SAT solvers

Computational Models of Argumentation

Half project report

Christophe Yang
Quentin Januel
Sylvain Declercq

2022

1 Implementation details

We have chosen to implement the solver in Rust. It is composed of two main parts.
The first one allows us to interact with several SAT solvers: it provides bindings famous C++ solvers such as minisat, manysat and glucose. We also added in our own SAT solver based on the DPLL algorithm, mainly for comparison purposes.
The second part uses the first one to solve computational problems. It first parses the input file as an argumentation framework, then it converts it to a suitable CNF formula, and finally it solves it using a SAT solver.
We made sure to follow the ICCMA solver interface[1].

2 Speed improvements

While the considered hard part of the solver is the SAT part, we had some speed issues on parts supposed to be fast.

2.1 Parsing

Our solver can parse argumentation frameworks in both the CNF and the Aspartix format. While the former is really efficient, the latter cannot be as fast because we the arguments and attacks declarations can be mixed. However we noticed it is a convention to always have the arguments at first and the attacks at the end of the file. If we know the file fulfils this convention, we can parse it much faster. So that's what we did, but we still kept the slower option just in case, which we named "loose-apx".

2.2 CNF generation

We could drastically improve the CNF generation by changing the data structure used to store the argumentation framework graph. At first, we represented them as a list of arguments and a list of attacks. Instead we now use a list of arguments with its attackers. This way, we can generate the CNF in a single pass.

3 SAT problem conversion

In order to convert an argumentation framework problem to a SAT instance, we used the formulas as described in Coquiaas.[2]

4 Glucose

The glucose solver appeared to be very slow, which was at first quite surprising since it was even smaller than minisat. We understood that's because glucose is doing a lot of preprocessing to have a better chance to solve really hard SAT instances in a reasonable amount of time. However most of the time, this preprocessing just makes the solver too slow.
That's why we have two versions of glucose, one with the preprocessing enabled and the other one without it. Thus, we can analyze in which cases the preprocessing should be enabled.

5 Portfolio

We also can start several solvers at the same time using threads. Thus if we don't know what solver to use, we can run all of them and stop when one of them finds a solution.

6 Frameworks generation

In the purpose of helping us with the experimentations, we reimplemented the three generators from the 2015 ICCMA[3], namely **gr**, **st** and **scc**.

7 ICCMA graphs

We also downloaded the graphs from the 2015 ICCMA. The first purpose of this was to use the graphs for benchmarking. But since they came with solutions, we also used them for unit testing. Our solver seems bug free, so we can start the experiments.

8 Experimentations protocol

Several factors can make the execution time vary. In order to avoid any bias, we shuffled the benchmark tests such that if the computer slows down at any time it wouldn't favor any particular solver. We also just mesure the SAT solving time instead of the whole time because the rest should be constant no matter the solver.

References

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

[2] 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.

[3] S. V. Matthias Thimm, "The first international competition on computational models of argumentation: Results and analysis," in *Artificial Intelligence Volume 252*, pp. 267–294, Elsevier, 2017.