Journal of Artificial Intelligence Research 49 (2014) 635-668 Submitted 01/14; published 04/14

Algorithms for Argumentation Semantics: Labeling Attacks as a Generalization of Labeling Arguments

Samer Nofal SAMER.NOFAL@GJU.EDU.JO
Dept. of Computer Science, German-Jordanian University
P.O. Box 35247, Amman 11180, Jordan

Katie Atkinson K.M.ATKINSON@LIVERPOOL.AC.UK
Paul E. Dunne P.E.DUNNE@LIVERPOOL.AC.UK
Dept. of Computer Science, University of Liverpool
Ashton Street, Liverpool L69 3BX, United Kingdom

Abstract

A Dung argumentation framework (AF) is a pair (A,R): A is a set of abstract arguments and $R \subseteq A \times A$ is a binary relation, so-called the *attack* relation, for capturing the conflicting arguments. "Labeling" based algorithms for enumerating *extensions* (i.e. sets of acceptable arguments) have been set out such that arguments (i.e. elements of A) are the only subject for labeling. In this paper we present implemented algorithms for listing extensions by labeling attacks (i.e. elements of R) along with arguments. Specifically, these algorithms are concerned with enumerating all extensions of an AF under a number of argumentation semantics: *preferred*, *stable*, *complete*, *semi stable*, *stage*, *ideal* and *grounded*. Our algorithms have impact, in particular, on enumerating extensions of AF-extended models that allow attacks on attacks. To demonstrate this impact, we instantiate our algorithms for an example of such models: namely argumentation frameworks with *recursive attacks* (AFRA), thereby we end up with unified algorithms that enumerate extensions of any AF/AFRA.

1. Introduction

Computational argumentation, covering its theory and applications, has attracted major attention in the AI research community, notably in the last twenty years (e.g. Bench-Capon & Dunne, 2007; Besnard & Hunter, 2008; Rahwan & Simari, 2009; Modgil, Toni, Bex, Bratko, Chesñevar, Dvořák, Falappa, Fan, Gaggl, García, González, Gordon, Leite, Možina, Reed, Simari, Szeider, Torroni, & Woltran, 2013). Dung's abstract argumentation frameworks (AFs) (Dung, 1995) are a widely studied model in which an AF is described by a pair (A,R): A is a set of abstract arguments and $R \subseteq A \times A$ is a binary relation, so-called the *attack* relation, to represent the conflicting arguments. A central notion in AFs is an *argumentation semantics*: a set of criteria that characterise the acceptable arguments; we define these criteria rigorously in section 2. For different reasons a number of argumentation semantics have been proposed in the literature. Explaining these reasons in detail is out of the scope of this paper; however, see the work of Baroni, Caminada, and Giacomin (2011a) for an excellent introduction to argumentation semantics.

Under various argumentation semantics, one might find multiple distinct *extensions* (defined in section 2). Labeling based algorithms (e.g. Dimopoulos, Magirou, & Papadimitriou, 1997; Doutre & Mengin, 2001; Modgil & Caminada, 2009) for listing all extensions have been developed such that arguments (i.e. elements of A) are the only target to be labeled. In this paper we illustrate how to enumerate extensions under several argumentation semantics by labeling *attacks* (i.e. elements of R)

along with arguments, instead of labeling arguments solely. This is particularly of interest in listing extensions of AF-extended formalisms that allow attacks on attacks (e.g. Modgil, 2009b; Gabbay, 2009; Baroni, Cerutti, Giacomin, & Guida, 2011b). As we show throughout the paper, the term "labeling based algorithms" for argumentation semantics is distinguished from the common term "labeling based semantics", although both concepts involve a labeling mapping. The former term (i.e. "labeling based algorithms") refers to the course of actions by which an extension enumeration process classifies arguments: those which might be in an extension from those which are excluded from the respective extension. This classification is essential in order to construct all concrete extensions of a given AF. The later term (i.e. "labeling based semantics") refers to an approach to describing (i.e. *not constructing*) extensions using a labeling mapping.

In section 2 we provide necessary background materials. In section 3 we review explicit algorithms for a selection of dominant argumentation semantics: *preferred*, *stable*, *complete*, *semi stable*, *stage*, *ideal* and *grounded*. These algorithms list extensions by labeling arguments only. Then in section 4 we develop, under the respective argumentation semantics, definite algorithms for enumerating extensions of an argumentation framework with *recursive attacks* (AFRA): an AF-extended model that allows attacks on attacks (Baroni et al., 2011b). These algorithms construct extensions by labeling attacks together with arguments. Since an AF is a special case of AFRA (Baroni et al., 2011b), the developed algorithms for AFRA also list extensions of an AF. In section 5 we report on experiments concerning the practical efficiency of the algorithms. Section 6 concludes the paper with a summary and a review of related work.

2. Preliminaries

We start with the definition of Dung's argumentation frameworks (Dung, 1995).

Definition 1. *(Dung's Argumentation Frameworks)*
An argumentation framework *(or* AF*) is a pair* (A,R) *where* A *is a set of arguments and* $R \subseteq A \times A$ *is a binary relation.*

We refer to $(x,y) \in R$ as x attacks y (or y is attacked by x). We denote by $\{x\}^-$ respectively $\{x\}^+$ the subset of A containing those arguments that attack (resp. are attacked by) the argument x, extending this notation in the natural way to *sets* of arguments, so that for $S \subseteq A$,

$$\begin{array}{rcl} S^- & = & \{\, y \in A \;:\; \exists\, x \in S \text{ s.t. } y \in \{x\}^- \} \\ S^+ & = & \{\, y \in A \;:\; \exists\, x \in S \text{ s.t. } y \in \{x\}^+ \} \end{array}$$

Given a subset $S \subseteq A$, then

- $x \in A$ is *acceptable* w.r.t. S if and only if for every $(y,x) \in R$, there is some $z \in S$ for which $(z,y) \in R$.
- S is *conflict free* if and only if for each $(x,y) \in S \times S$, $(x,y) \notin R$.
- S is *admissible* if and only if it is conflict free and every $x \in S$ is acceptable w.r.t. S.
- S is a *preferred extension* if and only if it is a maximal (w.r.t. $\subseteq$) admissible set.
- S is a *stable extension* if and only if it is conflict free and $S^+ = A \setminus S$.

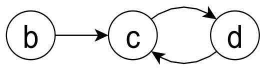

Figure 1: An argumentation framework.

- S is a *complete extension* if and only if it is an admissible set such that for each x acceptable w.r.t. S, $x \in S$.
- S is a *stage extension* if and only if it is conflict free and $S \cup S^+$ is maximal (w.r.t. $\subseteq$).
- S is a *semi stable extension* if and only if it is admissible and $S \cup S^+$ is maximal (w.r.t. $\subseteq$).
- S is the *ideal extension* if and only if it is the maximal (w.r.t. $\subseteq$) admissible set that is contained in every preferred extension.
- S is the *grounded extension* if and only if it is the least fixed point of $F(T) = \{x \in A \mid x\ is\ acceptable\ w.r.t.\ T\}$.

Preferred, complete, stable and grounded semantics are introduced in the work of Dung (1995), whereas stage semantics, ideal semantics and semi stable semantics are presented in the papers of Verheij (1996), Dung, Mancarella, and Toni (2007) and Caminada, Carnielli, and Dunne (2012) respectively. To give an example, consider the framework depicted in figure 1 where nodes represent arguments and edges correspond to attacks (i.e. elements of R). For this example $\{b,d\}$ is the preferred, grounded, stable, ideal, complete, semi stable and stage extension. Note that we do not intend by this example to show differences between semantics.

Offering an explicit means to weaken attacks, the formalisms of Modgil (2009b), Gabbay (2009) and Baroni et al. (2011b) extend AFs such that attacks (i.e. elements of R) are subject to attacks themselves. We present extension enumeration algorithms for an instance of such formalisms: namely argumentation frameworks with *recursive attacks* (AFRA) introduced by Baroni et al. (2011b).

Definition 2. *An argumentation framework with recursive attacks (*AFRA*) is a pair $(A,\overline{R})$ where A is a set of arguments and $\overline{R}$ is a set of pairs (x,y) such that $x \in A$ and $(y \in A$ or $y \in \overline{R})$.*

Let $x = (y,z) \in \overline{R}$ then we say that y is the source of x, denoted as $src(x) = y$, and z is the target of x, denoted as $trg(x) = z$.

Let $x \in A \cup \overline{R}$ and $y \in \overline{R}$ then we say that y *directly defeats* x if and only if $x = trg(y)$.

Let $x,\ y \in \overline{R}$ then we say y *indirectly defeats* x if and only if $src(x) = trg(y)$.

Let $x \in A \cup \overline{R}$ and $y \in \overline{R}$, we say y *defeats* x if and only if y directly or indirectly defeats x.

Given a subset $S \subseteq A \cup \overline{R}$, then

- S is conflict free if and only if there does not exist $x,y \in S$ s.t. x defeats y.
- An element $x \in A \cup \overline{R}$ is acceptable w.r.t. S if and only if for each $y \in \overline{R} : y$ defeats x, there is some $z \in S$ such that z defeats y.
- S is admissible if and only if S is conflict free and for each $x \in S,\ x$ is acceptable w.r.t. S.

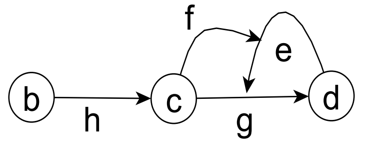

Figure 2: An argumentation framework with recursive attacks.

- S is a preferred extension if it is a maximal (w.r.t. $\subseteq$) admissible set.
- S is a stable extension if and only if it is conflict free and for each $x \in A \cup \overline{R} : x \notin S$, there exists $y \in S$ such that $y\ defeats\ x$.
- S is a complete extension if and only if it is admissible and every element of $A \cup \overline{R}$, which is acceptable w.r.t. S, belongs to S.
- S is a stage (resp. semi stable) extension if and only if S is conflict free (resp. admissible) and $S \cup \{x \mid \exists y \in S$ s.t. y defeats $x\}$ is maximal (w.r.t. $\subseteq$).
- S is the *ideal extension* if and only if it is the maximal (w.r.t. $\subseteq$) admissible set that is contained in every preferred extension.
- S is the *grounded extension* if and only if it is the least fixed point of $F(T) = \{x \in A \cup \overline{R} \mid x\ is\ acceptable\ w.r.t.\ T\}$.

Referring to figure 2, $\{b,d,h,e\}$ is the grounded, stable, preferred, ideal, complete, stage and semi stable extension.

We consider now the issue of expressing an AFRA as an AF. Let $H = (A, \overline{R})$ be an AFRA, then the corresponding AF $H' = (A', R')$ is defined such that $A' = A \cup \overline{R}$ and $R' = \{(x,y) \mid x,y \in A \cup \overline{R}\ and\ x\ defeats\ y\}$. For example, the corresponding AF of the AFRA depicted in figure 2 is described by $A' = \{b,c,d,e,f,g,h\}$ and $R' = \{(e,g),(f,e),(g,e),(g,d),(h,c),(h,f),(h,g)\}$.

3. Algorithms for a Selection of Argumentation Semantics

In this section we review explicit algorithms that list, under a number of argumentation semantics, all extensions of an AF by labeling arguments solely. Particularly, in subsection 3.1 we recall the algorithm of Nofal, Atkinson and Dunne (2014) for preferred semantics. In section 3.2 we present a new implementation of the algorithm of Dimopoulos et al. (1997) for stable semantics. Then we modify the algorithm of Nofal, Atkinson and Dunne (2014) to produce specific algorithms for complete, stage, semi stable and ideal semantics in subsections 3.3, 3.4, 3.5 and 3.6 respectively. In subsection 3.7 we present an implementation for building the grounded extension.

3.1 Enumerating Preferred Extensions of any AF

Algorithm 1 lists all preferred extensions of an AF. Algorithm 1 is taken from the work of Nofal, Atkinson and Dunne (2014) where it has been shown that the algorithm is likely to be more efficient than the algorithms of Doutre and Mengin (2001), and Modgil and Caminada (2009). We recall algorithm 1 because other implemented algorithms of the present paper can be seen as an extension of this algorithm. The algorithm is a backtracking procedure that traverses an abstract binary search tree. A core notion of the algorithm is related to the use of five labels: *IN*, *OUT*, *MUST_OUT*, *BLANK* and *UNDEC*. Informally, the IN label identifies arguments that might be in a preferred extension. The OUT label identifies an argument that is attacked by an IN argument. The BLANK label is for any unprocessed argument whose final label is not decided yet. The MUST_OUT label identifies arguments that attack IN arguments. The UNDEC label designates arguments which might not be included in a preferred extension because they might not be defended by any IN argument. To enumerate all preferred extensions algorithm 1 starts with BLANK as the default label for all arguments. This initial state represents the root node of the search tree. Then the algorithm forks to a left (resp. right) child (i.e. state) by picking an argument, that is BLANK, to be labeled IN (resp. UNDEC). Every time an argument, say x, is labeled IN some of the neighbour arguments' labels might change such that for every $y \in \{x\}^+$ the label of y becomes OUT and for every $z \in \{x\}^- \setminus \{x\}^+$ the label of z becomes MUST_OUT. This process, i.e. forking to new children, continues until for every $x \in A$ the label of x is not BLANK. At this point, the algorithm captures a preferred extension if and only if for every $x \in A$ the label of x belongs to {IN,OUT,UNDEC} and $\{x \mid$ the label of x is IN$\}$ is not a subset of a previously found preferred extension (if such exists). Then the algorithm backtracks to try to find all preferred extensions. It is important in these kinds of algorithms to exploit properties whereby we might bypass expanding a child of the search tree, thus considerable time might be saved. Algorithm 1 uses two pruning properties:

1. Algorithm 1 (lines 12-17) skips labeling an argument y IN (i.e. skips expanding a left child) if and only if there is $z \in \{y\}^-$ such that the label of z is not OUT while there is no $w \in \{z\}^-$ with the BLANK label. In other words, such z can not be labeled OUT later while for each $w \in \{z\}^-$ the label of w is OUT, MUST_OUT or UNDEC. Thus, it is more efficient to skip trying to include any argument that is attacked by such z in a preferred extension.

