2013-9

Computing the Grounded Semantics in all the Subgraphs of an Argumentation Framework: an Empirical Evaluation

Pierpaolo Dondio
Technological University Dublin, pierpaolo.dondio@tudublin.ie

Recommended Citation

Pierpaolo D.: Computing the Grounded Semantics in all the Subgraphs of an Argumentation Framework: an Empirical Evaluation. CLIMA, XIV Workshop on Computational Logic in Multi-Agent Systems, *Lecture Notes in Artificial Intelligence*, Springer, 2013. doi:10.1007/978-3-642-40624-9_8

Computing the Grounded Semantics in all the Subgraphs of an Argumentation Framework: An Empirical Evaluation

Pierpaolo Dondio[1]

[1] School of Computing, Dublin Institute of Technology, Kevin Street 2, Dublin, Ireland

pierpaolo.dondio@dit.ie

Abstract. Given an argumentation framework $AF = (Ar, R)$ – with Ar a finite set of arguments and $R \subseteq Ar \times Ar$ the attack relation identifying the graph G – we study how the grounded labelling of a generic argument $a \in Ar$ varies in all the subgraphs of G. Since this is an intractable problem of above-polynomial complexity, we present two non-naïve algorithms to find the set of all the subgraphs where the grounded semantic assigns to argument a a specific label $l \in \{in, out, undec\}$. We report the results of a series of empirical tests over graphs of increasing complexity. The value of researching the above problem is two-fold. First, knowing how an argument behaves in all the subgraphs represents strategic information for arguing agents. Second, the algorithms can be applied to the computation of the recently introduced probabilistic argumentation frameworks.

Keywords: Argumentation Theory, Semantics, Algorithms

1 Introduction

An abstract argumentation framework AF is a directed graph where nodes represent arguments and arrows represent the attack relation. AFs were introduced by Dung [2] to analyze properties of defeasible arguments.

The problem investigated in this paper is the following: given an argumentation framework $AF = (Ar, R)$ – with Ar a finite set of arguments and $R \subseteq Ar \times Ar$ the attack relation identifying the graph G – we study how the grounded labelling of a generic argument $a \in Ar$ varies in all the subgraphs of G. Since this is an intractable problem of above-polynomial complexity, we present two algorithms, one recursive and one modelled as a decision-tree, to find the set of all the subgraphs where the grounded semantic assigns to an argument a a specific label $l \in \{in, out, undec\}$.

The value of researching the above problem is two-fold. First, knowing how an argument behaves in all the subgraphs of an argumentation graph helps us to understand the sensitivity of the argument label to the removal of other arguments via further attacks. This represents strategic information for agents in pursuing a discussion, since they can identify which arguments should be attacked.

However, the main motivation is represented by the recently introduced probabilistic argumentation frameworks. In such frameworks, the computation of the probability of acceptance of arguments requires the identification of all the subgraphs where a certain label for an argument holds (this is known as the *constellation approach* [6]).

This first work only presents algorithms and results for grounded semantics. This is mainly due to space limitations and the fact that the versions of our algorithms for other semantics have not been yet implemented and therefore an empirical evaluation cannot be made. However, the idea behind the algorithms proposed is general enough to be applied to other semantics. Our recursive algorithm is based on constraints valid for any complete semantics and we have already presented a version for preferred semantics in [11]. The core mechanism of our decision-tree algorithm, based on splitting subgraphs and removing irrelevant arguments, is valid for any complete semantics and it can be extended to specific semantics by modifying the treatments of cyclic subgraphs.

The paper is organized as follows: section 2 presents the required background of abstract argumentation; section 3 sets the problem with the required definitions and presents a brute-force algorithm; section 4 describes the recursive algorithm; section 5 describes our decision-tree algorithm; section 6 reports the results of our experimental evaluation before the description of related works in section 7 and conclusions.

2 Background Definitions

Definition 1 (Abstract Argumentation Framework) Let U be the universe of all possible arguments. An argumentation framework is a pair (Ar, R) where Ar is a finite subset of U and $R \subseteq Ar \times Ar$ is called attack relation. We define an argument a **initial** if $\nexists b \in Ar \mid R(b, a)$, i.e. the argument is not attacked.

Let's consider $AF = (Ar, R)$ and $Args \subseteq Ar$.

Definition 2 (defense) $Args$ defends an argument $a \subseteq Ar$ iff $\forall$b ϵ Ar such that $R(b, a), \exists\ c \ \epsilon\ Args$ such that $R(c, b)$. The set of arguments defended by $Args$ is denoted $F(Args)$.

Definition 3 (indirect attack/defense) Let $a, b \in Ar$ and the graph G defined by (Ar, R). Then (1) a indirectly attacks b if there is an odd-length path from a to b in the attack graph G and (2) a indirectly defends b if there is an even-length path (with non-zero length) from a to b in G.

Labelling A semantics identifies a set of arguments that can survive the conflicts encoded by the attack relation R. In the labelling approach a semantics assigns a label to each argument. Following [4], the choice for the set of labels is: in, out or $undec$.

Definition 4 (Labelling/conflict free). Let $AF = (Ar, R)$ be an argumentation framework. A labelling is a total function L : $Ar \rightarrow \{in, out, undec\}$. We write $in(L)$ for $\{a \epsilon\ Ar \mid L(a) = in\}$, $out(L)$ for $\{a \epsilon\ Ar \mid L(a) = out\}$, and $undec(L)$ for $\{a \epsilon\ Ar \mid L(a) = undec\}$. We say that a labelling is conflict-free if no in-labelled argument attacks an (other or the same) in-labelled argument

Definition 5 (complete labelling). Let $AF = (Ar, R)$ be an argumentation framework. A complete labelling is a labelling that for every $a \in Ar$ holds that:

1. if a is labelled in then all attackers of a are labelled out
2. if all attackers of a are labelled out then a is labelled in
3. if a is labelled out then a has an attacker labelled in
4. if a has an attacker labelled in then a is labelled out
5. if a is labelled $undec$ then it has at least one attacker labelled $undec$ and it does not have an attacker labelled in.

Theorem 1, Grounded Labelling. (proved in [4]) Let $AF = (Ar, R)$ be an argumentation framework. L is the grounded labelling iff L is a complete labelling where $undec(L)$ is *maximal* (w.r.t. set inclusion) among all complete labellings of AF.

In figure 1 two argumentation graphs are depicted. The grounded semantics assigns the status of $undec$ to all the arguments of (A) (always when there are no initial arguments), while in (B) it assigns in to a and c, and out to b. Note how a reinstates c.

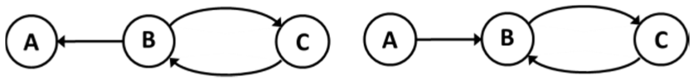


Figure 1. Two Argumentation Graphs (A) and (B)

3 Describing and Labelling Subgraphs

Given an argumentation framework $AF = (Ar, R)$ with $|Ar| = n$, and the graph G identified by Ar and R, we consider the set H of all the subgraphs of G. We focus on particular sets of subgraphs, i.e. elements of 2^H. Given $a \in Ar$, we define:

$$A = \{g \in H \mid a \in g\} \quad ; \quad \bar{A} = \{g \in H \mid a \notin g\}$$

that are respectively the set of subgraphs where argument a is present and the set of subgraphs where a is not present (note how we use $\bar{A}$ for the complementary set A^C). If $Ar = \{a_1, .., a_n\}$, a single subgraph g can be expressed by an intersection of n sets A_i or $\bar{A_i}$ $(0 < i \leq n)$ depending on whether the i^{th} argument a_i is or is not contained in g.

