# On Deciding Admissibility in Abstract Argumentation Frameworks

Samer Nofal[1], Katie Atkinson[2] and Paul E. Dunne[2]
[1]*Department of Computer Science, German Jordanian University, Jordan*
[2]*Department of Computer Science, University of Liverpool, U.K.*

Keywords: Argument-based Knowledge Base, Argument-based Reasoning, Computational Argumentation, Algorithms.

Abstract: In the context of abstract argumentation frameworks, the admissibility problem is about deciding whether a given argument (i.e. piece of knowledge) is admissible in a conflicting knowledge base. In this paper we present an enhanced backtracking-based algorithm for solving the admissibility problem. The algorithm performs successfully when applied to a wide range of benchmark abstract argumentation frameworks and when compared to the state-of-the-art algorithm.

## 1 INTRODUCTION

*Abstract argumentation frameworks* (AFs), introduced by (Dung, 1995), are a major topic within the field of knowledge representation and automated reasoning, see for example the reviews of (Modgil et al., 2013; Charwat et al., 2015; Simari and Rahwan, 2009; Atkinson et al., 2017; Baroni et al., 2011; Modgil and Caminada, 2009; Caminada and Gabbay, 2009). Particularly, AFs have been demonstrated as a powerful mechanism for decision-support systems (Heras et al., 2013; Hunter and Williams, 2012; Bench-Capon et al., 2015; Tamani et al., 2015), and for handling inconsistency in knowledge bases (Martinez and Hunter, 2009; Hecham et al., 2017; Amgoud and Cayrol, 2002; Croitoru and Vesic, 2013; Amgoud and Vesic, 2010).

An *abstract argumentation framework* is a pair $(A,R)$ where $A$ is a set of *abstract arguments* (i.e. pieces of knowledge) and $R \subseteq A \times A$ is the *attack* relation (representing conflicting knowledge). We say $x$ *attacks* $y$ (or $y$ is *attacked* by $x$) whenever $(x,y) \in R$. For a given set of arguments $B \subseteq A$, $B^-$ (respectively $B^+$) denotes the set of arguments that attack (respectively are attacked by) the arguments of $B$. Let $S \subseteq A$ be a set of arguments, then $S$ is *admissible* if and only if $S^- \subseteq S^+$ and $S^+ \cap S = \emptyset$. Let $x \in A$ be an argument, then $x$ is *admissible* if and only if $x$ is contained in an admissible set.

The problem of admissibility is to decide whether an argument, in a given AF, is admissible or not. Thus, the problem can be naturally solved by finding an admissible set containing the argument in question. For example, assume we desire to decide the admissibility of argument $b$ in the framework of figure 1, then we

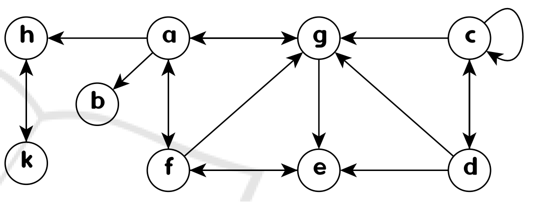


Figure 1: An example argumentation framework $AF_1$.

find $b$ admissible due to the admissibility of $\{b, f\}$.

It is known that the problem of admissibility is NP-complete (Dvorák and Dunne, 2017; Dunne, 2007). However, as noted in the survey of (Charwat et al., 2015), there are two approaches to the admissibility problem: *direct* and *reduction-based*. In the latter approach one might put the admissibility problem into a different form and then apply an off-the-shelf solver to decide admissibility for a given problem instance, see (Thimm and Villata, 2017) for reviews.

On the other hand, direct approaches to the admissibility problem are *ad hoc* algorithms. In this paper we present a new *ad hoc* algorithm for the admissibility problem. In the literature one can see a number of works that presented *ad hoc* algorithms for solving the admissibility problem. The work of (Doutre and Mengin, 2001) introduced an algorithm for the admissibility problem. Subsequently, the algorithm of (Doutre and Mengin, 2001) was re-presented in the article of (Cayrol et al., 2003). In fact, the algorithm of (Doutre and Mengin, 2001) is a depth-first search procedure that looks for an admissible set containing the argument in question. Later, (Verheij, 2007) presented a breadth-first search procedure for solving the admissibility problem. Afterwards, (Thang et al., 2009) introduced a unified breadth-first search procedure for deciding admissibility as well as for solving

Nofal, S., Atkinson, K. and Dunne, P.
On Deciding Admissibility in Abstract Argumentation Frameworks.
DOI: 10.5220/0008064300670075
In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019)*, pages 67-75
ISBN: 978-989-758-382-7

other decision problems related to AFs. The work of (Nofal et al., 2014) presented a new depth-first search algorithm that is likely faster than the previous algorithms of (Doutre and Mengin, 2001; Verheij, 2007; Thang et al., 2009), see (Nofal et al., 2014) for a comprehensive evaluation of the aforementioned algorithms. Recently, by a "look-ahead" mechanism the work of (Nofal et al., 2016) improved on the algorithm of (Nofal et al., 2014). Differently, (Dvořák et al., 2012) proposed a dynamic programming approach to the admissibility problem. The focus of (Dvořák et al., 2012) was to show the role of "fixed-parameter" tractable methods in the context of AFs. Lastly, we note that another line of research is more focused on building procedures for handling dynamic changes in AFs, see for example (Liao et al., 2011; Doutre and Mailly, 2018; Alfano et al., 2017).