2. Algorithm 1 (lines 19-22) skips labeling an argument y UNDEC (i.e. skips expanding a right child) if and only if for every $z \in \{y\}^-$ the current label of z is OUT or MUST_OUT. This is because if an admissible set, say S, is constructed while such y is UNDEC then $S \cup \{y\}$ is admissible also. Recall that preferred extensions are the maximal admissible sets, hence no need to label such y UNDEC.

Another fundamental issue to take into account is the selection of BLANK arguments that are to be labeled IN. The point behind adopting a selection strategy is to try to achieve a preferred extension more efficiently. This is critical when the problem is about constructing only *one* extension. Therefore, algorithm 1 (line 8) applies the following selection options:

1. Algorithm 1 tries to select first a BLANK argument, say y, that is not attacked at all or is attacked by OUT/MUST_OUT arguments only. The justification of this selection is related to the second pruning property used by the algorithm. Note that the earlier we pick such y to be labeled IN, the bigger part of the search tree to be avoided. Recall that such y will not lead to expanding a right child according to the second pruning property.

2. Otherwise the algorithm picks up a BLANK argument, say y, such that $|\{z : z \in \{y\}^+$ and the label of z is not OUT$\}|$ is maximal. The intuition is that maximising the number of OUT arguments will minimise the number of BLANK/MUST_OUT arguments. Thus, the

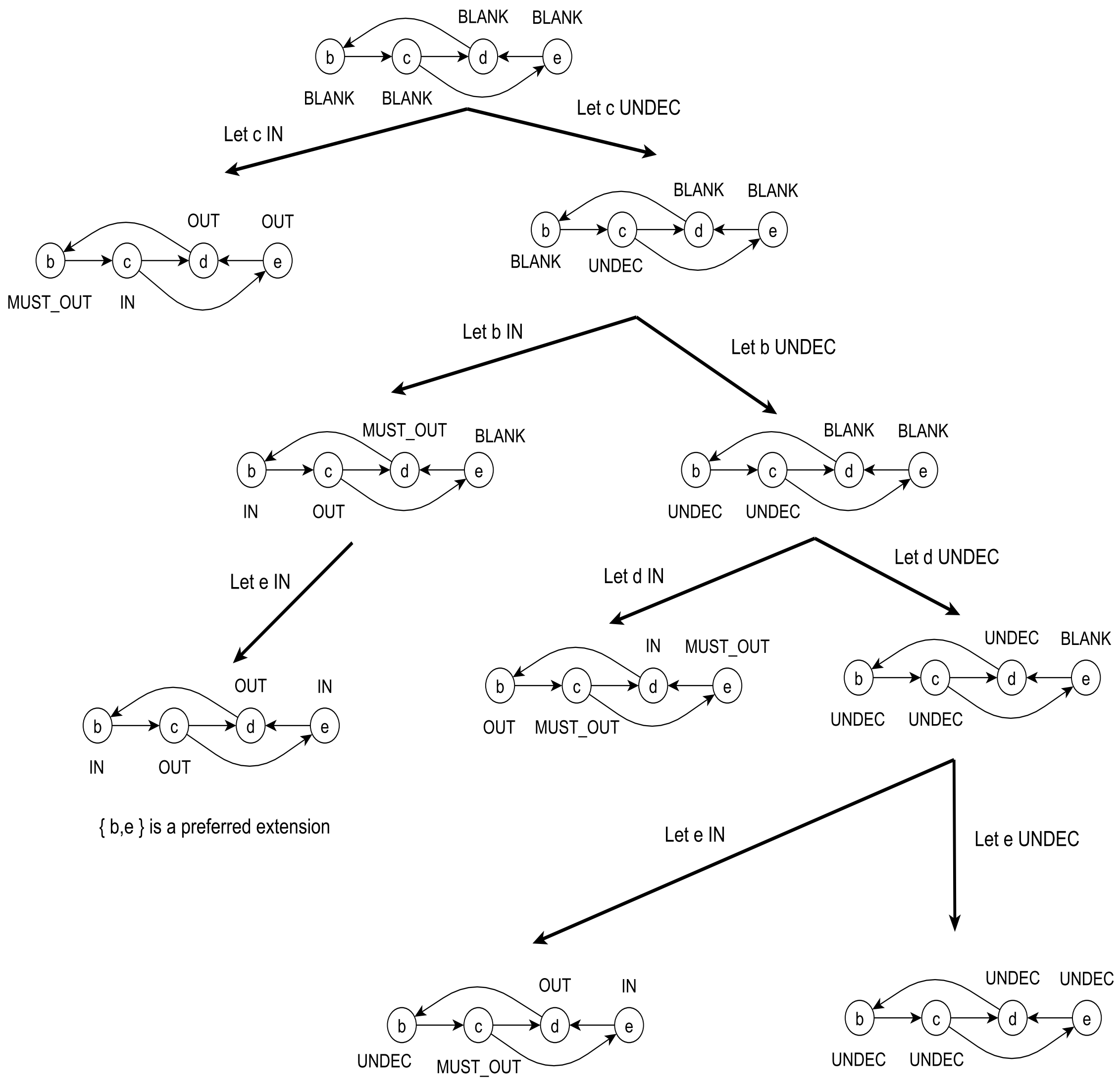

Figure 3: Enumerating preferred extensions of an AF using algorithm 1.

new generated state (i.e. child), due to selecting such y, is much closer to the state where a preferred extension is captured. Recall that a preferred extension is achieved if and only if for each $x \in A$ the label of x is IN, OUT or UNDEC.

Algorithm 1, like all algorithms in this paper, is self-contained and self-explanatory. Figure 3, however, illustrates algorithm 1 running on an AF.

3.2 Enumerating Stable Extensions of any AF

Algorithm 2 lists all stable extensions. Algorithm 2 can be seen as a new implementation of the algorithm of Dimopoulos et al. (1997). Algorithm 2 differs from algorithm 1 in two ways:

Algorithm 1: Enumerating all preferred extensions of an AF (A,R).

```
1  $Lab : A \to \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$; $Lab \leftarrow \emptyset$;
2  **foreach** $x \in A$ **do** $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $E_{preferred} \subseteq 2^A$; $E_{preferred} \leftarrow \emptyset$;
4  **call** find-preferred-extensions($Lab$);
5  **report** $E_{preferred}$ is the set of all preferred extensions;

6  **procedure** find-preferred-extensions($Lab$) **begin**
7  **while** $\exists y \in A : Lab(y) = BLANK$ **do**
8      **select** $y$ with $Lab(y) = BLANK$ and $\forall z \in \{y\}^- \; Lab(z) \in \{OUT, MUST\_OUT\}$,
       **otherwise select** $y$ with $Lab(y) = BLANK$ $s.t.$ $\forall z \in A : Lab(z) = BLANK, |\{x : x \in \{y\}^+ \wedge Lab(x) \neq OUT\}| \geq |\{x : x \in \{z\}^+ \wedge Lab(x) \neq OUT\}|$;
9      $Lab' \leftarrow Lab$;
10     $Lab'(y) \leftarrow IN$;
11     **foreach** $z \in \{y\}^+$ **do** $Lab'(z) \leftarrow OUT$;
12     **foreach** $z \in \{y\}^-$ **do**
13         **if** $Lab'(z) \in \{UNDEC, BLANK\}$ **then**
14             $Lab'(z) \leftarrow MUST\_OUT$;
15             **if** $\nexists w \in \{z\}^- : Lab'(w) = BLANK$ **then**
16                 $Lab(y) \leftarrow UNDEC$;
17                 **goto** line 7;
18     **call** find-preferred-extensions($Lab'$);
19     **if** $\exists z \in \{y\}^- : Lab(z) \in \{BLANK, UNDEC\}$ **then**
20         $Lab(y) \leftarrow UNDEC$;
21     **else**
22         $Lab \leftarrow Lab'$;
23 **if** $\nexists x : Lab(x) = MUST\_OUT$ **then**
24     $S \leftarrow \{x \mid Lab(x) = IN\}$;
25     **if** $\nexists T \in E_{preferred} : S \subseteq T$ **then** $E_{preferred} \leftarrow E_{preferred} \cup \{S\}$;
26 **end procedure**
```

1. Algorithm 2 uses four labels: IN, OUT, BLANK and MUST_OUT. The usage of these labels is as outlined in algorithm 1 with one distinction: the role of the UNDEC label used in algorithm 1 is now overloaded to the MUST_OUT label. Meaning, in algorithm 2 the MUST_OUT label is used also for labeling an argument, say x, trying to build a stable extension without x. This is because any argument, say x, outside a candidate stable extension should be attacked by an argument in the extension, hence x should be labeled MUST_OUT (not UNDEC as it is the case in algorithm 1.)

2. In algorithm 1, $P = \{w \mid \text{the label of } w \text{ is IN}\}$ is a preferred extension if and only if for each $x \in A$, the label of x is not BLANK nor MUST_OUT and P is not a subset of a previously found preferred extension. In algorithm 2 (line 24) the set $\{w \mid \text{the label of } w \text{ is IN}\}$ is a stable extension if and only if for every $x \in A$, the label of x is not BLANK nor MUST_OUT.

Algorithm 2: Enumerating all stable extensions of an AF (A,R).

```
1  $Lab : A \to \{IN, OUT, MUST\_OUT, BLANK\}$; $Lab \leftarrow \emptyset$;
2  **foreach** $x \in A$ **do** $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $E_{stable} \subseteq 2^A$; $E_{stable} \leftarrow \emptyset$;
4  **call** find-stable-extensions($Lab$);
5  **report** $E_{stable}$ is the set of all stable extensions;

6  **procedure** find-stable-extensions($Lab$) **begin**
7  **while** $\exists y \in A : Lab(y) = BLANK$ **do**
8      **select** $y$ with $Lab(y) = BLANK$ and $\forall z \in \{y\}^- \; Lab(z) \in \{OUT, MUST\_OUT\}$,
       **otherwise select** $y$ with $Lab(y) = BLANK$ s.t. $\forall z : Lab(z) = BLANK$, $|\{x : x \in \{y\}^+ \wedge Lab(x) \neq OUT\}| \geq |\{x : x \in \{z\}^+ \wedge Lab(x) \neq OUT\}|$;
9      $Lab' \leftarrow Lab$;
10     $Lab'(y) \leftarrow IN$;
11     **foreach** $z \in \{y\}^+$ **do** $Lab'(z) \leftarrow OUT$;
12     **foreach** $z \in \{y\}^-$ **do**
13         **if** $Lab'(z) = BLANK$ **then**
14             $Lab'(z) \leftarrow MUST\_OUT$;
15             **if** $\forall w \in \{z\}^- \; Lab'(w) \neq BLANK$ **then**
16                 $Lab(y) \leftarrow MUST\_OUT$;
17                 **goto** line 7;
18     **call** find-stable-extensions($Lab'$);
19     **if** $\exists z \in \{y\}^- : Lab(z) = BLANK$ **then**
20         $Lab(y) \leftarrow MUST\_OUT$;
21     **else**
22         $Lab \leftarrow Lab'$;
23 **if** $\forall x : Lab(x) \neq MUST\_OUT$ **then**
24     $S \leftarrow \{x \mid Lab(x) = IN\}$;
25     $E_{stable} \leftarrow E_{stable} \cup \{S\}$;
26 **end procedure**
```

3.3 Enumerating Complete Extensions of any AF

Algorithm 3 lists all complete extensions. Algorithm 3 is a modification of algorithm 1 that enumerates preferred extensions. In algorithm 1, $P = \{w \mid \text{the label of } w \text{ is IN}\}$ is a preferred extension if and only if for each $x \in A$, the label of x is not BLANK nor MUST_OUT and P is not a subset of a previously found preferred extension. In algorithm 3 (line 10) the set $\{w \mid \text{the label of } w \text{ is IN}\}$ is a complete extension if and only if

C1. for every $x \in A$, the label of x is not MUST_OUT and

C2. there is no z with UNDEC (or BLANK) label such that for every $y \in \{z\}^-$ the label of y is OUT.

Recall that a complete extension is an admissible set S such that for every x acceptable with respect to S, x belongs to S. Thus, the condition C1 ensures admissibility while C2 guarantees completeness.

3.4 Enumerating Stage Extensions of any AF

Algorithm 4 lists all stage extensions. Algorithm 4 is an alteration of algorithm 1 that enumerates preferred extensions. Algorithm 4 uses four labels: IN, OUT, UNDEC and BLANK. The usage of these labels is as outlined in algorithm 1 with one distinction: the role of the MUST_OUT label used in algorithm 1 is now overloaded to the UNDEC label. Meaning, in algorithm 4 the UNDEC label is used also for identifying arguments that attack an IN argument. This is because any argument attacks an argument of a stage extension is not necessarily attacked by an argument of the extension.

Algorithm 4 constructs conflict free subsets of A. In particular, algorithm 4 (line 26) keeps a record of the conflict free set $\{w \mid \text{the label of } w \text{ is IN}\}$ if and only if for each $x \in A$, the label of x is not BLANK. After constructing such conflict free subsets, algorithm 4 decides that a conflict free subset, say S, is a stage extension if and only if $S \cup S^+$ is maximal, see lines 5-9. As might be expected, argument selection and pruning strategies used in admissibility based semantics will not be applicable to stage semantics, which are based on conflict free sets. Therefore, as a pruning strategy we skip labeling an argument, say y, UNDEC if and only if for each $z \in \{y\}^+ \cup \{y\}^-$, the label of z is OUT or UNDEC. This is based on the following property: if a conflict free set, say S, will be captured while such y is UNDEC then $S \cup \{y\}$ is also conflict free, and hence, there is no need to label y with UNDEC since $S \cup \{y\} \supset S$; recall that algorithm 4 labels an argument UNDEC trying to build a stage extension excluding the argument. On selecting the next BLANK argument to be labeled IN, we consider the rule:

R1. select a BLANK argument y s.t. for each $z \in \{y\}^+ \cup \{y\}^-$, the label of z is OUT or UNDEC.

