize resource allocation, verify the correctness of software and hardware designs, and perform probabilistic inference in graphical models.

The input to a SAT solver is a Boolean formula in Conjunctive Normal Form (CNF), which is a standard representation for Boolean formulas. A Boolean formula in CNF consists of a conjunction

(AND) of one or more clauses, where each clause is a disjunction (OR) of one or more literals. A literal is a Boolean variable or its negation.

There are many different algorithms and approaches that have been developed for solving SAT, including local search, complete search, and incomplete search algorithms. The performance of SAT solvers can vary significantly depending on the structure and size of the formula being solved, as well as the specific algorithm being used.

There are many different SAT solvers available on the market, each with its own strengths and characteristics. Some of the most efficient and widely used SAT solvers include:

- MiniSat: MiniSat is a popular open-source SAT solver that is known for its simplicity and efficiency. It is based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and uses clause learning and conflict-driven clause learning (CDCL) to speed up the solving process.
- Z3: Z3 is a highly efficient SAT solver developed by Microsoft Research. It is based on the Conflict-Driven Clause Learning (CDCL) algorithm and supports a wide range of theories and decision procedures, making it well-suited for solving complex and multi-theory problems.
- CryptoMiniSat: CryptoMiniSat is an open-source SAT solver optimized for solving instances with many variables and few clauses. It is based on the CDCL algorithm and incorporates a number of advanced techniques, such as clause elimination, clause minimization, and dynamic restarting, to improve its performance.
- Lingeling: Lingeling is a highly efficient SAT solver based on the CDCL algorithm. It is known for its fast performance and low memory usage, making it well-suited for solving large and complex instances.
- Glucose: Glucose is an open-source SAT solver developed by the University of Manitoba and INRIA(short for Institut National de Recherche en Informatique et en Automatique) that is based on the Conflict-Driven Clause Learning (CDCL) algorithm. It is known for its fast performance and ability to solve difficult instances, making it a popular choice for many SAT-based applications.

In our solution we used Glucose4 which is a newer version of the Glucose SAT solver built on the basis of the Glucose3 (released in 2011). Overall, Glucose 4 is a more efficient and scalable version of the Glucose SAT solver that introduces several new features and improvements to the original Glucose 3 implementation.

We use Glucose4 through the PySAT module.

2 Our approach to solving the problem

Our code is based on the formulas presented in Besnard and Doutre (2004) (ref article) which was introduced in our course. This approach greatly facilitates the production of an efficient code that works.

2.1 Our code design

Our code is written in Python and is object-oriented. The responsibilities are assigned to classes as followed:

- the `SolverManager` class: a unique instance of this class is obtained through a call to `SolverManager.get_instance()` and is initialized with the problem parameters passed to the script:
  - the option `-p`: the type of problem solved, which can be one of SE-CO, SE-ST, DC-CO, DC-ST, DS-CO or DC-ST.
  - the option `-f`: the file describing the argumentation framework in a declarative language the arguments and the attack relationships.
  - the option `-a`: in the case of DC-** and DS-** problems, this option is the argument being tested.
- la classe `SATBasedSolver`: un objet de cette classe est créé par l'objet SolverManager, initialisé et reçoit le message de résoudre le problème par sa méthode .solve(problem, arg).

2.2 Computing stable extensions

Computing stable extensions using the result in [**?**] is straightforward. The formula given in the article is easily translated into CNF form which is the form required by PySAT solvers. From Besnard and Doutre we know that stable extensions are models of the following logical formula:

$$\Phi_{st}(F) = \wedge_{a \in A}(a \iff \wedge_{(b,a) \in R} \neg b)$$

This formula is easily transformed into a more explicit CNF through De Morgan laws:

$$\Phi_{st}(F) = \wedge_{a \in A}(\wedge_{(b,a) \in R}(\neg a \vee \neg b) \wedge a \vee (\vee_{(b,a) \in R} b))$$

In this form, this formula is a pretty direct interpretation of the definition of stable extensions into a logical formula. The formula expresses the fact that:

- the extension is a conflict-free set of arguments, which is expressed by the following part of the previous formula: $\wedge_{(b,a) \in R}(\neg a \vee \neg b)$. In other words, for each argument a , it can't be in the extension as well as one of its attackers. Thus no two arguments attack each other
- all arguments outside of the extension have at least one attacker from inside the extension. That is expressed by the following part of the previous formula:

  $$a \vee (\vee_{(b,a) \in R} b)$$

  which is equivalent to

  $$\neg a \implies \vee_{(b,a) \in R} b$$

  which says that if an argument is outside of the extension (expressed by the $\neg$) then it has at least one attacker in the extension. In particular an argument which is not attacked at all is always accepted.

This CNF formula is fed into the Glucose4 SAT solver offered by PySAT in our code in the following snippet:

Stable extension CNF formula

```
1
2  def stable_extensions(self, problem=False, arg=None):
3      """
4      problem is SATBasedSolver.SOME_EXTENSION
5      | SATBasedSolver.CREDULOUS | SATBasedSolver.SKEPTICAL
6      arg is the argument of which we are checking acceptability
7      """
8      # Build the clauses
9      cnf = self.conflict_free_sets_CNF()
10     for argument in self.arguments.values():
11         # If a certain argument is not accepted then
12         # one its attackers has been accepted.
13         cl=[argument]
14         if argument in self.attackers_list.keys():
15             cl.extend(self.attackers_list[argument])
16         cnf.append(cl)
17     return self.call_SAT_oracle_for_stable(cnf, problem, arg)
18
```

The last line makes a call to the following function:

The SAT solver call and the decision

```
def call_SAT_oracle_for_stable(self, cnf, problem, arg):
    solver = Solver(use_timer=True, name=SATBasedSolver.NAME, \
    bootstrap_with=cnf.clauses)
    solved = solver.solve()
    if solved:
        if problem==SATBasedSolver.SOME_EXTENSION:
            model = solver.get_model()
            solver.delete()
            return self.solution_for_print(model)
        else:
            for model in solver.enum_models():
                arg_in_extension = self.arguments[arg] in model
                if problem==SATBasedSolver.CREDULOUS and arg_in_extension:
                    solver.delete()
                    return "YES"
                elif problem==SATBasedSolver.SKEPTICAL and not arg_in_extension:
                    solver.delete()
                    return "NO"
    else:
        solver.delete()
        return "NO"
    solver.delete()
    if problem==SATBasedSolver.SOME_EXTENSION:
        return "NO"
    elif problem==SATBasedSolver.CREDULOUS:
        return "NO"
    elif problem==SATBasedSolver.SKEPTICAL:
        return "YES"
```

In our tests cases the SAT Solver has provided solutions for pretty large argumentation frameworks in a pretty reasonable time.

2.3 Computing complete extensions

Here again we use the formula given in Besnard and Doutre to partially solve the problem but only partially since the formula is not entirely in CNF form and transforming is completely into CNF results into a loss of performance. We thus generated conflict-free sets which are admissible using a CNF formula fed into the SAT solver. Then, we go over the models obtained to check which ones defend only the members of the extension. According to the article Besnard and Doutre, complete extensions of the argumentation framework are the models of the following formula:

$$\Phi_{co}(F) \equiv \Phi_{cf}(F) \wedge \Phi_{def}(F)$$

where

$$\Phi_{cf}(F) \equiv \wedge_{(a,b)\in R}(\neg a \vee \neg b)$$

which we met already in the stable extension formula. And,

$$\begin{aligned}\Phi_{def}(F) &\equiv \wedge_{a\in A}(a \iff \wedge_{(b,a)\in R}(\vee_{(c,b)\in R} c)) \\ &\equiv a \implies \wedge_{(b,a)\in R}(\vee_{(c,b)\in R} c) \wedge \neg a \implies \vee_{(b,a)\in R}(\wedge_{(c,b)\in R} \neg c)\end{aligned}$$

The previous line can be explained the conjunction of two conditions that complexte extensions has to very, on top of being conflict-free:

- complete extensions are admissible, which means that for each argument in the extension and each attack it receives, there is a defender inside the extension which defends it against that attack. That is the first part $a \implies \wedge_{(b,a)\in R}(\vee_{(a,b)\in R} c$. This admissibility part is easily transformed into

CNF as well and combined with the conflict-freeness clauses. It is solved very efficient with our SAT solver.

- the remaining part of the expression is $\neg a \implies \vee_{(b,a)\in R}(\wedge_{(a,b)\in R}\neg c)$ which says that for each argument outside of the extension, there exist one attack it receives against which it is not defended. This is equivalent to saying that the extension defends only its members, which is a required condition for complete extensions. We haven't found a way to transform this condition into CNF efficiently, so instead we check it iteratively using the following code:

Checking whether this admissible model defends only its own arguments

```
1
2  def contains_all_defended(self, model):
3      in_model=defaultdict(lambda: 0)
4      for arg in model:
5          if arg > 0:
6              in_model[arg]=1
7      for argument in model:
8          if argument < 0:
9              continue
10         for attacked in self.attacked_list[argument]:
11             for defended in self.attacked_list[attacked]:
12                 # this insider is defending an outsider
13                 if not in_model[defended]:
14                     # but is that outsider attacked
15                     # by another insider?
16                     attacked=False
17                     for att in self.attackers_list[defended]:
18                         # yes he is, then it's fine,
19                         # he deserves to be outside
20                         if in_model[att]:
21                             attacked=True
22                             break
23                     # no, he is not. Then, this is not a complete extension.
24                     if not attacked:
25                         return False
26     return True
```

Our solution first generates admissible extensions and then uses this function to check whether is defends only it's elements. For very large graphs, the number of admissible extensions is very large thus the performance of the algorithm suffers. The following code shows how the thing is glued together:

Generating admissible set using the SAT solver then checking whether for exclusive self-defense

```
def call_SAT_oracle_for_complete(self, cnf, problem, arg):
    solver = Solver(use_timer=True, name=SATBasedSolver.NAME, bootstrap_with=cnf.clauses)
    # generate admissible sets
    solved = solver.solve()
    if solved:
        if problem==SATBasedSolver.SOME_EXTENSION:
            for model in solver.enum_models():
                if self.contains_all_defended(model):
                    solver.delete()
                    return self.solution_for_print(model)
        else:
            for model in solver.enum_models():
                if self.contains_all_defended(model):
                    arg_in_extension = self.arguments[arg] in model
                    if problem==SATBasedSolver.CREDULOUS and arg_in_extension:
                        solver.delete()
                        return "YES"
                    elif problem==SATBasedSolver.SKEPTICAL and not arg_in_extension:
                        solver.delete()
                        return "NO"
    else:
        solver.delete()
        return "NO"
    solver.delete()
    if problem==SATBasedSolver.SOME_EXTENSION:
        return "NO"
    elif problem==SATBasedSolver.CREDULOUS:
        return "NO"
    elif problem==SATBasedSolver.SKEPTICAL:
        return "YES"

```

Due to the fact that PySAT does not offer support for DNF formulas, we have been forced to use this strategy. It yields good results on small and medium sized argumentation frameworks but is too long on large ones. To mitigate this slowness, we use the stable extension generator discussed above in cases where there are stable extensions

References

[BD04] Besnard and Doutre. Checking the acceptability of a set of arguments. 2004.