In this paper we present new enhancements that improve over the state-of-the-art *backtracking* algorithm presented by (Nofal et al., 2016). Therefore, in section 2 we recall the state-of-the-art algorithm of (Nofal et al., 2016). In section 3 we present our new algorithm. Then, we verify the running-time efficiency of the new algorithm in section 4. In section 5 we present a concrete scenario where deciding admissibility being vital for argument-based legal reasoning. In section 6 we conclude the paper.

## 2 THE EXISTING STATE-OF-THE-ART ALGORITHM

In this section we recall the algorithm of (Nofal et al., 2016) for deciding admissibility. We note that it would be inefficient if one decided the admissibility of some argument in a given AF by generating admissible sets one after another until a set, containing the argument in question, is found. Obviously, following this approach will result in a considerable wasted time in listing irrelevant sets that do not contain the argument in question. Moreover, using this approach one might waste time in computing admissible sets that are larger than needed. Take the framework of figure 1 and the problem of deciding the admissibility of argument $b$. Focusing on two admissible sets: $\{a,k\}$ and $\{f,b,d,h\}$, we note that the former set is irrelevant because it does not contain the query argument (i.e. $b$), whereas the latter set is larger than needed because the admissible set $\{f,b\}$ is sufficient for proving the admissibility of $b$. Hence, an efficient procedure for the admissibility problem would start with a

Algorithm 1: Algorithm 10 from Nofal et al (Nofal et al., 2016).

```
requires: an AF $H = (A,R)$ and a query argument
        $s \in A$.
ensures : a decision whether $s$ is admissible or not.
1  Function isAdmissible($label$)
2      propagate($label$);
3      if $label$ is admissible then return true;
4      if $label$ is hopeless then return false;
5      $label' \leftarrow label$;
6      select some $x \in \{y \mid label(y) = mustOut\}^-$
         with $label(x) = blank$;
7      in-trans($label', x$);
8      if isAdmissible($label'$)=true then return true;
9      und-trans($label, x$);
10     if isAdmissible($label$)=true then return true
         else return false;
11 Function main()
12     if $(s,s) \in R$ then report $s$ inadmissibile and
         exit;
13     $label : A \rightarrow \{in, out, und, mustOut, blank\}$;
14     $label \leftarrow \emptyset$;
15     foreach $x \in A$ do
16         $label \leftarrow label \cup \{(x, blank)\}$;
17         if $(x,x) \in R$ then $label(x) \leftarrow und$;
18     in-trans($label, s$);
19     if isAdmissible($label$)=true then report $s$
         admissible else report $s$ inadmissible;
```

one-argument set, containing the argument in question, and then incrementally try to include in the under construction set relevant arguments that are necessary to establish the admissibility of the argument in question. In particular, we may decide the admissibility of $b$ in AF$_1$ as follows:

1. We start with $S = \{b\}$, $S^- = \{a\}$, and $S^+ = \emptyset$. Since $S^- \nsubseteq S^+$, we expand $S$ trying to satisfy this admissibility condition: $S^- \subseteq S^+$.
2. Then, we include $f$ to get $S = \{f,b\}$, $S^- = \{a,e\}$, and $S^+ = \{a,g,e\}$. Now, $S$ is admissible since $S^- \subseteq S^+$ and $S^+ \cap S = \emptyset$.

After this introduction we present algorithm 1, which is algorithm 10 (with minor changes in the presentation) from (Nofal et al., 2016) for deciding admissibility. If we run the algorithm on a given AF $H = (A,R)$ with a query argument $s$ then the algorithm decides whether $s$ is admissible or not. We now specify the actions and structures of the algorithm. Starting at line 12 in the algorithm, if the query argument $s$ is self-attacking, then we conclude with $s$ being inadmissible. At line 13 we create a total function $label$ that maps every argument in $A$ to a label in $\{in, out, und, mustOut, blank\}$ according to the following rules.

**Remark 1** (Mapping Arguments). *Let $(A,R)$ be an* AF*, $label : A \rightarrow \{in,\ out,\ mustOut,\ und,\ blank\}$ be a* *total mapping, $x \in A$ be an argument with $label(x) = blank$ and $S \subseteq A$ be a set of arguments, then x might be re-mapped with respect to S as follows:*

- *If $x \in S$ then $label(x) \leftarrow in$.*
- *If $x \in S^+$ then $label(x) \leftarrow out$.*
- *If $x \in S^- \setminus S^+$ then $label(x) \leftarrow mustOut$.*
- *If $x \notin S \cup S^- \cup S^+$ then $label(x) \leftarrow und$.*

At lines 14-17 in the algorithm, we initialize *label* such that all arguments are mapped to *blank* except self-attacking arguments, which are mapped to *und*. Note that if a set of arguments, say *S*, contains some self-attacking argument, then *S* will violate this admissibility condition: $S^+ \cap S = \emptyset$.

Now, we come to the *in transition* routine (*in-trans* for short), see lines 7 & 18 in the algorithm. By applying the *in-trans* routine we re-map an argument to *in* and, subsequently re-map the neighbor arguments as described in the following definition.