R2. otherwise select a BLANK argument y such that $|\{x : x \in \{y\}^+ \cup \{y\}^- \text{ and the label of } x \text{ is BLANK}\}|$ is maximal.

Note the correlation between R1 and the applied pruning strategy: the earlier we label the argument selected by R1 with IN, the bigger the part of the search tree that will be bypassed. Regarding the benefit of R2, recall that the aim of argument selection is to accelerate achieving a goal state, which is a conflict free subset S such that $S \cup S^+$ is maximal and there is no $x \in A$ with the BLANK label. Indeed, R2 minimises the number of BLANK arguments by maximising the number of OUT/UNDEC arguments.

Algorithm 3: Enumerating all complete extensions of an AF (A,R).

```
1  $Lab : A \rightarrow \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$; $Lab \leftarrow \emptyset$;
2  **foreach** $x \in A$ **do** $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $E_{complete} \subseteq 2^A$; $E_{complete} \leftarrow \emptyset$;
4  **call** find-complete-extensions($Lab$);
5  **report** $E_{complete}$ is the set of all complete extensions;

6  **procedure** find-complete-extensions($Lab$) **begin**
7  **if** $\nexists y \in A : Lab(y) = MUST\_OUT$ **then**
8      **if** $\nexists x : Lab(x) \in \{UNDEC, BLANK\} \wedge \forall z \in \{x\}^- \ Lab(z) = OUT$ **then**
9          $S \leftarrow \{w \in A \mid Lab(w) = IN\}$;
10         $E_{complete} \leftarrow E_{complete} \cup \{S\}$;
11 **while** $\exists y \in A : Lab(y) = BLANK$ **do**
12     **select** $y$ with $Lab(y) = BLANK$ and $\forall z \in \{y\}^- \ Lab(z) \in \{OUT, MUST\_OUT\}$,
       **otherwise select** $y$ with $Lab(y) = BLANK \ s.t. \ \forall z \in A : Lab(z) = BLANK, |\{x : x \in$
       $\{y\}^+ \wedge Lab(x) \neq OUT\}| \geq |\{x : x \in \{z\}^+ \wedge Lab(x) \neq OUT\}|$;
13     $Lab' \leftarrow Lab$;
14     $Lab'(y) \leftarrow IN$;
15     **foreach** $z \in \{y\}^+$ **do** $Lab'(z) \leftarrow OUT$;
16     **foreach** $z \in \{y\}^-$ **do**
17         **if** $Lab'(z) \in \{UNDEC, BLANK\}$ **then**
18             $Lab'(z) \leftarrow MUST\_OUT$;
19             **if** $\nexists w \in \{z\}^- : Lab'(w) = BLANK$ **then**
20                 $Lab(y) \leftarrow UNDEC$;
21                 **goto** line 11;
22     **call** find-complete-extensions($Lab'$);
23     **if** $\exists z \in \{y\}^- : Lab(z) \in \{BLANK, UNDEC\}$ **then**
24         $Lab(y) \leftarrow UNDEC$;
25     **else**
26         $Lab \leftarrow Lab'$;
27 **end procedure**
```

Algorithm 4: Enumerating all stage extensions of an AF (A,R).

```
1  $Lab : A \rightarrow \{IN, OUT, UNDEC, BLANK\}$; $Lab \leftarrow \emptyset$;
2  foreach $x \in A$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $E_{stage} \subseteq \{Lab_1 \mid Lab_1 : A \rightarrow \{IN, OUT, UNDEC, BLANK\}\}$; $E_{stage} \leftarrow \emptyset$;
4  call find-conflict-free-sets($Lab$);
5  foreach $Lab_1 \in E_{stage}$ do
6      foreach $Lab_2 \in E_{stage}$ do
7          if $\{x : Lab_1(x) \in \{IN, OUT\}\} \subsetneq \{z : Lab_2(z) \in \{IN, OUT\}\}$ then
8              $E_{stage} \leftarrow E_{stage} \setminus \{Lab_1\}$;
9              continue to next iteration from line 5;
10 foreach $Lab_1 \in E_{stage}$ do
11     report $\{x : Lab_1(x) = IN\}$ is a stage extension ;

12 procedure find-conflict-free-sets($Lab$) begin
13 while $\exists y \in A : Lab(y) = BLANK$ do
14     select $y$ with $Lab(y) = BLANK$ such that $\forall z \in \{y\}^+ \cup \{y\}^-$ $Lab(z) \in \{OUT, UNDEC\}$,
       otherwise select $y$ with $Lab(y) = BLANK$ such that $\forall z : Lab(z) = BLANK$, $|\{x : x \in$
       $\{y\}^+ \cup \{y\}^- \wedge Lab(x) = BLANK\}| \geq |\{x : x \in \{z\}^+ \cup \{z\}^- \wedge Lab(x) = BLANK\}|$;
15     $Lab' \leftarrow Lab$;
16     $Lab'(y) \leftarrow IN$;
17     foreach $z \in \{y\}^+$ do $Lab'(z) \leftarrow OUT$;
18     foreach $z \in \{y\}^-$ do
19         if $Lab'(z) \in \{BLANK\}$ then
20             $Lab'(z) \leftarrow UNDEC$;
21     call find-conflict-free-sets($Lab'$);
22     if $\exists z \in \{y\}^+ \cup \{y\}^-$ with $Lab(z) = BLANK$ then
23         $Lab(y) \leftarrow UNDEC$;
24     else
25         $Lab \leftarrow Lab'$;
26 $E_{stage} \leftarrow E_{stage} \cup \{Lab\}$;
27 end procedure
```

3.5 Enumerating Semi Stable Extensions of any AF

Algorithm 5 enumerates all semi stable extensions. Again, algorithm 5 is a reproduction of algorithm 1. Actually, algorithm 5 firstly builds admissible sets. Then, the algorithm decides that an admissible set, say S, is a semi stable extension if and only if $S \cup S^+$ is maximal; see lines 6-10.

Algorithm 5: Enumerating all semi stable extensions of an AF (A,R).

```
1  $Lab : A \to \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$; $Lab \leftarrow \emptyset$;
2  foreach $x \in A$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $E_{semi-stable} \subseteq \{Lab_1 \mid Lab_1 : A \to \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}\}$;
4  $E_{semi-stable} \leftarrow \emptyset$;
5  call find-admissible-sets($Lab$);
6  foreach $Lab_1 \in E_{semi-stable}$ do
7      foreach $Lab_2 \in E_{semi-stable}$ do
8          if $\{x : Lab_1(x) \in \{IN, OUT\}\} \subsetneq \{z : Lab_2(z) \in \{IN, OUT\}\}$ then
9              $E_{semi-stable} \leftarrow E_{semi-stable} \setminus \{Lab_1\}$;
10             continue to next iteration from line 6;
11 foreach $Lab_1 \in E_{semi-stable}$ do
12     report $\{x : Lab_1(x) = IN\}$ is a semi stable extension ;

13 procedure find-admissible-sets($Lab$) begin
14 while $\exists y \in A : Lab(y) = BLANK$ do
15     select $y$ with $Lab(y) = BLANK$ and $\forall z \in \{y\}^- \; Lab(z) \in \{OUT, MUST\_OUT\}$,
       otherwise select $y$ with $Lab(y) = BLANK \; s.t. \; \forall z \in A : Lab(z) = BLANK, |\{x : x \in$
       $\{y\}^+ \wedge Lab(x) \neq OUT\}| \geq |\{x : x \in \{z\}^+ \wedge Lab(x) \neq OUT\}|$;
16     $Lab' \leftarrow Lab$;
17     $Lab'(y) \leftarrow IN$;
18     foreach $z \in \{y\}^+$ do $Lab'(z) \leftarrow OUT$;
19     foreach $z \in \{y\}^-$ do
20         if $Lab'(z) \in \{UNDEC, BLANK\}$ then
21             $Lab'(z) \leftarrow MUST\_OUT$;
22             if $\nexists w \in \{z\}^- : Lab'(w) = BLANK$ then
23                 $Lab(y) \leftarrow UNDEC$;
24                 goto line 14;
25     call find-admissible-sets($Lab'$);
26     if $\exists z \in \{y\}^- : Lab(z) \in \{BLANK, UNDEC\}$ then
27         $Lab(y) \leftarrow UNDEC$;
28     else
29         $Lab \leftarrow Lab'$;
30 if $\nexists x \in A : Lab(x) = MUST\_OUT$ then
31     $E_{semi-stable} \leftarrow E_{semi-stable} \cup \{Lab\}$;
32 end procedure
```

3.6 Constructing the Ideal Extension of any AF

Algorithm 6 builds the ideal extension. The algorithm is a modification of algorithm 1. Algorithm 6 (line 28) records $\{w \mid \text{the label of } w \text{ is IN}\}$ as an admissible set if and only if for each $x \in A$, the label of x is not BLANK nor MUST_OUT. However, the algorithm also constructs (line 27) $S = \{x \in A \mid$ there exists an admissible set T such that $x \in T^+\}$. After building a set of admissible sets and having S constructed, algorithm 6 considers an admissible set I as the ideal extension if and only if $I \cap S = \emptyset$; see lines 6-8. Recall that the ideal extension is the maximal (w.r.t. $\subseteq$) admissible set that is contained in every preferred extension. Satisfying the condition $I \cap S = \emptyset$ implies that the arguments of I are not attacked by any admissible set (see the definition of S above), which means I is contained in every preferred extension. To ensure such I is maximal, algorithm 6 collects admissible sets in descending order: from larger sets to smaller ones. In consequence, the algorithm checks the collected admissible sets with the condition $I \cap S = \emptyset$ starting from larger admissible sets to smaller ones.

3.7 Constructing the Grounded Extension of any AF

Algorithm 7 can be viewed as another implementation of the algorithm described by Modgil and Caminada (2009) for building the grounded extension.

4. Labeling Attacks as a Generalization of Labeling Arguments

In this section we illustrate how to enumerate extensions, under a number of argumentation semantics, by labeling attacks together with arguments instead of labeling arguments solely. To this end, we develop algorithms for listing extensions of an AFRA (Baroni et al., 2011b) under preferred, stable, complete, stage, semi stable, ideal and grounded semantics in subsections 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7 respectively. All these algorithms are basically a generalization of the algorithms presented in the previous section, hence these algorithms list extensions of any AF/AFRA.

4.1 Enumerating Preferred Extensions of any AF/AFRA

Algorithm 8 enumerates all preferred extensions of an AFRA. Algorithm 8 is a generalization of algorithm 1. The idea is based on using five labels: *IN*, *OUT*, *MUST_OUT*, *BLANK* and *UNDEC*. The BLANK label is the initial label for all arguments and attacks. A BLANK attack $y \in \overline{R}$ is labeled IN to indicate that y might be in a preferred extension. An argument x is labeled OUT if and only if there is $y \in \overline{R}$ with the label IN such that $trg(y) = x$. An attack $z \in \overline{R}$ is labeled OUT if and only if there is $y \in \overline{R}$ with the label IN such that $trg(y) \in \{z, src(z)\}$. A BLANK argument x is labeled IN, implying that x might be in a preferred extension, if and only if there is $y \in \overline{R}$ with the label IN such that $src(y) = x$ or for each $z \in \overline{R} : trg(z) = x$ the label of z is OUT. An attack y is labeled UNDEC to try to find a preferred extension excluding y. An attack z with the label BLANK/UNDEC is labeled MUST_OUT if and only if there is $y \in \overline{R}$ with the label IN such that $trg(z) \in \{y, src(y)\}$. Every time an attack is labeled IN the labels of some attacks and arguments might change accordingly, see lines 10-20 of algorithm 8. As a selection rule, line 8 represents the strategy by which the algorithm selects the next attack, that is BLANK, to be labeled IN. The rule and its grounds is in parallel to the selection rule applied in algorithm 1 for enumerating preferred extensions of an AF. Likewise, algorithm 8 applies two pruning tactics:

Algorithm 6: Constructing the ideal extension of an AF (A, R).

```
1  $Lab : A \to \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$; $Lab \leftarrow \emptyset$;
2  foreach $x \in A$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $E_{ideal} : \mathbb{Z} \to 2^A$;  $E_{ideal} \leftarrow \emptyset$;
4  $S \leftarrow \emptyset$;
5  call find-admissible-sets($Lab$);
6  foreach $i = 1 .. |E_{ideal}|$ do
7      if $E_{ideal}(i) \cap S = \emptyset$ then
8          report $E_{ideal}(i)$ is the ideal extension; exit;

9  procedure find-admissible-sets($Lab$) begin
10 while $\exists y \in A : Lab(y) = BLANK$ do
11     select $y$ with $Lab(y) = BLANK$ and $\forall z \in \{y\}^- \ Lab(z) \in \{OUT, MUST\_OUT\}$,
       otherwise select $y$ with $Lab(y) = BLANK \ s.t. \ \forall z \in A : Lab(z) = BLANK, |\{x : x \in \{y\}^+ \wedge Lab(x) \neq OUT\}| \geq |\{x : x \in \{z\}^+ \wedge Lab(x) \neq OUT\}|$;
12     $Lab' \leftarrow Lab$;
13     $Lab'(y) \leftarrow IN$;
14     foreach $z \in \{y\}^+$ do $Lab'(z) \leftarrow OUT$;
15     foreach $z \in \{y\}^-$ do
16         if $Lab'(z) \in \{UNDEC, BLANK\}$ then
17             $Lab'(z) \leftarrow MUST\_OUT$;
18             if $\nexists w \in \{z\}^- : Lab'(w) = BLANK$ then
19                 $Lab(y) \leftarrow UNDEC$;
20                 goto line 10;
21     call find-admissible-sets($Lab'$);
22     if $\exists z \in \{y\}^- : Lab(z) \in \{BLANK, UNDEC\}$ then
23         $Lab(y) \leftarrow UNDEC$;
24     else
25         $Lab \leftarrow Lab'$;
26 if $\nexists w \in A : Lab(w) = MUST\_OUT$ then
27     $S \leftarrow S \cup \{x \mid Lab(x) = OUT\}$;
28     $E_{ideal} \leftarrow E_{ideal} \cup \{(|E_{ideal}| + 1, \{z \mid Lab(z) = IN\})\}$;
29 end procedure
```

