A Forward Propagation Algorithm for the Computation of the Semantics of Argumentation Frameworks

Odinaldo Rodrigues(✉)

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

Abstract. In this paper we propose a novel algorithm for the computation of the semantics of argumentation frameworks. The algorithm can generate all complete extensions and thus can be used in problems involving the grounded, complete, preferred and stable semantics. The algorithm takes advantage of the constraints imposed on legal labelling functions to prune the search space of possible solutions.

1 Introduction

This paper describes a new algorithm for the computation of the semantics of argumentation frameworks based on the idea of *forward propagation* of **in** labels of accepted argument. The basic mechanism is very simple: the construction of complete extensions is done by attempting to re-label **in** all undecided (**und**) arguments that could *potentially* be labelled **in** by a labelling function and checking whether the resulting function can be made "legal".

The algorithm works on the strongly connected components (SCCs) of an argumentation framework which are arranged into layers following the direction of attacks. Because of the dependencies between the valid assignments of labels of attacking and attacked arguments, a solution for one layer may impose constraints on the possible solutions for SCCs of subsequent layers. In such cases, we say that the solution of one layer *conditions* the possible legal label assignments of the attacked SCC. So we take this idea further by looking at the consequences of legally labelling an argument **in** in an SCC: we search for labelling assignments of the SCC satisfying an increasing set of constraints. All solutions thus found are combined in the way described in [11].

We start with the undecided arguments of an SCC that could potentially be labelled **in** in *some* solution. By labelling one of these arguments **in**, we are forced to label all of its attackers **out** (i.e., reject them). If all attackers of an argument are re-labelled **out**, then the argument must be re-labelled **in**, imposing new constraints on the labels of the arguments that it attacks, and so forth. Forcing the attacker of an argument to be labelled **out** is done analogously by requiring that at least one of the attacker's attacker is labelled **in**, so the whole process can be done through a series of recursive forward propagation operations of **in** labels each of smaller complexity than the original one.

© Springer International Publishing AG, part of Springer Nature 2018
E. Black et al. (Eds.): TAFA 2017, LNAI 10757, pp. 120–136, 2018.
https://doi.org/10.1007/978-3-319-75553-3_8

Searching for extensions in this way has several advantages. The constraints can *prune* the search space considerably by ruling out assignments that violate the admissibility conditions. Thus, the algorithm is designed to incrementally "fill in" the gaps of an admissible but partially uncommitted labelling function by successively swapping labels from **und** to **in** or **out**, and as a result generating all complete extensions along the way. As a by-product, we can pick an argument of interest and attempt to construct a legal labelling assignment that labels the argument in a particular way (e.g., **in**), without necessarily having to look at all solutions of the SCC or the argumentation framework as a whole.

The rest of the paper is structured as follows. In Sect. 2, we provide some background material for the paper. This is followed by the presentation of the algorithm itself in Sect. 3.[1] In Sect. 4, we compare our algorithm with others in the literature. Section 5 provides some empirical evaluation of the algorithm and we conclude in Sect. 6 with a discussion and some future work.

2 Background

An abstract argumentation framework is a system for reasoning about arguments proposed by Dung [9] 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 denote it in the graph with an edge from X to Y. In what follows, $X^- = \{Y \in \mathcal{A} | (\text{Y,X}) \in \mathcal{R}\}$; and $X^+ = \{Y \in \mathcal{A} | (\text{X,Y}) \in \mathcal{R}\}$. For sets $E \subseteq \mathcal{A}$, E^- and E^+ are defined in an obvious way via set union. We write $E \to X$ as a shorthand for $X \in E^+$. The *path-equivalence relation* $\sim_{\mathcal{R}} \subseteq \mathcal{A}^{\in}$ 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. Arguments that have no attacks are always accepted. However, an attack from X to Y may not be sufficient to defeat Y, because X may itself be defeated, and thus the statuses of arguments need to be determined systematically. In Dung's original formulation, this is usually done through *acceptability* conditions for the arguments. A semantics can then be defined in terms of *extensions*—subsets of $\mathcal{A}$ with special properties. A set $E \subseteq \mathcal{A}$ is said to be *conflict-free* if for all elements $X, Y \in E$, we have that $(X, Y) \notin \mathcal{R}$. Although a conflict-free set only contains elements that do not attack each other, this does not necessarily mean that all arguments in the set are properly supported. Well-supported sets satisfy special *admissibility* criteria. An argument $X \in \mathcal{A}$ is *acceptable with respect to* E, if for all $Y \in X^-$, $E \cap Y^- \neq \varnothing$. A set E is *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

[1] For easier understanding the algorithm is broken into functional sub-components.

a *preferred extension* iff E is a $\subseteq$-maximal complete extension; and E is stable if E is preferred and $E \cup E^+ = \mathcal{A}$.

Dung's semantics can also 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 [4, 5, 15]. Let dom denote the domain of a function and λ a labelling function, we define $in(\lambda) = \{X \in \text{dom } \lambda | \lambda(\text{X}) = \mathbf{in}\}$; $und(\lambda) = \{X \in \text{dom } \lambda | \lambda(\text{X}) = \mathbf{und}\}$; and $out(\lambda) = \{X \in \text{dom } \lambda | \lambda(\text{X}) = \mathbf{out}\}$. The notion of extension is recovered from the set $in(\lambda)$ for some labelling function λ. Furthermore, 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 $X^- \cap in(\lambda) \neq \varnothing$. Finally, X is *super-illegally labelled* **in** if it is attacked by an argument that is legally labelled **in** or labelled **und** [12]. A labelling function is legal it does not illegally label any arguments.

2.1 Computing Extensions via Decomposition into SCCs

Baroni et al. proposed a general recursive schema for argumentation semantics in [1]. The schema employs the decomposition of an argumentation framework into SCCs and can be used to obtain Dung's admissibility-based semantics. Based on that, many researchers showed how to compute the extensions of argumentation frameworks under several semantics. Baumann adapted the Modgil-Caminada's algorithms [12] to compute extensions under the grounded, preferred and stable semantics in what he called "split" frameworks [2]. Preliminary experimental results of the advantages of these techniques were then shown in [3]. Liao described the use of the decomposition idea for computation of argumentation semantics in a more general way [11].

