Strong Admissibility, a Tractable Algorithmic Approach (proofs)

Martin Caminada
Cardiff University

Sri Harikrishnan
Cardiff University

April 7, 2022

Abstract

Much like admissibility is the key concept underlying preferred semantics, strong admissibility is the key concept underlying grounded semantics, as membership of a strongly admissible set is sufficient to show membership of the grounded extension. As such, strongly admissible sets and labellings can be used as an explanation of membership of the grounded extension, as is for instance done in some of the proof procedures for grounded semantics. In the current paper, we present two polynomial algorithms for constructing relatively small strongly admissible labellings, with associated min-max numberings, for a particular argument. These labellings can be used as relatively small explanations for the argument's membership of the grounded extension. Although our algorithms are not guaranteed to yield an absolute minimal strongly admissible labelling for the argument (as doing do would have implied an exponential complexity), our best performing algorithm yields results that are only marginally bigger. Moreover, the runtime of this algorithm is an order of magnitude smaller than that of the existing approach for computing an absolute minimal strongly admissible labelling for a particular argument. As such, we believe that our algorithms can be of practical value in situations where the aim is to construct a minimal or near-minimal strongly admissible labelling in a time-efficient way.

1 Introduction

In formal argumentation, one would sometimes like to show that a particular argument is (credulously) accepted according to a particular argumentation semantics, without having to construct the entire extension the argument is contained in. For instance, to show that an argument is in a preferred extension, it is not necessary to construct the entire preferred extension. Instead, it is sufficient to construct a set of arguments that is *admissible*. Similarly, to show that an argument is in the grounded extension, it is not necessary to construct the entire grounded extension. Instead, it is sufficient to construct a set of arguments that is *strongly admissible*.

The concept of strong admissibility was introduced by Baroni and Giacomin [1] as one of the properties to describe and categorise argumentation semantics. It was subsequently studied by Caminada and Dunne [4, 7] who further developed strong admissibility in both its set and labelling form. In particular, the strongly admissible sets (resp. labellings) were found to form a latice with the empty set (resp. the all-`undec` labeling) as its bottom element and the grounded extension (resp. the grounded labelling) as its top element [4, 7].

As a strongly admissible set (labelling) can be used to explain that a particular argument is in the grounded extension (for instance, by using the discussion game of [5]) a relevant question is whether one can identify an expanation that is *minimal*. That is, given an argument A that is in the grounded extension, how can one obtain:
(1) a strongly admissible set that contains A, of which the number of arguments is minimal among all strongly admissible sets containing A, and
(2) a strongly admissible labelling that labels A `in`, of which the number of `in` and `out` labelled arguments (its *size*, cf. [8]) is minimal among all strongly admissible labelings that label A `in`.

It has been found that the verification problem of (1) is NP-complete [12] whereas the the verification problem of (2) is co-NP-complete [8]. Moreover, it has also been observed that even computing a c-approximation for the minimum size of a strongly admissible set for a given argument is NP-hard for every $c \geq 1$. This is in sharp contrast with the complexity of the general verification problem of strong admissibility (i.e. verifying whether a set/labelling is strongly admissible, without the constraint that it also has to be minimal) which has been found to be polynomial [7].

The complexity results related to minimal strong admissibility pose a problem when the aim is to provide the user with a relatively small explanation of why a particular argument is in the grounded extension. For this, one can either apply an algorithmic approach that yields an absolute minimal explanation, but has an exponential runtime, or one can apply an algorithmic approach that has a less than exponential runtime, but does not come with any formal guarantees of how close the outcome is to an absolute minimal explanation [12]. The former approach is taken in [12]. The latter approach is taken in our current paper.

In the absence of a dedicated algorithm for strong admissibility, one may be tempted to simply apply an algorithm for computing the grounded extension or labelling instead (such as [13, 14]) if the aim is to do the computation in polynomial time. Still, from the perspective of minimality, this would yield the absolute worst outcome, as the grounded extension (labeling) is the maximal strongly admissible set (labelling). In the current paper we therefore introduce an alternative algorithm which, like the grounded semantics algoritms, runs in polynomial time but tends to produce a strongly admissible set (resp. labelling) that is that is significantly smaller than the grounded extension (resp. labelling). As the complexity results from [12] prevent us from giving any theory-based guarantees regarding how close the outcome of the algorithm is to an absolute minimal strongly admissible set, we will instead assess the performance of the algorithm using a wide range of benchmark examples.

The remaining part of the current paper is structured as follows. First, in Section 2 we give a brief overview of the formal concepts used in the current paper, including that of a strongly admissible set and a strongly admissible labelling. In Section 3 we then proceed to provide the proposed algorithm, including the associated proofs of correctness. Then, in Section 4 we assess the performance of our approach, and compare it with the results yielded by the approach in [12] both in terms of outcome and runtime. We round off with a discussion of our findings in Section 5.

2 Preliminaries

In the current section, we briefly restate some of the basic concepts in formal argumentation theory, including strong admissibility. For current purposes, we restrict ourselves to finite

argumentation frameworks.

Definition 1. *An* argumentation framework *is a pair* (Ar, att) *where* Ar *is a finite set of entities, called arguments, whose internal structure can be left unspecified, and* att *is a binary relation on* Ar*. For any* $x, y \in Ar$ *we say that* x attacks y *iff* $(x, y) \in att$*.*

As for notation, we use lower case letters at the end of the alphabet (such as x, y and z) to denote variables containing arguments, upper case letters at the end of the alphabet (such as X, Y and Z) to denote program variables containing arguments, and upper case letters at the start of the alphabet (such as A, B and C) to denote concrete instances of arguments.