Algorithm 7: Constructing the grounded extension of an AF (A, R).

```
1  $Lab : A \to \{IN, OUT, UNDEC\}$; $Lab \leftarrow \emptyset$;
2  foreach $w \in A$ do $Lab \leftarrow Lab \cup \{(w, UNDEC)\}$;
3  while $\exists x \ with \ Lab(x) = UNDEC : \forall y \in \{x\}^- \ Lab(y) = OUT$ do
4      foreach $x \ with \ Lab(x) = UNDEC : \forall y \in \{x\}^- \ Lab(y) = OUT$ do
5          $Lab(x) \leftarrow IN$;
6          foreach $z \in \{x\}^+$ do $Lab(z) \leftarrow OUT$;
7  report the grounded extension is $\{w \mid Lab(w) = IN\}$;
```

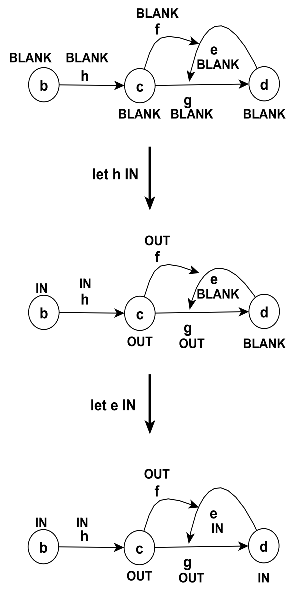

Figure 4: How algorithm 8 works on an AFRA.

1. Algorithm 8 (lines 17- 20) skips labeling an attack y IN (i.e. skips expanding a left child) if and only if

$$\exists z : trg(z) \in \{y, src(y)\} \; and\ the\ label\ of \; \text{z} \; is\ not\ OUT\ and$$
$$\nexists\ w\ with\ the\ label\ BLANK:\ trg(w) \ \in \ \{z, src(z)\}.$$

2. Algorithm 8 (lines 22- 25) skips labeling an attack y UNDEC (i.e. skips expanding a right child) if and only if for each $z \in \overline{R} : trg(z) \in \{y, src(y)\}$, the label of z is OUT or MUST_OUT.

To get the general idea of algorithm 8 see figure 4 that shows how the algorithm works on the AFRA depicted in figure 2.

4.2 Enumerating Stable Extensions of any AF/AFRA

Algorithm 9 enumerates all stable extensions. Actually, algorithm 9 is a modification of algorithm 8 that lists preferred extensions. However there are two differences:

1. Algorithm 9 uses four labels: IN, OUT, BLANK and MUST_OUT. The usage of these labels is as outlined in algorithm 8 with one difference: the role of the UNDEC label used in algorithm 8 is now overloaded to the MUST_OUT label. That is, in algorithm 9 the MUST_OUT label is used also for labeling an attack, say x, trying to build a stable extension without x. This is because any attack, say x, outside a candidate stable extension should be defeated by an attack in the extension, hence x should be labeled MUST_OUT.

2. In algorithm 8 we find a preferred extension, say P, if and only if for each $x \in A \cup \overline{R}$, x is not BLANK nor MUST_OUT and P is not a subset of a previously found preferred extension. In algorithm 9 we encounter a stable extension if and only if for each $x \in A \cup \overline{R}$, x is not BLANK nor MUST_OUT.

Algorithm 8: Enumerating all preferred extensions of an AFRA $(A, \overline{R})$.

```
1  $Lab : (A \cup \overline{R}) \to \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$; $Lab \leftarrow \emptyset$;
2  foreach $x \in A \cup \overline{R}$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $\overline{E}_{preferred} \subseteq 2^{A \cup \overline{R}}$; $\overline{E}_{preferred} \leftarrow \emptyset$;
4  call find-preferred-extensions($Lab$);
5  report $\overline{E}_{preferred}$ is the set of all preferred extensions;

6  procedure find-preferred-extensions($Lab$) begin
7  while $\exists y \in \overline{R} : Lab(y) = BLANK$ do
8      select $y \in \overline{R}$ with $Lab(y) = BLANK$ such that
       $\forall z \in \overline{R} : trg(z) \in \{y, src(y)\}$ $Lab(z) \in \{OUT, MUST\_OUT\}$, otherwise select $y \in \overline{R}$ with
       $Lab(y) = BLANK$ such that $\forall z \in \overline{R} : Lab(z) = BLANK$
       $|\{x : src(x) = trg(y) \wedge Lab(x) \neq OUT\}| \geq |\{x : src(x) = trg(z) \wedge Lab(x) \neq OUT\}|$;
9      $Lab' \leftarrow Lab$;
10     $Lab'(y) \leftarrow IN$;
11     $Lab'(src(y)) \leftarrow IN$;
12     $Lab'(trg(y)) \leftarrow OUT$;
13     if $trg(y) \in A$ then
14         foreach $z \in \overline{R} : src(z) = trg(y)$ do
15             $Lab'(z) \leftarrow OUT$;
16     foreach $z \in \overline{R} : Lab'(z) \in \{BLANK, UNDEC\} \wedge trg(z) \in \{y, src(y)\}$ do
17         $Lab'(z) \leftarrow MUST\_OUT$;
18         if $\nexists\, w \in \overline{R} : Lab'(w) = BLANK \wedge trg(w) \in \{z, src(z)\}$ then
19             $Lab(y) \leftarrow UNDEC$;
20             goto line 7;
21     call find-preferred-extensions($Lab'$);
22     if $\exists z \in \overline{R} : Lab(z) \in \{BLANK, UNDEC\} \wedge trg(z) \in \{y, src(y)\}$ then
23         $Lab(y) \leftarrow UNDEC$;
24     else
25         $Lab \leftarrow Lab'$;
26 if $\nexists w \in \overline{R} : Lab(w) = MUST\_OUT$ then
27     foreach $x \in A$ with $Lab(x) = BLANK$ s.t. $\forall z \in \overline{R} : trg(z) = x$ $(Lab(z) = OUT)$ do
       $Lab(x) \leftarrow IN$;
28     $S \leftarrow \{x \in A \cup \overline{R} \mid Lab(x) = IN\}$;
29     if $\forall T \in \overline{E}_{preferred}$ $(S \nsubseteq T)$ then
30         $\overline{E}_{preferred} \leftarrow \overline{E}_{preferred} \cup \{S\}$;
31 end procedure
```

Algorithm 9: Enumerating all stable extensions of an AFRA $(A, \overline{R})$.

```
1  $Lab : (A \cup \overline{R}) \rightarrow \{IN, OUT, MUST\_OUT, BLANK\}$;  $Lab \leftarrow \emptyset$;
2  foreach $x \in A \cup \overline{R}$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $\overline{E}_{stable} \subseteq 2^{A \cup \overline{R}}$;  $\overline{E}_{stable} \leftarrow \emptyset$;
4  call find-stable-extensions($Lab$);
5  report $\overline{E}_{stable}$ is the set of all stable extensions;

6  procedure find-stable-extensions($Lab$) begin
7  while $\exists y \in \overline{R} : Lab(y) = BLANK$ do
8      select $y \in \overline{R}$ with $Lab(y) = BLANK$ such that
       $\forall z \in \overline{R} : trg(z) \in \{y, src(y)\}\ Lab(z) \in \{OUT, MUST\_OUT\}$, otherwise select $y \in \overline{R}$ with
       $Lab(y) = BLANK$ such that $\forall z \in \overline{R} : Lab(z) = BLANK$
       $|\{x : src(x) = trg(y) \wedge Lab(x) \neq OUT\}| \geq |\{x : src(x) = trg(z) \wedge Lab(x) \neq OUT\}|$;
9      $Lab' \leftarrow Lab$;
10     $Lab'(y) \leftarrow IN$;
11     $Lab'(src(y)) \leftarrow IN$;
12     $Lab'(trg(y)) \leftarrow OUT$;
13     if $trg(y) \in A$ then
14         foreach $z \in \overline{R} : src(z) = trg(y)$ do
15             $Lab'(z) \leftarrow OUT$;
16     foreach $z \in \overline{R} : Lab'(z) = BLANK \wedge trg(z) \in \{y, src(y)\}$ do
17         $Lab'(z) \leftarrow MUST\_OUT$;
18         if $\nexists\, w \in \overline{R} : Lab'(w) = BLANK \wedge trg(w) \in \{z, src(z)\}$ then
19             $Lab(y) \leftarrow MUST\_OUT$;
20             goto line 7;
21     call find-stable-extensions($Lab'$);
22     if $\exists z \in \overline{R} : Lab(z) = BLANK \wedge trg(z) \in \{y, src(y)\}$ then
23         $Lab(y) \leftarrow MUST\_OUT$;
24     else
25         $Lab \leftarrow Lab'$;
26 if $\nexists w \in \overline{R} : Lab(w) = MUST\_OUT$ then
27     foreach $x \in A$ with $Lab(x) = BLANK$ s.t. $\forall z \in \overline{R} : trg(z) = x\ (Lab(z) = OUT)$ do
       $Lab(x) \leftarrow IN$;
28     $\overline{E}_{stable} \leftarrow \overline{E}_{stable} \cup \{\{x \in A \cup \overline{R} \mid Lab(x) = IN\}\}$;
29 end procedure
```

4.3 Enumerating Complete Extensions of any AF/AFRA

Algorithm 10 enumerates all complete extensions. Again, algorithm 10 is a modification of algorithm 8 that lists preferred extensions. In algorithm 8 we achieve a preferred extension, say P, if and only if for each $x \in A \cup \overline{R}$, the label of x is not BLANK nor MUST_OUT and P is not a subset of a previously found preferred extension. However, in algorithm 10 (line 7) we encounter a complete extension if and only if

C1. there is no $z \in \overline{R}$ with the MUST_OUT label and

C2. there does not exist $w \in \overline{R}$ such that

  (a) the label of w is UNDEC or BLANK and
  (b) for each $y \in \overline{R} : trg(y) \in \{w, src(w)\}$, the label of y is OUT.

Thus, C1 ensures admissibility while C2 guarantees completeness.

4.4 Enumerating Stage Extensions of any AF/AFRA

Algorithm 11 lists all stage extensions. The algorithm is a rewrite of algorithm 8. However, algorithm 11 uses four labels: IN, OUT, BLANK and UNDEC. The usage of these labels is as outlined in algorithm 8 with one difference: the role of the MUST_OUT label used in algorithm 8 is now overloaded to the UNDEC label. That is, in algorithm 11 the UNDEC label is used also for identifying attacks that attack an IN argument/attack. This is because any attack defeats an argument/attack of a stage extension is not necessarily defeated by an attack of the extension.

Algorithm 11 (lines 14-29) finds a set of conflict free subsets of $A \cup \overline{R}$ rather than constructing admissible subsets as done by algorithm 8. In algorithm 8 the set $\{w \in A \cup \overline{R} \mid \text{the label of } w \text{ is IN}\}$ is reported as an admissible set if and only if for each $x \in A \cup \overline{R}$, the label of x is not BLANK nor MUST_OUT. In algorithm 11 the set $\{w \in A \cup \overline{R} \mid \text{the label of } w \text{ is IN}\}$ is recorded as a conflict free set (i.e. a stage extension candidate) if and only if for each $x \in A \cup \overline{R}$, the label of x is not BLANK, see lines 14 & 31. After building a set of conflict free subsets, algorithm 11 decides that a conflict free subset $S \subseteq A \cup \overline{R}$ is a stage extension if and only if $S \cup \{x \mid \exists y \in S : y \text{ defeats } x\}$ is maximal, see lines 6-10. As we stated earlier, argument selection and pruning strategies used in semantics that are based on admissible sets will not be applicable to stage semantics, which are based on conflict free sets. Therefore, as a pruning strategy (line 29 of algorithm 11) we skip labeling an attack y UNDEC (i.e. skip expanding a right child) if and only if

$$\forall z \in \overline{R} : trg(z) \in \{y, src(y)\} \vee trg(y) \in \{z, src(z)\},$$
$$\textit{the label of z is OUT or UNDEC}.$$

This is based on the property that if a conflict free set, say S, is formed while such y is UNDEC then $S \cup \{y\}$ is also conflict free, and hence, there is no need to label y UNDEC since $S \cup \{y\} \supset S$. On selecting the next BLANK attack to be labeled IN, we apply the following rule (see line 15):

R1. select a BLANK attack y s.t. for each $z \in \overline{R} : trg(z) \in \{y, src(y)\} \vee trg(y) \in \{z, src(z)\}$, the label of z is OUT or UNDEC.

R2. otherwise select a BLANK attack y such that $|\{x : \text{the label of } x \text{ is BLANK and } (src(x) = trg(y) \vee trg(x) \in \{y, src(y)\})\}|$ is maximal.