The overall process can be summarised as follows. Firstly, the SCCs of an argumentation framework are arranged into layers following the direction of attack. Then the solutions for each layer are computed using an appropriate algorithm for the semantics at hand and the solutions of the previous layers. Finally, the solutions of subsequent layers are combined in a systematic way. To illustrate this idea, consider the argumentation framework $\mathcal{N}$ in Fig. 1 with SCCs $S_1 = \{X\}$, $S_2 = \{W, Y\}$ and $S_3 = \{A, B, C, D, E\}$. Following the attack relation, these SCCs can be arranged into two layers, the first containing S_1 and S_2 and the second containing S_3. The solutions of the SCCs in a given layer are all 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. For example, the computation of the solutions of S_3 depends on the labels assigned to X and W, and thus on the solutions of S_1 and S_2. As S_1 and S_2 have no external attackers, their solutions can be computed completely independently of the rest of the framework. S_2 has three 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**. $X = \mathbf{in}$ is the only solution to S_1, so each of the partial solutions to S_2 must be augmented with the assignment

$X = \textbf{in}$, giving all partial solutions to layer 0: $f_1 : X = Y = \textbf{in}, W = \textbf{out}$, $f_2 \colon X = W = \textbf{in}, Y = \textbf{out}$, and $f_3 \colon X = \textbf{in}, W = Y = \textbf{und}$.[2] [11].

Now consider the computation of the solutions for S_3. We say that S_3's solutions are *conditioned* by the labels of the external attackers X and W in the partial solutions f_1, f_2 and f_3. In any such solution, $X = \textbf{in}$, but the label of W could be either **out**, **in** or **und**. In order to generate all complete extensions for $\mathcal{N}$, each partial solution f_1, f_2 and f_3 needs to be *expanded* with the solutions for S_3 under the constraints that they impose.

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 $\lambda_S^f : S \longmapsto \{\textbf{out},\textbf{und},\textbf{in}\}$ is a legal labelling function whose set in(λ_S^f) is $\subseteq$-minimal with respect to all legal labelling functions conditioned by f.*

λ_S^f is the "minimal" (grounded) solution for S under f. It is a special case of forward propagation from external attackers starting with the all undecided labelling (all-**und**). The Discrete Gabbay-Rodrigues Iteration Schema [10] is an example of a method that can perform this propagation very efficiently.

Since $f_1(X) = \textbf{in}$ and $f_1(W) = \textbf{out}$, the search for the solutions for S_3 conditioned by f_1 consists of the search for all solutions to S_3 with the constraint $A = \textbf{out}$ or the search of all possible ways to "expand" $\lambda_S^{f_1}$ by swapping labels from **und** to **in** or **out**. Similarly, since the $f_2(X) = f_2(W) = \textbf{in}$, under f_2 we need to satisfy the constraint $A = B = \textbf{out}$. A similar reasoning applies to solution f_3 in which we have the "implicit" constraint $\lambda_S^{f_3}(B) \neq \textbf{in}$ (since $\lambda_S^{f_3}(W) = \textbf{und}$). More generally speaking, the whole process can be thought of as follows: given a 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.

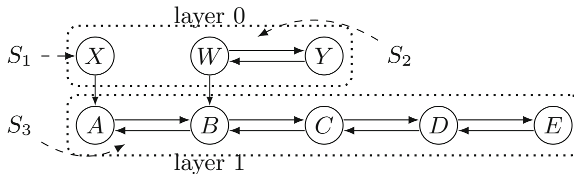


Fig. 1. A complex argumentation framework and its decomposition into layers.

Decomposition breaks the argumentation problem into smaller sub-problems, but an algorithm is still needed to find the solutions for each SCC. Modgil-Caminada's algorithm for preferred extensions is one algorithm that can be adapted for this [12].

[2] This is called the *horizontal combination* of solutions of the layer.

2.2 Modgil-Caminada's Algorithm for Preferred Extensions

For space limitations we cannot present Modgil-Caminada's algorithm in full, but we will describe it in general terms. This should suffice for our discussion.

Since preferred extensions are associated with maximal sets of arguments that are labelled **in**, Modgil-Caminada's algorithm starts with the labelling function that labels all arguments **in** (all-**in**) and then successively "corrects" illegally labelled arguments via a so-called *transition step*. Eventually, all illegal labels get corrected, and the set of arguments labelled **in** will correspond to an extension – those that are maximal will correspond to the preferred extensions.[3] A transition step consists of the following. If the argument X is illegally labelled **in**, then it is re-labelled **out**, if it can be legally re-labelled so. Otherwise it is re-labelled **und**. Afterwards, the labels of all arguments in X^+ that become illegally labelled **out** by the fact that X has been re-labelled from **in** to **out** or **und**, are then also changed to **und**. The algorithm applies transition steps as follows. If there is any argument X in λ that is super illegally labelled **in**, then the algorithm performs a single transition step on X generating a new labelling function λ' and then calls itself recursively from λ'. If there is no such argument, the algorithm will instead iterate through *all* arguments that are illegally labelled **in**; apply a transition step on each; and call itself recursively from the new labelling functions thus generated. Eventually, all labels will become legal and the algorithm will simply return the labelling functions with maximal sets of arguments labelled **in**.

In Sects. 4 and 5, we will see that the strategy used by Modgil-Caminada's algorithm may result in a very high number of operations.