When it comes to defining argumentation semantics, one can distinguish the *extension approach* and the *labelling approach* [6]. We start with the extensions approach.

Definition 2. *Let* (Ar, att) *be an argumentation framework,* $x \in Ar$ *and* $Args \subseteq Ar$*. We define* x^+ *as* $\{y \in Ar \mid x \text{ attacks } y\}$*,* x^- *as* $\{y \in Ar \mid y \text{ attacks } x\}$*,* $Args^+$ *as* $\bigcup\{x^+ \mid x \in Args\}$*, and* $Args^-$ *as* $\bigcup\{x^- \mid x \in Args\}$*. Args is said to be* conflict-free *iff* $Args \cap Args^+ = \emptyset$*. Args is said to* defend x *iff* $x^- \subseteq Args^+$*. The characteristic function* $F : 2^{Ar} \to 2^{Ar}$ *is defined as* $F(Args) = \{x \mid Args \text{ defends } x\}$*.*

Definition 3. *Let* (Ar, att) *be an argumentation framework.* $Args \subseteq Ar$ *is*

- *an admissible set iff* $Args$ *is conflict-free and* $Args \subseteq F(Args)$
- *a complete extension iff* $Args$ *is conflict-free and* $Args = F(Args)$
- *a grounded extension iff* $Args$ *is the smallest (w.r.t.* $\subseteq$*) complete extension*
- *a preferred extension iff* $Args$ *is a maximal (w.r.t.* $\subseteq$*) complete extension*

As mentioned in the introduction, the concept of strong admissibility was originally introduced by Baroni and Giacomin [1]. For current purposes we will apply the equivalent definition of Caminada [4, 7].

Definition 4. *Let* (Ar, att) *be an argumentation framework.* $Args \subseteq Ar$ *is* strongly admissible *iff every* $x \in Args$ *is defended by some* $Args' \subseteq Args \setminus \{x\}$ *which in its turn is again strongly admissible.*

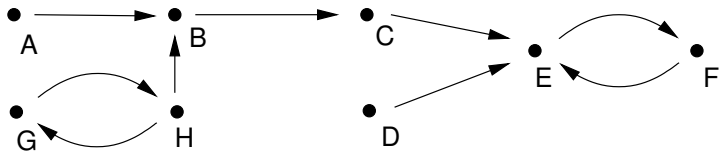

Figure 1: An example of an argumentation framework.

As an example (taken from [7]), in the argumentation framework of Figure 1 the strongly admissible sets are $\emptyset$, $\{A\}$, $\{A, C\}$, $\{A, C, F\}$, $\{D\}$, $\{A, D\}$, $\{A, C, D\}$, $\{D, F\}$, $\{A, D, F\}$ and $\{A, C, D, F\}$, the latter also being the grounded extension. The set $\{A, C, F\}$ is strongly admissible as A is defended by $\emptyset$, C is defended by $\{A\}$ and F is defended by $\{A, C\}$, each of which is a strongly admissible subset of $\{A, C, F\}$ not containing the argument it defends. Please notice that although the set $\{A, F\}$ defends argument C in $\{A, C, F\}$, it is in its turn

not strongly admissible (unlike $\{A\}$). Hence the requirement in Definition 4 for $Args'$ to be a *subset* of $Args \setminus \{A\}$. We also observe that although $\{C, H\}$ is an admissible set, it is not a *strongly* admissible set, since no subset of $\{C, H\} \setminus \{H\}$ defends H.

It can be shown that each strongly admissible set is conflict-free and admissible [7]. The strongly admissible sets form a lattice (w.r.t. $\subseteq$), of which the empty set is the bottom element and the grounded extension is the top element [7].

The above definitions essentially follow the extension based approach as described in [11]. It is also possible to define the key argumentation concepts in terms of argument labellings [2, 9].

Definition 5. *Let (Ar, att) be an argumentation framework. An* argument labelling *is a function $\mathcal{L}ab : Ar \to \{\texttt{in}, \texttt{out}, \texttt{undec}\}$. An argument labelling is called an* admissible labelling *iff for each $x \in Ar$ it holds that:*

- *if $\mathcal{L}ab(x) = \texttt{in}$ then for each y that attacks x it holds that $\mathcal{L}ab(y) = \texttt{out}$*
- *if $\mathcal{L}ab(x) = \texttt{out}$ then there exists a y that attacks x such that $\mathcal{L}ab(y) = \texttt{in}$*

$\mathcal{L}ab$ is called a complete labelling *iff it is an admissible labelling and for each $x \in Ar$ it also holds that:*

- *if $\mathcal{L}ab(x) = \texttt{undec}$ then there is a y that attacks x such that $\mathcal{L}ab(y) = \texttt{undec}$, and for each y that attacks x such that $\mathcal{L}ab(y) \neq \texttt{undec}$ it holds that $\mathcal{L}ab(y) = \texttt{out}$*

As a labelling is essentially a function, we sometimes write it as a set of pairs. Also, if $\mathcal{L}ab$ is a labelling, we write $\texttt{in}(\mathcal{L}ab)$ for $\{x \in Ar \mid \mathcal{L}ab(x) = \texttt{in}\}$, $\texttt{out}(\mathcal{L}ab)$ for $\{x \in Ar \mid \mathcal{L}ab(x) = \texttt{out}\}$ and $\texttt{undec}(\mathcal{L}ab)$ for $\{x \in Ar \mid \mathcal{L}ab(x) = \texttt{undec}\}$. As a labelling is also a partition of the arguments into sets of in-labelled arguments, out-labelled arguments and undec-labelled arguments, we sometimes write it as a triplet $(\texttt{in}(\mathcal{L}ab), \texttt{out}(\mathcal{L}ab), \texttt{undec}(\mathcal{L}ab))$.