In general, we can express a set of subgraphs combining some of the sets $A_1, .., A_n, \overline{A_1}, .., \overline{A_n}$. with the connectives $\{\cup, \cap\}$. We write AB to denote $A \cap B$ and $A + B$ for $A \cup B$. For instance, in figure 1 the single subgraph with only b and c present is denoted with $\bar{A}BC$, while the expression AB denotes a set of two subgraphs where arguments a and b are present and c can be either present or not.

We call a *clause* φ a finite intersection (or conjunction) of sets A_i, $\bar{A_i}$. We consider expressions of sets of subgraphs in their *disjunctive normal form*, i.e. as a finite disjunction of clauses $\varphi_1 + \varphi_2 + .. + \varphi_m$. An expression is said to be in *standard form* if $\varphi_j \cap \varphi_i = \emptyset$, for each $i \leq m, j \leq m, j \neq i$. The standard form is made of disjoint sets of subgraphs and it is of particular interest for its applications to probabilistic argumentation. As an example, let's consider the argumentation graph in fig.1 left. The clause $A + B$ is not in standard form. It identifies six out of eight possible subgraphs (the two left out are the one where a, b and c are not present and the one with a and b not present and c present). A standard form is for instance $A + \bar{A}B$.

3.1 Grounded Labelling of Subgraphs

Given a subgraph $g \in H$, the labelling of g simply follows the rules of the chosen semantics. We therefore define a *subgraph labelling* $\mathcal{L}$ as a total function over the Cartesian product of arguments in Ar and subgraphs in H, therefore $\mathcal{L}: Ar \times H \rightarrow \{in, out, undec\}$. When labelling a subgraph, we follow this choice: an argument a is automatically labelled out in all the subgraphs where a is not present (since it does not promote any claim) *or* when it is present but it is labelled out by the semantics, representing the effect on a of the other arguments.

In the case of grounded semantics there is only one labelling per subgraph g, that we call $\mathcal{L}(g)$ (we omit Ar). We call $in(\mathcal{L}(g))$, $out(\mathcal{L}(g))$, $undec(\mathcal{L}(g))$ the sets of arguments labelled in, out, $undec$ in the labelling $\mathcal{L}(g)$. In order to study how an argument behaves across subgraphs in H, we define the following sets of subgraphs:

$$A_{IN} = \{g \in H: a \in in(\mathcal{L}(g))\}\ ;\ A_{OUT} = \{g \in H: a \in out(\mathcal{L}(g))\}$$

$$A_U = \{g \in H: a \in undec(\mathcal{L}(g))\}$$

which represent all the subgraphs where argument a is labelled in, out or $undec$.

Example 1. Let's compute A_{IN} for the graph of figure 1 left. There are 3 arguments and 2^3 subgraphs; argument a is labelled in in all the subgraphs where it is present and b is not present (and c becomes irrelevant), i.e. the set of two subgraphs $A_{IN} = A\bar{B}$. It is $undec$ when all the arguments are present, the single subgraph $A_U = ABC$, while it is labelled out when it is not present or when b is present and c is not present, i.e. $A_{OUT} = \bar{A} + AB\bar{C}$ (the set of the remaining five subgraphs).

The following definition is needed in the presentation of our algorithms.

Definition 6 (Exclusively connected arguments). Given an argument a and an argumentation graph G, let's define $C_G(a)$ as the set of arguments connected to a, i.e. the set of all arguments x for which there is at least a path from x to a in G.

Given two arguments a and b, we also define the set of arguments *exclusively connected* to a via b, called $exC_{G,b}(a)$. $exC_{G,b}(a)$ is the set of arguments x for which there is no path from x to a when b is removed from graph G. Therefore, if G' is the subgraph of G obtained by removing b, $exC_{G,b}(a) = \{x \mid x \in C_G(a)\ \wedge x \notin C_{G'}(a)\}$

3.2 The Brute Force Approach

A brute force algorithm to solve our problem simply computes the grounded semantics in all the subgraphs of Ar and it assigns each subgraph to A_{IN}, A_{OUT} or A_U depending on the label of argument a in that subgraph.

```
Algorithm 1 – A brute force approach for computing $A_{IN}$, $A_{OUT}$, $A_U$
  for each subgraph $g$ of $G = (Ar, R)$
      for each argument $a$ in $g$
       assign a label $l(a)$ to $a$ in $g$ using the chosen semantics
       if $l(a) = in$ add $g$ to $A_{IN}$
       if $l(a) = out$ add $g$ to $A_{OUT}$
       if $l(a) = undec$ add $g$ to $A_U$
```

The complexity of the problem studied is above polynomial. There are 2^n possible subgraphs, and the computation of the grounded semantics in each subgraph requires a polynomial time, while other semantics such as the preferred are intractable (see [9]). The algorithms proposed in this paper aim to reduce the computational time by reducing the number of times the grounded semantics has to be computed, by identifying set of equivalent subgraphs in one step instead of individually.

The brute approach is not efficient in the computation of A_{IN} and it is not efficient in the way A_{IN} is expressed, that is a conjunction of single subgraphs. Let's consider the graph in figure 2 left. It can be computed that the expression of A_{IN} includes 56 subgraphs out of the potential 128 (in fact, there are 8 arguments and a total of 256 subgraphs, but we removed the 128 where a is not present).

In [11] we describe an alternative algorithm, which we optimize in the next section. The idea is that we do not need to consider all the subgraphs individually, but a set of subgraphs can be assigned to A_{IN}, A_{OUT} or A_U in a single step. For the graph of figure 2 left, the optimized algorithm of the next section produces the expression in standard form $A_{IN} = A\bar{B}\bar{D} + ABE\bar{D} + AB\bar{E}G\bar{D}$, composed of only three clauses.

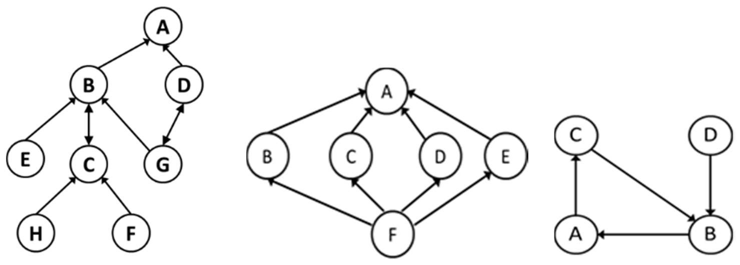


Figure 2. Three Argumentation Graphs

4 Computing A_{IN}: A recursive algorithm

This section presents an algorithm to compute A_{IN}, A_{OUT} under grounded semantics. Given a starting argument a and a label $l \in \{in, out\}$, we need to find the set of subgraphs where argument a is legally labelled l. The idea is to traverse the transpose graph (a graph with reversed arrows) from a down to its attackers, propagating the constraints of the grounded labelling. While traversing the graph, the various paths correspond to a set of subgraphs. The constraints needed are listed in definition 5 and theorem 1. If argument a – attacked by n arguments x_n – is required to be labelled in, we impose the set A_{IN} to be:

$$A_{IN} = A \cap \left(X_{1_{OUT}} \cap X_{2_{OUT}} \cap \ldots \cap X_{n_{OUT}}\right) \qquad \text{condition (1)}$$

i.e. argument a can be labelled *in* in the subgraphs where:

1. a is present - set A and
2. all the attacking arguments x_i are labelled out (sets $X_{i_{OUT}}$).