3 A New Algorithm for Enumeration and Decision Problems of Argumentation Semantics

Our algorithm's strategy takes advantage of the constraints that a legal labelling function must satisfy. These constraints come from two sources: *(i)* the labels of the external attacking arguments in the conditioning solutions (which already partially determine the SCC's solution); and *(ii)* the internal constraints arising from re-labelling the seed argument **in**. The constraints *help* to reduce the search space. The successful implementation of this strategy relies on an efficient propagation mechanism (see Sect. 6) and a bottom-up method for constructing all extensions.

This way of looking into the problem has two major implications. By generating all complete extensions, the method can be used in problems involving the grounded, complete, stable and preferred semantics. For the grounded semantics, all we need to do is to propagate the (unique) conditioning solution; for the preferred semantics, we generate alternative solutions but only keep those that maximise the set of nodes labelled **in**; and for the stable semantics we exclude preferred solutions with undecided nodes. Secondly, because we only work on an

[3] Unlike ours, Modgil-Caminada's algorithm does not guarantee the generation of all complete extensions.

individual argument at a time, we can define decision procedures for argument acceptability that do not need to necessarily generate all extensions.

In order to lighten the notation, we will drop the subscript and superscript in λ_S^f when the context makes the SCC S and the conditioning solution f clear. Given a partial solution λ conditioned by a solution f, an argument X of an SCC S can potentially be re-labelled from **und** to **in** if it satisfies the following conditions: (I1) $\lambda(X) = \mathbf{und}$; (I2) $X \notin X^-$ (it does not attack itself); and (I3) $\{Y \in X^- | f(Y) = \mathbf{und}\} = \varnothing$. [4] The set $possIns_S \subseteq S$ is the set of nodes satisfying conditions (I1)–(I3). Thus the starting point for Algorithm 1 is an SCC S; the set $possIns_S$; a conditioning solution f for previous layers; and a partial solution λ for S conditioned by f. The algorithm will compute the set Λ of all complete (or preferred)[5] labelling functions that "expand" λ by successively searching for complete/preferred labelling functions that label an element of $possIns_S$ **in**. Each search is done via Algorithm 4, which we now explain.

Algorithm 1 Finding extensions from a given set of arguments

Input: $possIns_S$, a SCC S, a conditioning labelling function f, a conditioned legal labelling function λ for S, and a set of candidate labelling functions Λ
Output: true (success) or false (failure) and an updated set Λ

```
1 Function findExtsFromArgs(S,possIns_S,f,λ,Λ)
2     while possIns_S ≠ ∅ do
3         Pick X ∈ possIns_S
4         possIns_S ← possIns_S \ {X}
5         findExtsFromArg(X, S, f, λ, Λ)
6     end while
7 end
```

In Sect. 2, we saw that an argument X is legally labelled **in** in a solution λ if all arguments that it attacks are labelled **out** and that if all arguments that attack X are labelled **out** then X must be labelled **in** in λ. Thus, to re-label X **in** we must re-label **out** all arguments that it attacks. By re-labelling some arguments **out**, we may also be forced to re-label **in** some other arguments, and so forth. We call this process the *forward propagation of the* **in** *label*. All attackers of X must also be labelled **out** for X to be legally labelled **in** in λ. Thus, all external attackers of X must be labelled **out** (by f) and all internal attackers that are still labelled **und** must be re-labelled **out**. This can be done by ensuring that every internal attacker Y that is labelled **und**, gets an attacker Z to be legally re-labelled **in**. We call the process of ensuring that all attackers of X are legally labelled **out** *the backward propagation of the* **in** *label*. Backward propagations can be done in terms of one or more forward propagations and this is the motivation for the title of the algorithm.

Although we start with "good enough" candidates, i.e., arguments satisfying (I1)–(I3), both types of propagations may fail, since we have no control over the assignments of conditioning solutions and the propagations may result in inconsistent label requirements. A failed propagation simply means that we

[4] We know that $\lambda(X) = \mathbf{und}$ by (I1), but we still want to make sure that X can be re-labelled **in** which is not the case if an external attacker $Y \in X^-$ has $f(Y) = \mathbf{und}$.

[5] The set Λ is updated according to the desired semantics (see Sect. 3.3).

cannot construct a legal labelling function meeting the required constraints, so we backtrack to any available alternatives. We now explain the details.

3.1 Propagating Forwards

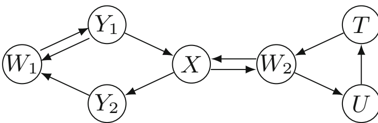
A forward propagation essentially requires changing the label of a *seed node* in a partial solution λ from **und** to **in** and then following the direction of attacks to re-label any nodes that may have thus have been rendered illegally labelled. One important aspect of a forward propagation is that (if successful) it will generate a single solution λ' from a partial solution λ which, by construction, has *less* undecided nodes than λ itself. A forward propagation is carried out by the function `propagateIN` in Algorithm 2. Figure 2 illustrates the labelling function λ' obtained as the result of a successful forward propagation from $X = \textbf{in}$ and $f = \varnothing$ and a failed forward propagation from $W_2 = \textbf{in}$ and $f = \varnothing$. The latter fails because by labelling W_2 **in**, we must label U **out**, which then requires T to be labelled **in**, which in turn requires W_2 to be labelled **out**, which is not possible.

Algorithm 2 Forward propagation of an IN label

Input: argument X to label **in**, its SCC S, a partial legal labelling function λ, and a conditioning labelling function f

Output: false if failure; or true if successful, with the new partial labelling function λ'

```
1  Function propagateIN(X,S,f,λ,λ')
2      if {Z ∈ X⁺ | λ(Z) = in} ≠ ∅ then
3          return false
4      else
5          λ' ← λ; λ'(X) = in
6          forall Y ∈ {Z ∈ X⁺ | λ'(Z) = und} do
7              λ'(Y) ← out
8          end forall
9          newIns← {Z ∈ S | λ'(Z) = und and for all Y ∈ Z⁻, λ'(Y) = out}
10         while newIns ≠ ∅ do
11             Pick W ∈ newIns
12             if propagateIN(W,S,f,λ',λ'') then
13                 λ'← λ''
14                 newIns← newIns\{W}
15             else
16                 return false
17             end if
18         end while
19         return true
20     end if
21 end
```



SCC S, $f = \varnothing$, λ is all-**und**

`propagateIN`$(X, S, f, \lambda, \lambda')$ succeeds with $in(\lambda') = \{X, U\}$, $out(\lambda') = \{T, W_2, Y_2\}$, $und(\lambda') = \{W_1, Y_1\}$

`propagateIN`$(W_2, S, f, \lambda, \lambda')$ fails since we cannot label W_2 both **in** and **out**

Fig. 2. Results of forward propagations from $X = \textbf{in}$ and $W_2 = \textbf{in}$.

