Université Paris Cité
UFR Mathématiques et Informatique

SAT Solvers Portfolio for Abstract Argumentation Master 1

Intelligence Artificielle Distribuée

Christophe Yang
Quentin Januel
Sylvain Declercq
Supervised by: Jean-Guy Mailly & Jérome Delobelle

2022

Contents

1 Introduction

Argumentation frameworks (AF for sort) are widely used in artificial intelligence since they can model formalized arguments and draw conclusions from them.
Indeed, an AF represents a debate with its arguments attacking each other. Given an AF, there are several problems that can be tackled such as "finding a coherent subset of *accepted* arguments(SE)" or "finding them all(EE)". Or "whether it is conceivable(DC), or necessary(DS), for given argument to be *accepted*". There are AF solvers designed to solve these problems, and the International Competition on Computational Models of Argumentation (ICCMA) [1] takes place every two years to compare their speed. The best submitted solvers in the competition are often based on SAT solvers: indeed, it turns out we can translate the AF problems to SAT instances. Therefore we need to solve these SAT instances in order to solve AF problems.
We will first demonstrate how one can make use of this method to build an AF solver.

It is important to note there is a very large amount of SAT solvers, each with their strengths and weaknesses. Some solvers may be better for specific problems, while being worse for others [2]. That is why we will also study the impact of the SAT solvers, and see if one can take advantage of a portfolio, i.e. having a AF solver that uses internally many SAT solvers to either launch them simultaneously on separate threads, or choose what SAT solver to use with a heuristic based on the AF structure.

2 Abstract Argumentation Theory of Dung

Formally[3], an **AF** is a pair $F = (A,\ R)$ where

- A is a set of arguments,
- $R \subseteq A^2$ represent attacks between arguments.

It can be thought as a directed graph where the nodes represent arguments and the edges represent attacks.

Intuitively, we would not want to have an accepted argument attacked by another accepted argument (for example).

An **extension** $E \subseteq A$ is set of jointly acceptable arguments, they can be thought as a "point of view" about the situation.
We say that an extension E attacks an argument a_1 if it contains an argument a_2 such that a_2 attacks a_1, i.e. if $\exists a_2 \in E,\ (a_2,\ a_1) \in R$. We denote $att(E)$ the set of arguments E attacks.
We also say that it defends an argument a_1 if it attacks all the arguments attacking a_1, i.e. if $\forall a_2 \in A,\ (a_2,\ a_1) \in R \implies a_2 \in att(E)$. We denote $def(E)$ the set of arguments E defends.

A **semantics** σ maps an AF to a set of extensions ($\sigma(F) \subseteq 2^A$).
The most common semantics to consider are:

- **Conflict-free**(CF): $\forall(a_1,\ a_2) \in E, (a_1,\ a_2) \notin R$,
- **Admissible**(AD): $E \in \sigma_{\mathbf{CF}}(F) \wedge E \subseteq def(E)$,

- **Complete**(CO): $E \in \sigma_{\mathbf{AD}}(F) \wedge def(E) \subseteq E$,
- **Preferred**(PR): $E \in \sigma_{\mathbf{CO}}(F) \wedge \forall E' \in \sigma_{\mathbf{CO}}(F),\ E' \subseteq E$,
- **Grounded**(GR): $E \in \sigma_{\mathbf{CO}}(F) \wedge \forall E' \in \sigma_{\mathbf{CO}}(F),\ E \subseteq E'$,
- **Stable**(ST): $E \in \sigma_{\mathbf{CF}}(F) \wedge A \backslash E \subseteq att(E)$.

Moreover, given a semantics σ we say that an argument a is

- **skeptically** accepted if $\forall E \in \sigma(F),\ a \in E$
- **credulously** accepted if $\exists E \in \sigma(F),\ a \in E$

In this paper, we will only focus on **CO**, **PR**, **GR** and **ST**.

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.

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.

Except for GR, finding extensions of these semantics is an NP problem, which means that there is probably no polynomial algorithm to solve it. That is why the ICCMA [1] has been organized since 2015 to find the best algorithms.

3 Logical encodings for abstract argumentation

ST and CO can be converted into SAT instances using the following CNF:

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

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

where A is the set of arguments and R the set of attacks [4].
Any model of φ_{ST} (resp. φ_{CO}) is a ST (resp. CO) extension and conversely. We can easily solve SE by finding one model and solve EE by enumerating all the models.

An argument is credulously accepted if it is accepted in at least one extension. So we can solve DC by adding the unit clause containing this argument and check if the CNF is SAT.