If a is required to be labelled out, the set of subgraphs is:

$$A_{OUT} = \bar{A} \cup A \cap \left(X_{1_{IN}} \cup X_{2_{IN}} \cup \ldots \cup X_{n_{IN}}\right) \qquad \text{condition (2)}$$

i.e. a is labelled out in all the subgraphs where it is not present or at least one of the attackers is labelled in. Therefore we recursively traverse the graph, finding the subgraphs that are compatible with the starting label of a. The sets $X_{n_{OUT}}$and $X_{n_{IN}}$ are found when terminal nodes are reached. When a terminal node x_T is reached the following conditions are applied:

1. if x_T is required to be in then $X_{T_{IN}} = X_T$
2. if node x_T is required to be out then $X_{T_{OUT}} = \overline{X_T}$

The way the algorithm treats cycles guarantees that only grounded complete labellings are identified. If a cycle is detected, the recursion path terminates, returning an empty set that also has the effect to discard all the sets of subgraphs linked with a logical AND (by condition 1) to the cyclic path. As described in [11], this treatment of cycles guarantees to discard $undec$ arguments not contributing to A_{IN} or A_{OUT} and to identify grounded complete labellings. We present the pseudo-code of the algorithm, while Table 1 describes the steps for computing A_{IN} in the graph of figure 2 right.

Algorithm 2 - The Recursive FindSet(A,L,P) Algorithm

```
A is a node, L a label (IN or OUT), P is the list of parent nodes, Cset
holds the partial result of the computation of conditions (1) and (2).
FindSet(A,L,P):
  if A in P:
     return empty_set // Cycle found
  if L = IN:
     if A terminal:
        return a // Terminal condition for IN Label
     else:
        add A to P
        for each child C of A
           Cset = Cset AND FindSet(C,OUT,P)
        return (a AND Cset)             // condition 1
  if L = OUT:
     if A terminal:
        return NOT(a) // Terminal condition for OUT Label
     else
        add A to P
        for each child C of A
           Cset = Cset OR FindSet(C,IN,P)
        return (NOT(a) OR (a AND Cset))  //condition 2
```

Table 1. Recursively applying Algorithm 2 on the graph of figure 2 right.

	Node, label	Constraint	Parent List	Comment
1↓	A_{IN}	$A_{IN} = A \cap B_{OUT}$	[]	*a must exist and b=OUT*
2↓	B_{OUT}	$B_{OUT} = \bar{B} \cup (B \cap (C_{IN} \cup D_{IN}))$	$[a]$	*b is out when b does not exist or b exists and c = in or d = in*
3=	C_{IN}	$C_{IN} = C \cap A_{OUT}$	$[a, b]$	*c=IN when c exists and a=OUT. Cycle with a,* $C_{IN} = \emptyset$
4=	D_{IN}	$D_{IN} = D$	$[a, b]$	*d is initial*
5↑	B_{OUT}	$B_{OUT} = \bar{B} \cup (B \cap \mathrm{D})$		
6↑	A_{IN}	$\mathrm{A_{IN}} = \mathrm{A} \cap (\overline{\mathrm{B}} \cup (\mathrm{B} \cap \mathrm{D})) = A\overline{\mathrm{B}} + ABD$		

4.1 Optimizations

Generating non-overlapping solutions. The $Recursive$ algorithm generates solutions not in standard form, composed by potentially overlapping clauses. If – as in the probabilistic frameworks – sets of disjoint subgraphs are required, a costly Boolean simplification is needed. This is an inclusion-exclusion problem of combinatorial complexity. It is also inefficient in that the recursive steps need to carry expressions longer than necessary.

A more efficient approach is to modify the algorithm so it produces solutions in a non-overlapping form by simplifying expressions during the computation. Let's analyse the two algorithm conditions:

1. $A_{IN} = A \cap \left(X_{1_{OUT}} \cap X_{2_{OUT}} \cap \dots \cap X_{n_{OUT}}\right)$ *condition (1)*
2. $A_{OUT} = \bar{A} \cup A \cap \left(X_{1_{IN}} \cup X_{2_{IN}} \cup \dots \cup X_{n_{IN}}\right)$ *condition (2)*

Condition 1 clearly generates disjoints sets if $X_{n_{OUT}}$ are expressed as disjoint sets. Regarding condition 2, since an expression such as $A + B + C +$.. can be rewritten as disjoint sets in the form $A + \bar{A}B + \bar{A}\,\bar{B}C +$.., we modify condition 2 as follows:

$$A_{OUT} = \bar{A} \cup A \cap \left(X_{1_{IN}} \cup \overline{X_{1_{IN}}}X_{2_{IN}} \cup \overline{X_{1_{IN}}}\ \overline{X_{2_{IN}}}\,X_{3_{IN}} \cup \dots \cup \left(\overline{X_{1_{IN}}} \dots \overline{X_{n-1_{IN}}}\right)X_{n_{IN}}\right) (2b)$$

In order to generate shorter expressions, the algorithm first computes $\overline{X_{1_{IN}}}$ for all the attackers, then it sorts the expressions of the set $\overline{X_{1_{IN}}}$ in ascending order by number of clauses contained in each expression and then it applies condition $2b$.

Optimizing condition 1: returning empty set. When the in-set of an argument has to be computed, all its attackers x_i must be labelled out (condition 1). Therefore, if a recursion step returns $X_{i_{OUT}} = \emptyset$, the algorithm immediately returns $A_{IN} = \emptyset$.

Exploiting Rebuttals. Argument b is a rebuttal of argument a iff $R(a, b)$ and $R(b, a)$. Rebuttals can be used to terminate a recursion branch earlier. In fact, if a and b are rebuttals, under grounded semantics neither of them can defeat the other (see [14] pag. 8). Therefore it is $A_{OUT} = \overline{A}$ instead of $A_{OUT} = \overline{A} + AB_{IN}$ as condition 2 would suggest in the general case. Therefore in the presence of a rebuttal argument b the set A_{OUT} results independent from B_{IN} (that increments A_U by forming a cycle), and the algorithm can spare itself the recursive computation of AB_{IN}. This implies a new terminal condition: while we are visiting node a, if a has a rebutting attacker b then the general condition $A_{OUT} = \overline{A} + AB_{IN}$ can be replaced by the condition $A_{OUT} = \overline{A}$, that terminates the recursion branch. Note how without this optimization the algorithm would eventually return $AB_{IN} = \emptyset$ in a further (and unnecessary) recursion step when the cycle with a is detected.