If the forward propagation from a node X is successful, we must then ensure that all of X's attackers are legally labelled **out** in order to guarantee that the solution is legal. This is done by a backward propagation.

3.2 Propagating Backwards

In the example in Fig. 2, it is easy to see that λ' is not legal, since $\lambda'(X) = \mathbf{in}$, $Y_1 \rightarrow X$, but $\lambda'(Y_1) = \mathbf{und}$. We can perform a backward propagation from X by performing one or more forward propagations using any of the attackers of each attacker of X as the seed. $X^- = \{Y_1, W_2\}$, so we want a labelling function that labels at least one of the arguments in Y_1^- *and* in W_2^- **in** (X itself already satisfies the latter). It is easy to see that the labelling function $\lambda'' = \{X = U = W_1 = \mathbf{in}, Y_2 = W_2 = T = Y_1 = \mathbf{out}\}$ satisfies these requirements.

Naturally, a backward propagation may also fail. Consider the network in Fig. 3. After a successful forward propagation from $X = \mathbf{in}$, $f = \varnothing$, and λ=all-**und**, we get the labelling function $\lambda' = \{X = \mathbf{in}, Y = \mathbf{out}, W_1 = W_2 = W_3 = \mathbf{und}\}$, which is not legal, since $W_3 \rightarrow X$ and $\lambda'(W_3) \neq \mathbf{out}$. So we attempt to backward propagate from X, $f = \varnothing$ and λ'. We need to label W_3 **out**, which requires labelling $W_2 = \mathbf{in}$, which is not possible since it attacks itself, and thus the backward propagation fails. What this means in practice is that X cannot be part of any extension (this reasoning can be used in decision problems).

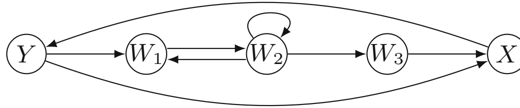
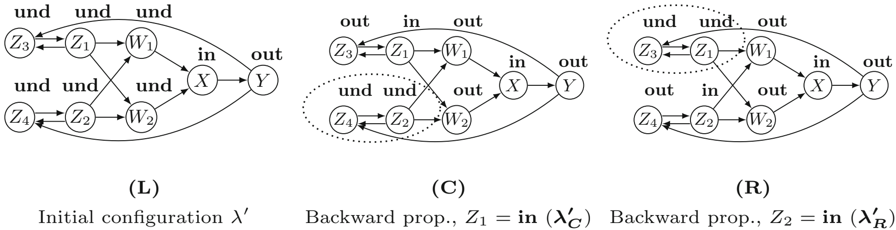


Fig. 3. Backward propagation.

Unlike a forward propagation, a backward propagation can generate multiple labelling functions. Consider the SCC S in the network in Fig. 4(L). A call to `propagateIN`$(X, S, f, \lambda, \lambda')$ will succeed with $\lambda' = \{X = \mathbf{in}, Y = \mathbf{out}, Z_1 = Z_2 = Z_3 = Z_4 = W_1 = W_2 = \mathbf{und}\}$. We must now legally label both W_1 and W_2 **out**. But here we have a choice between labelling Z_1 or Z_2 **in**. So both $\boldsymbol{\lambda'_C} = \{X = \mathbf{in}, Y = \mathbf{out}, Z_1 = \mathbf{in}, W_1 = W_2 = Z_3 = \mathbf{out}, Z_2 = Z_4 = \mathbf{und}\}$ (Fig. 4(C)) and $\boldsymbol{\lambda'_R} = \{X = \mathbf{in}, Y = \mathbf{out}, Z_2 = \mathbf{in}, W_1 = W_2 = Z_4 = \mathbf{out}, Z_1 = Z_3 = \mathbf{und}\}$ (Fig. 4(R)) are returned in Λ from an invocation to `propagateOUT`$(X, S, f, \lambda', \Lambda)$.



(L) Initial configuration λ' **(C)** Backward prop., $Z_1 = \mathbf{in}$ ($\boldsymbol{\lambda'_C}$) **(R)** Backward prop., $Z_2 = \mathbf{in}$ ($\boldsymbol{\lambda'_R}$)

Fig. 4. A sample network and two successful backward propagations from $X = \mathbf{in}$.

There are two more important considerations to make. First, all of the attackers of the seed node must be labelled **out**. Therefore any solution returned by `propagateOUT` must satisfy this requirement. Our implementation approach in Algorithm 3 was to work with two lists. *makeOuts* contains the nodes that still need to be labelled **out** and starts with all undecided attackers of the seed node (Algorithm 3, line 5). At least one solution must be found labelling all of these nodes **out**. If this is not possible, `propagateOUT` simply fails (Algorithm 3, line 27). This essentially complements `propagateIN` to guarantee the *correctness* of the algorithm. The solutions are stored in the list *sols*, which is initialised with the result of the forward propagation of the seed node (Algorithm 3, line 12). For each node in *makeOuts*, *sols* is replaced with a new set of satisfying solutions. Each successive node is then checked against all new solutions thus generated which, by construction, label **out** all of the previously removed nodes in *makeOuts*. If we successfully exhaust all of the nodes in *makeOuts*, then `propagateOUT` succeeds and returns all corresponding solutions (line 30). Otherwise, it fails and Λ is not updated.