Definition 6 ([10])**.** *Let $\mathcal{L}ab$ and $\mathcal{L}ab'$ be argument labellings of argumentation framework (Ar, att). We say that $\mathcal{L}ab \sqsubseteq \mathcal{L}ab'$ iff $\texttt{in}(\mathcal{L}ab) \subseteq \texttt{in}(\mathcal{L}ab')$ and $\texttt{out}(\mathcal{L}ab) \subseteq \texttt{out}(\mathcal{L}ab')$.*

Definition 7. *Let $\mathcal{L}ab$ be a complete labelling of argumentation framework (Ar, att). $\mathcal{L}ab$ is said to be*

- *the grounded labelling iff $\mathcal{L}ab$ is the (unique) smallest (w.r.t. $\sqsubseteq$) complete labelling*
- *a preferred labelling iff $\mathcal{L}ab$ is a maximal (w.r.t. $\sqsubseteq$) complete labelling*

We refer to the *size* of a labelling $\mathcal{L}ab$ as $|\texttt{in}(\mathcal{L}ab) \cup \texttt{out}(\mathcal{L}ab)|$. We observe that if $\mathcal{L}ab \sqsubseteq \mathcal{L}ab'$ then the size of $\mathcal{L}ab$ is smaller or equal to the size of $\mathcal{L}ab'$, but not necessarily vice versa. In the remainder of the current paper, we use the terms smaller, bigger, minimal and maximal in relation to the size of the respective labellings, unless stated otherwise.

The next step is to define a strongly admissible labelling. In order to do so, we need the concept of a min-max numbering [7].

Definition 8. *Let $\mathcal{L}ab$ be an admissible labelling of argumentation framework (Ar, att). A* min-max numbering *is a total function $\mathcal{MM}_{\mathcal{L}ab} : \texttt{in}(\mathcal{L}ab) \cup \texttt{out}(\mathcal{L}ab) \to \mathbb{N} \cup \{\infty\}$ such that for each $x \in \texttt{in}(\mathcal{L}ab) \cup \texttt{out}(\mathcal{L}ab)$ it holds that:*

- *if* $\mathcal{L}ab(x) = \texttt{in}$ *then* $\mathcal{MM}_{\mathcal{L}ab}(x) = max(\{\mathcal{MM}_{\mathcal{L}ab}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab(y) = \texttt{out}\}) + 1$ *(with* $max(\emptyset)$ *defined as* 0*)*
- *if* $\mathcal{L}ab(x) = \texttt{out}$ *then* $\mathcal{MM}_{\mathcal{L}ab}(x) = min(\{\mathcal{MM}_{\mathcal{L}ab}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab(y) = \texttt{in}\}) + 1$ *(with* $min(\emptyset)$ *defined as* ∞*)*

It has been proved that every admissible labelling has a unique min-max numbering [7]. A strongly admissible labelling can then be defined as follows [7].

Definition 9. *A* strongly admissible labelling *is an admissible labelling whose min-max numbering yields natural numbers only (so no argument is numbered* ∞*).*

As an example (taken from [7]), consider again the argumentation framework of Figure 1. Here, the admissible labelling $\mathcal{L}ab_1 = (\{A, C, F, G\}, \{B, E, H\}, \{D\})$ has min-max numbering $\{(A : 1), (B : 2), (C : 3), (E : 4), (F : 5), (G : \infty), (H : \infty)\}$, which means that it is not strongly admissible. The admissible labelling $\mathcal{L}ab_2 = (\{A, C, D, F\}, \{B, E\}, \{G, H\})$ has min-max numbering $\{(A : 1), (B : 2), (C : 3), (D : 1), (E : 2), (F : 3)\}$, which means that it is strongly admissible.

It has been shown that the strongly admissible labellings form a lattice (w.r.t. $\sqsubseteq$), of which the all-`undec` labelling is the bottom element and the grounded labelling is the top element [7].

The relationship between extensions and labellings has been well-studied [3, 9]. A common way to relate extensions to labellings is through the functions `Args2Lab` and `Lab2Args`. These translate a conflict-free set of arguments to an argument labelling, and an argument labelling to a set of arguments, respectively. More specifically, given an argumentation framework (Ar, att), and an associated conflict-free set of arguments $Args$ and a labelling $\mathcal{L}ab$, $\texttt{Args2Lab}(Args)$ is defined as $(Args, Args^+, Ar \setminus (Args \cup Args^+))$ and $\texttt{Lab2Args}(\mathcal{L}ab)$ is defined as $\texttt{in}(\mathcal{L}ab)$. It has been proven [9] that if $Args$ is an admissible set (resp. a complete, grounded or preferred extension) then $\texttt{Args2Lab}(Args)$ is an admissible labelling (resp. a complete, grounded or preferred labelling), and that if $\mathcal{L}ab$ is an admissible labelling (resp. a complete, grounded or preferred labelling) then $\texttt{Lab2Args}(\mathcal{L}ab)$ is an admissible set (resp. a complete, grounded or preferred extension). It has also been proven [7] that if $Args$ is a strongly admissible set then $\texttt{Args2Lab}(Args)$ is a strongly admissible labelling, and that if $\mathcal{L}ab$ is a strongly admissible labelling then $\texttt{Lab2Args}(\mathcal{L}ab)$ is a strongly admissible set.

