An Investigation into Reduction and Direct Approaches to the Computation of Argumentation Semantics

Odinaldo Rodrigues[1]

King's College London, Department of Informatics, odinaldo.rodrigues@kcl.ac.uk

Abstract. This paper compares the performance of the forward propagation algorithm proposed in [15] (which is used in the solver EqArgSolver) with two custom-built SAT-based argumentation solvers in the search for preferred extensions of abstract argumentation frameworks. The SAT-based solvers employ standard ways of computing argumentation semantics via the characterisation of the concept of extensions as models of propositional logic theories. As a result, the comparisons offer new insights about the employment and combination of reduction and direct approaches to the computation of argumentation semantics.

1 Introduction

Argumentation is a field of study involving several interdisciplinary areas, including logic, philosophy, linguistics and computer science, amongst others. In the late 1990s, Abstract Argumentation Frameworks were proposed as a means to formalise the relationships between arguments and evaluate the statuses of arguments under particular *semantics*. There are currently two leading approaches for the computation of these semantics: the so-called *reduction* and *direct* approaches.

A reduction approach translates a particular argumentation semantics and the structure of an argumentation framework into a problem specification in a related area for which a technique for finding solutions exists, then applies that technique and translates the results back into the argumentation context. An example of such an approach is to translate an argumentation problem to a propositional logic theory, then to employ a SAT solver to find models of the theory, and finally to translate back each model found by the solver into a possible solution to the original argumentation problem.

A direct approach to the computation of argumentation semantics, on the other hand, devises specific algorithms to solve the argumentation problem directly by operating on the argumentation graph itself.

Currently, SAT reduction-based solvers lead the performance tables [4]. However, recent advances in direct approaches [7, 15] have reduced the performance gap. The development of direct approach solvers is especially important when considering argumentation problems that cannot be translated to propositional logic (as is the case in some numerical argumentation frameworks).

In [15], we proposed a new forward propagation algorithm for the computation of several semantics of argumentation frameworks, offering significant performance improvements over the well-known Modgil-Caminada's labelling algorithms [12]. This new algorithm was employed in the solver EqArgSolver, submitted to the 2nd International Competition of Computational Models of Argumentation (ICCMA 2017, `http://argumentationcompetition.org`).

Although the new algorithm offers improvements over some of the alternatives, it is not expected to always fare better against a SAT-based solution. In particular, previous experiments [15, 14] indicated that the size and geometry of the argumentation graph may have a strong effect on its performance. This paper aims to shed some light into the employment of the algorithm in the computation of preferred extensions and how it compares in terms of performance with SAT-based reduction approaches. For the comparison, we implemented two new SAT-based argumentation solvers and analysed the performance of all solvers on a relatively large set of argumentation graphs of varied structure.

The first new SAT-based solver is a naive no-frills implementation: it simply translates the geometry of the whole argumentation graph and the conditions characterising a complete extension into a propositional logic theory, submits the theory to the SAT solver `Glucose` (`http://www.labri.fr/perso/lsimon/glucose/`) and then translates models of the theory back as complete extensions. Of these, the maximal ones (with respect to set inclusion) are selected as the preferred extensions. The second SAT-based solver provides a more direct comparison with EqArgSolver because it uses the same decomposition of the original argumentation framework into strongly connected components (SCCs). These are again translated into propositional logic theories. However, the theories are obviously smaller than they would otherwise be for the entire graph. SCC solutions are then combined in an appropriate way to provide solutions for the argumentation framework as a whole.

Our objective with this analysis was to answer a number of open questions: 1) How does the forward propagation algorithm compare with a SAT-based approach? 2) Does the density of attacks influence the performance of the forward propagation algorithm? 3) Does the decomposition of the argumentation framework affect the performance of the SAT-based solvers? 4) Are there circumstances in which one approach is better than the other? These answers can hopefully pave the way for future improvements in the development of direct and reduction-based solvers alike.

The rest of the paper is organised as follows. In Section 2, we introduce some important preliminary concepts. In Section 3, we describe EqArgSolver and the forward propagation algorithm. We then describe different ways in which argumentation semantics can be captured as propositional logic theories in Section 4. This is followed by the empirical evaluation in Section 5 and we finish the paper with some conclusions and future work in Section 6.

2 Background

An *abstract argumentation framework* is a system for reasoning about arguments proposed by Dung [8] and defined in terms of a directed graph $\langle \mathcal{A}, \mathcal{R} \rangle$, where $\mathcal{A}$ is a *finite* non-empty set of arguments and $\mathcal{R}$ is a binary relation on $\mathcal{A}$, called the *attack relation*. If $(X, Y) \in \mathcal{R}$, we say that X attacks Y and depict it with an arrow from X to Y. In what follows, $X^- = \{Y \in \mathcal{A} | (Y, X) \in \mathcal{R}\}$, i.e., the set of arguments attacking X; and $X^+ = \{Y \in \mathcal{A} | (X, Y) \in \mathcal{R}\}$, the set of arguments that X attacks. If $X^- = \varnothing$, then we say that X is a *source argument*. For sets $E \subseteq \mathcal{A}$, $E^- = \cup_{X \in E} X^-$ and $E^+ = \cup_{X \in E} X^+$. We write $E \rightarrow X$ as a shorthand for $X \in E^+$. The *path-equivalence relation* $\sim_{\mathcal{R}} \subseteq \mathcal{A}^2$ is defined as $X \sim_{\mathcal{R}} Y$ iff $X = Y$ or there is a path from X to Y and a path from Y to X in $\mathcal{R}$. A *strongly connected component* (SCC) is an equivalence class of arguments under $\sim_{\mathcal{R}}$.

One of the main purposes of an argumentation framework is to provide a way of reasoning about the *status* of its arguments, i.e., whether an argument is accepted or is defeated by other arguments. Source arguments, having no attacks, always persist. Arguments attacked by attackers that persist are defeated. An attacked argument may still persist, provided all of its attackers are defeated. Thus, the statuses of arguments are determined systematically. In Dung's original formulation, this leads to the concept of *extensions* – subsets of $\mathcal{A}$ with special properties described in terms of the concepts that follow.

A set $E \subseteq \mathcal{A}$ is said to be *conflict-free* if for all elements $X, Y \in E$, we have that $(X, Y) \notin \mathcal{R}$. Moreover, for an argument $X \in \mathcal{A}$ to be *acceptable with respect to a set* E, we want E to "protect" it, i.e., for all $Y \in X^-$, $E \cap Y^- \neq \varnothing$. A set E is then said to be *admissible* if it is conflict-free and all of its elements are acceptable with respect to itself. An admissible set E is a *complete extension* iff E contains all arguments which are acceptable with respect to itself; E is called a *preferred extension* iff E is a $\subseteq$-maximal complete extension; and E is a *stable extension* if E is preferred and $E \cup E^+ = \mathcal{A}$.