Re-using computations. Since an argumentation framework can be composed of an intricate set of links, the same node could be visited from different paths, and therefore the same label for the same argument may be computed more than once during the recursion. The idea is therefore to re-use the computed sets. However, this is not straightforward, since the expressions of X_{IN} (or X_{OUT}) might be different according to which path the recursion took before visiting x.

Let's presume we can reach node x with two computations 1 and 2, and we have already computed $X_{1_{IN}}$. We wonder when we can reuse the result sets $X_{1_{IN}}$ to compute $X_{1_{IN}}$. It is clearly $X_{1_{IN}} = X_{2_{IN}}$ if $C_1(x) = C_2(x)$, and the current version of the algorithm implements this simplification, by keeping a buffer of the previously solved recursion. Note how the condition $C_1(x) = C_2(x)$ is quite restrictive and it does not cover all the cases where previous computations, or part of them, can be reused. We leave further simplification for future research.

Example 2. We apply the recursive optimized algorithm to the graph of figure 2 left. Table 2 shows the computation performed. We comment on some of the differences with the baseline recursive algorithms of section 3. First, condition 1 splits the computation into two recursive steps. In step 1.1, the new condition $2b$ is applied to generate disjoints sets. The condition is further simplified by applying the rebuttals simplification that removes the term $B\overline{E_{IN}}\ \overline{G_{IN}}C_{IN}$ from the expression of B_{OUT}. Since c rebuts b, C_{IN} is irrelevant in the computation of B_{OUT} (note that would be relevant to the computation of B_{IN} or B_U, but these sets are not required by any recursive step).

Table 2. Computing A_{IN} using the optimized recursive algorithm for the graph of fig 2 left

1	$A_{IN} = AB_{OUT}D_{OUT}$	Condtion 1
1.1	$B_{OUT} = \bar{B} + BE_{IN} + B\overline{E_{IN}}G_{IN} + B\overline{E_{IN}}\overline{G_{IN}}C_{IN}$	Condition 2b (with reordering)
	$B_{OUT} = \bar{B} + BE_{IN} + B\overline{E_{IN}}G_{IN}$	2b after rebuttals detection. Since c rebuts b, c cannot label b out.
1.1.1	$E_{IN} = E$	Terminal node
1.1.2	$G_{IN} = G$	Terminal node
1.1	$B_{OUT} = \bar{B} + BE + B\bar{E}G$	Solution of the recursive step 1.1
1.2	$D_{OUT} = \bar{D} + DG_{IN}$	Condition 2b
	$D_{OUT} = \bar{D}$	Rebuttals optimization applied, g cannot defeat c
1	$A_{IN} = A(\bar{B} + BE + B\bar{E}G)\bar{D}$	Final Solution

5 *ADT*: Arguments Decision Tree algorithm

In many cases, the recursive algorithm reduces the computational effort required to compute A_{IN} in comparison with the brute force approach, but it is still prone to combinatorial explosion. For instance, for the graph of figure 2 centre the algorithm produces $A_{IN} = (\bar{B} + BF)(\bar{C} + CF)(\bar{D} + DF)\ (\bar{E} + EF)$, an expression with an exponential number of terms equal to 2^{n-2}, where n is the number of nodes.

In this section we describe a new algorithm modelled as a decision-tree, where at each step a node x is selected and the computation of A_{IN} is split in two disjoint graphs, one containing the node and the other not containing it ($A_{IN} = A'_{IN}X + A''_{IN}\bar{X}$).

Our idea is to select a node that reduces the complexity of the remaining subgraphs. We select the node x that makes the most number of nodes indifferent for the computation of A_{IN}, because these nodes are either (1) defeated by x in the subgraph containing x or (2) disconnected from a in the subgraph where x is not present. As an example, referring again to figure 2 centre, let's select node f for our tree split. In the subgraphs where node f is present, all the other nodes are defeated and a results labelled in. When f does not exist, the only possible subgraph is the one not containing all the attackers of a. Therefore $A_{IN} = F + \bar{F}\bar{B}\bar{C}\bar{D}\bar{E}$, which is a shorter and more manageable standard form expression.

The algorithm we present, called ADT, finds the sets A_{IN}, A_{OUT}, A_U in parallel; it is guaranteed to find disjoint sets and it works better than algorithm 2. First of all, we need to define the metric used to select the argument used for the split. We call this metric *dialectical strength*.

Definition 7. Given $G = (Ar, R)$ and an argument $a \in Ar$, the dialectical strength of an argument $x \in Ar$ w.r.t. a, called $DS_a(x)$, is defined as follows:

If x is initial, $DS_a(x)$ is the number of arguments that are defeated by x plus the arguments that result disconnected from a once the arguments defeated by x are removed from G. Therefore:

$$DS_a(x) = \left| \{x\} \cup A(x) \cup \bigcup_{y \in A(x)} exC_a(y) \right|$$

Where $A(x)$ is the set of all arguments attacked by x, i.e. $\forall x \in Ar, A(x) = \{a \in Ar | R(x, a)\}$. Note that, if x directly attacks a, then $DS_a(x) = |Ar|$. If x is not initial, $DS_a(x)$ is the number of arguments that are disconnected from a after x is removed. Therefore:

$$DS_a(x) = |\{x\} \cup exC_a(x)|$$

The argument with the highest DS_a is selected for the split. In the case of several arguments with the same DS_a, the node for the split is randomly selected.

In figure 2 centre, all the nodes have $DS_a = 1$, except argument f that has $DS_a(f) = 4$ (of course it is always $DS_a(a) = |Ar|$).

Once argument x is selected, the original graph G is split into $G_1 = GX$ and $G_2 = G\bar{X}$. For each subgraph the algorithm keeps a list of the nodes already used for the split and the constraint over each split node (i.e. if in the subgraph the argument is present or not present). At each step the algorithm removes the nodes defeated by argument x in G_1 and the nodes disconnected from a in G_2. Note how a chain effect can happen: by removing arguments, new initial nodes might be created that might defeat other arguments. Note how the number of nodes removed is equal to the dialectical strength DS Therefore, at each split ADT actually computes a set of 2^{DS-1} subgraphs that, as proven at the end of this section, are all equivalent for the labelling of a. Moreover, the computational complexity of ADT will strongly depend on the average value of the dialectical strength.

Regarding terminal conditions, ADT stops when one of the following terminal conditions is met:

1. If argument a is defeated, the branch of the tree will contribute to A_{OUT}
2. If argument a is isolated, the branch of the tree will contribute to A_{IN}, since a has no attackers.
3. If there are no more arguments for the split and neither of the above two are verified, the branch contributes to A_U since a cycle is detected.

Figure 3 proposes an illustrative example of the ADT algorithm applied to the graph of figure 2 right, followed by the pseudo-code of the algorithm.