An argument is skeptically accepted if it is accepted in all the extensions, or, said otherwise, if it is rejected in none of them. So we can solve DS by adding the unit clause containing the negation of this argument and check if the CNF is UNSAT.

Regarding GR, we can always find its only extension by performing a unit propagation on φ_{CO}. Finding all the extensions is the same thing as finding one, and DC and DS are just membership tests.

Finally, PR is a slightly more complex. In fact, this semantics has (probably) no CNF equivalent of polynomial size because SAT is $\sum_1^P$ and SE-PR is $\sum_2^P$. Instead, we will first use φ_{CO} to find all the complete extensions. And from there, we can:

- return the longest one for solving SE,
- filter the ones that are subsets of others for EE,
- do EE then check if the argument belongs to one for DC,
- do EE then check if the argument belongs to all for DS.

However there is also another algorithm: it consists in first finding one CO, and then adding repeatedly a clause of all the rejected arguments until the SAT instance is UNSAT. This has the effect to add accepted arguments until it is not possible anymore.
We compared the two methods but did not find any significant difference between the two.

4 Implementation

We decided to implement the solver in Rust for two main reasons: it is fast and reliable.
We first made a library [5] to use famous SAT solvers we will then compare. It includes minisat, manysat, glucose and maplesat (we also added a naive implementation of the DPLL algorithm).
We can also use it to run several solvers in parallel as a portfolio.
Then we made the AF solver itself [6]. It has been tested with all kinds of AF so we can be confident that it is likely bug free.

4.1 The pipeline

The pipeline consisted of three main steps:

1. Parse the AF,
2. Convert it into a SAT instance,
3. Solve the SAT instance.

4.2 TGF and APX parsing

An AF can be encoded in several formats. We decided to support two of them: TGF and APX.

The TGF format is composed of two parts separated by the line "#". On the first part, every line is an argument. On the second one, every line is an attack. We could simply parse the file by reading it line by line. We also used a hash table to speed up the process of finding an argument's index from its name, as we saw in the section 4.3 that attackers are stored using their index in the arguments list.

The APX format was a bit more tricky. Each line could either be of the form

```
arg(ARG_NAME).
```

to represent an argument, or of the form

```
att(ARG_NAME1, ARG_NAME2).
```

to represent an attack. Although the APX files often store first all the arguments and then the attacks, nothing in the format definition prevents that. So we could not parse the file with a single pass since we first need to have all the arguments.
Instead we defined the APX grammar and made a canonical LR parser (LR(1)).
However, as we just said, all the files we were dealing with used the convention of storing first all the arguments, so in order to not spoil that, we also made another APX parser which assumed that the arguments must come first. We ended up only using this faster parser.

4.3 Argumentation framework Data Structure

An AF is an directed graph, so we had to use a structure to store the arguments (nodes) and the attacks (edges). We first opted for the following structure:

```
{
    arguments: List<String>,
    attacks: List<(int, int)>,
}
```

However the conversion from an AF to a SAT instance was ridiculously slow, so we analysed the CNF formulas and figured out every time the conversion was looping, it did so on the attackers of a given argument.
We can see that with the Stable CNF for example, but it is also true for the Complete one.

$$\begin{aligned}\varphi_{\mathrm{ST}} : & \bigwedge_{a_i \in A} \left(a_i \vee \bigvee_{a_j \in A, (a_j, a_i) \in R} a_j \right), \\ & \bigwedge_{a_i \in A} \left[\bigwedge_{a_j \in A, (a_j, a_i) \in R} (\neg a_i \vee \neg a_j) \right]\end{aligned} \tag{3}$$

So we had to find another data structure that would provide direct access to the list of arguments and the list of attackers of any given argument. We did so by using the following data structure:

```
List<(String, List(int))>
```

4.4 Using external SAT solvers - Foreign Function Interface

Most of the famous SAT solvers are made in C++ on top of Minisat (which is, in its own, also a SAT solver). Hopefully Rust provides a Foreign Function Interface to call C. We first made bindings from C++ to C and then bindings from C to Rust by hacking the build steps. That gave us a nice Rust interface to use the following SAT solvers:

- Minisat [7]
- Manysat [8]
- Glucose [9]
- Maplesat [10]

These solvers are by default only capable of finding one model. However we needed to be able to enumerate all the models. For that purpose, we used the following algorithm:

```
function solve_all(CNF) {
    models <- []
    while is_sat(CNF) {
        model <- solve(CNF)
        models.push(model)
        CNF.add_clause(neg_clause(model))
    }
    return models
}
```

where `neg_clause(model)` is the clause containing all the atoms of the model, but negated. This has the effect of forbidding specifically the model we have just found, and not any other. So we can repeat that operation until no new model can be found.