Dung's semantics can alternatively be presented in terms of a Caminada *labelling function* of the form $\lambda : \mathcal{A} \longrightarrow \{\mathbf{in}, \mathbf{out}, \mathbf{und}\}$ satisfying certain conditions [2, 3, 16]. Let dom denote the domain of a function and λ a labelling function, we define $in(\lambda) = \{X \in \text{dom } \lambda | \lambda(X) = \mathbf{in}\}$; $und(\lambda) = \{X \in \text{dom } \lambda | \lambda(X) = \mathbf{und}\}$; and $out(\lambda) = \{X \in \text{dom } \lambda | \lambda(X) = \mathbf{out}\}$. We say that an argument X is *illegally labelled* **in** by λ, if $X^- \not\subseteq out(\lambda)$; X is *illegally labelled* **out** by λ, if $X^- \cap in(\lambda) = \varnothing$; and X is *illegally labelled* **und** by λ, if either $X^- \subseteq out(\lambda)$ or $X^- \cap in(\lambda) \neq \varnothing$. A labelling function is *legal* if it does not illegally label any arguments. A complete extension E_λ can be recovered from a legal labelling function λ by setting $E_\lambda = in(\lambda)$. Conversely, a labelling function λ_E can be defined from an extension E by setting $in(\lambda_E) = E$; $out(\lambda_E) = E^+$; and $und(\lambda_E) = \mathcal{A} \backslash (E \cup E^+)$.

2.1 Computing Extensions via the Decomposition into SCCs

In [1], Baroni *et. al* proposed a general recursive schema for argumentation semantics based on the strongly connected components (SCCs) of an argumen-

tation framework. The schema can be used to obtain Dung's admissibility-based semantics with the advantage that the original problem is reduced into components of smaller complexity. Based on the recursive schema, many researchers showed how to compute the extensions of argumentation frameworks under several semantics. Liao's presentation in [11] is particularly easy to understand.

The overall process can be summarised as follows. Firstly, the framework is divided into SCCs. The SCCs form a partition of the original framework and can be organised into successive layers using the attack relation. Each layer can be computed independently with input from the solutions obtained for previous layers and according to a particular semantics. The solutions to the layers thus obtained are combined in an appropriate way. This process is illustrated with an example.

Consider the argumentation framework $\mathcal{N}$ in Fig. 1 with the SCCs $SCC_1 = \{X\}$, $SCC_2 = \{W, Y\}$, $SCC_3 = \{A, B, C\}$ and $SCC_4 = \{D, E\}$. These SCCs can be arranged into three layers following the attack relation: the first one containing SCC_1 and SCC_2; the second one containing SCC_3; and the last one containing SCC_4. Within a given layer, solutions for SCCs are independent from each other, but the attacks between arguments of different layers create dependencies of the solutions of an SCC on the solutions of the SCCs attacking it. In this case, we say that the solutions of the preceding layers *condition* the solutions of the SCC. For example, the computation of the solutions of SCC_3 depends on the labels assigned to X and W, and thus on the solutions for SCC_1 and SCC_2 found in the computation of layer 0. The solutions for SCC_1 and SCC_2 do not depend on any previous computation because they have no external attackers. The only solution for SCC_1 is to label X **in**. However, SCC_2 (which is independent of SCC_1) has three solutions corresponding to legal assignments: one in which both W and Y are labelled **und** and the other two in which one of them is labelled **in** and the other is labelled **out**. The solutions for layer 0 are obtained by combining the solutions for its two SCCs: $f_1 \colon X = Y = \mathbf{in}, W = \mathbf{out}$, $f_2 \colon X = W = \mathbf{in}, Y = \mathbf{out}$, and $f_3 \colon X = \mathbf{in}, W = Y = \mathbf{und}$.[1]

Now let us turn to SCC_3, whose solutions are *conditioned* by the labels of the external attackers X and W in the solutions f_1, f_2 and f_3 for layer 0. In any such solution, $X = \mathbf{in}$, but the label of W could be either **in**, **out** or **und**. In order to generate, say all *complete* extensions for $\mathcal{N}$, each partial solution f_1, f_2 and f_3 needs to be *expanded* with the solutions for SCC_3 under the constraints that they impose. The following definition captures these constraints.

Definition 1 (Initial Conditioned Solution for an SCC). *Let f be a conditioning solution for an SCC S. The initial solution for S conditioned by f, in symbols $\lambda_S^f : S \mapsto \{\mathbf{out}, \mathbf{und}, \mathbf{in}\}$, is the legal labelling function whose set $in(\lambda_S^f)$ is $\subseteq$-minimal with respect to all legal functions conditioned by f.*

[1] As this combination of SCC solutions is done within the same layer, it is called the *horizontal combination* of solutions of the layer [11].

λ_S^f is the "minimal" (grounded) solution for S under the constraints imposed by f. For example, we know that in all partial solutions to layer 0, the label of the argument X must be **in**. Therefore, in any initial conditioned solution for SCC_3 the label of A must be **out**. As for the labels of B and C, they will depend on what the conditioning solution assigns to W. For example, if the conditioning solution is f_2, then the label of B must be **out** as well; f_1 allows it to be **in**, **out** or **und**; whereas f_3 only allows it to be **out** or **und**. In the case of f_1 and f_3, the initial conditioned solution for SCC_3 leaves the label of B undecided (**und**), which is the minimal commitment any legal assignment must satisfy (this minimises the set $in(\lambda)$ for any legal assignment λ). However, f_3 will prevent any assignment that gives the label **in** to B, because $f_3(W) =$ **und**. The initial conditioned solution can be seen as the result of propagating the labels of the external attackers through an SCC whose nodes are all initally labelled **und** (we call this the all-**und** labelling). The Discrete Gabbay-Rodrigues Iteration Schema [9] is an example of a method that can perform this propagation very efficiently.

Once an SCC is conditioned by a solution, the computation of all solutions to the SCC under that solution can be thought of as the search of all possible ways to "expand" the initial conditioned solution by legally swapping labels from **und** to **in** or **out**.

More generally speaking, the whole process can be thought of as follows: given an SCC S, a conditioning solution f, and a partial labelling function λ_S^f, compute the set Λ of all expansions of λ_S^f satisfying some constraints. In [15], we put forward a new algorithm to perform this computation. Obviously this can be done in different ways. For example, we can translate the constraints into propositional logic and use an external SAT solver to compute the models of the resulting theory (if any). Each model will correspond to a legal assignment satisfying the conditioning solution. Since the fastest argumentation solvers are currently all SAT-based, this paper aims to compare the SAT and our own direct approach under similar conditions to help understand the strengths and weaknesses of each.

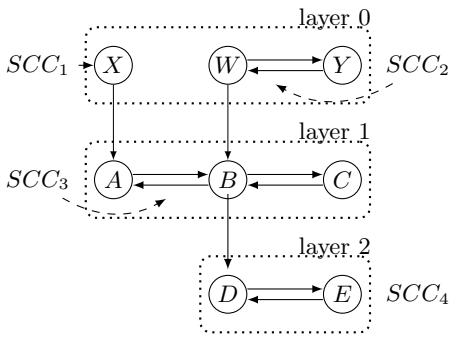


Fig. 1. A complex argumentation framework and its decomposition into layers.