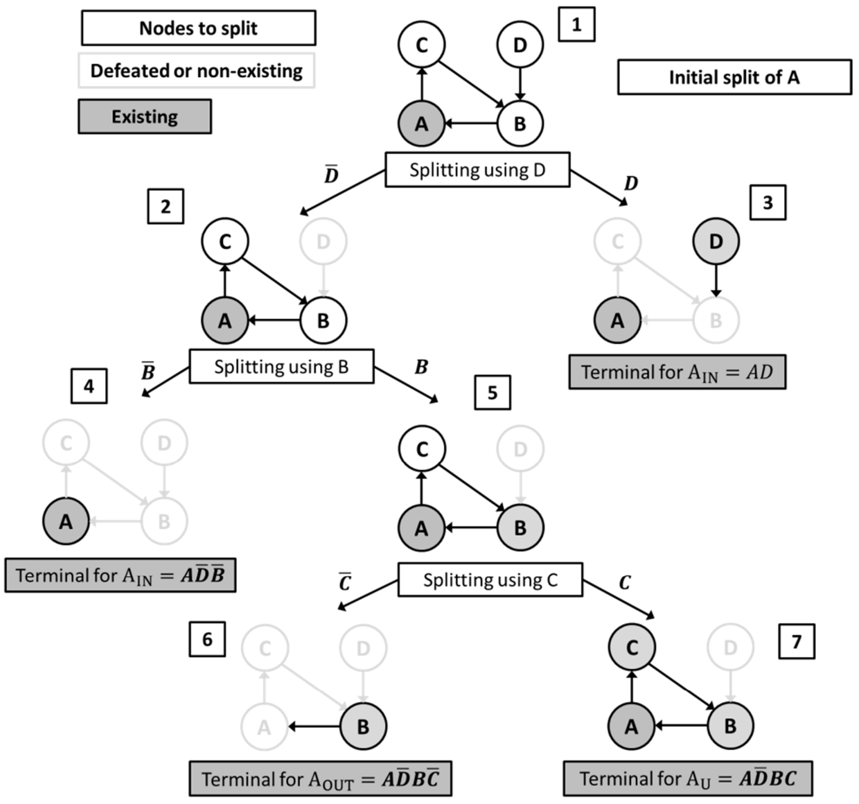


Figure 3. Visual Representation of the ADT Algorithm

At the beginning (not shown), the set $\bar{A}$ is trivially assigned to A_{OUT}, and we start from the situation where a is present (set of subgraphs A), depicted in subgraph 1 of figure 3. First, the DS_a of each argument is computed. Arguments d and b have both $DS_a = 3$ while c has $DS_a(c) = 1$. Therefore d is chosen.

In the subgraph (3), obtained by set d to present, b is defeated, a becomes initial and defeats c. Therefore a is isolated, the terminal condition for A_{IN} is reached and the path AD is added to A_{IN}. In the subgraph with d non-existent (2), no other node is disconnected. Since no terminal condition is reached, a new split is needed. Now b is selected. In the subgraph with b not present (4), argument a becomes isolated, and therefore the path $A\bar{D}\bar{B}$ is added to A_{IN}, while in the graph with b present (5) no arguments are disconnected. Only c is left for the split.

When c is present (subgraph 7), the terminal condition 3 is reached so $A\bar{D}BC$ contributes to A_U. Subgraph 6, with c not present in the subgraph, contributes to A_{OUT} (set of subgraphs $A\bar{D}B\bar{C}$) since b becomes initial and defeats a.

```
Algorithm 3 – ADT (Arguments Decision Tree Algorithm).
Inputs: Graph $G$, argument $a$ Output: ($A_{IN}$,$A_{OUT}$,$A_U$)
Initialize C to ∅. //C is the list of constraints on the split arguments
ADT($G$,$a$,$C$)
  If C is ∅ then C = $A$
  remove from G all the nodes disconnected from node $a$
  compute $I_G$, the list of initial nodes of G
  while ($\exists x$ in $I_G$ with $X$ is in $C$ )
      for each $x$ in $I_G$ with $X$ in $C$
          remove form G all the arguments attacked by $x$
          update the initial list $I_G$
      remove form G all the arguments not connected to $a$
      If $\nexists b$ so that R(b,a) then add $C$ to $A_{IN}$ and return
      If $a \notin G$ then add $C$ to $A_{OUT}$ and return
      If no more nodes to split then add $C$ to $A_U$ and return
  for each $x$ in $G$ and not in $C$ Compute the $DS_a(x)$
  select node $x$ with highest $DS_a(x)$
  split the subgraph: $G_1 = G \cup X$ and $G_2 = G \cup \bar{X}$
  call ADT($G_1$,$a$,$C \cup X$)
  call ADT($G_2$, $a$, $C \cup \bar{X}$)
```

Optimization. We optimized the ADT algorithm by keeping a buffer of the subgraphs that have already been computed. When, after a split, one of the remaining subgraph has been already encountered in the computation, its solution can be reused and joint with the constraints of the current branch. This operation is theoretically simpler than in the case of the $Recursive$ algorithm. For instance, considering the graph of figure 2 left, after we split using node g, the subgraph where g is present is reduced to the nodes $\{a, c\}$, but the same subgraph is obtained in the branch where g is not present by further splitting, using node e and selecting the branch where node e is present. The first branch has constraints G (g is present in all the subgraphs) while the second has constraints $\bar{G}E$ (g is not present and e is present). A solution S for the subgraph $\{a, c\}$ is computed only the first time the subgraph is encountered (branch G in our example), generating the clause GS that is added to the ADT output. When the same subgraph is encountered in the branch $\bar{G}E$, the solution S is reused and joint with the constraints of the branch, obtaining the new solution $G\bar{E}S$ that is also added to the ADT output. For instance, referring to the computation of A_{IN}, the solution for the subgraph $\{a, c\}$ is $A\bar{C}$, and this set is used to add the two clauses $GA\bar{C} + \bar{G}EA\bar{C}$ to the output of ADT for the set A_{IN}.

$\boldsymbol{ADT_{fast}}$**.** We implemented a version of the above ADT algorithm, called ADT_{fast}, where at each step the node used for the split is chosen randomly. The algorithm will be used to compare the impact of using the dialectical strength in the computation.

Soundness and Completeness. We end this section by proving the *soundness* and *completeness* of the ADT algorithm. Each of the clauses φ_j composing the output of the ADT algorithm identifies a set of subgraphs. We prove that all the subgraphs identified by a clause assign the same label to argument a and this label is correctly assigned under grounded semantics. The set of subgraphs associated with a clause φ_j have in common a subset of the arguments in Ar, the arguments present in the expression of φ_j. For instance, if $Ar = \{a, b, c, d, e\}$, the clause $AB\bar{C}$ identifies all the subgraphs having in common the presence of nodes a, b and the absence of node c. Nodes d and e are not specified, therefore their presence or absence is irrelevant and they identify a set of 4 different subgraphs associated with φ_j. We prove that these *irrelevant* arguments are actually irrelevant to the computation of the label of a and therefore all the subgraphs in φ_j assign the same label to a. ADT uses two conditions to identify irrelevant arguments. First, when the argument used for the split is removed, all the arguments resulting disconnected from a are irrelevant to the labelling of a. Second, in the subgraphs where an initial argument i is constrained to be present, all the arguments attacked by i are labelled out, and therefore they become irrelevant (as proven by [8], removing an out argument does not change the grounded extension). Therefore all the arguments marked as irrelevant do not alter the label of a and therefore we prove that all the subgraphs in φ_j assign the same label to a.