Algorithm 10: Enumerating all complete extensions of an AFRA $(A, \overline{R})$.

```
1  $Lab : (A \cup \overline{R}) \to \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$; $Lab \leftarrow \emptyset$;
2  foreach $x \in A \cup \overline{R}$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $\overline{E}_{complete} \subseteq 2^{A \cup \overline{R}}$; $\overline{E}_{complete} \leftarrow \emptyset$;
4  call find-complete-extensions($Lab$);
5  report $\overline{E}_{complete}$ is the set of all complete extensions;

6  procedure find-complete-extensions($Lab$) begin
7  if $\nexists v \in \overline{R}$ with $Lab(v) = MUST\_OUT$ and $\nexists w \in \overline{R} : Lab(w) \in \{UNDEC, BLANK\}$ with
   $\forall y \in \overline{R} : trg(y) \in \{w, src(w)\}$ $Lab(y) = OUT$ then
8      foreach $x \in A$ with $Lab(x) = BLANK$ s.t. $\forall z \in \overline{R} : trg(z) = x$ $(Lab(z) = OUT)$ do
       $Lab(x) \leftarrow IN$;
9      $\overline{E}_{complete} \leftarrow \overline{E}_{complete} \cup \{\{x \in A \cup \overline{R} \mid Lab(x) = IN)\}\}$;
10 while $\exists y \in \overline{R} : Lab(y) = BLANK$ do
11     select $y \in \overline{R}$ with $Lab(y) = BLANK$ such that
       $\forall z \in \overline{R} : trg(z) \in \{y, src(y)\}$ $Lab(z) \in \{OUT, MUST\_OUT\}$, otherwise select $y \in \overline{R}$ with
       $Lab(y) = BLANK$ such that $\forall z \in \overline{R} : Lab(z) = BLANK$
       $|\{x : src(x) = trg(y) \wedge Lab(x) \neq OUT\}| \geq |\{x : src(x) = trg(z) \wedge Lab(x) \neq OUT\}|$;
12     $Lab' \leftarrow Lab$;
13     $Lab'(y) \leftarrow IN$;
14     $Lab'(src(y)) \leftarrow IN$;
15     $Lab'(trg(y)) \leftarrow OUT$;
16     if $trg(y) \in A$ then
17         foreach $z \in \overline{R} : src(z) = trg(y)$ do
18             $Lab'(z) \leftarrow OUT$;
19     foreach $z \in \overline{R} : Lab'(z) \in \{BLANK, UNDEC\} \wedge trg(z) \in \{y, src(y)\}$ do
20         $Lab'(z) \leftarrow MUST\_OUT$;
21         if $\nexists w \in \overline{R} : Lab'(w) = BLANK \wedge trg(w) \in \{z, src(z)\}$ then
22             $Lab(y) \leftarrow UNDEC$;
23             goto line 10;
24     call find-complete-extensions($Lab'$);
25     if $\exists z \in \overline{R} : Lab(z) \in \{BLANK, UNDEC\} \wedge trg(z) \in \{y, src(y)\}$ then
26         $Lab(y) \leftarrow UNDEC$;
27     else
28         $Lab \leftarrow Lab'$;
29 end procedure
```

The aim of R1 is to maximise the gain of the applied pruning strategy. Meaning, the earlier we label the selected argument by R1 with IN, the greater the saving will be in terms of the part of the search tree pruned. Regarding R2, note that a goal state (i.e. conflict free set) is reached if and only if for each $x \in A \cup \overline{R}$, the label of x is not BLANK. Thus, R2 tries to maximise the number of OUT/UNDEC attacks/arguments, which implies minimising the number of BLANK attacks/arguments.

4.5 Enumerating Semi Stable Extensions of any AF/AFRA

Algorithm 12 lists all semi stable extensions. Algorithm 12 is a variation of algorithm 8 such that it basically constructs admissible sets. Algorithm 12 (line 31) records the set $\{w \mid \text{the label of } w \text{ is IN}\}$ as an admissible set (that is a semi stable extension candidate) if and only if for each $x \in A \cup \overline{R}$, the label of x is not BLANK nor MUST_OUT. After constructing a set of admissible sets, algorithm 12 decides that an admissible set S is a semi stable extension if and only if $S \cup \{x \mid \exists y \in S : y \text{ defeats } x\}$ is maximal, see lines 6-10.

4.6 Constructing the Ideal Extension of any AF/AFRA

Algorithm 13 builds the ideal extension. In particular, algorithm 13 finds admissible sets (lines 10-28) in the same way algorithm 8 does. However, in enumerating admissible sets algorithm 13 (line 31) also builds the set

$$S = \{x \in A \cup \overline{R} \mid \textit{there is y in an admissible set such that } trg(y) \in \{x, src(x)\}\}$$

After building a set of admissible sets and having such S constructed, algorithm 13 decides that an admissible set I is the ideal extension if and only if $I \cap S = \emptyset$, see lines 6-8. Recall that the ideal extension is the maximal (w.r.t. $\subseteq$) admissible set that is contained in every preferred extension. Satisfying the condition $I \cap S = \emptyset$ implies that the arguments/attacks of I are not defeated by any admissible set (see the definition of S above), which means I is contained in every preferred extension. To ensure such I is maximal, algorithm 13 collects admissible sets in descending order: from larger sets to smaller ones. In consequence, the algorithm checks the collected admissible sets with the condition $I \cap S = \emptyset$ starting from larger admissible sets to smaller ones.

4.7 Constructing the Grounded Extension of any AF/AFRA

Algorithm 14 builds the grounded extension. Algorithm 14 is actually a generalization of algorithm 7.

5. Practical Efficiency

All algorithms presented in this paper were implemented in C++ on a Fedora (release 13) based machine with 4 processors (Intel core i5-750 2.67GHz) and 16GB of memory. As an evaluation criterion, we considered the average elapsed time measured in seconds; the elapsed time was obtained by using the `time` command of Linux. We present experimental results for two purposes. First, we explore the efficiency of the algorithms of section 3. For the second purpose, we confirm that the generalized algorithms of section 4, which enumerate extensions by labeling attacks together with arguments, perform as efficiently as the algorithms of section 3, which enumerate extensions by labeling arguments alone.

Algorithm 11: Enumerating all stage extensions of an AFRA $(A, \overline{R})$.

```
1  $Lab : (A \cup \overline{R}) \to \{IN, OUT, UNDEC, BLANK\};\ Lab \leftarrow \emptyset$;
2  foreach $x \in A \cup \overline{R}$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $\overline{E}_{stage} \subseteq \{Lab_1 \mid Lab_1 : (A \cup \overline{R}) \to \{IN, OUT, UNDEC, BLANK\}\}$;
4  $\overline{E}_{stage} \leftarrow \emptyset$;
5  call find-conflict-free-sets($Lab$);
6  foreach $Lab_1 \in \overline{E}_{stage}$ do
7      foreach $Lab_2 \in \overline{E}_{stage}$ do
8          if $\{x : Lab_1(x) \in \{IN, OUT\}\} \subsetneq \{z : Lab_2(z) \in \{IN, OUT\}\}$ then
9              $\overline{E}_{stage} \leftarrow \overline{E}_{stage} \setminus \{Lab_1\}$;
10             continue to next iteration from line 6;
11 foreach $Lab_1 \in \overline{E}_{stage}$ do
12     report $\{x : Lab_1(x) = IN\}$ is a stage extension ;

13 procedure find-conflict-free-sets($Lab$) begin
14 while $\exists y \in \overline{R} : Lab(y) = BLANK$ do
15     select $y \in \overline{R}$ with $Lab(y) = BLANK$ s.t.
       $\forall z \in \overline{R} : trg(z) \in \{y, src(y)\} \vee trg(y) \in \{z, src(z)\}\ (Lab(z) \in \{OUT, UNDEC\})$,
       otherwise select $y \in \overline{R}$ with $Lab(y) = BLANK$ s.t.
       $\forall z \in \overline{R} : Lab(z) = BLANK\ |\{x : Lab(x) = BLANK \wedge (src(x) = trg(y) \vee trg(x) \in$
       $\{y, src(y)\})\}| \geq |\{x : Lab(x) = BLANK \wedge (src(x) = trg(z) \vee trg(x) \in \{z, src(z)\})\}|$;
16     $Lab' \leftarrow Lab$;
17     $Lab'(y) \leftarrow IN$;
18     $Lab'(src(y)) \leftarrow IN$;
19     $Lab'(trg(y)) \leftarrow OUT$;
20     if $trg(y) \in A$ then
21         foreach $z \in \overline{R} : src(z) = trg(y)$ do
22             $Lab'(z) \leftarrow OUT$;
23     foreach $z \in \overline{R} : Lab'(z) = BLANK \wedge trg(z) \in \{y, src(y)\}$ do
24         $Lab'(z) \leftarrow UNDEC$;
25     call find-conflict-free-sets($Lab'$);
26     if $\exists z \in \overline{R} : Lab(z) = BLANK \wedge (trg(z) \in \{y, src(y)\} \vee trg(y) \in \{z, src(z)\})$ then
27         $Lab(y) \leftarrow UNDEC$;
28     else
29         $Lab \leftarrow Lab'$;
30 foreach $x \in A$ with $Lab(x) = BLANK$ s.t. $\forall z \in \overline{R} : trg(z) = x\ (Lab(z) = OUT)$ do
   $Lab(x) \leftarrow IN$;
31 $\overline{E}_{stage} \leftarrow \overline{E}_{stage} \cup \{Lab\}$;
32 end procedure
```

Algorithm 12: Enumerating all semi stable extensions of an AFRA $(A,\overline{R})$.

```
1  $Lab : (A \cup \overline{R}) \rightarrow \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$;  $Lab \leftarrow \emptyset$;
2  **foreach** $x \in A \cup \overline{R}$ **do** $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
3  $\overline{E}_{semi-stable} \subseteq \{Lab_1 \mid Lab_1 : (A \cup \overline{R}) \rightarrow \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}\}$;
4  $\overline{E}_{semi-stable} \leftarrow \emptyset$;
5  **call** find-admissible-sets($Lab$);
6  **foreach** $Lab_1 \in \overline{E}_{semi-stable}$ **do**
7      **foreach** $Lab_2 \in \overline{E}_{semi-stable}$ **do**
8          **if** $\{x : Lab_1(x) \in \{IN, OUT\}\} \subsetneq \{z : Lab_2(z) \in \{IN, OUT\}\}$ **then**
9              $\overline{E}_{semi-stable} \leftarrow \overline{E}_{semi-stable} \setminus \{Lab_1\}$;
10             continue to next iteration from line 6;
11 **foreach** $Lab_1 \in \overline{E}_{semi-stable}$ **do**
12     **report** $\{x : Lab_1(x) = IN\}$ is a semi stable extension ;

13 **procedure** find-admissible-sets($Lab$) **begin**
14 **while** $\exists y \in \overline{R} : Lab(y) = BLANK$ **do**
15     **select** $y \in \overline{R}$ with $Lab(y) = BLANK$ s.t.
       $\forall z \in \overline{R} : trg(z) \in \{y, src(y)\}\ Lab(z) \in \{OUT, MUST\_OUT\}$, **otherwise select** $y \in \overline{R}$ with
       $Lab(y) = BLANK$ s.t. $\forall z \in \overline{R} : Lab(z) = BLANK\ |\{x : src(x) = trg(y) \wedge Lab(x) \neq$
       $OUT\}| \geq |\{x : src(x) = trg(z) \wedge Lab(x) \neq OUT\}|$;
16     $Lab' \leftarrow Lab$; $Lab'(y) \leftarrow IN$;  $Lab'(src(y)) \leftarrow IN$;
17     $Lab'(trg(y)) \leftarrow OUT$;
18     **if** $trg(y) \in A$ **then**
19         **foreach** $z \in \overline{R} : src(z) = trg(y)$ **do** $Lab'(z) \leftarrow OUT$;
20     **foreach** $z \in \overline{R} : Lab(z) \in \{BLANK, UNDEC\} \wedge trg(z) \in \{y, src(y)\}$ **do**
21         $Lab'(z) \leftarrow MUST\_OUT$;
22         **if** $\nexists\, w \in \overline{R} : Lab'(w) = BLANK \wedge trg(w) \in \{z, src(z)\}$ **then**
23             $Lab(y) \leftarrow UNDEC$; **goto** line 14;
24     **call** find-admissible-sets($Lab'$);
25     **if** $\exists z \in \overline{R} : Lab(z) \in \{BLANK, UNDEC\} \wedge trg(z) \in \{y, src(y)\}$ **then**
26         $Lab(y) \leftarrow UNDEC$;
27     **else**
28         $Lab \leftarrow Lab'$;
29 **if** $\nexists y \in \overline{R} : Lab(y) = MUST\_OUT$ **then**
30     **foreach** $x \in A$ with $Lab(x) = BLANK$ s.t. $\forall z \in \overline{R} : trg(z) = x\ (Lab(z) = OUT)$ **do**
       $Lab(x) \leftarrow IN$;
31     $\overline{E}_{semi-stable} \leftarrow \overline{E}_{semi-stable} \cup \{Lab\}$;
32 **end procedure**
```