We now describe how the solver EqArgSolver and the forward propagation algorithm work.

3 EqArgSolver and the Forward Propagation Algorithm

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 $\langle \mathcal{A}, \mathcal{R} \rangle$, 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 in some extension or in all extensions under one of those semantics. EqArgSolver follows the general process of computation described in the previous section, using the decomposition of the argumentation framework into SCCs.

Algorithm 1 gives a high-level overview of the computation performed by EqArgSolver, which we now briefly describe. The argumentation framework is first divided into SCCs and arranged into layers. All arguments are initially labelled **und** and then each layer is conditioned by each partial solution found for the previous layers using the discrete version of the *Gabbay-Rodrigues Iteration Schema* [9]. We have seen that this may impose some constraints on the labels of the attacked arguments in the layer. Since the first layer is not constrained by any solutions, the iteration schema simply propagates the **in** labels of the source arguments (which have no attacks and therefore must be labelled **in**). The top-level non-trivial SCCs will be left with all arguments undecided. Some of these could potentially be labelled **in** in a larger extension (line 12). The newly proposed forward propagation algorithm [15] described below, ensures that all such arguments are systematically tried for inclusion generating all possible partial solutions for the SCC (line 13). The partial solutions thus obtained[2] are then combined using what Liao calls the *horizontal and vertical combinations of partial solutions* [10] (lines 14 and 16, respectively). All newly generated solutions are then applied to the next layer and this process is repeated until all layers are processed.

For example, in the argumentation framework of Fig. 1, the discrete iteration schema will produce the solution $X =$ **in** for SCC_1, and the solution $W = Y =$ **und** for SCC_2. The forward propagation algorithm would then operate on SCC_2 to try to attempt to label as **in** any viable argument left undecided in SCC_2. We have seen that we can either leave both W and Y undecided or to assign the label **in** to one and the label **out** to the other. The horizontal combination of the solutions for SCC_1 and SCC_2 (line 14) would then generate all solutions to layer 0, i.e., f_1, f_2 and f_3. This process repeats for the next layer as explained in the previous section. We now briefly describe the forward propagation algorithm itself in a bit more detail.

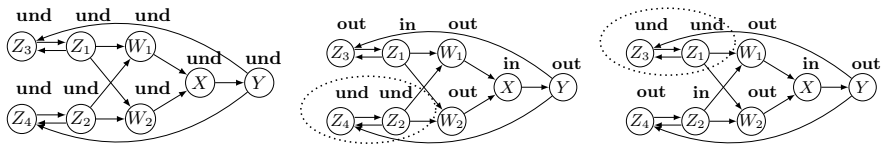
[2] Some filtering to eliminate solutions not leading to maximal extensions in preferred/stable semantics problems is also done, although this is not shown in Algorithm 1. For full details, refer to [14].

```
1  EqArgSolver
2      Read and validate graph G
3      Decompose G into SCCs and arrange them into layers L = {L_0, ..., L_{k-1}}
4      Sols←{all-und}
5      for i ← 0 to k − 1 do                    /* Iterate through layers */
6          newSols←∅
7          foreach f ∈ Sols do
8              λ←GR-ground(L_i, f); TSB← trivial SCC block of L_i
9              LayerSols←{λ ↓ TSB}
10             S← non-trivial SCCs in L_i
11             foreach S ∈ S do
12                 possIns← candidate in-nodes of S according to λ
13                 SCC-sols←findExtsFromArgs(possIns, S, f, λ ↓ S)
14                 Horizontally combine SCC-sols with solutions in LayerSols
15             end foreach
16             Add vertical combination of f with each γ ∈ LayerSols to
                newSols
17         end foreach
18         Sols ←newSols
19     end for
20 end
```

Algorithm 1: EqArgSolver's overall processing sequence.

The Forward Propagation Algorithm.

The forward propagation algorithm is a new algorithm proposed in [15] allowing the generation of all complete extensions of an argumentation framework. It takes advantage of the constraints imposed by what it means to legally label an argument in order to prune the search space of feasible solutions. The algorithm is invoked by the procedure `findExtsFromArgs` at line 13 of Algorithm 1 and provides a huge performance improvement over Modgil-Caminada's algorithm (see [15] for a comparison). Essentially, each choice made for an argument being labelled **in** restricts the set of candidate labelling functions further. The more attacks there are in the argumentation graph the higher the number of such constraints to be satisfied. In [15], we suggested that the performance of the algorithm should therefore increase proportionally to the density of attacks in the graph. The empirical evaluation in Section 5 clearly confirms this conjecture.



(L) Initial configuration λ

(C) Labelling $X = \mathbf{in}$, option $Z_1 = \mathbf{in}$ ($\boldsymbol{\lambda_1}$)

(R) Labelling $X = \mathbf{in}$, option $Z_2 = \mathbf{in}$ ($\boldsymbol{\lambda_2}$)

Fig. 2. A sample SCC illustrating the forward propagation algorithm.

As an example of execution, suppose we had at layer 0 the SCC shown in Fig. 2 (L). Initially, all of its arguments would be undecided. Some of these arguments could potentially belong to an extension. The forward propagation algorithm would attempt to label **in** all viable arguments. Suppose we start with argument X (the order is not important). By labelling X **in** we would be forced to label **out** the argument Y that X attacks as well as X's attackers W_1 and W_2. In order to legally label the latter two arguments **out** we would also be required to label either Z_1 or Z_2 **in**. Again for Z_1 to be legally labelled **in**, Z_3 must be labelled **out** ($\boldsymbol{\lambda_1}$), and for Z_2 to be labelled **in**, Z_4 must be labelled **out** ($\boldsymbol{\lambda_2}$). This process is repeated until all options are considered. In the case of $Z_1 =$ **in** ($\boldsymbol{\lambda_1}$), we would still have the two sub-options $Z_2 =$ **in**, $Z_4 =$ **out** ($\boldsymbol{\lambda_{11}}$) and $Z_2 =$ **out**, $Z_4 =$ **in** ($\boldsymbol{\lambda_{12}}$). This would result in all legal labellings in which X is labelled **in** (see Table 1).

For *all* complete extensions to be generated, the algorithm will attempt to label **in** all viable nodes in the SCC. If the SCC of Fig. 2 is at layer 0, this will be all of its nodes. In the case of the SCC_3 of Fig. 1 with conditioning solution f_2, there would be nothing to be done, because the iteration schema would force A and B to be labelled **out** and this would force C to be labelled **in**, leaving no argument left undecided. In general, the algorithm would only attempt to label **in** the suitable candidate arguments. There are further subtle requirements to be satisfied, and sometimes the forward propagation may fail, leaving just the initial solution which will be legal because the iteration schema always provides a legal assignment when it starts from a legal assignment. The full details can be found in [15].