ADT also assigns the correct label under grounded semantics, since its second condition and the three terminal conditions described above actually implement the basic step of the algorithm for grounded labelling described by Modgil and Caminada in [14, page 8] and therefore ADT generates correct grounded labellings.

In order to prove ADT *completeness*, we observe that the ADT algorithm considers the entire problem space, since all the arguments that are not found irrelevant to the labelling of a are split. Therefore in all the 2^n subgraphs of G argument a is labelled by the ADT algorithm.

6 Evaluation

We implemented our algorithms in Python 2.7, and we performed a set of initial experiments on a *Windows* 7 machine with 3Gb RAM and *Core I3 Intel* processor. We implemented the following algorithms:

1. $\boldsymbol{Brute}$ – the brute force approach.
2. $\boldsymbol{ADT}$ – the decision-tree based algorithm using the dialectical strength as splitting criterion.
3. $\boldsymbol{ADT_{fast}}$ – the ADT algorithm where splitting nodes are selected randomly.
4. ***Rec*** $(Recursive)$ – the optimized recursive algorithm. All the optimization of section 4 were implemented.

Our first evaluation tests two aspects of the computation of A_{IN}: computational time and length of the output expression. The evaluation described in this paper does not claim to be exhaustive. It focuses on the generic case of random graphs; it does not study particular class of graphs nor does it test hybrid approaches.

Random Graphs Generation. We generate different acyclic and cyclic graphs of increasing complexity both in terms of number of nodes and density. Graph instances have been generated as follows. Given n arguments, we assign an incremental index i to each argument and we generate a tree with node a as root, to guarantee that for each argument there is at least a path to a. Then, in the case of acyclic graphs, random links are added until the required density is reached. In order to generate only acyclic graphs, the links are added only if they go from a node with a higher index to a node with a lower index. In the case of cyclic graph, links are added randomly with no restrictions. However, we require each random graph to at least contain a cycle. Note that the density for an acyclic graph is computed over $n(n-1)$ (instead of $\frac{n(n-1)}{2}$ used for the acyclic case) to take into consideration the presence of symmetric attacks.

6.1 Experimenting with the length of A_{IN}

This set of experiments tests the ability of each algorithm to express a standard-form solution for A_{IN} in the most compact way. We use as a metric the length l of the expression of A_{IN}, defined as the number of clauses contained in its standard-form expression. Results reported are the average of a set of 1000 executions of each algorithm using graphs differentiated by number of nodes, density and type (cyclic or acyclic).

In the brute force approach, the length of the solution equates to the number of subgraphs in A_{IN}. Table 3 shows results for the brute force approach. No data for graphs with more than 15 nodes are available due to the long computational time needed by this algorithm (a single 15-node with a 0.3 density takes about 12 minutes).

Table 3. Length of A_{IN}, brute force approach

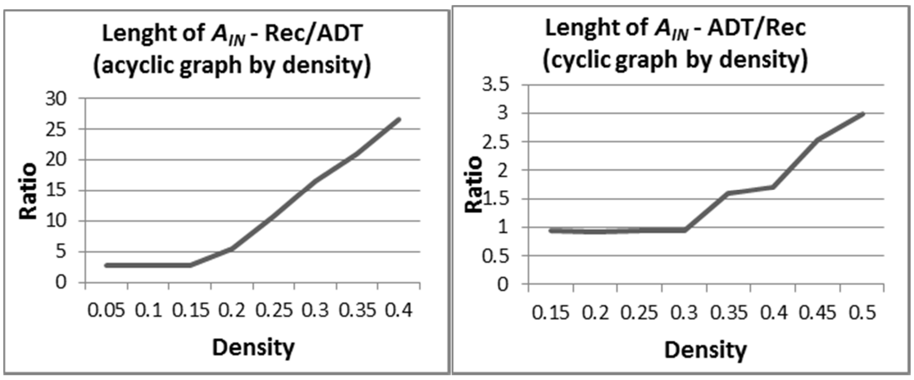
Nodes	6	7	8	9	10	11	12	13	14	15
Length of A_{IN}	12	23	44	85	158	335	618	1421	2219	4853

Graphs 1-4 show the behaviour of the other algorithms. We divide the analysis into cyclic and acyclic graphs. Overall, the ADT algorithm shows the best performance, even if its performance is not consistent with the type of graph (cyclic or acyclic). Graph 3 shows how the ADT algorithm is extremely efficient for acyclic graphs, and the gap with the other algorithm increases rapidly. For a 20-node graph, ADT output is on average 42.1 clauses against the 659.4 of the $Recursive$ algorithm.

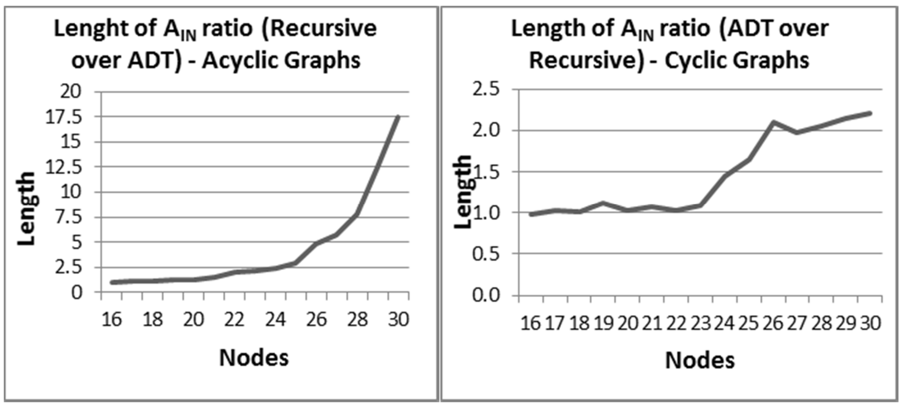
Again, Graphs 1 and 2 (left) show the ratio (by density and by number of nodes) between the length of the solution expressed by the ADT algorithm and the second best algorithm, the $Recursive$ algorithm, for acyclic graphs.

Graph 1 left shows how the ratio by density increases almost linearly, showing how the ADT algorithm becomes more efficient with high density acyclic graphs. This could be explained by the fact that, when the number of links increases, each node is likely to attack a larger set of nodes, and therefore nodes' dialectical strength DS increases and the split subgraphs that result are smaller and easier to compute.
The introduction of the dialectical strength is also proved to be efficient, since the ADT_{fast} algorithm (i.e. that in which nodes for the split are randomly selected) produces much longer expressions, already 22 times longer for a 20-node graph.