4.5 Portfolio

We had to be able to run several solvers in parallel in what we call ”portfolios”. To do so, we made a structure that takes solvers as parameters and returns a portfolio. When using the portfolio instead of a simple solver, it calls each individual solver in a separate thread and exits when the first solver is done.

4.6 Verbose mode

Only the last step of the pipeline, namely, the SAT instance solving, was dependent of the SAT solver. And since we only mattered in the impact of the SAT solver choice, we only wanted to compare the benchmarks of this step.
For that, we added a verbose mode that, if enabled, logged the benchmarks of each individual step of the pipeline so then when we did our experiments, we could retrieve the only time that mattered.

4.7 Unit testing

The whole code is quite complex, so we implemented unit tests to make sure it was bug free. For that purpose, we downloaded all the generated graphs from the first ICCMA [1]. These graphs were given along with the set of all the extensions for each semantics. We ran the solver for all the graphs with all the possible parameters (task, semantics, SAT solver, APX or TGF, ...) and so we can be pretty confident everything is working fine.

5 Experimental evaluation

We used Python to do our experiments as it is best suited for data analysis and data visualization due to its large choice of libraries. We used `matplotlib` for the plotting and `networkx` for the AF generations. We also reimplemented the three AF generators of the 2015 ICCMA (*gr*, *st* and *scc*).

5.1 Protocol

The protocol has the purpose to find out if one of our SAT solvers is better (or worse) than the other ones depending on the AF. If it is the case, we will only run the faster one (or a portfolio of the fastest ones) using an heuristic.

The first step of the protocol is to generate AF by varying a specific characteristic of the generator. For example, we can use a complete AF generator and vary the number of nodes. Then we can process all theses AF with all our solvers and we plot the curve with the characteristic as the x-coordinate and the time as the y-coordinate. Finally, we manually compare the times spent between the different solvers.

For the DC and DS tasks, we had to choose the argument to run the solver. However the result would potentially be very different whether the argument is accepted or rejected. Therefore we decided to do a preprocessing to find one accepted argument and one rejected argument. We then run the solver with each and compute the average time of the two.

While some problems were rather fast, others would be too long for us to be computed. To address that issue, we set up a timeout to 10 minutes, and if a problem happened to timeout, we assume it would have taken 10 times longer (i.e. 100 minutes). This is called the Penalized Average Runtime 10 (PAR10).

We did not have access to very powerful computers, but this was not crucial since we are only doing comparisons here. However, the computers can have performance drops at random points in time due to background processes. So in order to minimise this noise, we used the two following techniques:

- we execute each task in a random order,
- we execute several times each task and compute the average time.

We tried the same experiments several times with and without these methods, and we have indeed concluded that they reduced the noise.

5.2 Benchmark generation

We generated benchmarks for all the solvers, all the tasks and all the semantics for the following AF types:

1. Random internet, varying the number of nodes from 1,000 to 10,000 with a step of 100,
2. Erdos Renyi, varying the probability of attack from 0 to 1 with a step of 0.2, fixing the number of nodes to 1,000,
3. Watts Strogatz, varying the k nearest neighbour from 1,000 to 3,000 with a step of 100, fixing the number of nodes to 3,000 and the probability of rewiring each edge to 0.2.

5.3 Histograms

We also ran the solver with the AF from the 2015 ICCMA. Since we did not have any characteristic we could smoothly vary to plot a curve, we instead plotted histograms.

5.4 Results

First, let's take a look at the results obtained for the problem EE-ST with random internet graphs. The varying characteristic here is the number of nodes.