**Definition 1** (In-trans)**.** *Let $(A,R)$ be an* AF, *$label : A \rightarrow \{in, out, mustOut, und, blank\}$ be a total mapping and $x \in A$ be an argument with $label(x) = blank$, then in-trans($label$,x) is defined by the following set of actions:*

1. *$Label(x) \leftarrow in$.*
2. *For each $y \in \{x\}^+$ do $label(y) \leftarrow out$.*
3. *For each $y \in \{x\}^-$ with $label(y) \neq out$ do $label(y) \leftarrow mustOut$.*

At line 9 in the algorithm, we re-map an argument to *und* by an *undecided transition* (*und-trans* for short) as defined below.

**Definition 2** (Und-trans)**.** *Let $(A,R)$ be an* AF, *$label : A \rightarrow \{in, out, mustOut, und, blank\}$ be a total mapping and $x \in A$ be an argument with $label(x) = blank$, then we define und-trans($label$,x) by the action: $label(x) \leftarrow und$.*

Now, we define *propagate*$(label)$, see line 2 in the algorithm. By invoking *propagate*$(label)$ we apply the *in-trans* routine on some arguments as we describe in the next definition.

**Definition 3** (Mappings Propagation)**.** *Let $(A,R)$ be an* AF, *$label : A \rightarrow \{in, out, mustOut, und, blank\}$, then propagate($label$) is defined by the following actions:*

1. *If there is no x with $label(x) = blank$ such that for all $y \in \{x\}^-$ $label(y) \in \{out, mustOut\}$ then halt.*
2. *Select some x with $label(x) = blank$ such that for all $y \in \{x\}^-$ $label(y) \in \{out, mustOut\}$.*
3. *In-trans($label$,x).*
4. *Go to step 1.*

Referring to line 3 in the algorithm, we describe *admissible mappings* that correspond to admissible sets.

**Definition 4** (Admissible Mappings)**.** *Let $(A,R)$ be an* AF, *$label : A \rightarrow \{in, out, mustOut, und, blank\}$ be a total mapping, then label is admissible if and only if for all $x \in A$ $label(x) \neq mustOut$.*

On the other hand, we define *hopeless mappings* that correspond to inadmissible sets, see line 4 in the algorithm.

**Definition 5** (Hopeless Mappings)**.** *Let $(A,R)$ be an* AF, *$label : A \rightarrow \{in, out, mustOut, und, blank\}$ be a total mapping, then label is hopeless if and only if there is $x \in A$ with $label(x) = mustOut$ such that for all $y \in \{x\}^-$ $label(y) \in \{out, mustOut, und\}$.*

At this stage, we are ready to present a progression of the algorithm in deciding the admissibility of *b* in the framework of figure 1:

1. at lines 15-17 in the algorithm, we initialize *label* with $\{(a,blank), (b,blank), (c,und), (d,blank), (e,blank), (f,blank), (g,blank), (h,blank), (k,blank)\}$.
2. at line 18, we apply *in-trans*$(label,b)$, and so, *label* is now equal to $\{(a,mustOut), (b,in), (c,und), (d,blank), (e,blank), (f,blank), (g,blank), (h,blank), (k,blank)\}$.
3. at line 19, we invoke isAdmissible($label$). The actions of this routine are:
   3.1. at line 2, we apply propagate($label$). As a result, *label* remains unchanged, equal to $\{(a,mustOut), (b,in), (c,und), (d,blank), (e,blank), (f,blank), (g,blank), (h,blank), (k,blank)\}$.
   3.2. at lines 3 & 4, we note that *label* is not admissible nor hopeless.
   3.3. at line 5, we copy *label* into *label*$'$. Thus, *label*$'$ is now equal to $\{(a,mustOut), (b,in), (c,und), (d,blank), (e,blank), (f,blank), (g,blank), (h,blank), (k,blank)\}$.
   3.4. at line 6, we select *g* from $\{g, f\}$.
   3.5. at line 7, we apply *in-trans*$(label', g)$, and so, *label*$'$ is changed to $\{(a,out), (b,in), (c,mustOut), (d,mustOut), (e,out), (f,mustOut), (g,in), (h,blank), (k,blank)\}$.
   3.6. at line 8, we call isAdmissible($label'$) to take the actions:
      3.6.1. at line 2, we invoke propagate($label'$). In effect, *label*$'$ remains unchanged, equal to $\{(a,out), (b,in), (c,mustOut), (d,mustOut), (e,out), (f,mustOut), (g,in), (h,blank), (k,blank)\}$.

3.6.2. at line 4, we find that $label'$ is hopeless, and so we return false.

3.7. at line 9, we apply und-trans($label, g$) to get $label = \{(a, mustOut), (b, in), (c, und), (d, blank), (e, blank), (f, blank), (g, und), (h, blank), (k, blank)\}$.

3.8. at line 10, we call isAdmissible($label$), and so, the next actions are:

3.8.1. at line 2, we call propagate($label$). Thus, $label$ remains unchanged, equal to $\{(a, mustOut), (b, in), (c, und), (d, blank), (e, blank), (f, blank), (g, und), (h, blank), (k, blank)\}$.

3.8.2. at line 5, we copy $label$ into $label'$. Hence, $label'$ is now equal to $\{(a, mustOut), (b, in), (c, und), (d, blank), (e, blank), (f, blank), (g, und), (h, blank), (k, blank)\}$.

3.8.3. at line 6, the only choice is argument $f$.

3.8.4. at line 7, we call *in-trans*($label', f$). Now $label'$ becomes equal to $\{(a, out), (b, in), (c, und), (d, blank), (e, out), (f, in), (g, out), (h, blank), (k, blank)\}$.