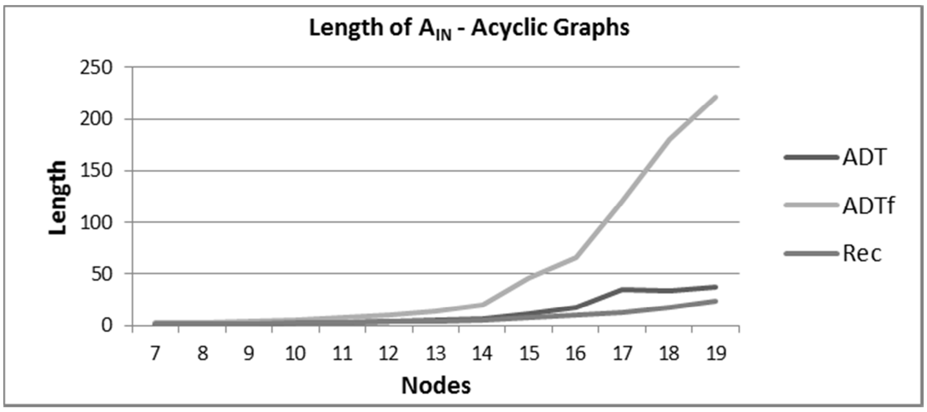
However, the situation is different for cyclic graphs. The $Recursive$ algorithm shows similar or better performance than ADT, as shown in Graph 4 and Graphs 1 and 2 right. Graphs 1 and 2 right now show an inverse ratio ($Recursive$ algorithm over ADT). The presence of cycles and rebuttals increase the likelihood that some recursive branches quickly generate an empty return set, and consequently the length of the solution decreases. Moreover, when the number of cycles increases, the dialectical strength is no longer effective, since the number of initial arguments diminishes and the number of arguments disconnected from the root node a after the generic node x is removed – i.e. $|exC(x)|$ – diminishes as well or it could likely be empty.



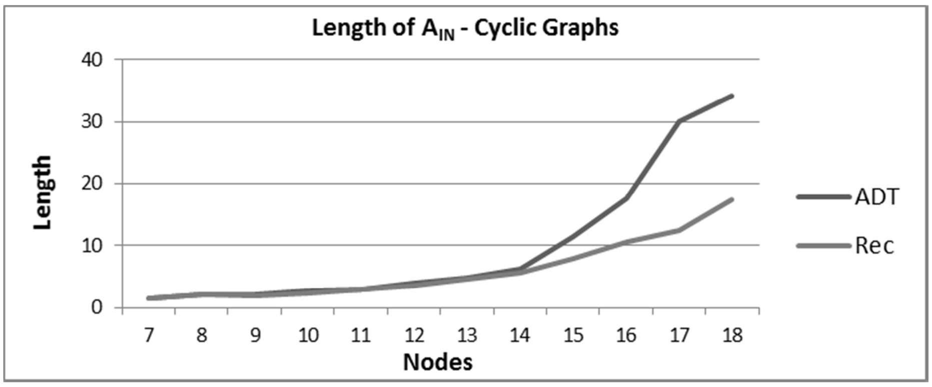
Graph1. Length of the solution by density



Graph 2. Length of the solutions by nodes



Graph 3. Length of the solutions – Acyclic Graphs

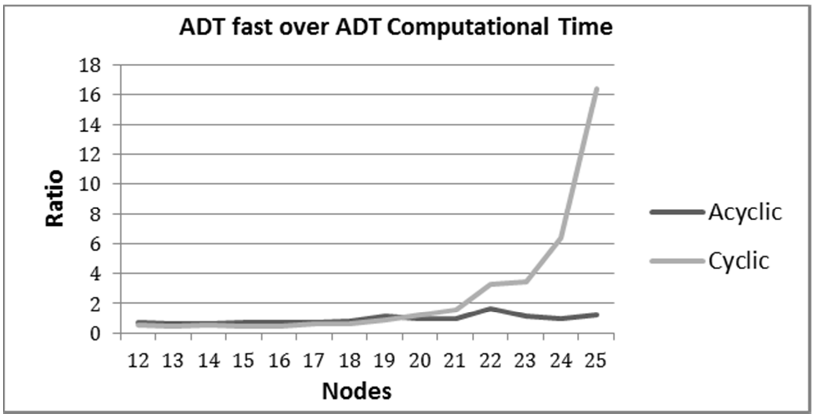


Graph 4. Length of the solutions – Cyclic Graphs

6.2 Computational Time

This second set of experiments tests the efficiency of the above algorithms in terms of computational time. Again, the brute force approach is by far the slowest. In a 14-node graph with 0.3 density, the average computing time is about 45 times longer than the $Recursive$ algorithm, while it increases to 650 times for a 15-node graph.

The ADT_{fast} algorithm is also considerably slower than the others. For a 25-node acyclic graph it is on average 15 times slower than the ADT, while it is more than 200 times slower for a cyclic graph compared to the $Recursive$ algorithm.

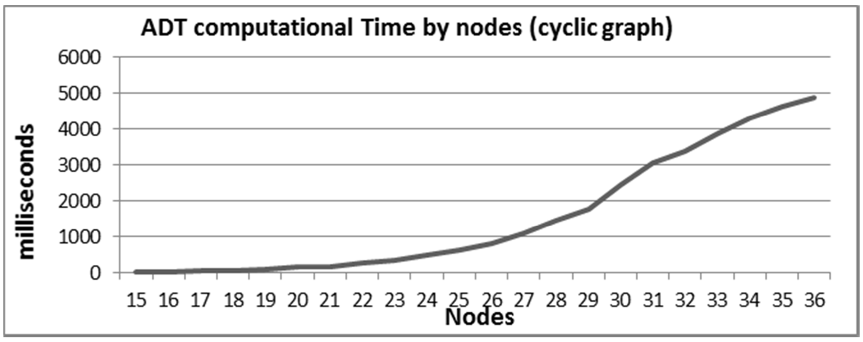


Graph 5. ADT versus ADT fast computational Time

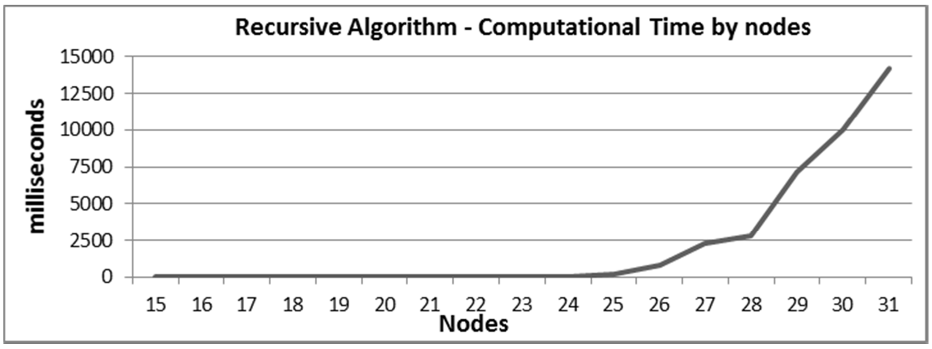
It is interesting to compare the performance of ADT versus ADT_{fast} in order to understand the impact of the dialectical strength as splitting criterion. Following a similar pattern encountered in the length-based experiment, the gap between ADT and ADT_{fast} is highly significant for both the acyclic graph and the cyclic graph with low density. ADT is already 10 times faster with a 23-node acyclic graph, while for a cyclic graph the computational time is comparable and it does not show a clear trend. The reason for this is mainly because in an acyclic (or quasi-acyclic) graph, the dialectical strength DS of the arguments is high and this effectively reduces the complexity of the split subgraphs. In a cyclic graph, the set exC is small or empty and few nodes are removed during a split. Therefore the choice of a splitting node is less important and the overhead of computing the dialectical strength is not justified.