	X	Y	W_1	W_2	Z_1	Z_2	Z_3	Z_4
$\boldsymbol{\lambda_1}$	**in**	**out**	**out**	**out**	**in**	**und**	**out**	**und**
$\boldsymbol{\lambda_2}$	**in**	**out**	**out**	**out**	**und**	**in**	**und**	**out**
$\boldsymbol{\lambda_{11}}$	**in**	**out**	**out**	**out**	**in**	**in**	**out**	**out**
$\boldsymbol{\lambda_{12}}$	**in**	**out**	**out**	**out**	**in**	**out**	**out**	**in**
$\boldsymbol{\lambda_{21}}$	**in**	**out**	**out**	**out**	**in**	**in**	**out**	**out**
$\boldsymbol{\lambda_{22}}$	**in**	**out**	**out**	**out**	**out**	**in**	**in**	**out**

Table 1. All possible complete labelling functions in which the label of argument X of the SCC in Fig. 2 is **in**.

At the point where the forward propagation algorithm is invoked (Algorithm 1, line 13), we can place any suitable procedure that computes all legal label assignments to the SCC taking into account a particular conditioning solution. In Section 4.2 we will show how to use a SAT solver to perform this computation. The replacement of the SAT solver procedure for the forward propagation algorithm yielded a new SAT-based solver that we called `glucsccsolver`, after its use of `Glucose` and the decomposition into SCCs.

4 Computing the Semantics via a SAT Solver

It is pretty straightforward to search for complete extensions of an argumentation framework using an existing SAT solver. The main idea is to characterise the legal labelling conditions as a theory in propositional logic and search for models of the theory using the solver. A model can then be translated back as an extension via an appropriate mapping. In this Section, we show one of many possible characterisation and discuss possible optimisations for the stable semantics. In addition, we show how to restrict the labelling conditions to individual SCCs conditioned by solutions, allowing the combination of the technique described in Section 2.1 with a SAT solver. The formalisations given here were used to implement the two SAT-based solvers `glucsolver` and `glucsccsolver` discussed in Section 5.

We focus our presentation on the complete semantics, because it can be used in the computation of the grounded, preferred and stable semantics as well (see Section 2). We must also stress that a more efficient SAT-based implementation would take advantange of the intrinsic characteristics of the underlying SAT solver to optimise the search for solutions. The characterisation below is merely illustrative.

4.1 Translating the Complete Semantics Conditions to Propositional Logic

Let $\langle \mathcal{A}, \mathcal{R} \rangle$ be an argumentation framework. In order to encode the semantics as a logical theory, we need a propositional logic language with the usual logical connectives and $3 * |\mathcal{A}|$ symbols where for each $X \in \mathcal{A}$, the symbols $in(X)$, $out(X)$ and $und(X)$ denote that the argument X is either labelled **in**, **out**, or **und**, respectively. The formulae below, which we refer to as the COMPLETE conditions, are then defined for all arguments $X \in \mathcal{A}$.

We start with the uniqueness labelling conditions implicit in the definition of a function.

(UNIQUE) $(\text{in}(X) \vee \text{out}(X) \vee \text{und}(X)) \wedge \neg(\text{in}(X) \wedge \text{out}(X)) \wedge$
$\neg(\text{in}(X) \wedge \text{und}(X)) \wedge \neg(\text{out}(X) \wedge \text{und}(X))$

We then need to capture the conditions under which arguments are legally labelled **in** and **out**:

(IN-C1) $\bigwedge_{Y \in X^-} \text{out}(Y) \rightarrow \text{in}(X)$

(IN-C2) $\text{in}(X) \rightarrow \bigwedge_{Y \in X^+} \text{out}(Y)$

(OUT-C1) $\bigvee_{Y \in X^-} \text{in}(Y) \rightarrow \text{out}(X)$

(OUT-C2) $\text{out}(X) \rightarrow \bigvee_{Y \in X^-} \text{in}(Y)$

(IN-C1) says that if all attackers of an argument are labelled **out**, the argument must be labelled **in**. (IN-C2) says that if an argument is labelled **in**, then

all of the arguments that it attacks must be labelled **out**. (OUT-C1) says that if any attacker of an argument is labelled **in**, the argument must be labelled **out**. Finally, (IN-C2) says that if and argument is labelled **out**, then at least one of its attackers must be labelled **in**.

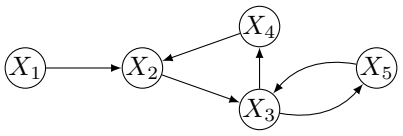
It is easy to see that all source arguments, having no attackers, will be labelled **in** in any model that satisfies (IN-C1). This is because the empty conjunction of **out**'s is equivalent to $\top$, which means those arguments must be labelled **in**. This will force any arguments attacked by those arguments to be labelled **out** by (IN-C2) and so forth.

In [5], Cerutti *et al.* showed that a propositional logic formalisation similar to (UNIQUE), (IN-C1), (IN-C2), (OUT-C1) and (OUT-C2) precisely captures the notion of a complete extension. The authors divided the set $\mathcal{A}$ into the two parts $C, A \backslash C$ (for some $C \subseteq \mathcal{A}$). The perceptive reader will note that formulae (1)–(5) of [5, Definition 14] correspond to (UNIQUE), (IN-C1), (IN-C2), (OUT-C1) and (OUT-C2) and that NC_i^x of [5, Proposition 5] is $\top$ when $C = \mathcal{A}$. Therefore, (1)∧(2)∧(3)∧(4)∧(5) is sufficient to capture the notion of a complete extension semantically in propositional logic.

A SAT solver such as `Glucose` accepts a formula in *conjunctive normal form* (CNF) which are conjunctions of disjunctions of literals. Each clause is a disjunction of positive and negative literals l_k of the form $\bigvee_m l_m \vee \bigvee_n \neg l_n$. Let $X^- = \{Y_1, \ldots, Y_{x_i}\}$ and $X^+ = \{W_1, \ldots, W_{x_j}\}$. The conditions above can be re-written as disjunctive clauses as follows:

(UNIQUE)	$\text{in}(X) \vee \text{out}(X) \vee \text{und}(X)$
	$\neg\text{in}(X) \vee \neg\text{out}(X)$
	$\neg\text{in}(X) \vee \neg\text{und}(X)$
	$\neg\text{out}(X) \vee \neg\text{und}(X)$
(IN-C1)	$\neg\text{out}(Y_1) \vee \neg\text{out}(Y_2) \vee \ldots \vee \neg\text{out}(Y_{x_i}) \vee \text{in}(X)$
(IN-C2)	$\neg\text{in}(X) \vee \text{out}(W_1)$
	$\neg\text{in}(X) \vee \text{out}(W_2)$
	$\vdots$
	$\neg\text{in}(X) \vee \text{out}(W_{x_j})$
(OUT-C1)	$\neg\text{in}(Y_1) \vee \text{out}(X)$
	$\neg\text{in}(Y_2) \vee \text{out}(X)$
	$\vdots$
	$\neg\text{in}(Y_{x_i}) \vee \text{out}(X)$
(OUT-C2)	$\neg\text{out}(X) \vee \text{in}(Y_1) \vee \text{in}(Y_2) \vee \ldots \vee \text{in}(Y_{x_i})$

