EqArgSolver – System Description

Odinaldo Rodrigues

Department of Informatics,
King's College London,
The Strand, London, WC2R 2LS, UK
odinaldo.rodrigues@kcl.ac.uk

Abstract. This document provides an outline of EqArgSolver, a solver for enumeration and decision problems in argumentation theory. EqArgSolver is implemented from scratch as a self-contained application in C++ without the use of any other external solver (e.g., SAT, ASP, CSP) or libraries. Even though the solver uses the discrete version of the Gabbay-Rodrigues Iteration Schema [4] for the propagation of labels through an argumentation network, all operations are performed directly on the graph and thus EqArgSolver uses a *direct approach* to argumentation problems in the sense of [2].

1 Introduction

EqArgSolver is a computer application that can be used to solve the following *enumeration* and *decision* problems in argumentation theory: *i)* Given an argumentation network $\langle S, R \rangle$, to produce one or all of the extensions of the network under the grounded, complete, preferred or stable semantics; and *ii)* Given an argument $X \in S$, to decide whether X is accepted credulously or sceptically according to one of these semantics. Problems to EqArgSolver must be submitted following `probo`'s syntax (see [1]).

EqArgSolver builds and expands on the prototype GRIS [8] submitted to the 1st International Competition on Computational Models of Argumentation (`http://argumentationcompetition.org/2015/`), but includes two technical advances that result in significant improvements in performance and functionality. Firstly, EqArgSolver uses the discrete version of the Gabbay-Rodrigues Iteration Schema (dGR-iteration schema) [4], which can be implemented in a much more efficient way than its full-fledged counterpart [3]. Secondly, the component in GRIS responsible for computing preferred extensions (and based on Caminada and Modgil's algorithm for the computation of preferred labellings [6]) has been replaced by a novel algorithm that can compute *all* complete extensions and hence EqArgSolver is now also able to handle the complete and preferred semantics. An ad hoc mechanism for checking whether a preferred extension is also stable enhances EqArgSolver's functionality further allowing the solver to handle decision and enumeration problems in the grounded, complete, preferred and stable semantics.

EqArgSolver computes extensions by labelling arguments as accepted (**in**), rejected (**out**), or undecided (**und**). However, instead of the labels **in**, **out**, and

und, EqArgSolver uses the numerical values $\{0 = \textbf{out}, 1 = \textbf{und}, 2 = \textbf{in}\}$ and hence all labelling manipulations are done in terms of integer operations, which can be implemented very efficiently. Without loss of generality, we will refer to the values as labels.

The solver initially decomposes the input argumentation network into strongly connected components (SCCs) and arranges the components into layers as described in [5]. Each layer is then processed in succession. The dGR-iteration schema is employed in what we call a *grounding* module: the module propagates a (base) solution $\boldsymbol{f}$ for arguments in layers up to $k-1$ under a particular semantics to the nodes of a SCC in layer k. Provided $\boldsymbol{f}$ corresponds to a legal assignment, the result of the propagation to the arguments in the SCC will also be legal. However, the grounding process may leave some arguments in the SCC with label **und**. These undecided arguments could have been labelled **in** or **out** in some (larger) complete extension and it is this part of the process that is carried out by the new proposed algorithm. The partial solutions thus obtained are then combined to compute all extensions of the network as a whole using what Liao calls *horizontal and vertical combinations of partial solutions* [5].

2 System Overview

EqArgSolver initially reads the problem specification passed as command line arguments, validates it and then validates the input graph itself, exiting in case of error.

The basic workflow of the computation is depicted in Fig. 1. The solver starts by computing the SCCs of the network using a slightly adapted version of Tarjan's algorithm [7] and arranging them into layers that can be used in successive computation steps similar to what is described in [5]. Once the network is arranged into layers, in decision problems the solver can identify at what layer the computation can be terminated according to the depth of the input argument. Hence layers are successively processed until the maximum depth needed to establish the solution to the problem submitted is reached. This strategy proves particularly efficient in decision problems where the argument in question belongs to a layer of low depth. A number of other "shortcuts" allowing early termination are employed according to the semantics of the problem at hand.

The computation of the solutions to the problems in the grounded semantics does not require the decomposition of the network into layers. In principle, the dGR-iteration schema can be applied to the entire network to produce its grounded extension upon completion. However, since the decomposition of the network into SCCs and their arrangement into layers can be performed very efficiently, the extra decomposition cost is offset by performance gains obtained through the computation by layers in all but a few special cases, and is therefore our preferred choice for all semantics. It is technically possible to bypass the decomposition *and* compute the grounded extension more efficiently by simply excluding from the computation some nodes as soon as their labels converge. However, the computation of the grounded semantics is so efficient as it is, that we decided not to optimise it further in this version.

Generating All Complete Extensions. As we mentioned, the grounding module may leave some nodes with label **und** which in a larger extension could potentially be labelled **in**. Our algorithm attempts to label **in** all such undecided nodes, propagating the results as required. When this is employed judiciously, it generates all complete extensions. A careful analysis of the solutions generated can then be employed to identify the extensions that are preferred and stable.