3 The Algorithms

In the current section, we present an algorithmic approach for computing a relatively small[1] strongly admissible labelling. For this, we provide three different algorithms. The first algorithm (Algorithm 1) basically constructs a strongly admissible labelling bottom-up, starting with the arguments that have no attackers and continuing until the main argument (the argument for which one want to show membership of a strongly admissible set) is labelled `in`. The second algorithm (Algorithm 2) then takes the output of the first algorithm and tries to prune it. That is, it tries to identify only those `in` and `out` labelled arguments that are actually needed in the strongly admissible labelling. The third algorithm (Algorithm 3) then combines Algorithm 1 (which is used as the construction phase) and Algorithm 2 (which is used as the pruning phase).

[1] Small with respect to the size of the labelling.

3.1 Algorithm 1

The basic idea of Algorithm 1 is to start constructing the grounded labelling bottom-up, until we reach the main argument (that is, until we reach the argument that we are trying to construct a strongly admissible labelling for; this argument should hence be labelled `in`). As such, the idea is to take an algorithm for computing the grounded labeling (e.g. [13] or [14]) and modify it accordingly. We have chosen the algorithm of [14] for this purpose, as it has been proved to run faster than some of the alternatives (such as [13]). We had to adjust this algorithm in two ways. First, as mentioned above, we want the algorithm to stop once it hits the main argument, instead of continuing to construct the entire grounded labelling. Second, we want it to compute not just the strongly admissible labelling itself, but also its associated min-max numbering.

Obtaining the min-max numbering is important, as it can be used to show that the obtained admissible labelling is indeed *strongly* admissible, through the absence of ∞ in its min-max numbering. Additionally, the min-max numbering is also needed for some of the applications of strong admissibility, in particular the Grounded Discussion Game [5] where the combination of a strongly admissible labelling and its associated min-max numbering serves as a roadmap for obtaining a winning strategy.

Instead of first computing the strongly admissible labelling and then proceeding to compute the min-max numbering, we want to compute both the strongly admissible labelling and the min-max numbering in just a single pass, in order to achieve the best performance.

To see how the algorithm works, consider again the argumentation framework of Figure 1. Let C be the main argument. At the start of the first iteration of the while loop (line 21) it holds that $\mathcal{L}ab = (\{A, D\}, \emptyset, \{B, C, E, F, G, H\})$, $\mathcal{MM}_{\mathcal{L}ab} = \{(A : 1), (D : 1)\}$ and `unproc_in` $= [A, D]$. At the first iteration of the while loop, the argument in front of `unproc_in` (A) is selected (line 22). This then means that B gets labelled `out` and C gets labelled `in` Hence, the algorithm hits the main argument (C) at line 33 and terminates. This yields a labelling $\mathcal{L}ab = (\{A, C, D\}, \{B\}, \{E, F, G, H\})$ and associated min-max numbering $\mathcal{MM}_{\mathcal{L}ab} = \{(A : 1), (B : 2), (C : 3), (D : 1)\}$.

We now proceed to prove some of the formal properties of the algorithm. The first property to be proved is termination.

Theorem 1. *Let $AF = (Ar, att)$ be an argumentation framework and A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. It holds that the algorithm terminates.*

Proof. As for the first loop (the for loop of lines 9-18) we observe that it terminates as the number of arguments in Ar is finite.

As for the second loop (the while loop of lines 21-37) we first observe that no argument can be added to `unproc_in` more than once (that is, once an argument has been added to `unproc_in`, it can never be added again). This is because for an argument to be added, it has to be labelled `undec` (line 27) whereas after adding it, it will be labelled `in` (line 31). Moreover, once an argument is labelled `in`, it will stay labelled `in` as there is nothing in the algorithm that will change it. Given that (1) there is only a finite number of arguments in Ar, (2) each argument can be added to `unproc_in` at most once, and (3) each iteration of the while loop removes an argument from `unproc_in`, it follows that the loop has to terminate. □

Next, we need to show that the algorithm is correct. That is, we need to show that the algorithm yields a strongly admissible labelling $\mathcal{L}ab$ that labels A `in`, together with its

Algorithm 1 Construct a strongly admissible labelling that labels A `in` and its associated min-max numbering.

Input: An argumentation framework $AF = (Ar, att)$,
an argument $A \in Ar$ that is in the grounded extension of AF.
Output: A strongly admissible labelling $\mathcal{L}ab$ where $A \in \texttt{in}(\mathcal{L}ab)$,
the associated min-max numbering $\mathcal{MM}_{\mathcal{L}ab}$.