3.8.5. at line 8, we call isAdmissible($label'$) to apply the actions:

3.8.5.1. at line 2, we call propagate($label'$). Note that $label'$ does not change and so remains equal to $\{(a, out), (b, in), (c, und), (d, blank), (e, out), (f, in), (g, out), (h, blank), (k, blank)\}$.

3.8.5.2. at line 3, we find $label'$ admissible, and so, we return true.

3.8.6. at line 8, we return true.

3.9. at line 10, we return true.

4. at line 19, we report $b$ admissible.

Building on the old algorithm, next we develop a faster algorithm.

## 3 THE NEW ALGORITHM

As we did in the previous section, we introduce the new algorithm by using a top-down presentation, which means we give first the algorithm and then we specify its structures. Algorithm 2 is our new algorithm for the admissibility problem. If we run the algorithm on a given AF $H = (A, R)$ with a query argument $s$ then the algorithm decides whether $s$ is admissible or not.

We note that the old algorithm and the new algorithm have a similar high-level organization. However, we refine a number of constructs as we elaborate

Algorithm 2: The new algorithm.

```
   requires: an AF $H = (A, R)$ and a query argument
             $s \in A$.
   ensures : a decision whether $s$ is admissible or not.
1  Function isAdmissible($label, toIn, toUnd, undAtt, blankAtt$)
2      if propagate($label, toIn, toUnd, undAtt, blankAtt$)=false then return false;
3      if $label$ is admissible then return true;
4      $label' \leftarrow label, toIn' \leftarrow toIn, toUnd' \leftarrow toUnd$;
5      $undAtt' \leftarrow undAtt, blankAtt' \leftarrow blankAtt$;
6      select some $x \in \{y \mid label(y) = mustOut\}^-$ with $label(x) = blank$;
7      if in-trans($x, label', toIn', toUnd', undAtt', blankAtt'$)=false then go to line 9;
8      if isAdmissible($label', toIn', toUnd', undAtt', blankAtt'$)=true then return true;
9      if und-trans($x, label, toIn, toUnd, undAtt, blankAtt$)=false then go to line 11;
10     if isAdmissible($label, toIn, toUnd, undAtt, blankAtt$)=true then return true;
11     return false;
12 Function main()
13     if $(s, s) \in R$ then report $s$ inadmissibile and exit;
14     $label : A \rightarrow \{in, out, und, mustOut, blank, mustIn, mustUnd\}$;
15     $label \leftarrow \emptyset$, $toIn \leftarrow \emptyset$, $toUnd \leftarrow \emptyset$, $undAtt \leftarrow \emptyset$, $blankAtt \leftarrow \emptyset$;
16     foreach $x \in A$ do
17         $label(x) \leftarrow blank$, $undAtt(x) \leftarrow 0$, $blankAtt(x) \leftarrow |\{x\}^-|$;
18         if $(x, x) \in R$ then $label(x) \leftarrow mustUnd$, $toUnd \leftarrow toUnd \cup \{x\}$;
19         if $|\{x\}^-| = 0$ then $label(x) \leftarrow mustIn$, $toIn \leftarrow toIn \cup \{x\}$;
20     $label(s) \leftarrow mustIn$, $toIn \leftarrow toIn \cup \{s\}$;
21     if isAdmissible($label, toIn, toUnd, undAtt, blankAtt$)=true then $s$ is admissible;
22     else $s$ is not admissible;
```

throughout this section. Therefore, we will focus on the differences between the old algorithm and the new one.

At lines 14-19 in the new algorithm, we create and initialize five structures: $label$, $toIn$, $toUnd$, $blankAtt$ and $undAtt$. We illustrate all these structures next.

Observe that in the new algorithm we follow the basic mapping rules of the old algorithm, see remark 1. However, we add two additional labels: *mustIn* and *mustUnd*. In particular, for a given AF $(A, R)$, $label$ is now a total function that maps every argument in $A$ to a label in $\{in, out, mustOut, mustIn, mustUnd, und, blank\}$. Actually, we map an argument to *mustIn* for one of two reasons as described next.

**Definition 6** (mustIn Arguments). *Let $(A,R)$ be an* AF*, $label : A \to \{in, out, mustOut, mustIn, mustUnd, und, blank\}$ be a total mapping and $x \in A$ be an argument with $label(x) = blank$. Then, $x$ will be re-mapped to mustIn (or equivalently we say $x$ must be in) if and only if:*

- *For every $y \in \{x\}^- \setminus \{x\}^+$ $label(y) \in \{out, mustOut\}$, or*
- *There is $y \in \{x\}^+$ with $label(y) = mustOut$ such that $|\{z \in \{y\}^- : label(z) = blank\}| = 1$.*

We map an argument to *mustUnd* for the following reason.

**Definition 7** (mustUnd Arguments). *Let $(A,R)$ be an* AF*, $label : A \to \{in, out, mustOut, mustIn, mustUnd, und, blank\}$ be a total mapping and $x \in A$ be an argument with $label(x) = blank$. Then, $x$ will be re-mapped to mustUnd (or equivalently we say $x$ must be und) if and only if there is $y \in \{x\}^-$ with $label(y) \in \{blank, und, mustUnd\}$ such that for all $z \in \{y\}^-$ $label(z) \in \{out, mustOut, und, mustUnd\}$.*

Eventually, *mustIn* and *mustUnd* arguments will be re-mapped to *in* and *und* respectively, but we delay this to optimize the mapping propagation process as we elaborate shortly. At line 15 in the algorithm, we use *toIn* and *toUnd* sets, which respectively collect *mustIn* and *mustUnd* arguments to allow for an efficient access to them.