ADT *vs* $Recursive$. For acyclic graphs, thanks to the high dialectical strength of the arguments, the ADT algorithm is faster. ADT is already 100 times faster for a 20-node graph. On our machine setting, the average computational time needed to compute an acyclic graph goes above 60 seconds between 50-55 nodes. Graph 6 shows the computational time in terms of number of nodes. The computational time grows with a quite constant slope after about 25 nodes.

For cyclic graphs, the $Recursive$ algorithm takes advantage of the presence of rebuttals and cycles, which reduce some of the recursive steps. The $Recursive$ algorithm is already 25 times faster for a 15-node and 60 times faster for a 25-node graph. The ADT algorithm remains better up to a density of 0.1.



Graph 6. ADT Computational Time by number of nodes



Graph 7. Recursive Algorithm computational time

The $Recursive$ algorithm goes above the 60-second threshold at 38 arguments. Graph 7 shows the computational time of the $Recursive$ algorithm by number of nodes. We notice how the algorithm has a rapid increase after 25 nodes, much faster than the ADT increase for acyclic graphs. An explanation could be that, since the $Recursive$ algorithm is based on paths visited on the graph, it is sensitive to the number of links rather than to the number of nodes, and the number of links grows like n^2 rather than n. However, the experimental analysis calls for a theoretical complexity analysis that is at the top of our research agenda.

Overall, our results suggest defining a hybrid approach exploiting both the ADT (good for acyclic or quasi-acyclic graphs) and the $Recursive$ algorithms (good for cyclic graphs), depending on the characteristics of the graph. Another observation is about the computation of the dialectical strength, which could be optimized and made more effective in the presence of cyclic graphs (for instance by considering the effect of removing a couple of nodes instead of a single node).

7 Related Works

The research presented in this paper is inspired by the recently introduced Probabilistic Argumentation Framework. The original paper by Li [3] introduces the formalism but it does not present any computational algorithm beyond the brute force approach. The author proposes an approximate method using a *Montecarlo* simulation for grounded semantic. Other papers in the field (Hunter [6], Trimm [7], Dung [2]) do not investigate computational aspects. This paper continues our research in [11], where we presented the baseline non-optimized recursive algorithm.

To the author's best knowledge, there is no other study that directly approaches the problem of subgraph-based computation in the context of probabilistic argumentation. Even for abstract argumentation in general, experimental evaluations of algorithms represents a small corpora. The work by Nofal at al. [13] represents one such work. As the author notes, "*although experimental analysis of algorithms is a well-established in other domains, such methodology is given a little attention in the context of AFs*" [13]. We mention also the experimental thesis by Charwat [10] based on tree-decomposition of AFs. Therefore, our paper contributes to the experimental analysis of abstract argumentation algorithms.

However, the algorithms proposed in this paper decompose the computation of the grounded semantic, and they can be described as a study on how an argument label behaves when arguments are added (or removed) from an argumentation graph. In particular we refer to the work by Boella [8], that studied how the grounded extension changes with the addition of a new argument. Indeed our algorithm – especially the ADT algorithm – relies on similar mechanisms and theoretical foundations. The work in [8] is extended by Cayrol [12] to the case of preferred semantics and the removing of arguments or attacking links.

In abstract argumentation there are works that employ similar techniques to ours. The work by Baumann [9] et al. provides an experimental evaluation of computing extensions semantics by splitting the argumentation graph into subparts that are then combined to obtain a final solution. Their systematic empirical evaluation shows that the performance of algorithms may drastically improve when splitting is applied.

8 Conclusions and Future Works

In this paper we initiated an investigation of how the label assignment of an argument varies in all the subgraphs of an argumentation framework. We presented a recursive algorithm and a tree-based computation. We started to evaluate the algorithms experimentally, showing how they drastically improve performance compared to a brute-force approach. We claim to have provided enough evidence to justify further investigations. In particular, the ADT algorithm is proven to be efficient in expressing solutions using the minimal number of clauses, and effective in computing acyclic and quasi-acyclic graphs. The $Recursive$ algorithm shows the best computational efficiency for cyclic graphs, and on average it can compute cyclic graphs of up to 35/40 nodes. However, this last result might not fit all the applications, and the number of nodes could be small in some contexts. Interesting future research trajectories include the theoretical complexity analysis of the algorithms, which has not been addressed in this work. Regarding extensions to other semantics, we have already described an extension to preferred semantics for the recursive algorithms, while defining the preferred version of the ADT should not present difficulties. Moreover, we intend to focus on the definition of a hybrid approach that uses the ADT and the $Recursive$ algorithms together. Specific classes of graphs have also to be studied. It appears reasonable to the author that natural argumentation graphs could show specific patterns in terms of density and type of cycles – mostly rebuttal cycles – that could differ from randomly-generated graphs. Finally, attention might also be devoted to the application of the above algorithms to probabilistic argumentation frameworks.

References

1. P. Dung, “On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games,” Artificial Intelligence, vol. 77, pp. 321–357, 1995
2. P. Dung, P. Thang. Towards (Probabilistic) Argumentation for Jury-based Dispute Resolution. COMMA 2010. IOS Press, Amsterdam, 171-182
3. Hengfei Li, Nir Oren, Timothy J. Norman. Probabilistic Argumentation Frameworks. 1st TAFA, JICAI 2011, Barcellona, Spain
4. P. Baroni, M. Caminada, M. Giacomin: An introduction to argumentation semantics. Knowledge Eng. Review 26(4): 365-410 (2011)
5. Dunne, Paul E., and Michael Wooldridge. "Complexity of abstract argumentation." Argumentation in Artificial Intelligence. Springer US, 2009. 85-104.
6. A. Hunter. A probabilistic approach to modeling uncertain logical arguments, International Journal of Approximate Reasoning, 54(1):47-81, 2013.
7. Thimm M. Probabilistic Semantics for Abstract Argumentation, Proceedings. of 20th European Conference of Artificial Intelligence, IOS Press, 2012, pp. 750-755
8. Boella, Guido, Souhila Kaci, and Leendert van der Torre. "Dynamics in argumentation with single extensions: Abstraction principles and the grounded extension." Symbolic and Quantitative Approaches to Reasoning with Uncertainty. Springer, 2009. 107-118.
9. Baumann, Ringo. "Splitting an argumentation framework." Logic Programming and Nonmonotonic Reasoning. Springer Berlin Heidelberg, 2011. 40-53.
10. Charwat, Günther. "Tree-Decomposition based Algorithms for Abstract Argumentation Frameworks.", Thesis, Vienna University of Technology, February 2012
11. Dondio, P , Probabilistic Argumentation Frameworks: Basic Properties and Computation, Highlights on Practical Applications of Multi-Agent Systems, 263-279, 2013, Springer
12. Cayrol, C, F. Dupin, M. Lagasquie-Schiex. "Change in abstract argumentation frameworks: adding an argument." Journal of Artificial Intellgence Research 38.1 (2010): 49-84.
13. Samer Nofal, Paul E. Dunne, Katie Atkinson: Towards Experimental Algorithms for Abstract Argumentation. COMMA 2012: 217-228
14. Modgil, Sanjay, and M. Caminada. "Proof theories and algorithms for abstract argumentation frameworks." Argumentation in artificial intelligence. Springer US, 2009. 105-129.