Algorithm 3 Backward propagation of an IN label

Input: argument X labelled **in**, its SCC S, a conditioning labelling function f, and a labelling function λ' obtained from propagating $X =$ **in** forward

Output: false or true with a set of new partial labelling functions Λ

```
1  Function propagateOUT(X,S,f,λ',Λ)
2    if there exists W ∈ X⁻ such that W is in a previous layer and f(W) ≠ out or there exists
       W ∈ X⁻ such that W ∈ S and f(W) = in then
3      return false
4    else
5      makeOuts(X)← {W ∈ X⁻ | W ∈ S and λ'(W) = und}
6      forall W ∈ makeOuts(X) do
7        makeIns(W)← {Z ∈ W⁻ | Z ∈ S and λ'(Z) = und}
8        if makeIns(W) = ∅ then
9          return false
10       end if
11     end forall
12     sols← {λ'}
13     while makeOuts(X) ≠ ∅ do
14       Pick W ∈ makeOuts(X) such that |makeIns(W)| is minimal
15       makeOuts(X)← makeOuts(X)\{W}
16       newSols← ∅
17       forall λ' ∈ sols do
18         forall Y ∈ makeIns(W) do
19           if findExtsFromArg(Y,f,λ',newSols) then
20             success← true
21           end if
22         end forall
23       end forall
24       if success then
25         sols← newSols
26       else
27         return false
28       end if
29     end while
30     Λ← newSols
31     return true
32   end if
33 end
```

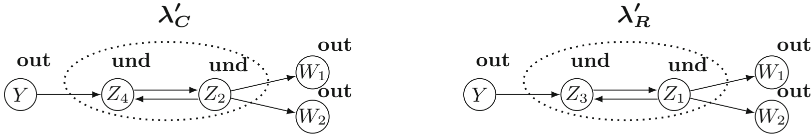


Fig. 5. Undecided sub-cycles within solutions.

The story does not end here though, and this takes us to the second important consideration which has to do with *completeness*. The result of a successful backward propagation may still leave some nodes of an SCC in what are effectively *induced sub-SCCs ring-fenced by* **out**-*labelled nodes*. Consider the network of Fig. 4(L) again. In order to legally label X **in** we need to label W_1 and W_2 **out**. We have seen that this can be done by labelling either Z_1 or Z_2 **in**, giving us the solutions $\boldsymbol{\lambda'_C}$ (extension $\{X, Z_1\}$) and $\boldsymbol{\lambda'_R}$ ($\{X, Z_2\}$) of Fig. 4(C) and (R), respectively. However, $\boldsymbol{\lambda'_C}$ leaves Z_2 and Z_4 undecided, whereas $\boldsymbol{\lambda'_R}$ leaves Z_1 and Z_3 undecided. In order to break these cycles (and hence guarantee completeness w.r.t. all complete extensions), all we have to do is to simply treat Z_4–Z_2 and Z_3–Z_1 as "sub-SCCs" and restart the whole process from the same original conditioning solution but now with initial conditioned solutions $\boldsymbol{\lambda'_C}$ and $\boldsymbol{\lambda'_R}$ (see Fig. 5). This is implemented in lines 8 and 10 of Algorithm 4. In our example, $\boldsymbol{\lambda'_C}$ will generate sub-solutions $Z_4 = \mathbf{in}$, $Z_2 = \mathbf{out}$ and $Z_4 = \mathbf{out}$, $Z_2 = \mathbf{in}$; whereas $\boldsymbol{\lambda'_R}$ will generate sub-solutions $Z_3 = \mathbf{in}$, $Z_1 = \mathbf{out}$ and $Z_3 = \mathbf{out}$, $Z_1 = \mathbf{in}$. The search will eventually terminate because recursive calls are only made with initial solutions containing less **und** labels than their parents' and the fact that the argumentation graph is finite.

3.3 Combining All Steps

Algorithm 1 will attempt to label **in** all candidate arguments that can be possibly labelled **in**. We then generate all possible solutions starting from each of these arguments with Algorithm 4. This requires to attempt to propagate forward from $X = \mathbf{in}$ (line 3). If this is successful, it will generate a new labelling function λ'' with at least two less undecided arguments than λ'. We then attempt to propagate backwards from λ'' (line 5), to guarantee that all attackers of X are legally labelled **out**. If this is successful, it will generate a number of possible solutions Λ', which we add to the current set of solutions (line 7). These solutions may still leave some undecided nodes, so we restart the process from each solution σ in Λ' and the remaining candidate undecided nodes (lines 8 and 10), adding again the results to the set of solutions (line 12). At this point, we can filter out the solutions that do not yield preferred extensions if needed (see Algorithm 5).

Algorithm 4 Finding extensions from a given argument

Input: argument X to label **in**, its SCC S, a conditioning labelling function f, a legal labelling function λ, and a set of candidate labelling functions Λ

Output: false or true with an updated set of candidate labelling functions Λ

```
1  Function findExtsFromArg(X,S,f,λ,Λ)
2    λ' ← λ; λ'(X) = in
3    if propagateIN(X,S, f,λ',λ'') then
4      Λ' ← ∅
5      if propagateOUT(X,f,λ'',Λ') then
6        for σ ∈ Λ' do
7          updateExts(σ,Λ)
8          possIns_S ← {Y ∈ S|σ(Y) = und, Y ∉ Y+, {X ∈ Y−\S|f(X) = und} = ∅}
9          if possIns ≠ ∅ then
10           Λ'' ← ∅; findExtsFromArgs(S,possIns_S,f,σ,Λ'')
11           for σ' ∈ Λ'' do
12             updateExts(σ',Λ)
13           end for
14         end if
15       end for
16     else
17       return false
18     end if
19   else
20     return false
21   end if
22 end
```

Algorithm 5 Updating the set of candidate solutions

Input: a solution λ and a set of candidate solutions Λ

Output: an updated set of candidate solutions Λ, according to the semantics

```
1 Function updateExts(λ,Λ)
2   if preferred semantics then
3     Remove all solutions γ in Λ whose set of in-nodes is contained in the set of in-nodes of λ
4   end if
5   Λ ← Λ ∪ {λ}
6 end
```

Proposition 1 (Soundness and Completeness). *Let S be an SCC, f an admissible conditioning labelling function, and λ a labelling function for S conditioned by f (cf. Definition 1), then (1) all labelling functions returned by Algorithm 1 are legal; and (2) these are all the legal labelling functions for S.*

Proof. Omitted, but soundness comes from the fact that lines 7 and 12 of Algorithm 4 only add legal labelling functions and completeness from the facts that all alternative solutions are tried in line 2 of Algorithm 1 and line 10 of Algorithm 4.