Example 1. Consider the argumentation framework below:



The framework would yield the following clauses for the node X_1:

(UNIQUE)	$\text{in}(X_1) \vee \text{out}(X_1) \vee \text{und}(X_1)$ $\neg\text{in}(X_1) \vee \neg\text{out}(X_1), \neg\text{in}(X_1) \vee \neg\text{und}(X_1), \neg\text{out}(X_1) \vee \neg\text{und}(X_1)$
(IN-C1)	$\text{in}(X_1)$
(IN-C2)	$\neg\text{in}(X_1) \vee \text{out}(X_2)$

Since X_1 has no attackers, the attack disjunction in (IN-C1) is false, leaving just the positive literal $\text{in}(X_1)$; in (IN-C2), we only have one clause for the node that X_1 attacks, i.e., X_2; finally, (OUT-C1) $\equiv \top$, and (OUT-C2) $\equiv \text{out}(X_1) \rightarrow \bot \equiv \neg\text{out}(X_1)$, since X_1 has no attacks and hence the disjunction of its attackers is false. Since (IN-C1) + (UNIQUE) $\vdash \neg\text{out}(X_1)$, this is not needed.

Any model of a theory with the clauses above would satisfy $\text{in}(X_1)$ and $\text{out}(X_2)$ (because of (IN-C1) and (IN-C2)). As for X_3–X_5, one of the possibilities would be to make $\text{und}(X_3) = \text{und}(X_4) = \text{und}(X_5) = \mathit{true}$. An extension would simply be the set of nodes X for which the proposition $\text{in}(X) = \mathit{true}$. In this case, the model described would correspond to the complete (but not preferred) extension $\{X_1\}$.

`Glucose` would provide a model (if any) according to some internal computation strategy. In order to obtain further models, one can re-submit the original set of clauses *plus* a clause negating the previous solution. As the solution is presented as the set of literals that are true (to be understood as a conjunction of literals describing the model), all we need to do is to add the negation of these literals as a new clause and re-submit the call to `Glucose`. Eventually, no more models will be found and `Glucose` will return the word "UNSAT". This simply means that no more models (i.e., complete extensions) can be found.

We saw above that each model will correspond to a complete extension that is not necessarily preferred. At the *meta-level*, we will need to filter out these extensions keeping only those that are maximal w.r.t. set inclusion in order to obtain the preferred extensions.

4.2 Conditions for SCCs Conditioned by Solutions

We can employ the same principles presented in Section 2.1 and write a modified theory that can be used to find all solutions for a non-trivial SCC.[3] These solutions can then be combined horizontally and vertically as before to yield

[3] Trivial SCCs are dealt with directly by the discrete Gabbay-Rodrigues Iteration Schema (see [14] for details).

solutions for the argumentation framework as a whole. The advantage of doing this is that the theorem prover would have to deal with a smaller theory and this may speed up the computation of solutions of larger argumentation frameworks. The drawback is that more external calls to the solver will be needed. Obviously there is a trade-off between the two approaches. The results in Section 5 indicate that there is an advantage of doing so when the size of the network is relatively large. For now, let us describe what is needed.

Consider the argumentation framework of Fig. 1 again depicted on the left of Fig. 3 and suppose we wished to find the solutions to SCC_3 alone, which is in layer 1. In Section 2.1 we saw that the solutions to SCC_3 are conditioned by the solutions f_1, f_2 and f_3 to layer 0, where $f_1(X) = \mathbf{in}$, $f_1(W) = f_1(Y) = \mathbf{und}$; $f_2(X) = f_2(W) = \mathbf{in}$, $f_2(Y) = \mathbf{out}$; and $f_3(X) = f_3(Y) = \mathbf{in}$, $f_3(W) = \mathbf{out}$.

The part of the argumentation framework relevant to the computation of SCC_3's solutions is shown in Fig. 3 (R).

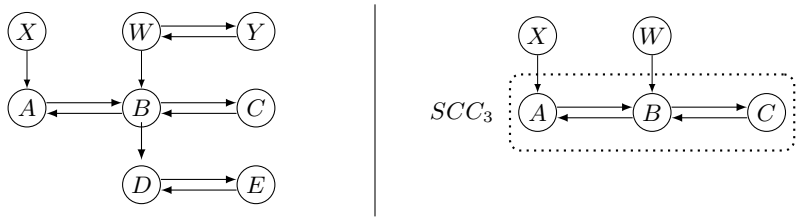


Fig. 3. The argumentation framework of Fig. 1 and its relevant part to the computation of the solutions to SCC_3

The only modification we need to do is to write special formulae for the nodes attacking the SCC (which have fixed values) and write the usual formulae for the nodes in the SCC itself, eliminating those associated with attacks going out of the SCC. Let us start with the former. In this example, the formulae need to describe what the labels of X and W are in the corresponding solution.

Axioms for the External Attackers of the SCC

The values of the incoming attacks are determined by the particular conditioning solution. This means one formula for each attacking argument (each being written as three clauses). For example, for solution f_1, we would need the two formulae:

$$\mathrm{in}(X) \wedge \neg\mathrm{out}(X) \wedge \neg\mathrm{und}(X), \mathrm{und}(W) \wedge \neg\mathrm{in}(W) \wedge \neg\mathrm{out}(W)$$

Alternatively, one could write the actual label of the argument, e.g., $\mathrm{in}(X)$ and then instantiate (UNIQUE) with X, but this overcomplicates matters somewhat. Similarly, although (IN-C2) could be applied to X, forcing A to be **out**, this can be taken care of by (OUT-C1). So for all external attackers Y_i of an SCC $\mathcal{S}$, we only need to write one of

(ATT-IN) $\text{in}(Y_i) \wedge \neg\text{out}(Y_i) \wedge \neg\text{und}(Y_i)$

(ATT-OUT) $\neg\text{in}(Y_i) \wedge \text{out}(Y_i) \wedge \neg\text{und}(Y_i)$

(ATT-UND) $\neg\text{in}(Y_i) \wedge \neg\text{out}(Y_i) \wedge \text{und}(Y_i)$

depending on whether the attacker Y_i is labelled **in**, **out**, or **und**, respectively, by the corresponding conditioning solution.

Axioms for the Nodes in the SCC

For all nodes in the SCC itself, we need the axioms (UNIQUE), (IN-C1), (OUT-C1) and (OUT-C2). As for (IN-C2), we only need the axioms where the attacked node belongs to the SCC. For example, for the full argumentation framework, we would normally write the axioms

$$\text{in}(B) \rightarrow \text{out}(A) \wedge \text{out}(C) \wedge \text{out}(D)$$

which in clausal form would yield the three clauses

$$\neg\text{in}(B) \vee \text{out}(A), \neg\text{in}(B) \vee \text{out}(C), \neg\text{in}(B) \vee \text{out}(D)$$

However, D is outside of the SCC, so its value is not relevant in the solution to SCC_3, so instead, we simply write

$$\neg\text{in}(B) \vee \text{out}(A), \neg\text{in}(B) \vee \text{out}(C)$$

because the nodes A and C that B attacks are within the same SCC as B.