Also, at line 15 we use the total mappings: *undAtt* and *blankAtt*. For a given AF $(A,R)$ with a total mapping $label : A \to \{in, out, mustOut, mustIn, mustUnd, und, blank\}$, *undAtt* maps every argument $x \in A$ to $|\{y \in \{x\}^- : label(y) = und\}|$, while *blankAtt* maps every argument $x \in A$ to $|\{y \in \{x\}^- : label(y) \in \{blank, mustIn, mustUnd\}\}|$. The purpose of these mappings is to speed the process of checking the conditions under which an argument is mapped to *mustUnd* or *mustIn*. Further, these mappings streamline the computations around detecting hopeless labellings.

**Remark 2** (*undAtt* and *blankAtt*). *Let $(A,R)$ be an* AF*, $label : A \to \{in, out, mustOut, mustIn, mustUnd, und, blank\}$ be a total mapping, $\mathbb{N}$ be the set of non-negative integers, $undAtt : A \to \mathbb{N}$ with $blankAtt : A \to \mathbb{N}$ be total mappings and $x \in A$ be an argument. Then,*

- *Checking if $label(x) = blank$, $blankAtt(x) = 0$ and $undAtt(x) = 0$, is equivalent to checking whether $x$ must be in.*
- *Checking if $label(x) = mustOut$, $blankAtt(x) = 1$ and $\exists y \in \{x\}^-$ with $label(y) = blank$, is equivalent to checking whether $y$ must be in.*
- *Checking if $label(x) \in \{blank, mustUnd, und\}$, $blankAtt(x) = 0$, and $y \in \{x\}^+$ with $label(y) = blank$ is equivalent to checking whether $y$ must be und.*
- *Checking if $label(x) = mustOut$ and $blankAtt(x) = 0$ is equivalent to checking whether label is hopeless.*

Hence, remark 2 shows how we actually check if an argument has to be re-mapped to *mustIn* or *mustUnd*, or if the current mapping is hopeless.

At line 18, we map self-attacking arguments to *mustUnd* and then include them in *toUnd*. Then, at line 19 we map every $x$ with $|\{x\}^-| = 0$ to *mustIn* and then add them to *toIn*. Referring to line 7 in the algorithm, we re-define the *in-trans* routine as in the following specification.

**Definition 8** (In-trans Routine). *Let $(A,R)$ be an* AF*, $label : A \to \{in, out, mustIn, mustOut, und, mustUnd, blank\}$ be a total mapping, $x \in A$ be an argument with $label(x) \in \{blank, mustIn\}$, $toIn \subseteq A$ with $toUnd \subseteq A$ be sets of arguments, $\mathbb{N}$ be the set of non-negative integers, and $undAtt : A \to \mathbb{N}$ with $blankAtt : A \to \mathbb{N}$ be total mappings. Then, in-trans($x, label, toIn, toUnd, undAtt, blankAtt$) is defined by the set of actions:*

```
1. $label(x) \leftarrow in$.
2. for each $y \in \{x\}^+ \cup \{x\}^-$ with $label(y) \neq out$ do:
  2.1. for each $z \in \{y\}^+$ do:
    2.1.1. if $label(y) = und$, then $undAtt(z) \leftarrow undAtt(z) - 1$.
    2.1.2. if $label(y) = blank$, then $blankAtt(z) \leftarrow blankAtt(z) - 1$.
    2.1.3. if $label(z) = mustOut$ and $blankAtt(z) = 0$, then return false.
    2.1.4. if $z$ must be in, then $toIn \leftarrow toIn \cup \{z\}$ and $label(z) \leftarrow mustIn$.
    2.1.5. for each $v \in \{z\}^+$ s.t. $v$ must be und do:
      2.1.5.1. $toUnd \leftarrow toUnd \cup \{v\}$.
      2.1.5.2. $label(v) \leftarrow mustUnd$.
    2.1.6. if there is $v \in \{z\}^-$ s.t. $v$ must be in, then:
      2.1.6.1. $toin \leftarrow toIn \cup \{v\}$.
      2.1.6.2. $label(v) \leftarrow mustIn$.
  2.2. if $y \in \{x\}^-$, then $label(y) \leftarrow mustOut$.
  2.3. if $y \in \{x\}^+$, then $label(y) \leftarrow out$.
  2.4. if $label(y) = mustOut$ and $blankAtt(y) = 0$, then return false.
3. return true.
```

Referring to line 9 in the algorithm, we re-define *und-trans*.