4 Analysis and Comparisons with Other Work

In Sect. 2, we briefly described the Modgil-Caminada's algorithm for preferred extensions and mentioned that it could behave very inefficiently. In fact, Charwat *et al.* pointed out that for the class of argumentation frameworks $\langle \mathcal{A}, \mathcal{A}^2 \rangle$, the algorithm produces $n!$ branches (where $n = |\mathcal{A}|$), all with the same extension [8]. Since each node in each branch of execution corresponds to a transition step, the total number of transition steps is at least twice as many. In fact, it is $n! + \sum_{i=1}^{n-1} \frac{n!}{(n-i)!} \geq 2n!$, to be precise. Although arguably unrealistic, this

class of argumentation frameworks is particularly hard for Modgil-Caminada's algorithm, but it is dealt with trivially by our algorithm, requiring only n steps to identify that no nodes can be possibly labelled **in** and then producing the empty extension. This is because the higher the degree of attacks, the higher the degree of constraints and hence the lower the number of alternatives to check by our algorithm. Perhaps a more interesting class of frameworks to compare is what we call *bi-directed cycle graphs* involving a cycle with all nodes in both directions (see Fig. 6 (Start)). We now discuss the behaviour of both algorithms for this class of graphs. Modgil-Caminada's algorithm would start with the all-**in** labelling function and hence all nodes would be initially illegally labelled **in**. None is super-illegally labelled **in**, so the algorithm would iterate through *all* nodes, performing a transition step on each one and then recursively calling itself with the labelling functions resulting from the transitions. For the sake of argument, let us assume that the algorithm picks node A_1 first. A_1's label would be changed from **in** to **out**. As it would become legally labelled **out** and none of the nodes that it attacks is labelled **out**, the first transition step would result in the labelling TS1 of Fig. 6.[6] None of the nodes in TS1 are legally labelled **in** or super-illegally labelled **in**, so the algorithm would then again iterate through all nodes that remain illegally labelled **in** (4 in total). In the branches that pick a node adjacent to A_1, say A_2, the following would happen. The algorithm would change A_2's label to **out** (which is illegal), and then to **und**. A_1 is the only node that A_2 attacks and it is labelled **out**, but it is legally labelled so. The algorithm would then choose from one of the remaining illegally labelled nodes (of which there would be 3). If it agains picks an adjacent node, say A_3, it would change its label to **out** and then to **und**, and this process would continue until all nodes were re-labelled **und**. This sequence of transitions is depicted in graphs TS1, TS2,..., TS5 of Fig. 6. The algorithm would eventually pick A_3 or A_4 as an alternative choice to A_2 and in those branches it would eventually produce the preferred extensions. However, the number of recursive calls would still remain *close* to factorial (See Fig. 7). Our algorithm by contrast would start with all nodes labelled **und** and pick any initial seed node. In enumeration problems the choice is actually irrelevant as all eligible undecided nodes are attempted. In decision problems, we can start with an argument of interest and continue only if an appropriate extension can be constructed. If we start by propagating $A_1 =$ **in**, we are immediately forced to label A_2 and A_5 **out**, giving us only two further choices to generate the preferred extensions, i.e., either to label $A_3 =$ **in** or to label $A_4 =$ **in**. Figure 7 shows the number of transition steps performed by Modgil-Caminada's algorithm in bi-directed cycle graphs of up to 24 nodes and the number of recursive calls in our own algorithm (both implemented in EqArgSolver). For comparison, we included the factorial and 2^x functions.

In [6], Cerutti et al.'s proposed a meta-algorithm that decomposes the original argumentation framework into SCCs and uses a "base algorithm" at the base of the recursion to solve the original problem at the SCC level. As an illustration of the approach, the base algorithm employed a SAT solver. It should be possible

[6] There is an analogous branch for all other arguments A_2,...,A_5.