4.3 Putting it All Together

We built two SAT-based argumentation solvers based on the translations provided in Sections 4.1 and 4.2 above. Both employ `Glucose` as an external SAT solver. The first solver uses a unified approach, i.e., it translates the whole argumentation framework as a set of clauses and then invokes `Glucose` successively to compute all models of that set. Each model is then translated back as a single extension, with those that are maximal being the preferred ones. We called this solver `glucsolver`.

The second solver makes use of EqArgSolver's decomposition of the argumentation framework into SCCs and their arrangement into layers. As `glucsolver`, it also invokes `Glucose` but only to find models of the logical translations of the non-trivial SCCs. This means the theories it has to solve are smaller, although a higher number of external calls to `Glucose` need to be made. We called this solver `glucsccsolver`.

These two solvers allowed us to compare the impact of the decomposition into SCCs in the SAT reduction-based approach and their relationship with our direct approach employing the forward propagation algorithm.

Before we move on to the evaluation of the benchmark results in Section 5, it is worth emphasising that for the special case of the stable semantics, the logical encoding can be greatly (and trivially) simplified as shown next.

4.4 Optimising for the Stable Semantics

We had to utilise three propositional symbols for each argument because classical logic is two-valued whereas Dung's semantics is three-valued. The proposition $\mathrm{in}(X)$ can only encode, say, whether or not the argument X is labelled **in**. If it is not labelled **in**, it could still be labelled **out** or **und**, so we need more symbols to distinguish between these two states. However, in the case of the *stable semantics*, extensions correspond to assignments in which arguments can only be in one of two states: **in** or **out** (as no arguments can be labelled **und**). In this case, one propositional symbol per argument will suffice. Accordingly, we could choose the propositional symbol "X" to represent the fact that the label of the argument X is **in**, and then infer from the formula $\neg X$ that the label of the argument X is **out**. Under this assumption, the COMPLETE conditions could then be simplified as the conditions that follow (which we refer to, collectively, as STABLE):

(ST IN-C1) $\bigwedge_{Y \in X^-} \neg Y \rightarrow X$

(ST IN-C2) $X \rightarrow \bigwedge_{Y \in X^+} \neg Y$

(ST OUT-C1) $\bigvee_{Y \in X^-} Y \rightarrow \neg X$

(ST OUT-C2) $\neg X \rightarrow \bigvee_{Y \in X^-} Y$

Proposition 1. *If a model w satisfies conditions* STABLE*, then the set of arguments $\{X | w \models X\}$ corresponds to a stable extension.*

Proof. A model of COMPLETE *corresponds to a complete extension. When a complete extension is stable, then it attacks all arguments outside of it. Therefore, all arguments outside of the extension must be labelled* **out** *and hence no argument is labelled* **und**. *By* (UNIQUE) *and the fact that no argument is labelled* **und**, *any model w corresponding to a stable extension will satisfy $in(X) \leftrightarrow \neg out(X)$. We can simply use the formula "X" for $in(X) \equiv \neg out(X)$ and "$\neg X$" for $\neg in(X) \equiv out(X)$. From this,* (ST IN-C1)*,* (ST IN-C2)*,* (ST OUT-C1) *and* (ST OUT-C2) *will correspond to* (IN-C1)*,* (IN-C2)*,* (OUT-C1) *and* (OUT-C2)*.*

Since (ST IN-C1) $\equiv$ (ST OUT-C2) and (ST IN-C2) $\equiv$ (ST OUT-C1) (via the contraposition of the implication), we only really need an appropriate combination of these, for example (ST IN-C1) and (ST IN-C2).

5 Empirical Evaluation

We conducted three sets of experiments to evaluate different aspects of the performance of the forward propagation algorithm proposed in [15] in relation to a generic search for preferred extensions of an argumentation framework using the `Glucose` SAT solver (`http://www.labri.fr/perso/lsimon/glucose/`).

Our objective with the experiments was not to conduct an extensive empirical evaluation between solvers. Indeed this is being done by ICCMA 2017

(`http://argumentationcompetition.org/`). We merely proposed to draw some conclusions about the behaviour of the forward propagation algorithm with respect to a "generic" SAT-based approach.

Experimental Setup

All graphs in the experiments were generated using `probo`'s SCC generator [6], which generates graphs with a bounded number of SCCs and a given probability of attacks between arguments. We were not too concerned about attacks between SCCs, because the search for solutions for an SCC lies at the base of the recursive problem, so our main focus was on the SCC cardinality and the density of attacks *within* SCCs. Thus, in all experiments we set the probability of attacks between arguments of *different* SCCs to 0.1, but varied the internal probability of attacks between arguments within an SCC between 0.1 and 1.0. This variation allowed us to confirm a conjecture we made in [15] suggesting that the performance of the forward propagation algorithm should increase proportionally to the probability of attacks between arguments. Intuitively, the higher this probability, the higher the density of attacks in the graph, and hence the more the number of constraints that are added to the labelling functions and, consequently, the smaller the search space of feasible solutions. Our experiments confirmed this conjecture (we will revisit this later).

The graphs thus generated were submitted to the three solvers: EqArgSolver [14] (which employs the forward propagation algorithm); `glucsolver` which employs `Glucose` on the translation of the whole argumentation graph as a SAT problem (as described in Section 4.1); and finally `glucsccsolver`, which employs a combination of the decomposition technique into SCCs and `Glucose` restricted to theories encoding single non-trivial SCCs (as described in Section 4.2). The solvers ran on a PC with an Intel i7 4690K processor and 32Gb RAM.

Fig. 4, Fig. 5 and Fig. 6 show the comparative average execution time per graph. We discuss the findings in three categories of graphs as follows.

Small cardinality, small number of SCCs

In the first set of experiments, we randomly generated $6,000$ graphs with up to 50, 100, 120 140, 160 and 200 nodes. For each cardinality boundary, we generated $1,000$ graphs with up to 2 SCCs each, 100 for each probability 0.1, 0.2, ..., 1.0. We then searched for preferred extensions using the three solvers. The performance of the solvers for this set of graphs is depicted in Fig. 4. From the results, we can conclude the following:

1. There was not much difference in performance between the SAT solvers with or without decomposition of the graphs into SCCs. This is due to the fact that the graphs are relatively small and only contain up to 2 SCCs, so the performance gain obtained by searching for models in smaller theories could not be offset by the extra effort used in the decomposition into SCCs.

2. The SAT-based solvers outperformed the forward propagation algorithm when the probability of internal attack was roughly below 0.5. After a certain probability threshold (of around 0.5), the forward propagation algorithm outperformed the SAT-based solvers in all cardinalities.
3. The performance of the forward propagation algorithm *did* increase proportionally with the increase in the probability of attacks between arguments.

