EqArgSolver 2019 – System Description

Odinaldo Rodrigues

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

1 Introduction and System Overview

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 those semantics. These tasks are commonly denoted by SE-σ, EE-σ, DC-σ and DS-σ for $\sigma \in$ {grounded,complete,preferred,stable}.[1]

The solver follows the general process of computation described in [2], which requires the decomposition of the framework into SCCs and the arrangement of these into layers following the direction of attacks. A more comprehensive description can be found in [5]. Since the last ICCMA (in 2017), EqArgSolver has gone through a few improvements that we describe in this paper.

EqArgSolver uses `probo`'s syntax [1]. The problem specification is fully validated before the computation proceeds. Algorithm 1 gives a high-level overview of this computation process. Some shortcuts allowing early termination are also employed but not described here (see [5] for more details).

The framework is first divided into SCCs and arranged into layers according to Fig. 1 (line 3). The starting point is an initial partial solution labelling all arguments as undecided (all-**und**, line 4). The solutions to each layer expand on the previous layers' solutions to include the new labelling assignments for the layer's arguments.

The composition of a typical layer is shown in Fig. 1. It consists of a block of trivial SCCs that are mutually dependent and operated on in one step, and a set of non-trivial SCCs that are independent from each other (and hence can potentially be computed in parallel). Before working on a layer, each partial solution generated for the preceding layer is propagated to the layer's SCCs in order to *condition* its arguments' labels — a process that we call *grounding with a solution*. Grounding will fully and uniquely determine the labels of the arguments in the trivial SCC block (line 9) but will only provide a minimal solution for the arguments in the non-trivial SCCs. As it turns out, some of the arguments left undecided in these SCCs by the grounding could potentially be labelled

[1] The grounded semantics has exactly one extension so SE and EE, and DC and DS are equivalent, and only SE and DC are referred.

in in a larger extension (line 12). An algorithm based on the one proposed in [3] is employed to systematically attempt to include all such arguments in an extension. This will generate all solutions associated with complete extensions for the SCC (line 13). These solutions are only partial to the argumentation framework as a whole and are "horizontally" and "vertically" combined, i.e., with solutions for other SCCs within the layer and solutions for previous layers, respectively (see [2]). This is done is lines 14 and 16 of Algorithm 1, respectively.[2]

This process is repeated until all relevant layers are processed. The resulting solutions are then output as extensions by simplying ignoring the arguments with **out** or **und** labels (line 20).

Strictly speaking, the computation of the solutions to the problems in the grounded semantics does not require the decomposition of the framework into layers. However, since the decomposition of the framework into SCCs and their arrangement into layers can be performed very efficiently, the extra decomposition cost is offset by the performance gain obtained through the computation by layers in all but a few special cases, and is therefore our preferred choice for all semantics. Further optimisation here is possible but left as future work.

```
Input: Graph $G$
Output: Extensions of $G$
1  EqArgSolver
2      Read and validate graph $G$
3      Decompose $G$ into SCCs and arrange them into layers $L_0, L_1, \ldots, L_{k-1}$
4      $Sols \leftarrow \{$all-und$\}$
5      for $i \leftarrow 0$ to $k-1$ do                    /* Iterate through layers */
6          $newSols \leftarrow \varnothing$
7          foreach $f \in Sols$ do
8              $\lambda \leftarrow$ GR-ground($L_i, f$); $TSB \leftarrow$ trivial SCC block of $L_i$
9              $LayerSols \leftarrow \{\lambda \downarrow TSB\}$
10             $\mathcal{S} \leftarrow$ non-trivial SCCs in $L_i$
11             foreach $S \in \mathcal{S}$ do
12                 $possIns \leftarrow$ candidate in-nodes of $S$ according to $\lambda$
13                 $SCC$-sols $\leftarrow$ findExtsFromArgs($possIns, S, f, \lambda \downarrow S$)
14                 Horizontally combine $SCC$-sols with solutions in $LayerSols$
15             end foreach
16             Add vertical combination of $f$ with each $\gamma \in LayerSols$ to $newSols$
17         end foreach
18         Sols $\leftarrow newSols$
19     end for
20     Output $Sols$
21 end
```

Algorithm 1: EqArgSolver's overall processing structure.

[2] There are particular variations for the preferred and stable semantics not shown in Algorithm 1.

2 Functionality and Design Choices

EqArgSolver accepts input graphs using the *trivial graph format*, which consists of a text file where arguments are provided as a sequence of argument designators, one per line, followed by the separator "#" in its own line, followed by a list of pairs of argument designators, one pair per line, where the first element of the pair is the attacking argument and the second element is the attacked one.