1: // We start with the type definitions
2: $\mathcal{L}ab : Ar \rightarrow \{\texttt{in}, \texttt{out}, \texttt{undec}\}$
3: $\mathcal{MM}_{\mathcal{L}ab} : \texttt{in}(\mathcal{L}ab) \cup \texttt{out}(\mathcal{L}ab) \rightarrow \mathbb{N} \cup \{\infty\}$
4: $\texttt{undec\_pre} : Ar \rightarrow \mathbb{N}$
5: $\texttt{unproc\_in} : [X_1, ...X_n]$ ($X_i \in Ar$ for each $1 \leq i \leq n$) // list of arguments
6:
7: // Next, we initialize and process the arguments that have no attackers
8: $\texttt{unproc\_in} \leftarrow []$
9: **for** each $X \in Ar$ **do**
10: $\quad \mathcal{L}ab(X) \leftarrow \texttt{undec}$
11: $\quad \texttt{undec\_pre}(X) \leftarrow |X^-|$
12: $\quad$ **if** $\texttt{undec\_pre}(X) = 0$ **then**
13: $\quad\quad$ add X to the rear of `unproc_in`
14: $\quad\quad \mathcal{L}ab(X) \leftarrow \texttt{in}$
15: $\quad\quad \mathcal{MM}_{\mathcal{L}ab}(X) \leftarrow 1$
16: $\quad\quad$ **if** $X = A$ **then** return $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$
17: $\quad$ **end if**
18: **end for**
19:
20: // We proceed to process the arguments that do have attackers
21: **while** `unproc_in` is not empty **do**
22: $\quad$ let X be the argument at the front of `unproc_in`
23: $\quad$ remove X from `unproc_in`
24: $\quad$ **for** each $Y \in X^+$ with $\mathcal{L}ab(Y) \neq \texttt{out}$ **do**
25: $\quad\quad \mathcal{L}ab(Y) \leftarrow \texttt{out}$
26: $\quad\quad \mathcal{MM}_{\mathcal{L}ab}(Y) \leftarrow \mathcal{MM}_{\mathcal{L}ab}(X) + 1$
27: $\quad\quad$ **for** each $Z \in Y^+$ with $\mathcal{L}ab(Z) = \texttt{undec}$ **do**
28: $\quad\quad\quad \texttt{undec\_pre}(Z) \leftarrow \texttt{undec\_pre}(Z) - 1$
29: $\quad\quad\quad$ **if** $\texttt{undec\_pre}(Z) = 0$ **then**
30: $\quad\quad\quad\quad$ add Z to the rear of `unproc_in`
31: $\quad\quad\quad\quad \mathcal{L}ab(Z) \leftarrow \texttt{in}$
32: $\quad\quad\quad\quad \mathcal{MM}_{\mathcal{L}ab}(Z) \leftarrow \mathcal{MM}_{\mathcal{L}ab}(Y) + 1$
33: $\quad\quad\quad\quad$ **if** $Z = A$ **then** return $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$
34: $\quad\quad\quad$ **end if**
35: $\quad\quad$ **end for**
36: $\quad$ **end for**
37: **end while**
38:
39: // If we get here, A is not in the grounded extension,
40: // so we may want to print an error message

associated min-max numbering $\mathcal{MM}_{\mathcal{L}ab}$. In order to do so, we first need to state and prove a number of lemmas. We start with showing that $\mathcal{L}ab$ is admissible in every stage of the algorithm.

Lemma 2. *Let $AF = (Ar, att)$ be an argumentation framework and A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. It holds that during any stage in the algorithm, $\mathcal{L}ab$ is an admissible labelling.*

Proof. Consider the value of $\mathcal{L}ab$ at an arbitrary point during the execution of Algorith 1. According to the definition of an admissible labelling (Definition 5) we need to prove two things, for an arbitrary argument $x \in Ar$:

1. if $\mathcal{L}ab(x) = \texttt{in}$ then for each y that attacks x it holds that $\mathcal{L}ab(y) = \texttt{out}$
   Suppose $\mathcal{L}ab(x) = \texttt{in}$. We distinguish two cases:
   (a) x was labelled `in` at line 14. This implies that $\texttt{und\_pre}(x) = 0$ in line 12, which implies that x has no attackers. Therefore, trivially $\mathcal{L}ab(y) = \texttt{out}$ for each $y \in Ar$ that attacks x.
   (b) x was labelled `in` at line 31. This implies that $\texttt{undec\_pre}(x) = 0$ in line 29, which implies that each attacker y of x has been relabelled to `out`. To see that this is the case, let n be the number of attackers of x (that is, $n = |x^-|$). It follows that $\texttt{undec\_pre}(x)$ is initially n (line 11) and at least 1 (otherwise x would have been labelled `in` at line 14 instead of at line 31). In order for $\texttt{undec\_pre}(x)$ to have fallen to 0 (line 29) it will need to have decremented (at line 28) n times (as no other line changes the value of $\texttt{undec\_pre}(x)$). Each time this happens at line 28, an attacker of x that wasn't previously labelled `out` (line 24) is labelled `out` (line 25). Therefore, by the time $\texttt{undec\_pre}(x)$ became 0, it follows that *all* attackers of x have become labelled `out`.
2. if $\mathcal{L}ab(x) = \texttt{out}$ then there exists a y that attacks x such that $\mathcal{L}ab(y) = \texttt{in}$
   Suppose $\mathcal{L}ab(x) = \texttt{out}$. This implies that x was labelled `out` at line 25, which implies that an attacker y of x was an element of `unproc_in`. This means that at some point, argument y was added to `unproc_in`. This could have happened at line 13 or 30. In both cases, it follows that (line 14 and 31) y is labelled `in`.

□

The next lemma presents an intermediary result that will be needed further on in the proofs.