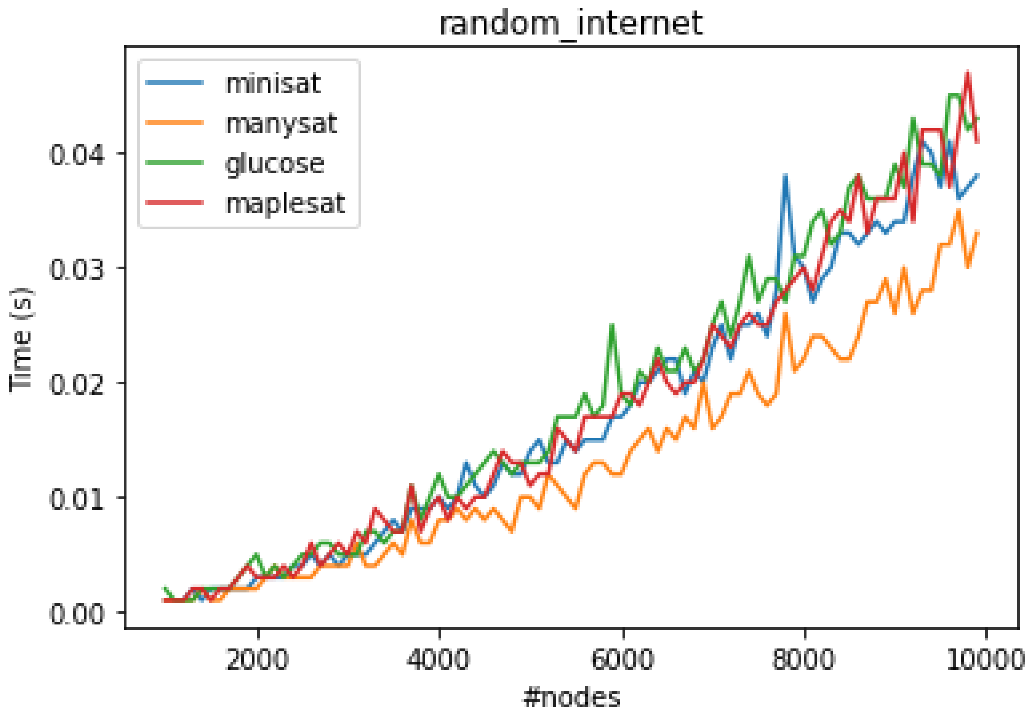

Figure 1: random_internet_EE-ST

Manysat is clearly the winner here, and tends to get better as the number of nodes increases. However there is no solver that seems to be better (or worse) depending on the number of nodes.

When the varying characteristic is too simple like the number of nodes, it does probably not change the AF structure enough. So instead, let's observe the results obtained for the problem DC-PR with Erdos Renyi graphs where the varying characteristic is the probability of having an attack between two arguments.

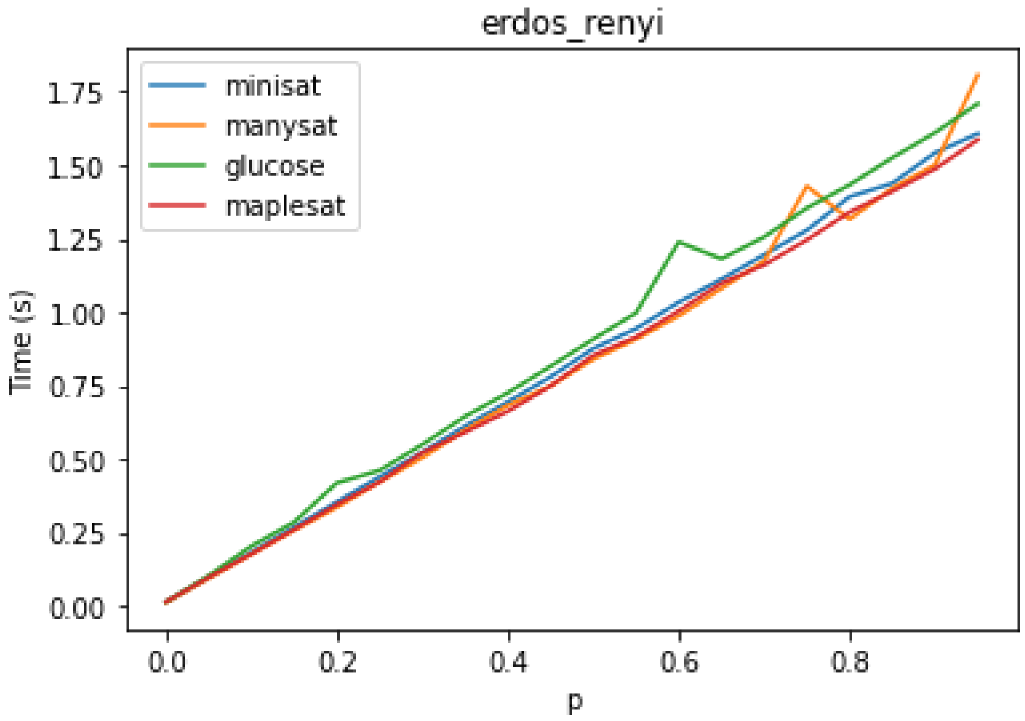

Figure 2: DC-PR with Erdos Renyi

Once again, this does not give us any useful information.
This is also the case of all the other benchmarks we generated that can be found in the appendix (citation needed).

