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 paper provides a general overview of EqArgSolver, a solver for enumeration and decision problems in argumentation theory. The solver is implemented from the ground up as a self-contained application in C++ without the use of any other external solver (e.g., SAT, ASP, CSP) or libraries.

1 Introduction

EqArgSolver is a computer application that can be used to solve *enumeration* and *decision* problems in argumentation theory. EqArgSolver builds and expands on the prototype GRIS [8] submitted to the 1st International Competition on Computational Models of Argumentation (ICCMA, [1]). It includes two technical advances that result in significant improvements in performance [7] 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preferred extensions (and based on Modgil and Caminada's algorithm for the computation of preferred labellings [6]) has been replaced by a novel and more efficient algorithm [7] that can compute *all* complete extensions. This allows EqArgSolver to handle the following two types of problems: *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.

The solver follows the general process of computation described in [5], which requires the decomposition of the framework into SCCs and the arrangement of these into layers following the direction of attacks between the arguments.

The dGR-iteration schema is employed in what we call a *grounding* module that propagates a (conditioning) solution $\boldsymbol{f}$ (under a particular semantics) to an attacked SCC in a subsequent layer. Provided $\boldsymbol{f}$ is a legal assignment, i.e., an assignment in which all arguments are correctly labelled, the result of the propagation will also be legal. The numerical computations of the dGR-iteration schema are optimised by using the integer values $0 =$ **out** (rejected), $1 =$ **und** (undecided), and $2 =$ **in** (accepted). It is worth emphasising, that although the dGR-iteration schema is employed in these computational tasks, EqArgSolver actually uses a *direct approach* to the problem (in the sense of [2]), i.e., argumentation problems are solved via operations performed directly on the graph. We will refer to the labels 0/**out**, 1/**und** and 2/**in** interchangeably.

The newly proposed algorithm [7] ensures that all arguments left undecided by the propagation of solutions to SCCs are systematically tried for inclusion in some extension (this is explained in more detail in Section 2).

EqArgSolver has been submitted to the 2nd iteration of ICCMA, whose results will be announced at the 2017 International Workshop on Theory and Applications of Formal Argument (TAFA-17).

2 System Overview

EqArgSolver accepts problems submitted according to `probo`'s syntax (see [1]). The problem specification is fully validated before the computation proceeds.

Algorithm 1 gives a high-level overview of this computation, which we now briefly describe. Some shortcuts allowing early termination are omitted for space limitations. The network is first divided into SCCs and arranged into layers. 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. 2 (L). 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. 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 argument values – a process we call *grounding*. Grounding will fully determine all of the values of the arguments in the trivial block (line 9) but not all of the values of the arguments in the non-trivial SCCs. Some of these arguments will be left undecided although they could potentially be labelled **in** in a larger extension (line 12). A newly proposed algorithm [7] ensures that all such arguments are systematically tried for inclusion generating a number of partial solutions for the SCC (line 13). The partial solutions thus obtained[1] are then combined using what Liao calls the *horizontal and vertical combinations of partial solutions* [5] (lines 14 and 16, respectively). This process is repeated until all relevant layers are processed.

Strictly speaking, the dGR-iteration schema can be applied to the entire argumentation framework (without decomposition) to compute the grounded extension. 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. Further optimisation here is possible but left as future work.

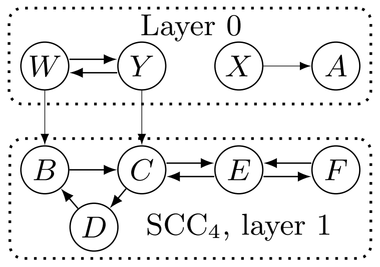
How Grounding Works. Propagation of the conditioning values of a solution is done using the dGR-iteration schema, whose behaviour we can only outline

[1] Some filtering to eliminate solutions not leading to maximal extensions in preferred/stable semantics problems is also done, although this is not shown in Algorithm 1. For full details, refer to [7].

due to space limitations. Each node $X \in S$ gets an equation describing its value at iteration $i+1$ ($V_{i+1}(X)$) based on the nodes' values at iteration i. All nodes get initial value **und** (i.e., $V_0(X) = 1$, for all $X \in S$). Let $Att(X)$ denote the attackers of X. The general form of the equations is $V_{i+1}(X) = 2 - \max_{Y \in Att(X)}\{V_i(Y)\}$. The sequence of values of all nodes will converge in linear time producing a legal assignment that corresponds to an extension (see [4] for details). Fig. 1, depicts a sample argumentation framework, its associated system of equations, and the behaviour of the schema in several grounding scenarios until convergence is achieved. In layer 0 the schema produces the solution $X = 2$, $A = 0$, and $W, Y = 1$. It is easy to see that besides X, arguments W and Y could also be included in (distinct) complete extensions. The new algorithm is invoked at this point considering all candidate arguments that could potentially be labelled **in**. This imposes some constraints on any additional candidate partial solutions. In this example, it will produce the two remaining partial solutions to layer 0: $W = 2, Y = 0$ and $W = 0, Y = 2$. Propagating layer 0's three conditioning solutions to layer 1 is done again by grounding SCC$_4$ with the solutions. The result of these groundings can be seen in Fig. 1 (R). It produces extensions $\{X\}$, $\{X, W\}$ and $\{X, Y, D\}$.

Generating All Complete Extensions. As mentioned, the grounding module may leave some nodes with label **und** which could potentially be labelled **in** yielding a larger extension (e.g., nodes W, Y in layer 0 or the nodes in layer 1 of Fig. 1). Our algorithm attempts to label **in** all such undecided nodes, propagating the results as required. When this is employed judiciously, it not only generates all remaining complete extensions, but also offers significant performance gains because the legal labelling of an argument imposes constraints that help prune the search space of feasible solutions. Full details are given in [7].