Lemma 3. *Let $AF = (Ar, att)$ be an argumentation framework and A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. It holds that for each argument x that is added to* `unproc_in`*, $\mathcal{MM}_{\mathcal{L}ab}(x) \geq 1$*

Proof. We prove this by induction over the number of arguments that are added to `unproc_in` during the execution of the while loop of lines 21-37.

BASIS (n=0) Suppose the while loop has not yet added any arguments to `unproc_in`. This means that any argument x that was added to `unproc_in` was added by the for loop (lines 9-18). This could only have been done at line 13. Line 15 then implies that $\mathcal{MM}_{\mathcal{L}ab}(x) = 1$ so trivially $\mathcal{MM}_{\mathcal{L}ab}(x) \geq 1$.

STEP Suppose that at a particular point, the while loop has added n (≥ 0) arguments to `unproc_in` and that for each argument x that has been added to `unproc_in` (either by the while loop of lines 21-37 or by the for loop of lines 9-18) it holds that $\mathcal{MM}_{\mathcal{L}ab}(x) \geq 1$. We distinguish two cases:

- x was added to `unproc_in` previously. From the induction hypothesis it follows that $\mathcal{MM}_{\mathcal{L}ab}(x) \geq 1$ at the moment x was added. As Algorithm 1 does not change any value of $\mathcal{MM}_{\mathcal{L}ab}$ once it is assigned, it follows that $\mathcal{MM}_{\mathcal{L}ab}(x) \geq 1$ still holds at the current point.
- x is the argument that is currently being added to `unproc_in` (so $x = Z$ at line 30). This implies that Z is labelled `in` at line 31 and is numbered $\mathcal{MM}_{\mathcal{L}ab}(Y) + 1$ at line 32. Following line 26, it holds that $\mathcal{MM}_{\mathcal{L}ab}(Y) = \mathcal{MM}_{\mathcal{L}ab}(X) + 1$, with X being an `in` labelled attacker of Y that was added to `unproc_in` previously. We can therefore apply the induction hypothesis and obtain that $\mathcal{MM}_{\mathcal{L}ab}(X) \geq 1$, which together with the earlier observed facts that $\mathcal{MM}_{\mathcal{L}ab}(Z) = \mathcal{MM}_{\mathcal{L}ab}(Y) + 1$ (line 32) and $\mathcal{MM}_{\mathcal{L}ab}(Y) = \mathcal{MM}_{\mathcal{L}ab}(X) + 1$ (line 26) implies that $\mathcal{MM}_{\mathcal{L}ab}(Z) \geq 3$ which trivially implies that $\mathcal{MM}_{\mathcal{L}ab}(Z) \geq 1$. Hence (as $x = Z$) we obtain that $\mathcal{MM}_{\mathcal{L}ab}(x) \geq 1$.

□

Algorithm 1 (especially line 22 and line 30) implements a FIFO queue for the `in` labelled arguments it processes. This is an important difference with the algorithm of [14], which uses a set for this purpose. Using a set is fine if the aim is merely to compute a strongly admissible labelling (as is the case for [14] where the aim is to compute the grounded labelling). However, if the aim is also to compute the associated min-max numbering, having a set as the basic data structure could compromise the algorithm's correctness.

As an example, consider again the argumentation framework of Figure 1. Let F be the main argument. Now suppose that `unproc_in` is a set instead of a queue. In that case, at the start of the first iteration of the while loop (line 21) it holds that $\mathcal{L}ab = (\{A, D\}, \emptyset, \{B, C, E, F, G, H\})$, $\mathcal{MM}_{\mathcal{L}ab} = \{(A : 1), (D : 1)\}$ and `unproc_in` $= \{A, D\}$. At the first iteration of the while loop, an argument X from `unproc_in` is selected (line 22). As a set has no order, it would be possible to select A (so $X = A$). This then means that B gets labelled `out` and C gets labelled `in`. Hence, at the end of the first iteration of the while loop (and therefore at the start of the second iteration of the while loop) it holds that $\mathcal{L}ab = (\{A, C, D\}, \{B\}, \{E, F, G, H\})$, $\mathcal{MM}_{\mathcal{L}ab} = \{(A : 1), (B : 2), (C : 3), (D : 1)\}$ and `unproc_inn` $= \{C, D\}$. At the second iteration of the while loop, suppose C is the selected argument (so $X = C$). This means that E gets labelled `out` and F gets labelled `in`. Hence, at the moment the algorithm hits the main argument (F, at line 33) and terminates, it holds that $\mathcal{L}ab = (\{A, C, D, F\}, \{B, E\}, \{G, H\})$ and $\mathcal{MM}_{\mathcal{L}ab} = \{(A : 1), (B : 2), (C : 3), (D : 1), (E : 4), (F : 5)\}$. Unfortunately $\mathcal{MM}_{\mathcal{L}ab}$ is incorrect. This is because `out` labelled argument E is numbered 4, whereas its two `in` labelled attackers C and D are numbered 3 and 1, respectively, so the correct min-max number of E should be 2 instead of 4, which implies that the correct min-max number of F should be 3 instead of 5.

One of the conditions of a min-max numbering is that the min-max number of an `out` labelled argument should be the minimal value of its `in` labelled attackers, plus 1. This seems to require that the min-max number of the `in` labelled attackers is already known,

before assigning the min-max number of the `out` labelled argument. At the very least, it would seem that the min-max number of an `out` labelled argument would potentially need to be recomputed each time the min-max number of one of its `in` labelled attackers becomes known. Yet, Algorithm 1 does none of this. It determines the min-max number of an `out` labelled argument as soon as the min-max number of its first `in` labelled attacker becomes known (line 26) without waiting for the min-max number of any other `in` labelled attacker becoming available. Yet, Algorithm 1 still somehow manages to always yield the correct min-max numbering.