Algorithm 13: Constructing the ideal extension of an AFRA $(A, \overline{R})$.

```
Lab : $(A \cup \overline{R}) \rightarrow \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}$;  $Lab \leftarrow \emptyset$;
foreach $x \in A \cup \overline{R}$ do $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
$\overline{E}_{ideal} : \mathbb{Z} \rightarrow 2^{A \cup \overline{R}}$;  $\overline{E}_{ideal} \leftarrow \emptyset$;
$S \leftarrow \emptyset$;
call find-admissible-sets($Lab$);
foreach $i$ :  $1$ to $|\overline{E}_{ideal}|$ do
    if $\forall x \in \overline{E}_{ideal}(i)$ $(x \notin S)$ then
        report $\overline{E}_{ideal}(i)$ is the ideal extension; exit;

procedure find-admissible-sets($Lab$) begin
while $\exists y \in \overline{R} : Lab(y) = BLANK$ do
    select $y \in \overline{R}$ with $Lab(y) = BLANK$ s.t.
    $\forall z \in \overline{R} : trg(z) \in \{y, src(y)\}$ $Lab(z) \in \{OUT, MUST\_OUT\}$, otherwise select $y \in \overline{R}$ with
    $Lab(y) = BLANK$ such that $\forall z \in \overline{R} : Lab(z) = BLANK$
    $|\{x : src(x) = trg(y) \wedge Lab(x) \neq OUT\}| \geq |\{x : src(x) = trg(z) \wedge Lab(x) \neq OUT\}|$;
    $Lab' \leftarrow Lab$;
    $Lab'(y) \leftarrow IN$;
    $Lab'(src(y)) \leftarrow IN$;
    $Lab'(trg(y)) \leftarrow OUT$;
    if $trg(y) \in A$ then
        foreach $z \in \overline{R} : src(z) = trg(y)$ do
            $Lab'(z) \leftarrow OUT$;
    foreach $z \in \overline{R} : Lab'(z) \in \{BLANK, UNDEC\} \wedge trg(z) \in \{y, src(y)\}$ do
        $Lab'(z) \leftarrow MUST\_OUT$;
        if $\nexists w \in \overline{R} : Lab'(w) = BLANK \wedge trg(w) \in \{z, src(z)\}$ then
            $Lab(y) \leftarrow UNDEC$;
            goto line 10;
    call find-admissible-sets($Lab'$);
    if $\exists z \in \overline{R} : Lab(z) \in \{BLANK, UNDEC\}$ and $trg(z) \in \{y, src(y)\}$ then
        $Lab(y) \leftarrow UNDEC$;
    else
        $Lab \leftarrow Lab'$;
if $\nexists w \in \overline{R} : Lab(w) = MUST\_OUT$ then
    foreach $x \in A$ with $Lab(x) = BLANK$ s.t. $\forall z \in \overline{R} : trg(z) = x$ $(Lab(z) = OUT)$ do
    $Lab(x) \leftarrow IN$;
    $S \leftarrow S \cup \{x \in A \cup \overline{R} \mid Lab(x) = OUT\}$;
    $\overline{E}_{ideal} \leftarrow \overline{E}_{ideal} \cup \{(|\overline{E}_{ideal}| + 1, \{z \mid Lab(z) = IN\})\}$;
end procedure
```

Algorithm 14: Constructing the grounded extension of an AFRA $(A, \overline{R})$.

```
1  $Lab : (A \cup \overline{R}) \rightarrow \{IN, OUT, UNDEC\};\ Lab \leftarrow \emptyset$;
2  foreach $w \in A \cup \overline{R}$ do $Lab \leftarrow Lab \cup \{(w, UNDEC)\}$;
3  while $\exists x \in \overline{R}$ with $Lab(x) = UNDEC$ s.t. $\forall y \in \overline{R} : trg(y) \in \{x, src(x)\}\ (Lab(y) = OUT)$ do
4      foreach $x \in \overline{R}$ with $Lab(x) = UNDEC$ s.t. $\forall y \in \overline{R} : trg(y) \in \{x, src(x)\}\ (Lab(y) = OUT)$
       do
5          $Lab(x) \leftarrow IN$;
6          $Lab(src(x)) \leftarrow IN$;
7          $Lab(trg(x)) \leftarrow OUT$;
8          if $trg(x) \in A$ then
9              foreach $z \in \overline{R} : trg(x) = src(z)$ do
10                 $Lab(z) \leftarrow OUT$;
11 foreach $x \in A$ with $Lab(x) = UNDEC$ s.t. $\forall z \in \overline{R} : trg(z) = x\ (Lab(z) = OUT)$ do
   $Lab(x) \leftarrow IN$;
12 report the grounded extension is $\{w \in A \cup \overline{R} \mid Lab(w) = IN\}$;
```

We compared the algorithms with dynPARTIX, which is an implemented system based on the dynamic programming algorithm of Dvořák, Pichler, and Woltran (2012b). Given an AF, dynPARTIX basically computes a tree decomposition of the AF then the extensions are enumerated based on the tree decomposition. The algorithm used in dynPARTIX is fixed-parameter tractable such that its time complexity depends on the tree width of the given AF while it is linear in the size of the AF (Dvořák et al., 2012b). Since dynPARTIX computes only extensions under preferred, stable and complete semantics, figures 5, 6 & 7 depict respectively the efficiency of algorithms 1, 2 & 3 versus dynPARTIX. In summary, the figures show that these algorithms are likely to be more efficient than dynPARTIX. In running the experiments that are represented by these figures, we set a time limit of 120 seconds for every execution. Out of 1000 runs, dynPARTIX encountered 316 timeouts in enumerating preferred extensions and 827 timeouts in enumerating complete extensions. These timeouts are plotted within the figures as 120 seconds. This explains the steady behavior of dynPARTIX that can be noted, particularly, in figure 7. To see the performance of algorithms 1-7 in contrast to the behavior of algorithms 8-14 we present figures 8-14 respectively. In profiling algorithms 1-7, we reported running times including the time needed to get the corresponding AF of an AFRA. Note that such process (i.e. expressing an AFRA as an AF) runs in polynomial time: at worst quadratic time. The figures 8-14 plot the running times for 1000 instances of AFRA randomly generated with $|A| = 5$ and $\overline{R} = R_1 \cup R_2$ s.t. $R_1 \subseteq A \times A$ and $R_2 \subseteq A \times R_1$. For these instances $|\overline{R}|$ grows from 0 to 100 as the probability, which was used for setting attacks in the random generation, goes over $\{0.01, 0.02, 0.03, ..., 1\}$. Note that instances with $|A| = 5$ should not be considered quite small. For example, a randomly generated AFRA with $|A| = 5$ and $|\overline{R}| = 49$ has a corresponding AF with $|A| = 54$ and $|R| = 190$. Again, we emphasize that the aim of the experiments is to compare the performance of algorithms 1-7 with the performance of algorithms 8-14. We do not mean by the experiments to check the scalability of these algorithms, although an important issue to be examined. Said that, it is not crucial in such evaluation to consider very large frameworks or higher level of recursive attacks. Back to the results of the experiments, the only case that involved exceeding the 120-second time limit occurred in enumerating semi stable extensions. Referring

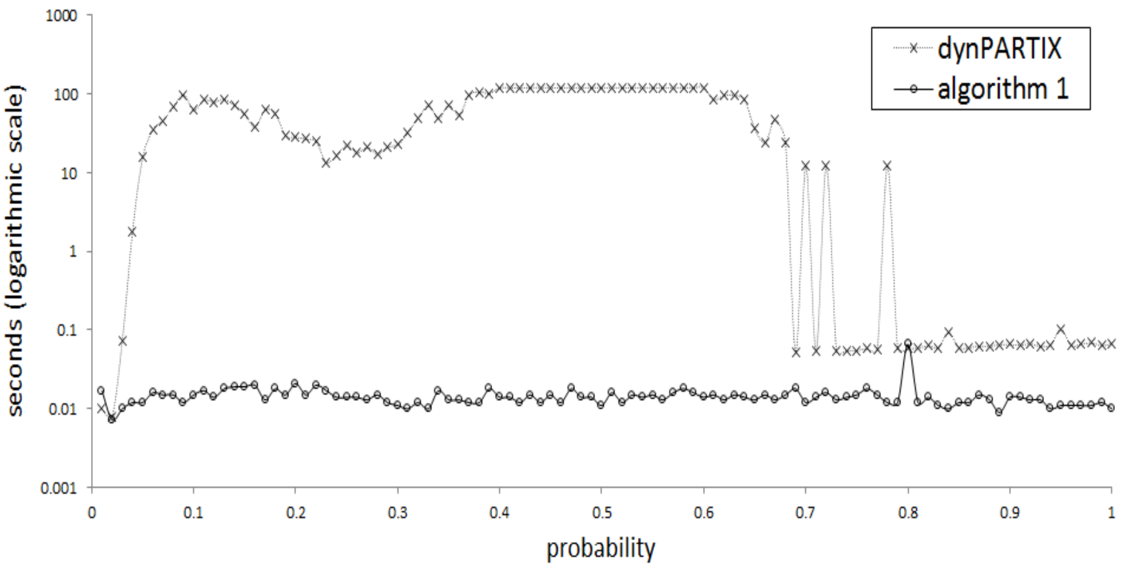

Figure 5: Enumerating all preferred extensions of 1000 instances of AF with $|A|$=40, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p (i.e. the probability that x attacks y for any $x, y \in A$).

to figure 12 we note that algorithm 5 (resp. 12) encountered 68 (resp. 66) timeouts. The bottom line conclusion of these figures is: enumerating extensions of an AFRA by labeling attacks together with arguments seems to be as efficient as enumerating the extensions of the corresponding AF via labeling arguments alone.

6. Discussion and Conclusion

We started the paper by refining implemented algorithms[1] for enumerating all extensions of Dung's argumentation frameworks (AFs) under a number of argumentation semantics: preferred, stable, complete, stage, semi stable, ideal and grounded. Algorithms for all these semantics, except stage and grounded semantics, share a similar core structure: they all basically build admissible sets in order to construct extensions. In the case of stage semantics, the algorithm actually constructs conflict free sets for the purpose of listing stage extensions, and hence, the algorithm applies a slightly different approach to expanding the search tree as we elaborated earlier. Concerning the grounded semantics, the presented algorithm builds the grounded extension in polynomial time. Furthermore, we explored the practical efficiency of these algorithms by profiling their performance running on a wide spectrum of AF instances: from sparse instances to dense ones. In essence these algorithms construct extensions by using a total function that maps arguments *solely* to a set of labels reflecting different states as we illustrated in the paper. Then, we generalized these algorithms by using a total mapping that labels *attacks* together with arguments. We implemented the generalized algorithms to enumerate extensions of an AFRA, which is an AF-extended model that

1. C++ implementations can be found at http://sourceforge.net/projects/argtools/files/

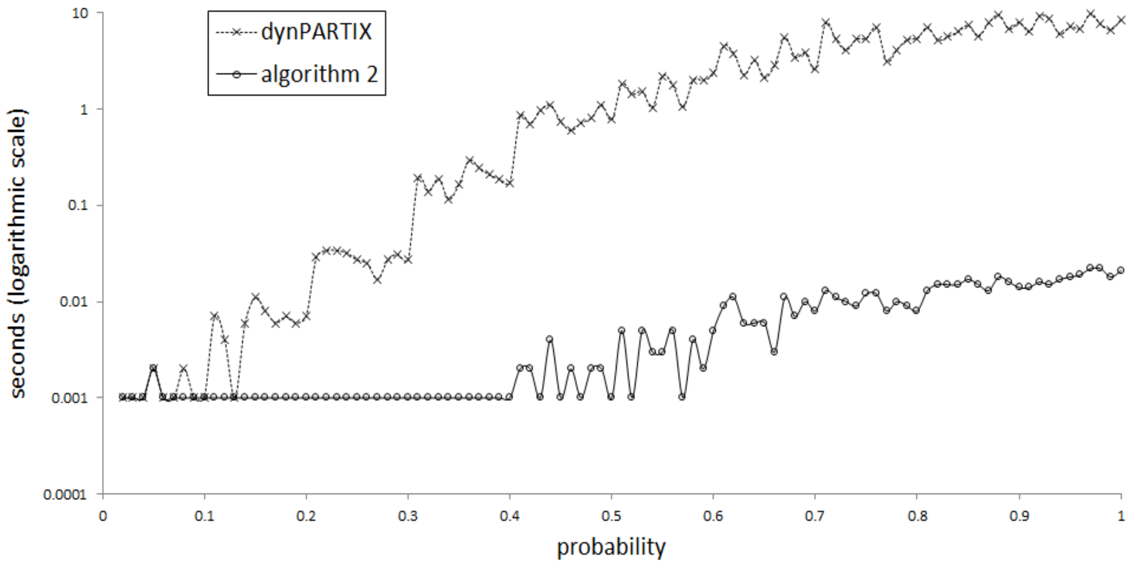

Figure 6: Enumerating all stable extensions of 1000 instances of AF with $|A|$=60, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

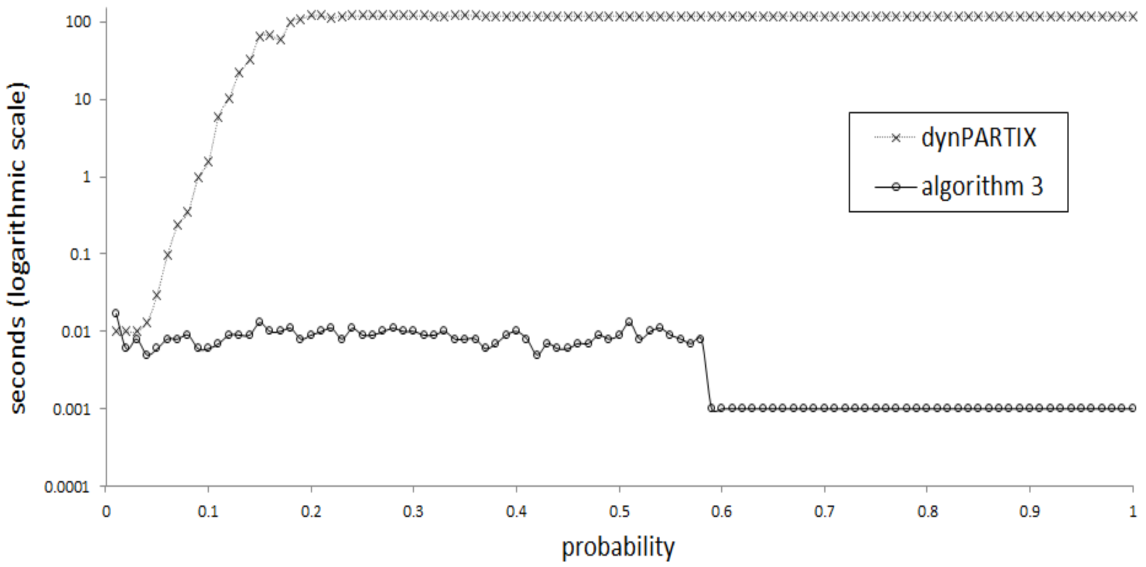

Figure 7: Enumerating all complete extensions of 1000 instances of AF with $|A|$=30, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