Regarding the histograms, manysat tends to be the fastest solver on most of the configurations. In Figure 3, the AF are generated with the *gr* algorithm from the ICCMA:

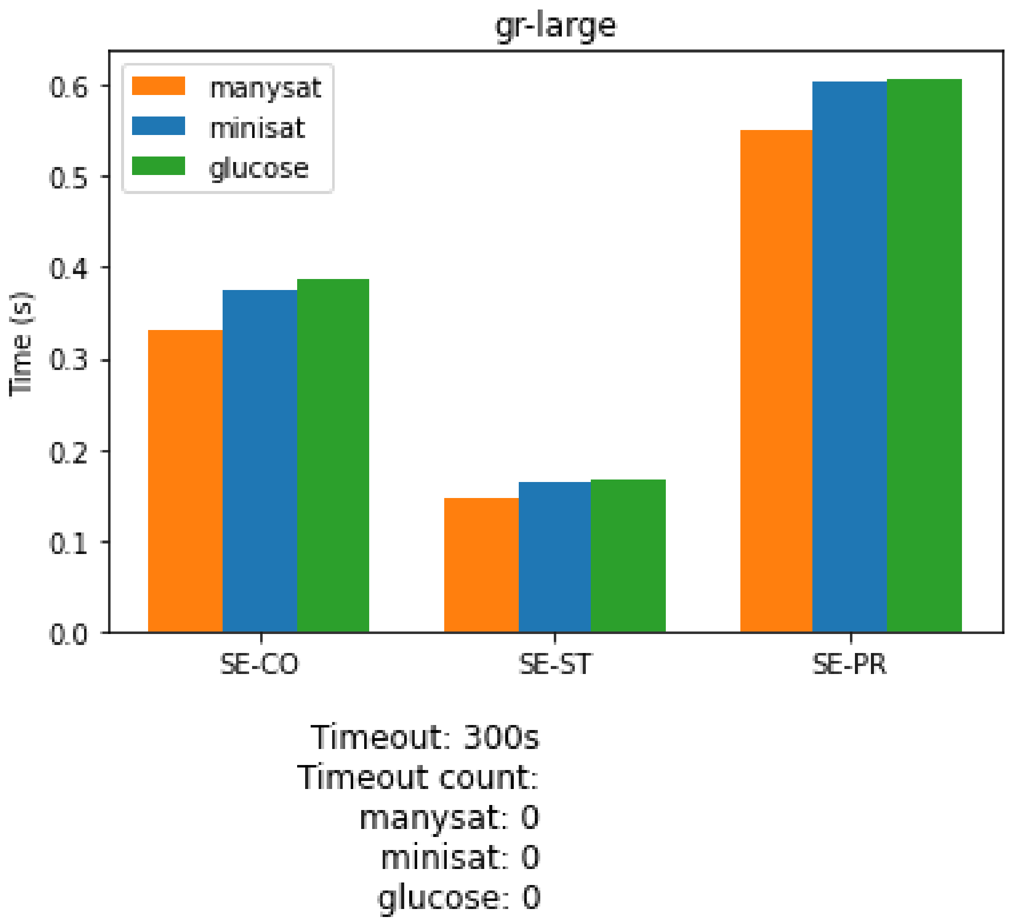

Figure 3: Scenario where manysat is faster

However, the best solver is sometimes glucose in specific scenarios. In Figure 4, the AF are generated with the *scc* algorithm from the ICCMA:

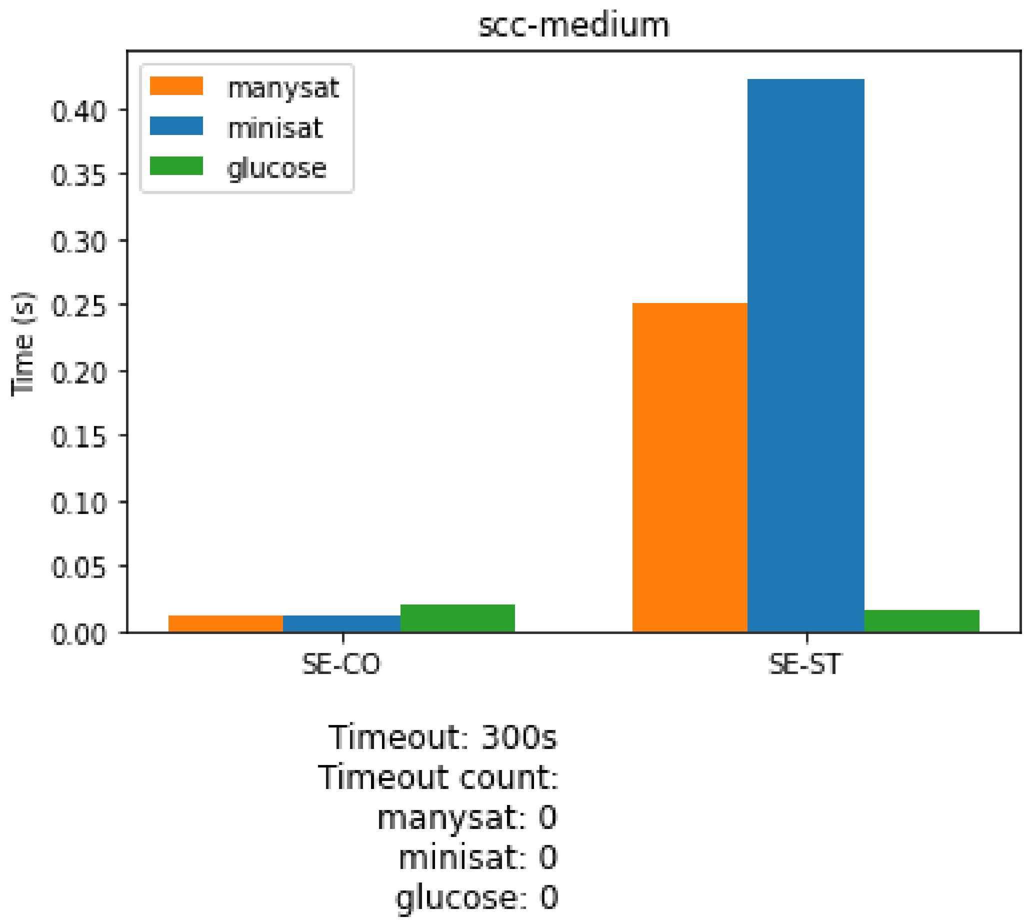

Figure 4: Scenario where glucose is faster

However a further analysis would be required to determine what elements in these algorithms are responsible for this difference.

6 Conclusion

As we saw, we could not identify a graph characteristic that makes a given SAT solver sometimes better, sometimes worse than the other ones.

However we managed to build a useful set of tools for doing further researches and exploration. The project has been constantly designed to be able to support new implementations. Indeed, new SAT solvers can easily be implemented, as long with new experimentations.

We had to stop the experiments here due to time constraints but as we said in the Introduction, it is already known the choice of the SAT solver has an impact on the AF solver performances [2, 11]. So we hope these researches will be continued in order to identify such characteristics.

References

[1] "The international competition on computational models of argumentation." `http://argumentationcompetition.org/`.

[2] S. Gning and J.-G. Mailly, "On the impact of SAT solvers on argumentation solvers," in *3rd International Workshop on Systems and Algorithms for Formal Argumentation co-located with the 8th International Conference on Computational Models of Argument (SAFA@COMMA'20)*, pp. 68–73, 2020.

[3] P. M. Dung, "On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games," *Artificial intelligence*, vol. 77, no. 2, pp. 321–357, 1995.

[4] J.-M. Lagniez, E. Lonca, and J.-G. Mailly, "Coquiaas: A constraint-based quick abstract argumentation solver," in *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 928–935, IEEE, 2015.

[5] "Our rust sat solver library." `https://github.com/QuentinJanuel/sat-portfolio`.

[6] "Our af solver." `https://github.com/QuentinJanuel/TER`.

[7] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*, pp. 502–518, Springer, 2003.

[8] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: solver description," Tech. Rep. MSR-TR-2008-83, May 2008.

[9] "The glucose official website." `https://www.labri.fr/perso/lsimon/glucose/`.

[10] "The maplesat official website." `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`.

[11] J. Klein and M. Thimm, "Revisiting SAT techniques for abstract argumentation," in *Computational Models of Argument - Proceedings of COMMA 2020, Perugia, Italy, September 4-11, 2020* (H. Prakken, S. Bistarelli, F. Santini, and C. Taticchi, eds.), vol. 326 of *Frontiers in Artificial Intelligence and Applications*, pp. 251–262, IOS Press, 2020.