The key to understanding how Algorithm 1 manages to always yield the correct min-max numbering is that the `in` labelled arguments are processed in the order of their min-max numbers. That is, once an `in` labelled attacker is identified, any subsequently identified `in` labelled attacker will have a min-max number greater or equal to the first one and will therefore not change the minimal value (in the sense of Definition 8, first bullet point). This avoids having to recalculate the min-max number of an `out` labelled attacker once more of its `in` labelled arguments become available, therefore speeding up the algorithm.

To make sure that arguments are processed in the order of their min-max numbers, we need to apply a FIFO queue instead of the set that was applied by [14]. The following two lemmas (Lemma 4 and Lemma 5) state that the `in` labelled arguments are indeed added and removed to the queue in the order of their min-max numbers. These properties are subsequently used to prove the correctness of the computed min-max numbering (Lemma 6 and Theorem 10).

Lemma 4. *Let $AF = (Ar, att)$ be an argumentation framework and A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. The order in which arguments are added to* `unproc_in` *is non-descending w.r.t. $\mathcal{MM}_{\mathcal{L}ab}$. That is, if argument x_1 is added to* `unproc_in` *before argument x_2 is added to* `unproc_in`*, then $\mathcal{MM}_{\mathcal{L}ab}(x_1) \leq \mathcal{MM}_{\mathcal{L}ab}(x_2)$.*

Proof. We first observe that this property is satisfied just after finishing the for loop of lines 9-18. This is because the for loop makes sure that for each argument x, $\mathcal{MM}_{\mathcal{L}ab}(x) = 1$ (line 15) so it is trivially satisfied that if x_1 is added before x_2, then $\mathcal{MM}_{\mathcal{L}ab}(x_1) \leq \mathcal{MM}_{\mathcal{L}ab}(x_2)$. We proceed the proof by induction over the number of arguments added by the while loop (lines 21-37).

BASIS (n=0) Suppose no arguments have yet been added to `unproc_in` by the while loop. In that case, all arguments that have been added to `unproc_in` were added by the for loop (lines 9-18) for which we have observed that the property holds.

STEP (n+1) Suppose the property holds after n arguments have been added to `unproc_in` by the while loop. We now show that if the while loop adds another argument (n+1) to `unproc_in`, the property still holds. In the while loop, only line 30 adds an argument to `unproc_in`. Let Z_{new} be the argument (n+1) that is currently added and let Z_{old} be an argument that was previously added. We distinguish two cases:

1. Z_{old} has been added by the while loop (so at a previous run of line 30). Let Y_{new} be the `out` labelled attacker of Z_{new} at line 25 and Y_{old} be the `out` labelled attacker of Z_{old} at line 25. Let X_{new} be the `in` labelled attacker of Y_{new} at line 22 and let X_{old} be the `in` labelled attacker of Y_{old} at line 22. It holds that either

(a) X_{old} was added to `unproc_in` before X_{new} (at a previous iteration of the while loop, in which case it follows from our induction hypothesis that $\mathcal{MM}_{\mathcal{L}ab}(X_{old}) \leq \mathcal{MM}_{\mathcal{L}ab}(X_{new})$, or

(b) $X_{new} = X_{old}$, in which case it trivially holds that $\mathcal{MM}_{\mathcal{L}ab}(X_{old}) \leq \mathcal{MM}_{\mathcal{L}ab}(X_{new})$.

In either case, we obtain that $\mathcal{MM}_{\mathcal{L}ab}(X_{old}) \leq \mathcal{MM}_{\mathcal{L}ab}(X_{new})$. Furthermore, as it holds that
$\mathcal{MM}_{\mathcal{L}ab}(Y_{old}) = \mathcal{MM}_{\mathcal{L}ab}(X_{old}) + 1$ (line 26)
$\mathcal{MM}_{\mathcal{L}ab}(Y_{new}) = \mathcal{MM}_{\mathcal{L}ab}(X_{new}) + 1$ (line 26)
$\mathcal{MM}_{\mathcal{L}ab}(Z_{old}) = \mathcal{MM}_{\mathcal{L}ab}(Y_{old}) + 1$ (line 32)
$\mathcal{MM}_{\mathcal{L}ab}(Z_{new}) = \mathcal{MM}_{\mathcal{L}ab}(Y_{new}) + 1$ (line 32)
it follows that $\mathcal{MM}_{\mathcal{L}ab}(Z_{old}) \leq \mathcal{MM}_{\mathcal{L}ab}(Z_{new})$.

2. Z_{old} has been added by the for loop of lines 9-18. In that case, it holds that $\mathcal{MM}_{\mathcal{L}ab}(Z_{old}) = 1$ (line 15). As $\mathcal{MM}_{\mathcal{L}ab}(Z_{new}) \geq 1$ (Lemma 3) it directly follows that $\mathcal{MM}_{\mathcal{L}ab}(Z_{old}) \leq \mathcal{MM}_{\mathcal{L}ab}(Z_{new})$.