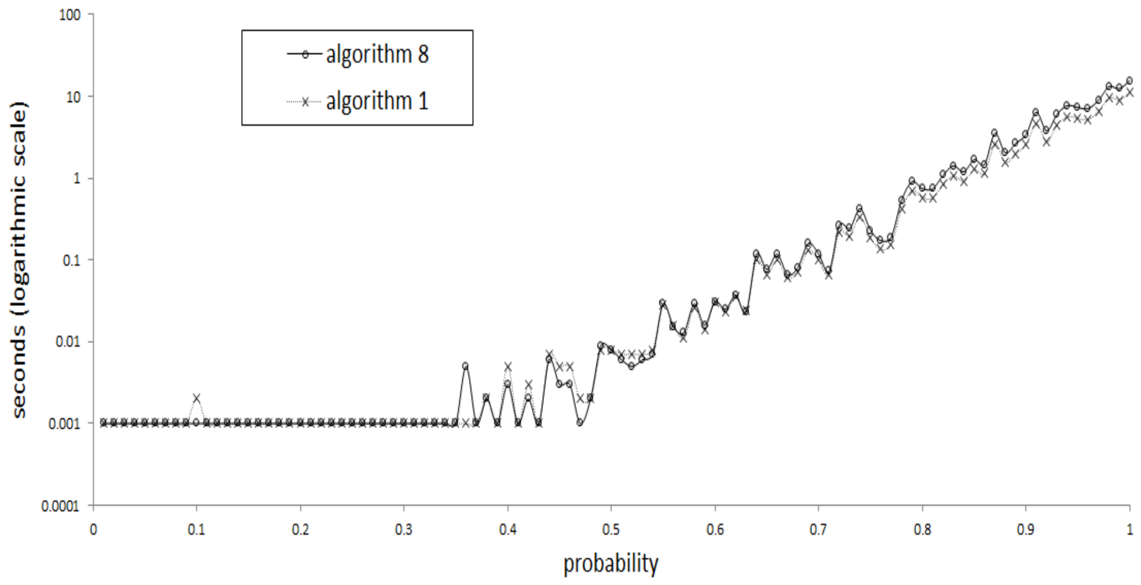

Figure 8: Enumerating all preferred extensions of 1000 instances of AFRA, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

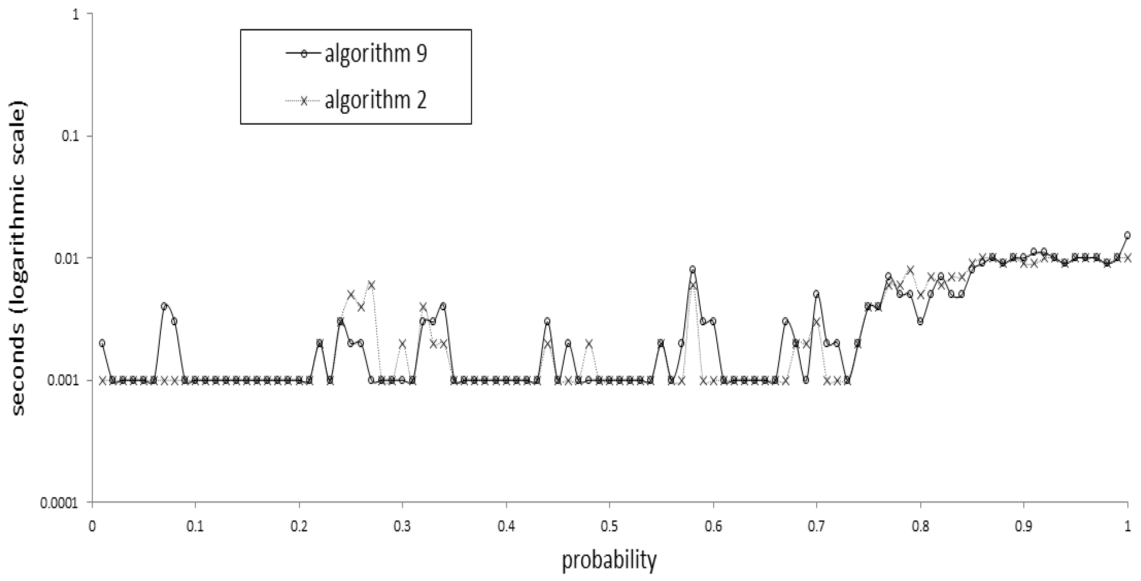

Figure 9: Enumerating all stable extensions of 1000 instances of AFRA, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

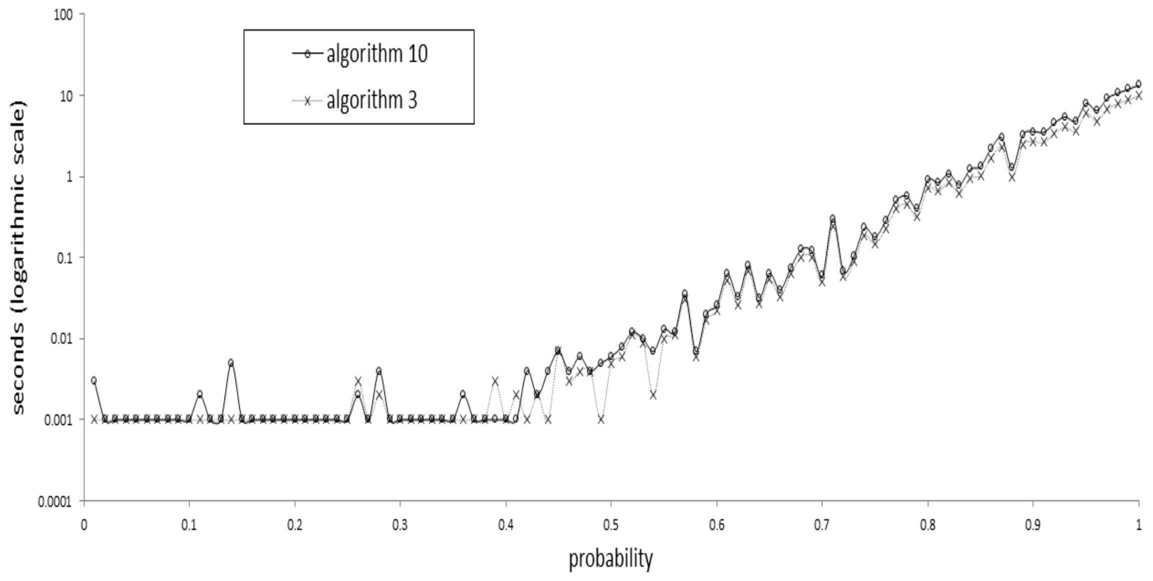

Figure 10: Enumerating all complete extensions of 1000 instances of AFRA, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

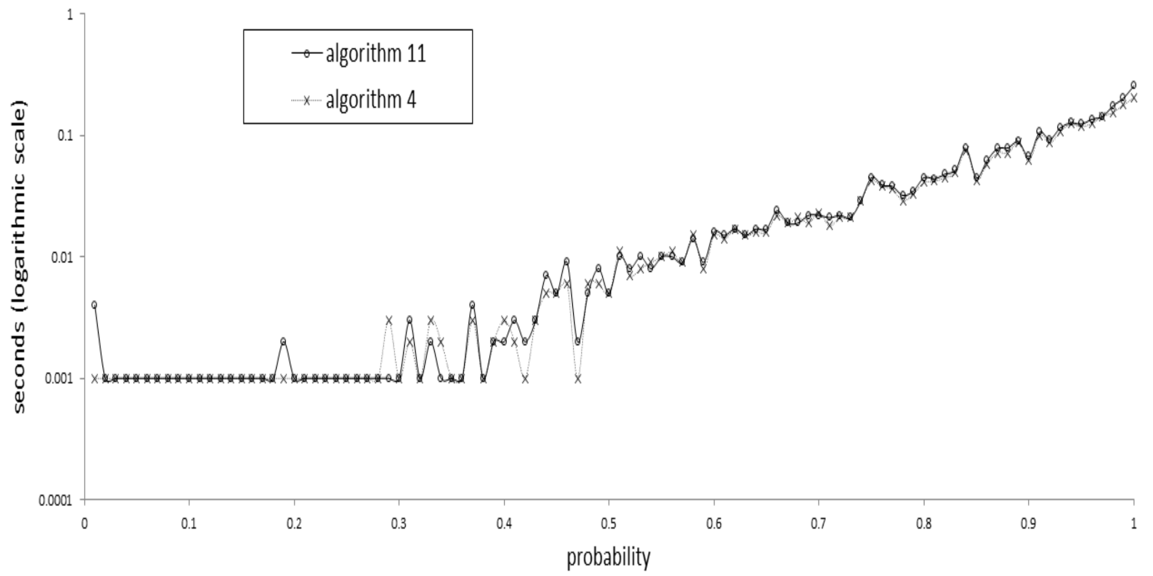

Figure 11: Listing all stage extensions of 1000 instances of AFRA, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

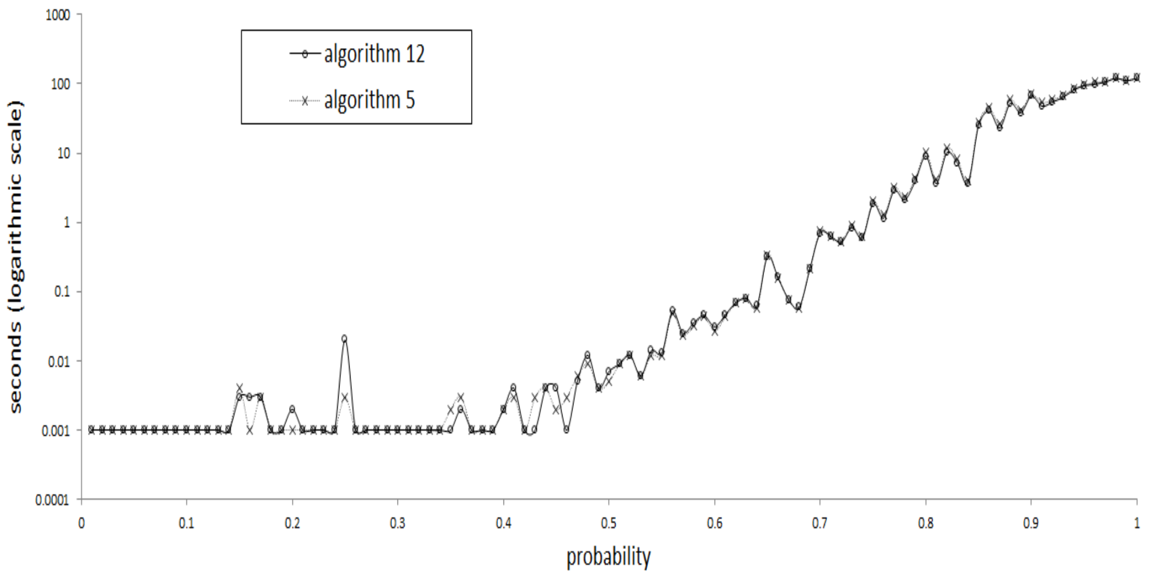

Figure 12: Listing all semi stable extensions of 1000 instances of AFRA, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

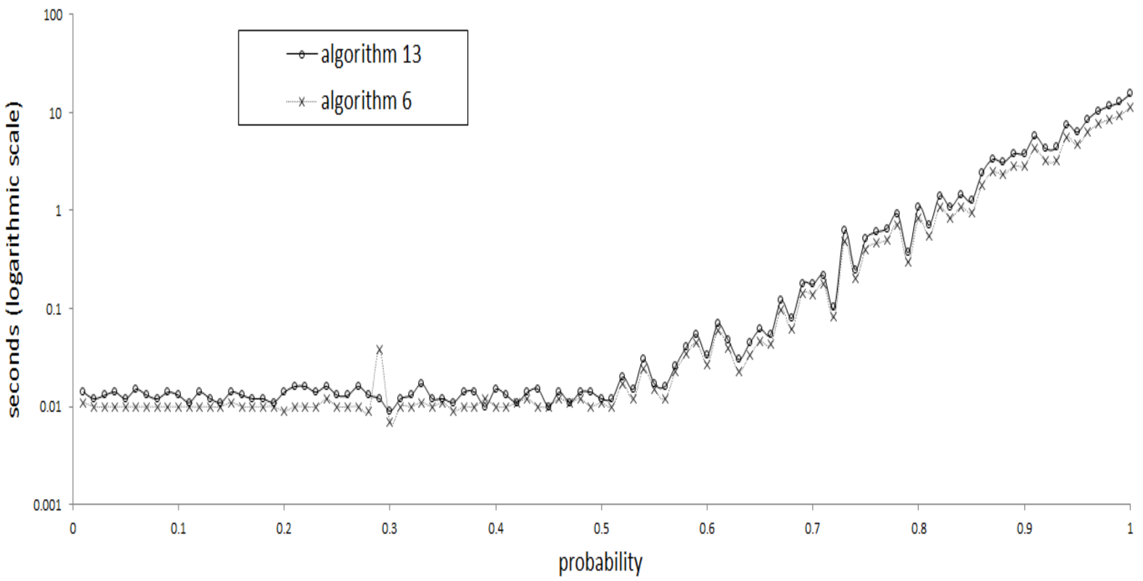

Figure 13: Constructing the ideal extension of 1000 instances of AFRA, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