Argumentation Framework:



Equations:

$V_{i+1}(X) = 2$

$V_{i+1}(A) = 2 - V_i(X)$

$V_{i+1}(W) = 2 - V_i(Y)$

$V_{i+1}(Y) = 2 - V_i(W)$

$V_{i+1}(B) = 2 - \max\{V_i(W), V_i(D)\}$

$V_{i+1}(C) = 2 - \max\{V_i(B), V_i(Y), V_i(E)\}$

$V_{i+1}(E) = 2 - \max\{V_i(C), V_i(F)\}$

$V_{i+1}(D) = 2 - \max\{V_i(C)\}$

$V_{i+1}(F) = 2 - \max\{V_i(E)\}$

Results of grounding:

Layer 0

	X	A	W	Y
V_0	1	1	1	1
V_1	2	1	1	1
V_2	2	0	1	1
	in	**out**	**und**	**und**

Layer 1, sol $W = 1$, $Y = 1$

	B	C	D	E	F
V_0	1	1	1	1	1
	und	**und**	**und**	**und**	**und**

Layer 1, sol $W = 2$, $Y = 0$

	B	C	D	E	F
V_0	1	1	1	1	1
V_1	0	1	1	1	1
	out	**und**	**und**	**und**	**und**

Layer 1, sol $W = 0$, $Y = 2$

	B	C	D	E	F
V_0	1	1	1	1	1
V_1	1	0	1	1	1
V_2	1	0	2	1	1
V_3	0	0	2	1	1
	out	**out**	**in**	**und**	**und**

Fig. 1. Examples of grounding invocations.

```
EqArgSolver
    Read and validate graph $G$
    Decompose $G$ into SCCs and arrange them into layers $\mathcal{L} = \{L_0, \dots, L_{k-1}\}$
    $Sols \leftarrow \{$all-und$\}$
    for $i \leftarrow 0$ to $k-1$ do                    /* Iterate through layers */
        $newSols \leftarrow \varnothing$
        foreach $f \in Sols$ do
            $\lambda \leftarrow$ GR-ground($L_i, f$); $TSB \leftarrow$ trivial SCC block of $L_i$
            $LayerSols \leftarrow \{\lambda \downarrow TSB\}$
            $\mathcal{S} \leftarrow$ non-trivial SCCs in $L_i$
            foreach $S \in \mathcal{S}$ do
                $possIns \leftarrow$ candidate in-nodes of $S$ according to $\lambda$
                $SCC$-sols$\leftarrow$findExtsFromArgs($possIns, S, f, \lambda \downarrow S$)
                Horizontally combine $SCC$-sols with solutions in $LayerSols$
            end foreach
            Add vertical combination of $f$ with each $\gamma \in LayerSols$ to $newSols$
        end foreach
        Sols $\leftarrow newSols$
    end for
end
```

Algorithm 1: EqArgSolver's overall processing sequence.

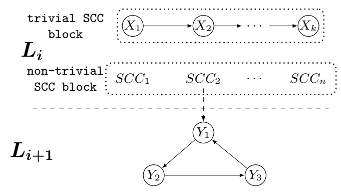
3 Functionality and Design Choices

As we mentioned, EqArgSolver can tackle enumeration and decision problems (sceptical and credulous) of the grounded, complete, preferred and stable semantics. EqArgSolver can also provide solutions for the *Dung's Triathlon*, i.e., to compute in sequence the grounded extension, all stable extensions, and all preferred extensions of an argumentation framework. Graphs must be supplied as a *trivial graph format* text file, consisting of 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, a pair per line, where the first element of the pair is the attacking argument and the second element is the attacked argument.

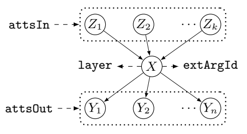
Each argument in EqArgSolver is assigned an internal identifier (an unsigned integer) and the argumentation graph is represented internally as an enhanced adjacency list. The argument data structure used is shown in Fig. 2 (C) and Fig. 2 (R). `layer` is the graph layer assigned by the decomposition; `extArgId` is the external argument identifier (the string given in the graph's input file); and `attsIn` and `attsOut` give, respectively, the list of incoming and outgoing attacks of the argument. This data structure is associated with the internal argument identifier using C++'s associative container `unordered_map`.

Storing both directions of the attack relation in the vectors `attsIn` and `attsOut` makes it more efficient to traverse the graph as needed. Similarly, a second associative container is created using the external node identifier as key and the internal node identifier as value (this is useful in decision problems). A (partial) solution is just a mapping from node identifiers to unsigned integers.

Layer structure | Node definition | Node structure



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



(L) (C) (R)

Fig. 2. Data representation in EqArgSolver

In order to avoid resizing of the associative container at creation time (which in large graphs can be very inefficient), EqArgSolver looks ahead in the input graph file to count the total number of arguments. It then creates a hash map with a sufficiently large number of buckets to represent the entire graph. This ensures that even graphs with many tens of thousands of nodes can be created in a just a few seconds.

Many further improvements can still be made to EqArgSolver. In tests, we have identified a number of randomly generated graphs with problem instances that were particularly difficult to solve. Work is under way to understand why these graphs are challenging and to refine the complete extension generator algorithm further to avoid the multiple generation of the same solution in different search branches.

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] 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] 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.

[8] M. Thimm and S. Villata. System descriptions of the 1st International Competition on Computational Models of Argumentation (ICCMA'15). *CoRR*, abs/1510.05373, 2015.