Université de Paris

SAT solvers
Computational Models of Argumentation

Christophe Yang
Quentin Januel
Sylvain Declercq

2022

Contents

1 Introduction

Argumentation frameworks are widely used in artificial intelligence since they can model formalized arguments and draw conclusions from them.
In this paper, we will describe how we made our argumentation solver using SAT solvers. It has already been shown that the choice of the SAT solver is important (citation needed), so we will also compare SAT solvers for several argumentation framework instances. The goal is to see if we can apply an heuristic to decide which solver to use in which scenario.

2 Background notions

2.1 Argumentation framework

An argumentation framework is an oriented graph where the nodes represent arguments and the edges represent attacks.

2.2 Extension and semantics

An extension is a subset of nodes that represent the accepted arguments. It is defined depdending on a given semantics. In this paper, we will focus on the following semantics: **complete**, **stable**, **grounded** and **preferred** [**?**].
We will refer to them as CO, ST, GR and PR from now on. Except for GR, finding extensions of these semantics is a hard problem (NP). That's why the **ICCMA**[**?**] has been organized since 2015 to find the best algorithms.

2.3 Tasks

For each semantics, there are four different tasks we are interested in:
- **SE**: find one extension, no matter which one
- **EE**: find all the extensions
- **DC**: decide if a given argument is credulously accepted
- **DS**: decide if a given argument is skeptically accepted

An argument is said to be credulously accepted if it is accepted in at least one of the extensions, and skeptically accepted if it is accepted in all of them.
That leaves us with the 16 following problems: SE-CO, SE-ST, SE-GR, SE-PR, EE-CO, EE-ST, EE-GR, EE-PR, DC-CO, DC-ST, DC-GR, DC-PR, DS-CO, DS-ST, DS-GR, DS-PR.

3 Logical encodings for abstract argumentation

We can convert each semantics to a SAT instance.
For instance, here is the complete semantics as a SAT instance: *inserthere* Finding one extension is then equivalent to finding a model of the SAT instance, and finding all the extensions is equivalent to enumerating all its models.
For the DC task, we add a clause forcing the argument to be accepted, and then check if the model is satisfiable.
For the DS task, we add a clause forcing the argument to be rejected, and then check if the model

is unsatisfiable.

We have the following CNF:

$$\begin{aligned}
\text{ST} : & \bigwedge_{a_i \in A} \left(a_i \vee \bigvee_{a_j \in A, (a_j, a_i) \in R} a_j \right), \\
& \bigwedge_{a_i \in A} \left[\bigwedge_{a_j \in A, (a_j, a_i) \in R} (\neg a_i \vee \neg a_j) \right]
\end{aligned} \tag{1}$$

$$\begin{aligned}
\text{CO} : & \bigwedge_{a_i \in A} (\neg a_i \vee \neg P_{a_i}), \\
& \bigwedge_{a_i \in A} \left(a_i \vee \bigvee_{a_j \in A, (a_j, a_i) \in R} \neg P_{a_j} \right), \\
& \bigwedge_{a_i \in A} \bigwedge_{a_j \in A, (a_j, a_i) \in R} (\neg a_i \vee P_{a_j}), \\
& \bigwedge_{a_i \in A} \left(\neg P_{a_i} \vee \bigvee_{a_j \in A, (a_j, a_i) \in R} a_j \right), \\
& \bigwedge_{a_i \in A} \bigwedge_{a_j \in A, (a_j, a_i) \in R} (P_{a_i} \vee \neg a_j)
\end{aligned} \tag{2}$$

where A is the set of arguments and R the set of attacks[**?**].
The grounded semantics is trivial: it is always unique and we are guaranteed to find it by doing a unit propagation, hence its linear time complexity.
The preferred semantics is more complex. There are two different approaches we will compare in this paper.
The first one is to compute all the complete extensions and
- for SE, return the longest one
- for EE, filter the complete extensions that are subsets of other complete extensions
- for DC, same as EE but then check if the argument belongs to ne preferred extension
- for DS, same as EE but then check if the argument belongs to all preferred extensions
The second one is to find one complete extension and to try to add arguments to it until not possible.

4 Implementation

The decided to implement the solver in Rust for two main reasons: it is fast and reliable.
We first made a library[**?**] to use famous SAT solvers we will then compare. It allows for minisat, manysat, glucose and maplesat.
We can also use it to run several solvers in parallel as a portfolio.
Then we made the argumentation solver itself[**?**]. It has been tested with all kinds of argumentation frameworks so we can be confident that it is likely bug free.

5 Experimental results

When benchmarking, we will have a time limit for each problem. If a given solver takes more than this time, we will consider it took 10 times more time.
Also, each problems requires three steps: parse the argumentation framework, convert it to a SAT instance and solve it. We will only measure the latter one since the first two will not vary depending of the choice of the SAT solver. . . .

6 Related works

7 Conclusion