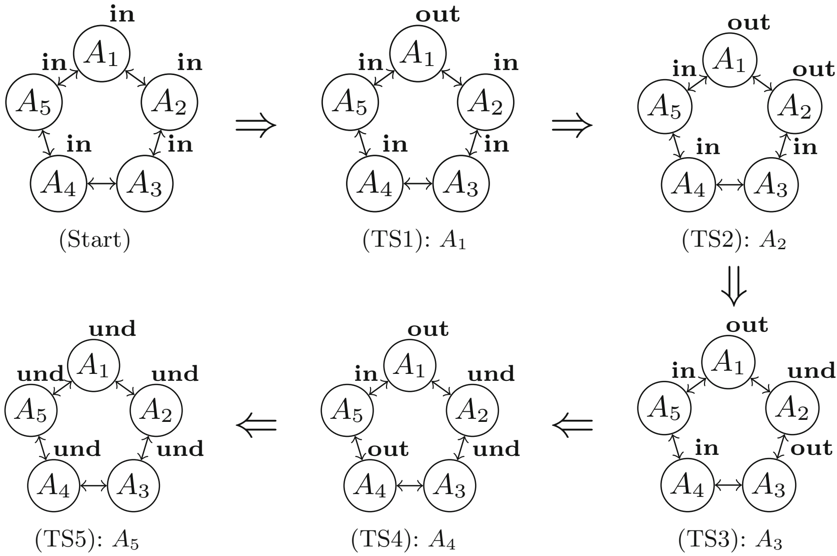


Fig. 6. Bi-directed cycle graph behaviour

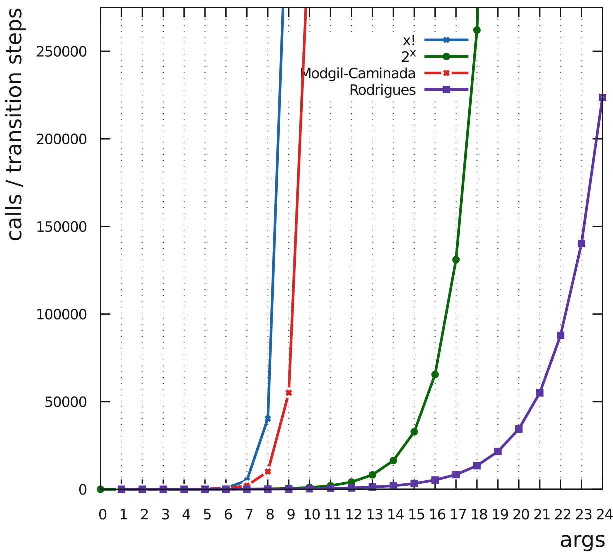


Fig. 7. Transition steps × calls to `propagateIN` and `propagateOUT`

to swap the algorithm here proposed for the call to the SAT solver [6, Line 19, Algorithm 2] or vice-versa using an appropriate translation of the problem, since a conditioning solution simply constrains the set of possible models. This investigation will appear in a forthcoming paper. Finally, Nofal et al. proposed

algorithms for decision problems in the preferred semantics [13]. The algorithm presented here is not restricted to this semantics only. We will however compare the approaches of these algorithms and ours in future work.

5 Empirical Evaluation

Apart from the special cases discussed above, we also conducted some experiments to compare Modgil-Caminada's algorithm with ours in randomly generated graphs. Our objective was not to conduct an extensive empirical evaluation between general solvers, as this will be done by the 2nd International Competition of Computational Models of Argumentation (ICCMA), but merely to provide a first-hand evaluation of the two labelling approaches. In order to eliminate any implementation factors that could directly affect the comparison between the two, they were both embedded within two versions of EqArgSolver which was invoked for the preferred semantics only. For further comparison we also recorded the results provided by Tweetysolver v1.2, which also uses decomposition into SCCs but uses a SAT solver for solutions. Tweetysolver was chosen because it is an off-the-peg easy-to-deploy solver and a "good enough" initial marker for the performance of SAT-based solvers *in this class of problems*.

We generated 3 datasets of 1,000 graphs each with maximum cardinality of 15, 25 and 35 nodes using `probo`'s SCC generator. The maximum number of SCCs in each graph was set to 2. Each dataset was divided into 10 sets of 100 graphs with probability $p = 0.1$, $p = 0.2$, ..., $p = 1$ of a node attacking another within an SCC. We submitted the 3,000 graphs thus generated to the solvers running on a PC with an Intel i7 4690 K processor and 32 Gb RAM. The left of Fig. 8 shows the comparative average time per graph successfully solved by each solver and the right shows the percentage of instances timed out within 180 s.

The graphs turned out to be rather too small to effectively stress test EqArgSolver using our algorithm. However, they clearly show the differences in performance between the two algorithms (and Tweetysolver). Both the version of EqArgSolver using our algorithm as well as Tweetysolver successfully solved all graphs submitted within the time limit. As expected, the version using Modgil-Caminada's algorithm timed out more frequently the more nodes the datasets contained. For graphs with up to 15 nodes, it timed out in roughly 10% of the problems, increasing to 40% of timeouts in graphs with up to 25 nodes; and then to 70% timeouts in graphs with up to 35 nodes. The actual average time per graph successfully solved varied rather erratically in the version using Modgil-Caminada's algorithm and this deserves further investigation. Our algorithm was clearly the fastest (just above 0 ms per graph on average). The execution times for Tweetysolver stayed relatively constant at around 1,000–1,250 ms per graph in all datasets. This shows some advancements in catching up with SAT reduction approaches.[7]

[7] A more robust SAT-based argumentation solver would employ special techniques to maximise the performance of the underlying SAT solver.

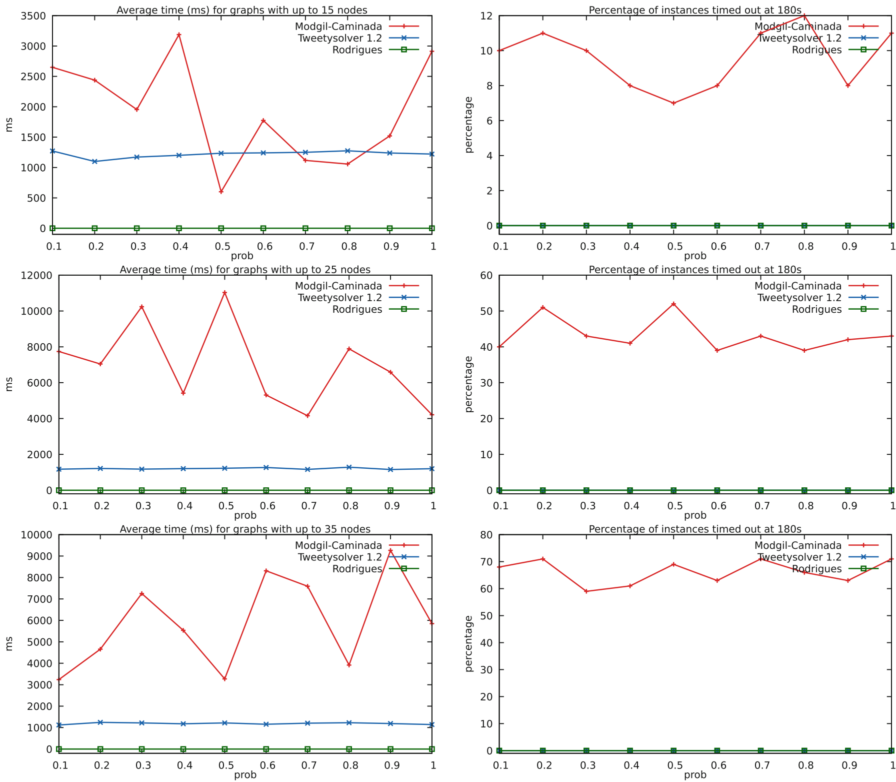


Fig. 8. Average execution time and % of time-outs for graphs with up to 15, 25 and 35 nodes.