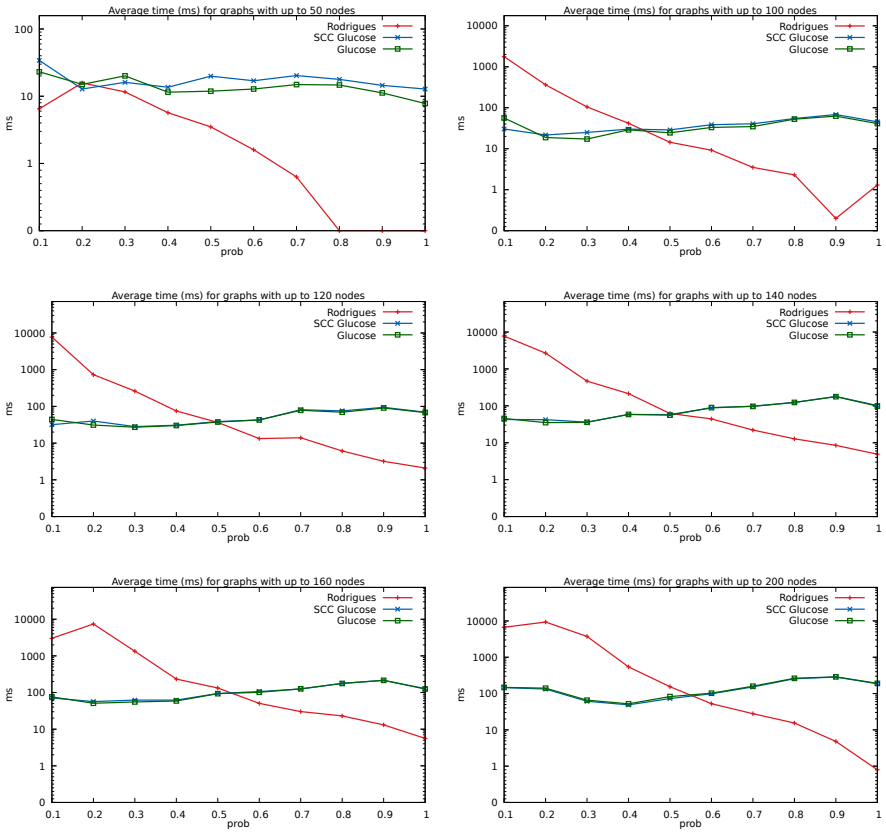


Fig. 4. Average computation time for graphs with up to 50–200 nodes and up to 2 SCCs.

Medium cardinality, small number of SCCs

We then generated a second set of graphs with a larger upper bound on the number of arguments varying from 300 to 600 in increments of 100 and a slightly larger upper bound on the number of SCCs of 5. Again we set the probability of internal attacks from 0.1 to 1.0 in 0.1 increments. The results are depicted in Fig. 5. From this set we can conclude the following:

1. Again there was no significant difference in performance between the SAT solvers with or without decomposition of the graphs into SCCs. We again

conclude that the relatively small number of SCCs was not sufficient to take advantage of the decomposition into SCCs.
2. The SAT-based solvers still outperform the forward propagation algorithm when the probability of internal attack is roughly below 0.5. After a certain threshold, the forward propagation algorithm then outperforms the SAT-based solvers.
3. There is more variance in performance between the SAT-based solvers and the one using the forward propagation algorithm and the performance gap is somewhat reduced between the three. This may suggest that there may be a graph cardinality threshold after which the performance of the SAT-based solvers degrade.
4. Again, the performance of the forward propagation algorithm *did* increase proportionally with the increase in the probability of attacks between arguments.

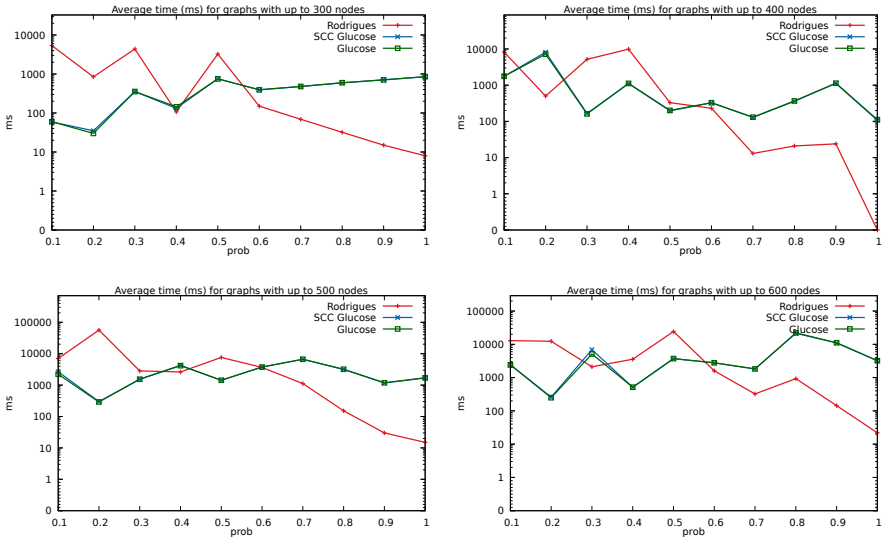


Fig. 5. Average computation time for graphs with 300–600 nodes and up to 5 SCCs.

Large cardinality, medium number of SCCs

Subsequently, we randomly generated 100 graphs with up to 2,500 arguments each and a number of SCCs between 35 and 50 each, again with internal probability of attacks between arguments from 0.1 to 1.0 in 0.1 increments. The results of this set of experiments are depicted in Fig. 6, from which we can conclude:

1. The SAT solver using the decomposition outperformed the one without it independently of the internal probability of attacks. This confirms the

result ($I2$) given in [5], which asserted that there would be a threshold after which the decomposition into SCCs provides performance benefits for SAT-reduction solvers.
2. The forward propagation algorithm outperformed both SAT solvers independently of the internal probabilities of attacks.

Although the graph in Fig. 6 still shows some improvement in EqArgSolver's performance with the increase in the probability of attacks, this was not very noticeable. There may be many reasons for this. Firstly, the sample was relatively small. Secondly, the large number of SCCs means that the relative size of each individual SCC may not be large enough to show a more marked variation. This investigation will continue in future work.

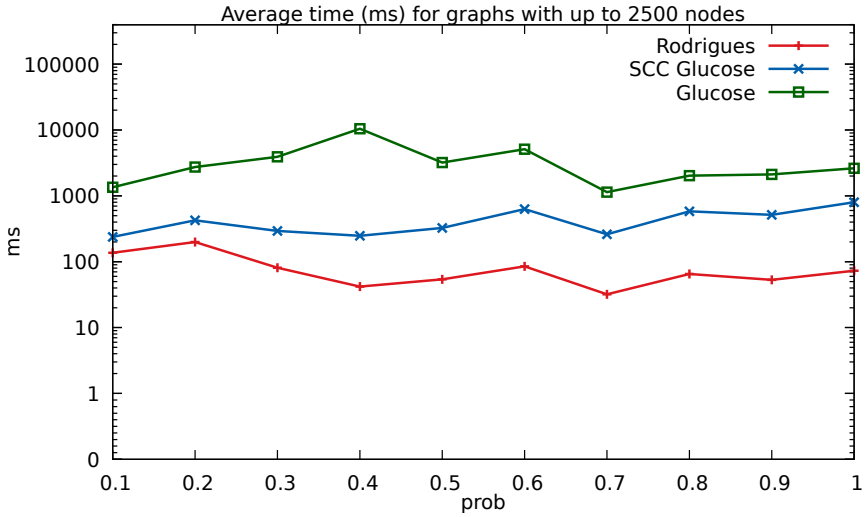


Fig. 6. Average computation time for graphs with up to 2,500 nodes and in between 35 and 50 SCCs.