**Definition 9** (Und-trans Routine). *Let $(A,R)$ be an* AF*, $label : A \to \{in, out, mustIn, mustOut, und,$*

*$mustUnd$, $blank\}$ be a total mapping, $x \in A$ be an argument with $label(x) \in \{blank, mustUnd\}$, $toIn \subseteq A$ with $toUnd \subseteq A$ be sets of arguments, $\mathbb{N}$ be the set of non-negative integers, and $undAtt : A \to \mathbb{N}$ with $blankAtt : A \to \mathbb{N}$ be total mappings. Then, und-trans($x$, $label$, $toIn$, $toUnd$, $undAtt$, $blankAtt$) is defined by the set of actions:*

```
1. $label(x) \leftarrow und$.
2. for each $y \in \{x\}^+$ do:
  2.1. $undAtt(y) \leftarrow undAtt(y) + 1$.
  2.2. $blankAtt(y) \leftarrow blankAtt(y) - 1$.
  2.3. if $label(y) = mustOut$ with $blankAtt(y) = 0$,
       then return false.
  2.4. for each $z \in \{y\}^+$ s.t. $z$ must be und do:
    2.4.1. $toUnd \leftarrow toUnd \cup \{z\}$.
    2.4.2. $label(z) \leftarrow mustUnd$.
  2.5. if there is $z \in \{y\}^-$ s.t. $z$ must be in, then:
    2.5.1. $toIn \leftarrow toIn \cup \{z\}$.
    2.5.2. $label(z) \leftarrow mustIn$.
3. return true.
```

Referring to line 2 in the algorithm, we refine the *propagate* routine.

**Definition 10** (Propagate Routine)**.** *Let $(A,R)$ be an* AF*, $label$: $A \to \{in, out, mustIn, mustOut, und, mustUnd, blank\}$ be a total mapping, $toIn \subseteq A$ with $toUnd \subseteq A$ be sets of arguments, $\mathbb{N}$ be the set of non-negative integers, and $undAtt : A \to \mathbb{N}$ with $blankAtt : A \to \mathbb{N}$ be total mappings. Then, propagate($label$,$toIn$,$toUnd$,$undAtt$,$blankAtt$) is defined by the following set of actions:*

```
1. while $toIn \neq \emptyset$ or $toUnd \neq \emptyset$ do:
  1.1. while $toIn \neq \emptyset$ do:
    1.1.1. remove an argument $x$ from $toIn$.
    1.1.2. if in-trans($x$, $label$, $toIn$, $toUnd$, $undAtt$,
           $blankAtt$)=false, then return false.
  1.1. while $toUnd \neq \emptyset$ do:
    1.1.1. remove an argument $x$ from $toUnd$.
    1.1.2. if und-trans($x$, $label$, $toIn$, $toUnd$, $undAtt$,
           $blankAtt$)=false, then return false.
2. return true.
```

We note that the speedup of the new algorithm comes mainly from refining the routines: *in-trans*, *und-trans*, and *propagate*; together with the structures: $label$, $toIn$, $toUnd$, $undAtt$ and $blankAtt$.

To see how these refined constructs have led to a faster admissibility deciding, let us have a closer look at two major actions: checking hopeless mappings, and mappings propagation. In the old algorithm, these two actions are done at a global scope, hence one needs to scan exhaustively all arguments to check hopelessness or to propagate mappings. Since these two actions are repeated enormously often at run-time, they are extremely expensive. However, not all arguments actually need to be checked (for hopelessness or propagation) because not all of them have been affected by the last re-mappings that happened during *in-trans* or *und-trans*. This is exactly what we have addressed in the new algorithm: we restrict the scope of these two major actions (i.e. hopelessness check and mappings propagation) to a relatively small subset of arguments, which roughly are the neighbors of the arguments that recently have been re-mapped. In the next section we verify practically the running-time performance of the new algorithm.

However, before closing this section we recall two other problems that are equivalent to the admissibility problem. To elaborate on this correspondence, we define *preferred* and *complete* arguments.

**Definition 11** (Preferred Arguments)**.** *Let $(A,R)$ be an* AF *and $S \subseteq A$ be a set of arguments. Then, $S$ is a preferred extension of* AF *if and only if $S$ is a set-inclusion-maximal admissible set. Further, we call $x \in A$ preferred if and only if $x$ is in a preferred extension.*

**Definition 12** (Complete Arguments)**.** *Let $(A,R)$ be an* AF *and $S \subseteq A$ be an admissible set, then $S$ is a complete extension of* AF *if and only if for each $x \notin S$ $x \in S^+$ or $\{x\}^- \not\subseteq S^+$. Further, we call $x \in A$ complete if and only if $x$ is in a complete extension.*

Therefore, it is not hard to see that every admissible set of a given AF is preferred/complete extension or it can be expanded to become one. Thus, deciding the admissibility of a given argument is equivalent to deciding if the argument is preferred or complete.

## 4 VERIFICATION

We verified the running-time performance of the new algorithm[1] by using benchmark B of the second international competition on computational models of argumentation 2017 (ICCMA17)[2]. The benchmark is a set of 350 different AFs. However, for evaluating the admissibility problem 300 AFs of benchmark B have been considered in ICCMA17, where 50 AFs were excluded due to triviality. In fact, ICCMA17 duplicated the hardest 50 AFs of the benchmark by generating two different queries on every AF. Thus, in total there

[1]The C++ source code is available for free download at: https://sourceforge.net/projects/argtools.

[2]ICCMA17 is organized by Sarah A. Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran. See http://argumentationcompetition.org/2017.

were 350 problem instances using only 300 AFs. To evaluate our algorithm we used a machine with Intel-core-i7 processor alongside four gigabytes of system memory. Note that the environment of ICCMA17 has a more powerful Intel-xeon processor and, four gigabytes of system memory were allocated for each problem instance. Following ICCMA17, we set a timeout of 10 minutes for each problem instance.