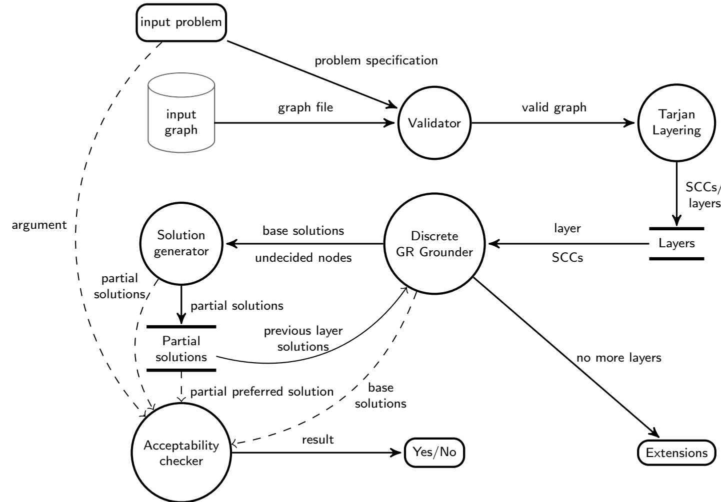

Fig. 1. EqArgSolver's basic workflow.

EqArgSolver's URL: `http://nms.kcl.ac.uk/odinaldo.rodrigues/eqargsolver`

3 Functionality and Design Choices

As mentioned in Section 1, EqArgSolver can tackle enumeration and decision problems (skeptical and credulous) of the following argumentation semantics: grounded, complete, preferred and stable. In addition, EqArgSolver can also provide solutions for *Dung's Triathlon*, i.e., to compute in succession the grounded extension, all stable extensions, and all preferred extensions of an argumentation network $\langle S, R \rangle$.

In terms of input graph format, EqArgSolver only accepts the *trivial graph format*, which is basically a text file containing a sequence of node designators one per line, followed by the separator "#" in its own line, and then followed by a list of pairs of nodes, a pair per line, where the first element of each pair is the identifier of the source node of an edge in the graph (the attacker) and the second element is the identifier of its target node (the node being attacked).

In an abstract way, one can think of EqArgSolver's internal graph representation as an enhanced adjacency list. The extra bits in the list provide information to speed up a number of operations that are used frequently in the computation of solutions. More specifically, each argument is assigned an internal identifier which is an unsigned integer. Each internal identifier is then associated with the following data structure:

```
struct
ArgNode_T {
  int layer;
  ExtArgId_T extArgId;
  vector<IntArgId_T> attsIn;
  vector<IntArgId_T> attsOut;
};
```

where `layer` gives the graph layer to which the associated argument belongs; `extArgId` gives the external argument identifier (the string given in the input file); and `attsIn` and `attsOut` give, respectively, the list of incoming attacks and the list of outgoing attacks of the argument.

Each internal node identifier is associated with a corresponding structure of the above type using an associative container (a C++ `unordered_map`). For efficiency, a second associative container is created using the external node identifier as key and having the internal node identifier as value.

In order to avoid resizing of the container, which in large graphs can be very inefficient, EqArgSolver looks ahead at the number of nodes in the graph so that a sufficient number of buckets in the hash map is allocated before the graph is actually created. This ensures that even graphs of up to 100,000 nodes can be created in a few seconds.

A number of further improvements can be made to EqArgSolver. In tests, we have identified a number of randomly generated graphs that proved particularly difficult to handle. Work is under way to refine the complete extension generator algorithm further.

References

1. F. Cerutti, N. Oren, H. Strasse, M. Thimm, and M. Vallati. The first international competition on computational models of argumentation (ICCMA15). `http://argumentationcompetition.org/2015/index.html`, 2015.
2. G. Charwat, W. Dvořák, S. A. Gaggl, J. P. Wallner, and S. Woltran. Methods for solving reasoning problems in abstract argumentation – A survey. *Artificial Intelligence*, 220:28 – 63, 2015.
3. D. M. Gabbay and O. Rodrigues. Equilibrium states in numerical argumentation networks. *Logica Universalis*, pages 1–63, 2015.
4. D. M. Gabbay and O. Rodrigues. Further applications of the Gabbay-Rodrigues iteration schema in argumentation and revision theories. In C. Beierle, G. Brewka, and M. Thimm, editors, *Computational Models of Rationality*, volume 29, pages 392–407. College Publications, 2016.
5. B. Liao. *Efficient Computation of Argumentation Semantics*. Elsevier, 2014.
6. S. Modgil and M. Caminada. Proof theories and algorithms for abstract argumentation frameworks. In Guillermo Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 105–129. Springer US, 2009.
7. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.
8. M. Thimm and S. Villata. System descriptions of the first international competition on computational models of argumentation (ICCMA'15). *CoRR*, abs/1510.05373, 2015.