6 Conclusions and Future Work

The motivation for the development of the forward propagation algorithm proposed in [15] came from the need to replace Modgil-Caminada's algorithm in the solver GRIS [13]. Modgil-Caminada's algorithm proved very inefficient for all but the simplest graphs and can only compute the preferred extensions. The forward propagation algorithm can compute all complete extensions and is able to decide on argument acceptability without necessarily having to generate all extensions. This new algorithm was used in the solver EqArgSolver [14], which was submitted to ICCMA 2017 (see `http://argumentationcompetition.org/`).

Solvers using SAT-reduction approaches took the top spots in the 1st ICCMA and we would expect them to continue to outperform solvers using direct

approaches for some time, but there are application areas where solvers using a direct approach are the only alternatives, e.g., in certain argumentation frameworks where attacks and arguments receive values in the $[0, 1]$ interval. Therefore continued research in direct algorithms remains important for the development of the field. The objective of this paper was to shed some light into the performance differences between the SAT approaches and the forward propagation algorithm, paving the way for future performance enhancements arising from this analysis.

We showed how to build a rudimentary meta argumentation solver based on the `Glucose` SAT solver, which we called `glucsolver`. It simply does a naive translation of the entire argumentation graph and the COMPLETE labelling conditions as a logical theory and uses the models found by `Glucose`'s for the theory to provide all complete extensions of the argumentation framework. The maximal extensions of these will correspond to the argumentation framework's preferred extensions. We then provided two optimisation techniques: in problems involving the *stable* semantics, we showed how to simplify the translation of the argumentation problem into propositional logic by minimising the number of required propositional variables. Secondly, we showed how to restrict the logical conditions for complete labellings of an SCC when they are conditioned by a solution. This means that the SAT solver can be invoked to find solutions restricted to SCCs and hence work on smaller theories. We called the second argumentation solver employing this technique `glucsccsolver`. It simply replaces the forward propagation algorithm in EqArgSolver with a call to `Glucose` for each theory representing an individual non-trivial SCC and hence provides a pathway for a more direct comparison between the two approaches.

We ran three sets of experiments with each of the above solvers. Our experiments confirmed that the density of attacks within SCCs is inversely proportional to the execution time of the forward propagation algorithm. This means that this algorithm is particularly useful in dense graphs. The results also indicate that the decomposition into SCCs is only advantageous when the number of SCCs is relatively large (confirming an observation made earlier by [5]). For larger graphs, EqArgSolver using the forward propagation algorithm outperformed the SAT-based solvers. This could arise from several factors. `Glucose` may suffer some performance degradation when the underlying theory is large. This would explain why `glucsccsolver` outperforms `glucsolver`, since each call to `Glucose` invoked by `glucsccsolver` is done on a smaller logical theory. The performance gain is however offset by the extra overhead incurred by the additional external system calls and would explain why EqArgSolver outperformed both SAT-based solvers. In a more robust SAT-based implementation, the SAT solver would be embedded in the meta-solver to eliminate the extra effort in extenal system calls and its invocation would be carefully fine-tuned for the problem at hand.

Our next step is to try to incorporate into the forward propagation algorithm techniques that in the logical context allow a faster search for models of the underlying logical theory. If we can translate these into algorithmic con-

straints or heuristics, they may help us prune the search space of solutions further, improving the overall performance of the propagation algorithm.

References

[1] Baroni, P., Giacomin, M., Guida, G.: SCC-recursiveness: a general schema for argumentation semantics. Artificial Intelligence 168(1), 162 – 210 (2005)

[2] Caminada, M.: A labelling approach for ideal and stage semantics. Argument and Computation 2(1), 1–21 (2011)

[3] Caminada, M., Gabbay, D.M.: A logical account of formal argumentation. Studia Logica 93(2-3), 109–145 (2009)

[4] Cerutti, F., Oren, N., Strass, H., Thimm, M., Vallati, M.: Summary report of the first international competition on computational models of argumentation. AI Magazine 37(1), 102 (2016)

[5] Cerutti, F., Giacomin, M., Vallati, M., Zanella, M.: A SCC recursive meta-algorithm for computing preferred labellings in abstract argumentation. In: 14th Intl. Conference on Principles of Knowledge Representation and Reasoning (KR) (2014)

[6] Cerutti, F., Oren, N., Strass, H., Thimm, M., Vallati, M.: A benchmark framework for a computational argumentation competition (demo paper). In: Proceedings of the Fifth International Conference on Computational Models of Argumentation (COMMA'14) (September 2014)

[7] Charwat, G., Dvořák, W., Gaggl, S.A., Wallner, J.P., Woltran, S.: Methods for solving reasoning problems in abstract argumentation a survey. Artificial Intelligence 220, 28 – 63 (2015)

[8] Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artificial Intelligence 77, 321–357 (1995)

[9] Gabbay, D.M., Rodrigues, O.: Further applications of the Gabbay-Rodrigues iteration schema. In: Beierle, C., Brewka, G., Thimm, M. (eds.) Computational Models of Rationality, vol. 29, pp. 392–407. College Publications (2016)

[10] Liao, B.: Efficient Computation of Argumentation Semantics. Elsevier (2014)

[11] Liao, B.: Toward incremental computation of argumentation semantics: A decomposition-based approach. Annals of Mathematics and Artificial Intelligence 67(3), 319–358 (2013), http://dx.doi.org/10.1007/s10472-013-9364-8

[12] Modgil, S., Caminada, M.: Proof theories and algorithms for abstract argumentation frameworks. In: Simari, G., Rahwan, I. (eds.) Argumentation in Artificial Intelligence, pp. 105–129. Springer US, Boston, MA (2009)

[13] Rodrigues, O.: GRIS system description. In: Thimm, M., Villata, S. (eds.) System Descriptions of the 1st International Competition on Computational Models of Argumentation. pp. 37–40. Cornell University Library (2015)

[14] Rodrigues, O.: EqArgSolver system description. In: Proceedings of the 4th Intl. Conference on Theory and Applications of Formal Argumentation. TAFA'17 (2017), to appear

[15] Rodrigues, O.: A forward propagation algorithm for the computation of the semantics of argumentation frameworks. In: Proceedings of the 4th Intl. Conference on Theory and Applications of Formal Argumentation. TAFA'17 (2017), to appear

[16] Wu, Y., Caminada, M.: A labelling-based justification status of arguments. Studies in Logic 3(4), 12–29 (2010)