As this paper presents a direct approach to the admissibility problem, we compare with the three direct-based systems: ArgTools v1.0 (Nofal et al., 2015), heureka (Geilen and Thimm, 2017), and ArgEqSolver (Rodrigues, 2017). ArgTools v1.0 implements algorithm 1, and it solved 234 problem instances. The new algorithm (algorithm 2) is currently implemented within the ArgTools project, and it solved 297 problem instances. To the best of our knowledge, the specifications of admissibility checking in heureka and ArgEqSolver are not published. However, the two systems solved at best 148 problem instances. For reduction-based solvers, at best 304 problem instances were solved in ICCMA17. Lastly, note that the ICCMA17 solvers were evaluated (with respect to the admissibility problem) under the task: DC-PR (and respectively DC-CO), which stands for Decide Credulous acceptance under PReferred (respectively COmplete) semantics. Table 1 summarizes the total running times of the new algorithm compared to the systems mentioned above.

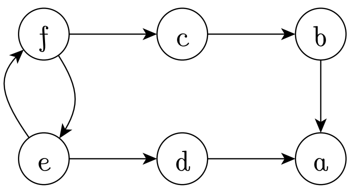
Table 1: Experimental performance.

| system | elapsed time (seconds) | #solved problem instances |
|---|---|---|
| Algorithm 2 | 36,833 | 297 |
| ArgTools v1.0 | 75,358 | 234 |
| heureka | 126,156 | 148 |
| ArgEqSolver | 122,604 | 148 |

## 5 AN EXAMPLE WITH LEGAL REASONING

In this section we provide an application of the admissibility checking using the following hypothetical exchange of allegations (as presented in (Jakobovits and Vermeir, 1999) and which is actually adapted from (Gordon, 1993)):

> The plaintiff and the defendant have both loaned money to Miller for the purchase of an oil tanker, which is the collateral for both loans. Miller has defaulted on both loans, and the practical question is which of the two lenders will first be paid from the proceeds of the sale of the ship. One subsidiary issue is whether the plaintiff perfected his security interest in the ship or not.



$$\begin{tikzcd} f \arrow[r] \arrow[d, bend left] & c \arrow[r] & b \arrow[d] \\ e \arrow[u, bend left] \arrow[r] & d \arrow[r] & a \end{tikzcd}$$

Figure 2: The AF associated with the legal case.

- Argument $a$ by the plaintiff: My security interest in Miller's ship was perfected. A security interest in goods may be perfected by taking possession of the collateral (UCC Article 9). I have possession of Miller's ship.
- Argument $b$ by defendant: Ships are not goods for the purposes of Article 9.
- Argument $c$ by the plaintiff: Ships are movable, and movable things are goods according to UCC Article 9.
- Argument $d$ by the defendant: According to the Ship Mortgage Act, a security interest in a ship may only be perfected by filing a financing statement.
- Argument $e$ by the plaintiff: The Ship Mortgage Act does not apply, since the UCC is newer and therefore has precedence.
- Argument $f$ by the defendant: The Ship Mortgage Act is federal law, which has precedence over state law such as UCC.

Figure 2 shows these arguments and the attack relation between them.

Let us now check the admissibility of the plaintiff's claim regarding her security interest in Miller's ship being perfected (argument $a$ in Figure 2.) Subsequently, the question now is whether we can find an admissible set $S$ of arguments including $a$. Let us start with $S = \{a\}$. Thus, $S^- = \{b, d\}$. As $S^- \not\subseteq S^+$, we need to expand $S$ further by an argument from $(S^-)^-$. Let $c$ join $S$. Now, $S = \{a, c\}$. But still $S^- \not\subseteq S^+$. Therefore, include $e$ in $S$. At this point, $S = \{a, c, e\}$, $S^- = \{b, d, f\}$ and $S^- \subseteq S^+$. As now $S$ is admissible, the plaintiff's claim for perfecting her security interest in the Miller's ship is admissible.

## 6 CONCLUSION

We implemented an improved algorithm for the admissibility problem. Using benchmark B of the international competition ICCMA17, we evaluated the new algorithm and practically verified that it outperforms the old algorithm. As the speed gain is due to a number of useful structures that optimize the underlying actions of the new algorithm, we plan to investigate the possibility of utilizing additional constructs that might enhance further the admissibility-deciding procedures. In particular, referring to line 6 in the new algorithm, we currently select an argument by searching in the whole set of arguments (i.e. $A$). An open issue is finding a cost-effective mechanism to restrict the search to a subset of candidate arguments.

## ACKNOWLEDGMENT

This work is supported by the scientific research deanship of the German Jordanian University.

## REFERENCES

Alfano, G., Greco, S., and Parisi, F. (2017). Efficient computation of extensions for dynamic abstract argumentation frameworks: An incremental approach. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 49–55.

Amgoud, L. and Cayrol, C. (2002). Inferring from inconsistency in preference-based argumentation frameworks. *Journal of Automated Reasoning*, 29(2):125–169.

Amgoud, L. and Vesic, S. (2010). Handling inconsistency with preference-based argumentation. In Deshpande, A. and Hunter, A., editors, *Scalable Uncertainty Management*, pages 56–69, Berlin, Heidelberg. Springer Berlin Heidelberg.

Atkinson, K., Baroni, P., Giacomin, M., Hunter, A., Prakken, H., Reed, C., Simari, G. R., Thimm, M., and Villata, S. (2017). Towards artificial argumentation. *AI Magazine*, 38(3):25–36.