Each argument in EqArgSolver is assigned an internal identifier (an unsigned integer) and the argumentation graph is represented internally as a type of enhanced adjacency list where each argument node contains vectors `attsIn` and `attsOut` giving, respectively, the list of incoming and outgoing attacks of the argument.

3 General Improvements Since 2017

The 2019 version offers a number of improvements over the one submitted in 2017:

1. Inefficient behaviour in some exceptional conditions were identified by analysing the results in the 2017 competition. Alternative solutions able to deal with these cases were implemented.
2. The overall implementation has moved towards a more object-oriented approach, with graphs and solutions encapsulated within class objects. This improves readability and software maintenance.
3. Each argument's internal identifier is now determined by the argument's position within the framework's topological structure. The numbers in bold in Fig. 1 give an example. This means the graph has to be first analysed before the argument's internal identifers are defined, but it greatly facilitates the representation of solutions (see next point).
4. Until 2017, solutions were represented using C++'s associative container `unordered_map` to map argument identifers to the labels **in**, **out**, and **und**. Due to the growing cardinality of the enumerations of the problems since the first competition, the use of `unordered_map` became very inefficient in terms of memory requirements [4]. For this reason, solutions in the 2019 version are now represented as simple `vectors` of unsigned integers instead. Although this seems a trivial modification, its implementation requires a careful design due to the modular nature by which the solutions are computed. The vertical and horizontal combinations must ensure that all solutions are precisely juxtaposed. Due to the canonical ordering of all arguments employed, which as mentioned above satisfies the natural topological order of the framework, the position of the solution of each SCC within a "global" solution vector is uniquely determined (see Fig. 1).

4 Docker Repository

The solver is available from the docker respository: `odinaldo/eqargsolver`.

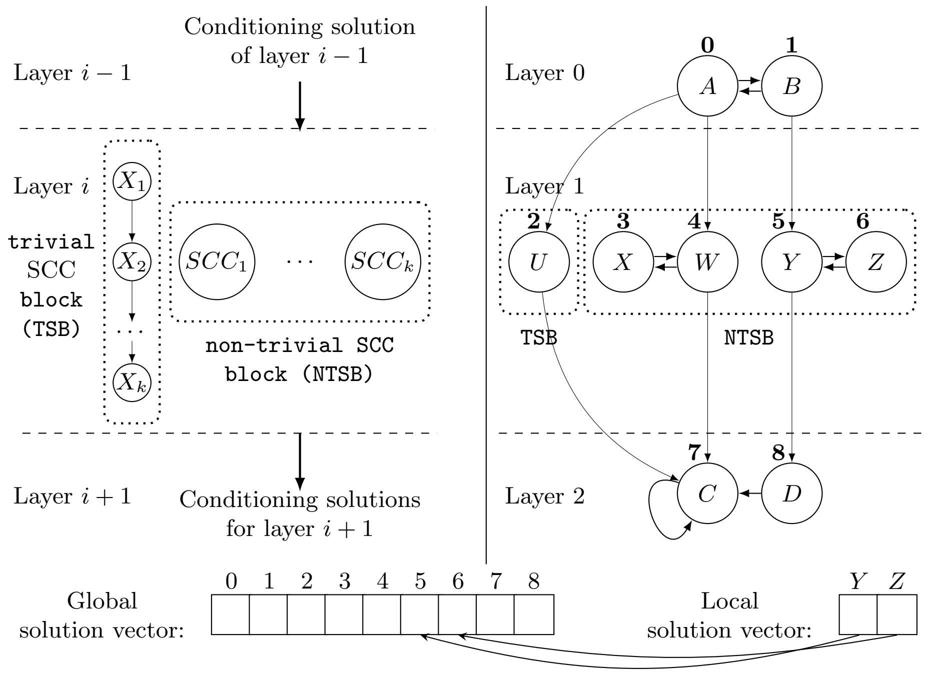

Fig. 1. Relationship between partial solutions and layers of the argumentation framework.

References

[1] F. Cerutti, N. Oren, H. Strasse, M. Thimm, and M. Vallati. The First International Competition on Computational Models of Argumentation (ICCMA'15). 2015. `http://argumentationcompetition.org/2015/index.html`.

[2] B. Liao. *Efficient Computation of Argumentation Semantics*. Academic Press, 2014.

[3] O. Rodrigues. A forward propagation algorithm for semantic computation of argumentation frameworks. In E. Black, S. Modgil, and N. Oren, editors, *Theory and Applications of Formal Argumentation*, Melbourne, Australia, 2017. Springer.

[4] O. Rodrigues. Representing and comparing large sets of extensions of abstract argumentation frameworks. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC'19)*, 2019.

[5] Odinaldo Rodrigues. Eqargsolver - system description. In *Proceedings of the 4th Intl. Conference on Theory and Applications of Formal Argumentation*, pages 150–158, 2018.