6 Conclusions and Future Work

It is well known that the computation of grounded extensions is simply a matter of propagation of the **in** labels of unattacked arguments, which can be done very efficiently using the Discrete Gabbay-Rodrigues Iteration Schema [10]. In this paper we proposed a novel algorithm for the computation of all other complete extensions by looking for solutions to the SCCs of an argumentation framework. With minor modifications the algorithm can be used for the preferred and stable semantics as well.

The motivation for the development of this algorithm came from the following. In the solver GRIS [14], we used Modgil-Caminada's algorithm to compute the preferred extensions of an argumentation framework. However, Modgil-Caminada's algorithm proved very inefficient for all but the simplest graphs and can only compute the preferred extensions. We wanted a more efficient algorithm that could compute all complete extensions and that could also check argument acceptability without necessarily having to generate all extensions. The algorithm here proposed achieves all that and successfully replaced

Modgil-Caminada's algorithm in the solver EqArgSolver, which we submitted to the 2nd ICCMA (see http://argumentationcompetition.org/).

Given that solvers using reduction-based approaches to the computation of argumentation semantics took the top spots in the 1st ICCMA, the reader might ask if the development of direct algorithms and tools for argumentation semantics is worthwhile or whether we should simply concentrate on improving the reduction-based techniques. We would side with Cerutti et al.'s to argue that both approaches have a role to play [7] and combining them could be advantageous. In addition, we would claim that direct approaches are the only alternative in applications for which a translation to logic is either not possible at all or very cumbersome, e.g., in certain numerical argumentation networks.

We tested the new algorithm over tens of thousands of graphs of cardinality of up to 100,000 nodes. Rather than the number of nodes in the framework as a whole, it is the complexity and the number of SCCs involved that can stress a solver using the algorithm. Although some of these characteristics are unavoidable and intrinsic to the problem, the complexity could be reduced in our algorithm by avoiding multiple generation of the same solution arising in different search branches. As it stands, we attempt to label **in** every candidate argument in an SCC in order to guarantee the completeness of the set of solutions found, but this could be improved. Optimisations in this area are under investigation.

A further point to make is that within an SCC we can start the algorithm at an argument of interest to aid in decision problems of argument acceptability. It should also be possible to work backwards from a specific argument to see if an extension containing it can be constructed. This is work in progress.

Finally, each **in** labelling of a node forces the arguments that it attacks to be labelled **out**, which means that in each non-trivial SCC, each forward propagation reduces the complexity of the original problem by at least two arguments, but possibly many more in cases where the seed node attacks multiple arguments. We therefore expect the probability of attacks between nodes within an SCC to be inversely proportional to the execution time of our algorithm. This needs to be fully demonstrated and we also want to compare the performance of our algorithm with Nofal et al.'s [13] which, as for Modgil-Caminada's algorithm, can only generate the preferred extensions.

References

1. Baroni, P., Giacomin, M., Guida, G.: SCC-recursiveness: a general schema for argumentation semantics. Artif. Intell. **168**(1), 162–210 (2005)
2. Baumann, R.: Splitting an argumentation framework. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS (LNAI), vol. 6645, pp. 40–53. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20895-9_6
3. Baumann, R., Brewka, G., Wong, R.: Splitting argumentation frameworks: an empirical evaluation. In: Modgil, S., Oren, N., Toni, F. (eds.) TAFA 2011. LNCS (LNAI), vol. 7132, pp. 17–31. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29184-5_2

4. Caminada, M.: A labelling approach for ideal and stage semantics. Argum. Comput. **2**(1), 1–21 (2011)
5. Caminada, M., Gabbay, D.M.: A logical account of formal argumentation. Stud. Log. **93**(2–3), 109–145 (2009)
6. Cerutti, F., Giacomin, M., Vallati, M., Zanella, M.: A SCC recursive meta-algorithm for computing preferred labellings in abstract argumentation. In: 14th International Conference on Principles of Knowledge Representation and Reasoning (2014)
7. Cerutti, F., Vallati, M., Giacomin, M.: Where are we now? State of the art and future trends of solvers for hard argumentation problems. In: Baroni, P., Gordon, T., Scheffler, T. (eds.) Proceedings of COMMA, Frontiers in Artificial Intelligence and Applications, vol. 287, pp. 207–218. IOS Press (2016)
8. Charwat, G., Dvořák, W., Gaggl, S.A., Wallner, J.P., Woltran, S.: Methods for solving reasoning problems in abstract argumentation a survey. Artif. Intell. **220**, 28–63 (2015)
9. Dung, P.M.: On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. Artif. Intell. **77**, 321–357 (1995)
10. 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)
11. Liao, B.: Toward incremental computation of argumentation semantics: a decomposition-based approach. Ann. Math. Artif. Intell. **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, Boston (2009). https://doi.org/10.1007/978-0-387-98197-0_6
13. Nofal, S., Atkinson, K., Dunne, P.E.: Algorithms for decision problems in argument systems under preferred semantics. Artif. Intell. **207**, 23–51 (2014)
14. 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)
15. Wu, Y., Caminada, M.: A labelling-based justification status of arguments. Stud. Log. **3**(4), 12–29 (2010)