Baroni, P., Caminada, M., and Giacomin, M. (2011). An introduction to argumentation semantics. *Knowledge Eng. Review*, 26(4):365–410.

Bench-Capon, T. J. M., Atkinson, K., and Wyner, A. Z. (2015). Using argumentation to structure e-participation in policy making. *T. Large-Scale Data- and Knowledge-Centered Systems*, 18:1–29.

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

Cayrol, C., Doutre, S., and Mengin, J. (2003). On decision problems related to the preferred semantics for argumentation frameworks. *J. Log. Comput.*, 13(3):377–403.

Charwat, G., Dvorák, W., Gaggl, S. A., Wallner, J. P., and Woltran, S. (2015). Methods for solving reasoning problems in abstract argumentation - A survey. *Artif. Intell.*, 220:28–63.

Croitoru, M. and Vesic, S. (2013). What can argumentation do for inconsistent ontology query answering? In Liu, W., Subrahmanian, V. S., and Wijsen, J., editors, *Scalable Uncertainty Management*, pages 15–29, Berlin, Heidelberg. Springer Berlin Heidelberg.

Doutre, S. and Mailly, J. (2018). Constraints and changes: A survey of abstract argumentation dynamics. *Argument & Computation*, 9(3):223–248.

Doutre, S. and Mengin, J. (2001). Preferred extensions of argumentation frameworks: Query answering and computation. In *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, pages 272–288.

Dung, P. M. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358.

Dunne, P. E. (2007). Computational properties of argument systems satisfying graph-theoretic constraints. *Artif. Intell.*, 171(10-15):701–729.

Dvorák, W. and Dunne, P. E. (2017). Computational problems in formal argumentation and their complexity. *FLAP*, 4(8).

Dvořák, W., Pichler, R., and Woltran, S. (2012). Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37.

Geilen, N. and Thimm, M. (2017). Heureka: A general heuristic backtracking solver for abstract argumentation. In *second international competition on computational argumentation models*.

Gordon, T. F. (1993). The pleadings game: Formalizing procedural justice. In *Proceedings of the 4th International Conference on Artificial Intelligence and Law*, ICAIL '93, pages 10–19, New York, NY, USA. ACM.

Hecham, A., Arioua, A., Stapleton, G., and Croitoru, M. (2017). An empirical evaluation of argumentation in explaining inconsistency-tolerant query answering. In *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017*.

Heras, S., Atkinson, K., Botti, V. J., Grasso, F., Julián, V., and McBurney, P. (2013). Research opportunities for argumentation in social networks. *Artif. Intell. Rev.*, 39(1):39–62.

Hunter, A. and Williams, M. (2012). Aggregating evidence about the positive and negative effects of treatments. *Artificial Intelligence in Medicine*, 56(3):173–190.

Jakobovits, H. and Vermeir, D. (1999). Dialectic semantics for argumentation frameworks. In *Proceedings of the 7th International Conference on Artificial Intelligence and Law*, ICAIL '99, pages 53–62, New York, NY, USA. ACM.

Liao, B. S., Jin, L., and Koons, R. C. (2011). Dynamics of argumentation systems: A division-based method. *Artif. Intell.*, 175(11):1790–1814.

Martinez, M. V. and Hunter, A. (2009). Incorporating classical logic argumentation into policy-based inconsistency management in relational databases. In *The Uses of Computational Argumentation, Papers from* *the 2009 AAAI Fall Symposium, Arlington, Virginia, USA, November 5-7, 2009*.

Modgil, S. and Caminada, M. (2009). Proof theories and algorithms for abstract argumentation frameworks. In (Simari and Rahwan, 2009), pages 105–129.

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., and Woltran, S. (2013). The added value of argumentation. In Ossowski, S., editor, *Agreement Technologies*, volume 8 of *Law, Governance and Technology Series*, pages 357–403. Springer Netherlands.

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

Nofal, S., Atkinson, K., and Dunne, P. E. (2015). Argtools: a backtracking-based solver for abstract argumentation. In *first international competition on computational argumentation models*.

Nofal, S., Atkinson, K., and Dunne, P. E. (2016). Looking-ahead in backtracking algorithms for abstract argumentation. *Int. J. Approx. Reasoning*, 78:265–282.

Rodrigues, O. (2017). A forward propagation algorithm for the computation of the semantics of argumentation frameworks. In *Theory and Applications of Formal Argumentation - 4th International Workshop, TAFA 2017, Melbourne, VIC, Australia, August 19-20, 2017, Revised Selected Papers*, pages 120–136.

Simari, G. R. and Rahwan, I., editors (2009). *Argumentation in Artificial Intelligence*. Springer.

Tamani, N., Mosse, P., Croitoru, M., Buche, P., Guillard, V., Guillaume, C., and Gontard, N. (2015). An argumentation system for eco-efficient packaging material selection. *Computers and Electronics in Agriculture*, 113(0):174 – 192.

Thang, P. M., Dung, P. M., and Hung, N. D. (2009). Towards a common framework for dialectical proof procedures in abstract argumentation. *J. Log. Comput.*, 19(6):1071–1109.

Thimm, M. and Villata, S. (2017). The first international competition on computational models of argumentation: Results and analysis. *Artif. Intell.*, 252:267–294.

Verheij, B. (2007). A labeling approach to the computation of credulous acceptance in argumentation. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 623–628.