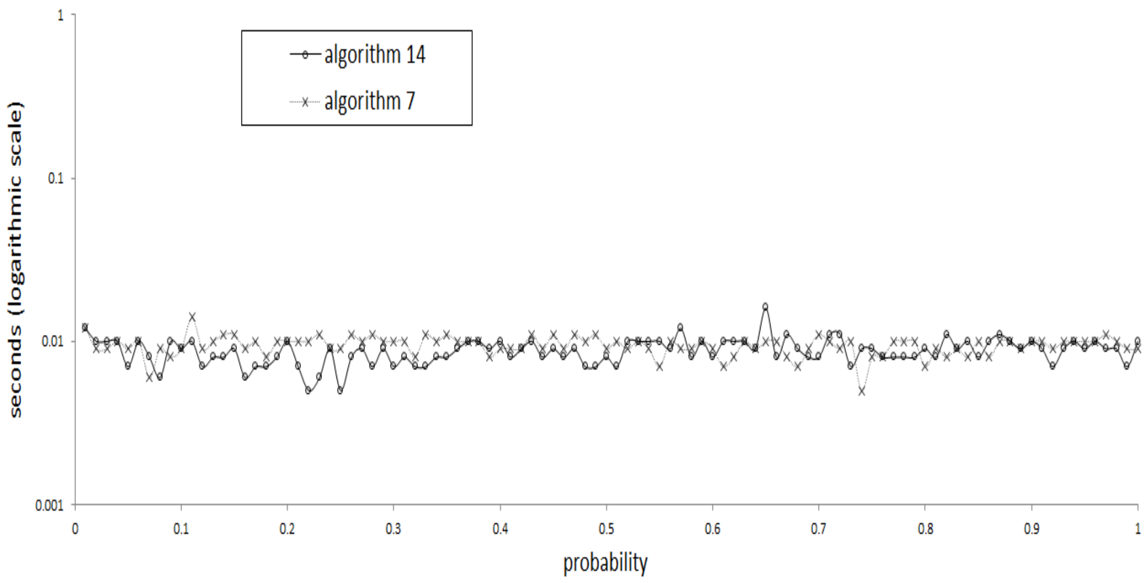

Figure 14: Constructing the grounded extension of 1000 instances of AFRA, for each $p \in \{0.01, 0.02, 0.03, ..., 1\}$ we tracked the average elapsed time for 10 instances generated randomly with a probability p.

allows attacks on attacks. In other words, we offered a unified approach to enumerating extensions of any AF/AFRA, given the fact that an AF is a special case of AFRA (Baroni et al., 2011b). On the other hand, we showed how labeling attacks alongside arguments can be potentially used as a basis for enumerating extensions of related formalisms that allow attacks on attacks (e.g. Modgil, 2009b; Gabbay, 2009); nonetheless this is to be confirmed by further research. In fact, extensions of an instance of such formalisms can be listed by working on the corresponding AF. However, we demonstrated how to enumerate extensions of an AFRA by applying labeling directly on its native form without compromising the running time efficiency. We omitted the soundness/completeness proof of the presented algorithms since it follows immediately from the proof of the algorithm of Nofal, Atkinson and Dunne (2014) for preferred semantics. The algorithms presented in this paper do not handle frameworks with self-attacking arguments perfectly. However, the algorithms can be easily modified such that the initial label for any self-attacking argument is UNDEC instead of BLANK. For instance, the only change necessary to be made in algorithm 1 is to modify line 2 as follows

```
foreach $x \in A$ do
    if $(x,x) \in R$ then $Lab \leftarrow Lab \cup \{(x, UNDEC)\}$;
    else $Lab \leftarrow Lab \cup \{(x, BLANK)\}$;
```

In general, the UNDEC label (instead of the BLANK label) should be the default label for any argument/attack that can not be in any extension, like self-attacking arguments and their outgoing attacks because simply such arguments present conflict by themselves. Recall that only BLANK arguments/attacks are to be tried with the IN label.

For future work, we plan to study additional options for argument selection. Also, we intend to evaluate further strategies for pruning the search space.

We discuss now related work. The existing algorithms of Doutre and Mengin (2001) and Modgil and Caminada (2009) for listing preferred extensions can also be re-engineered towards enumerating extensions under other argumentation semantics. For example, the papers of Caminada (2007, 2010) presented algorithms for enumerating semi stable, respectively stage, extensions building up on the algorithm of Modgil and Caminada (2009). However, the algorithms of the present paper are based on the algorithm of Nofal, Atkinson and Dunne (2014) for enumerating preferred extensions, which is likely to be more efficient than existing algorithms (Nofal et al., 2014). We give now some examples of related work on labeling-based semantics. The theory of Caminada and Gabbay (2009) defined argumentation semantics by using a total mapping $\lambda : A \rightarrow \{IN, OUT, UNDEC\}$ such that, broadly speaking, the IN labeled arguments correspond to an extension, say S, while the OUT labeled arguments correspond to S^+ and the UNDEC labeled arguments correspond to $A \setminus (S \cup S^+)$. It is not hard to see the connection between our algorithms and the theory of Caminada and Gabbay (2009). For example, in algorithm 1 we capture a preferred extension when all arguments are mapped to one of those labels: IN, OUT and UNDEC. Listing other works that present labeling-based semantics, the paper of Modgil (2009a) defined labeling-based semantics for the extended AFs of Modgil (2009b) while Villata, Boella, and van der Torre (2011) described argumentation semantics in terms of attacks and arguments. Also, the work of Gabbay (2009) set argumentation semantics for the AF-extended model of Barringer, Gabbay, and Woods (2005) that among other features allow attacks on attacks. On the topic of extension computation in general, the study of Li, Oren, and Norman (2012) examined approximation versus exact computations, whereas the experiments of Baumann, Brewka, and Wong (2012), Liao, Lei, and Dai (2013) evaluated the effect of splitting an AF on the computation of preferred extensions. The work of Dondio (2013) studied, under the grounded semantics, how the acceptance status of an argument varies in all the subgraphs of the given AF. Computational complexity of argumentation semantics are widely studied (see e.g. Dimopoulos, Nebel, & Toni, 2000; Dunne, 2007, 2009; Ordyniak & Szeider, 2011). Another line of research concerns encoding computational problems of AFs into other formalisms and then solving them by using a respective solver (e.g. Besnard & Doutre, 2004; Nieves, Cortes, & Osorio, 2008; Egly, Gaggl, & Woltran, 2010; Amgoud & Devred, 2011; Dvořák, Järvisalo, Wallner, & Woltran, 2012a; Cerutti, Dunne, Giacomin, & Vallati, 2013; Charwat, Dvořák, Gaggl, Wallner, & Woltran, 2013), such approaches are called reduction based methods. We stress that the focus of this paper was on algorithmic based implementations of argumentation semantics.

Acknowledgments

We thank the anonymous reviewers for the comments that improved the presentation of this work.

References

Amgoud, L., & Devred, C. (2011). Argumentation frameworks as constraint satisfaction problems. In Benferhat, S., & Grant, J. (Eds.), *SUM*, Vol. 6929 of *Lecture Notes in Computer Science*, pp. 110–122. Springer.

Baroni, P., Caminada, M., & Giacomin, M. (2011a). An introduction to argumentation semantics. *The Knowledge Engineering Review*, *26*(4), 365–410.

Baroni, P., Cerutti, F., Giacomin, M., & Guida, G. (2011b). Argumentation framework with recursive attacks. *International Journal of Approximate Reasoning*, *52*(1), 19–37.

Barringer, H., Gabbay, D., & Woods, J. (2005). Temporal dynamics of support and attack networks: From argumentation to zoology. In Hutter, D., & Stephan, W. (Eds.), *Mechanizing Mathematical Reasoning*, Vol. 2605 of *Lecture Notes in Computer Science*, pp. 59–98. Springer.

Baumann, R., Brewka, G., & Wong, R. (2012). Splitting argumentation frameworks: An empirical evaluation. In Modgil, S., Oren, N., & Toni, F. (Eds.), *First International Workshop on Theory and Applications of Formal Argumentation 2011*, Vol. 7132 of *Lecture Notes in Computer Science*, pp. 17–31. Springer.

Bench-Capon, T., & Dunne, P. (2007). Argumentation in artificial intelligence. *Artificial Intelligence*, *171*, 619–641.

Besnard, P., & Doutre, S. (2004). Checking the acceptability of a set of arguments. In Delgrande, J., & Schaub, T. (Eds.), *NMR*, pp. 59–64.

Besnard, P., & Hunter, A. (2008). *Elements of Argumentation*. MIT press.

Caminada, M. (2007). An algorithm for computing semi-stable semantics. In Mellouli, K. (Ed.), *ECSQARU*, Vol. 4724 of *Lecture Notes in Computer Science*, pp. 222–234. Springer.

Caminada, M. (2010). An algorithm for stage semantics. In Baroni, P., Cerutti, F., Giacomin, M., & Simari, G. (Eds.), *COMMA*, Vol. 216 of *Frontiers in Artificial Intelligence and Applications*, pp. 147–158. IOS Press.

Caminada, M., Carnielli, W., & Dunne, P. (2012). Semi-stable semantics. *J. Log. Comput.*, *22*(5), 1207–1254.

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

Cerutti, F., Dunne, P., Giacomin, M., & Vallati, M. (2013). A sat-based approach for computing extensions in abstract argumentation. In *TAFA, Second International Workshop on Theory and Applications of Formal Argumentation*.

Charwat, G., Dvořák, W., Gaggl, S., Wallner, J., & Woltran, S. (2013). Implementing abstract argumentation - a survey. Tech. rep. DBAI-TR-2013-82, Technische Universität Wien, Database and Artificial Intelligence Group.

Dimopoulos, Y., Magirou, V., & Papadimitriou, C. (1997). On kernels, defaults and even graphs. *Annals of Mathematics and Artificial Intelligence*, *20*, 1–12.

Dimopoulos, Y., Nebel, B., & Toni, F. (2000). Finding admissible and preferred arguments can be very hard. In Cohn, A., Giunchiglia, F., & Selman, B. (Eds.), *KR*, pp. 53–61. Morgan Kaufmann.

Dondio, P. (2013). Computing the grounded semantics in all the subgraphs of an argumentation framework: an empirical evaluation. In *CLIMA, XIV Workshop on Computational Logic in Multi-Agent Systems*.

Doutre, S., & Mengin, J. (2001). Preferred extensions of argumentation frameworks: Query answering and computation. In Goré, R., Leitsch, A., & Nipkow, T. (Eds.), *IJCAR*, Vol. 2083 of *Lecture Notes in Computer Science*, pp. 272–288. Springer.

Dung, P. (1995). On the acceptability of arguments and its fundamental role in non monotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, *77*(2), 321–357.

Dung, P., Mancarella, P., & Toni, F. (2007). Computing ideal skeptical argumentation. *Artificial Intelligence*, *171*(10-15), 642–674.

Dunne, P. (2007). Computational properties of argument systems satisfying graph-theoretic constraints. *Artificial Intelligence*, *171*, 701–729.

Dunne, P. (2009). The computational complexity of ideal semantics. *Artificial Intelligence*, *173*(18), 1559 – 1591.

Dvořák, W., Järvisalo, M., Wallner, J. P., & Woltran, S. (2012a). Complexity-sensitive decision procedures for abstract argumentation. In Brewka, G., Eiter, T., & McIlraith, S. (Eds.), *KR*. AAAI Press.

Dvořák, W., Pichler, R., & Woltran, S. (2012b). Towards fixed-parameter tractable algorithms for abstract argumentation. *Artificial Intelligence*, *186*, 1–37.

Egly, U., Gaggl, S., & Woltran, S. (2010). Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, *1*(2), 147–177.

Gabbay, D. (2009). Semantics for higher level attacks in extended argumentation frames part 1: Overview. *Studia Logica*, *93*(2-3), 357–381.

Li, H., Oren, N., & Norman, T. (2012). Probabilistic argumentation frameworks. In Modgil, S., Oren, N., & Toni, F. (Eds.), *First International Workshop on Theory and Applications of Formal Argumentation 2011*, Vol. 7132 of *Lecture Notes in Computer Science*, pp. 1–16. Springer.

Liao, B., Lei, L., & Dai, J. (2013). Computing preferred labellings by exploiting sccs and most sceptically rejected arguments. In *TAFA, Second International Workshop on Theory and Applications of Formal Argumentation*.

Modgil, S. (2009a). Labellings and games for extended argumentation frameworks. In Boutilier, C. (Ed.), *IJCAI*, pp. 873–878.

Modgil, S. (2009b). Reasoning about preferences in argumentation frameworks. *Artificial Intelligence*, *173*, 901–934.

Modgil, S., & Caminada, M. (2009). Proof theories and algorithms for abstract argumentation frameworks. In Rahwan, I., & Simari, G. (Eds.), *Argumentation in Artificial Intelligence*, pp. 105–129. Springer.

Modgil, S., Toni, F., Bex, F., Bratko, I., Chesñevar, C., Dvořák, W., Falappa, M., Fan, X., Gaggl, S., García, A., González, M., Gordon, T., Leite, J., Možina, M., Reed, C., Simari, G., Szeider, S., Torroni, P., & Woltran, S. (2013). The added value of argumentation. In Ossowski, S. (Ed.), *Agreement Technologies*, Vol. 8 of *Law, Governance and Technology Series*, pp. 357–403. Springer Netherlands.

Nieves, J., Cortes, U., & Osorio, M. (2008). Preferred extensions as stable models. *Theory and Practice of Logic Programming*, *8*(4), 527–543.

Nofal, S., Atkinson, K., & Dunne, P. (2014). Algorithms for decision problems in argument systems under preferred semantics. *Artif. Intell.*, *207*, 23–51.

Ordyniak, S., & Szeider, S. (2011). Augmenting tractable fragments of abstract argumentation. In Walsh, T. (Ed.), *Proceedings of the 22nd International Joint Conference on Artificial Intelligence IJCAI 2011*, pp. 1033–1038.

Rahwan, I., & Simari, G. (2009). *Argumentation in Artificial Intelligence*. Springer.

Verheij, B. (1996). Two approaches to dialectical argumentation: admissible sets and argumentation stages. In *Proceedings of the Eighth Dutch Conference on AI*, pp. 357–368.

Villata, S., Boella, G., & van der Torre, L. (2011). Attack semantics for abstract argumentation. In Walsh, T. (Ed.), *Proceedings of the 22nd International Joint Conference on Artificial Intelligence IJCAI 2011*, pp. 406–413.