□

Lemma 5. *Let $AF = (Ar, att)$ be an argumentation framework and A an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. The order in which arguments are removed from* `unproc_in` *is non-descending w.r.t. $\mathcal{MM}_{\mathcal{L}ab}$. That is, if argument x_1 is removed from* `unproc_in` *before argument x_2 is removed from* `unproc_in`*, then $\mathcal{MM}_{\mathcal{L}ab}(x_1) \leq \mathcal{MM}_{\mathcal{L}ab}(x_2)$.*

Proof. This follows directly from Lemma 4, together with the fact that additions to and removals from `unproc_in` are done according to the FIFO (First In First Out) principle. □

We proceed to show the correctness of $\mathcal{MM}_{\mathcal{L}ab}$ in an inductive way. That is, we show that $\mathcal{MM}_{\mathcal{L}ab}$ is correct at the start of each iteration of the while loop. We then later need to do a bit of additional work to state that $\mathcal{MM}_{\mathcal{L}ab}$ is still correct at the moment we jump out of the while loop using the return statement.

Lemma 6. *Let $AF = (Ar, att)$ be an argumentation framework and A an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. At the start of each iteration of the while loop, it holds that $\mathcal{MM}_{\mathcal{L}ab}$ is a correct min-max numbering of $\mathcal{L}ab$.*

Proof. We prove this by induction over the number of loop iterations.

As for the basis of the induction (n=1), let us consider the first loop iteration. This is just after the for loop of lines 9-18 has finished. We need to prove that $\mathcal{MM}_{\mathcal{L}ab}$ is a correct min-max numbering of $\mathcal{L}ab$ According to the definition of a min-max numbering (Definition 8) we need to prove that for every x in Ar:

1. if $\mathcal{L}ab(x) =$ `in` then $\mathcal{MM}_{\mathcal{L}ab}(x) = max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks x and $\mathcal{L}ab(y) =$ `out`$\})+1$
Suppose $\mathcal{L}ab(x) =$ `in`. This means that x has been labelled `in` by the for loop of lines 9-18, which implies that x does not have any attackers and is numbered 1. That is, $\mathcal{MM}_{\mathcal{L}ab}(x) = 1$ and $max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks x and $\mathcal{L}ab(y) =$ `out`$\}) = 0$ (by definition). Therefore $\mathcal{MM}_{\mathcal{L}ab}(x) = max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks x and $\mathcal{L}ab(y) =$ `out`$\}) + 1$

2. if $\mathcal{L}ab(x) = \texttt{out}$ then $\mathcal{MM}_{\mathcal{L}ab}(x) = min(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y \text{ attacks } x \text{ and } \mathcal{L}ab(y) = \texttt{in}\})+1$
This is trivially the case, as at the end of the for loop (lines 9-18) no argument is labelled out.

As for the induction step, suppose that at the start of a particular loop iteration, $\mathcal{MM}_{\mathcal{L}ab}$ is a correct min-max numbering of $\mathcal{L}ab$. We need to prove that if there is a next loop iteration, then at the start of this next loop iteration it is still the case that $\mathcal{MM}_{\mathcal{L}ab}$ is a correct min-max numbering of $\mathcal{L}ab$. For this, we need to prove that at the end of the current loop iteration, for any $x \in Ar$ it holds that:

1. if $\mathcal{L}ab(x) = \texttt{in}$ then $\mathcal{MM}_{\mathcal{L}ab}(x) = max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y \text{ attacks } x \text{ and } \mathcal{L}ab(y) = \texttt{out}\})+1$
We distinguish two cases:

   (a) x was already labelled `in` at the start of the current loop iteration. Then, as $\mathcal{L}ab$ is an admissible labelling at each point of the algorithm (Lemma 2) each attacker y of x is labelled `out` by $\mathcal{L}ab$. These attackers are still labelled `out` at the end of the current loop iteration (once an argument is labelled `out`, it stays labelled `out`). Also, the value $\mathcal{MM}_{\mathcal{L}ab}(y)$ of these `out` labelled attackers remains unchanged. Hence, from the fact that $\mathcal{MM}_{\mathcal{L}ab}(x) = max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y \text{ attacks } x \text{ and } \mathcal{L}ab(y) = \texttt{out}\}) + 1$ at the start of the current iteration, it follows that $\mathcal{MM}_{\mathcal{L}ab}(x) = max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y \text{ attacks } x \text{ and } \mathcal{L}ab(y) = \texttt{out}\}) + 1$ at the end of the current iteration.

   (b) x became labelled `in` during the current loop iteration. In that case, x was labelled `in` at line 31 (with $Z = x$). So $Z = x$ in $\mathcal{MM}_{\mathcal{L}ab}(Z) = \mathcal{MM}_{\mathcal{L}ab}(Y) + 1$ (line 32). We therefore need to show that $\mathcal{MM}_{\mathcal{L}ab}(Y) = max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y \text{ attacks } Z \text{ and } \mathcal{L}ab(y) = \texttt{out}\})$. As Y is an `out` labelled attacker of Z, we already know that $max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y \text{ attacks } Z \text{ and } \mathcal{L}ab(y) = \texttt{out}\})$ will be *at least* $\mathcal{MM}_{\mathcal{L}ab}(Y)$. We now proceed to show that $max(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y \text{ attacks } Z \text{ and } \mathcal{L}ab(y) = \texttt{out}\})$ will be *at most* $\mathcal{MM}_{\mathcal{L}ab}(Y)$. That is, for each `out` labelled attacker y of Z we show that $\mathcal{MM}_{\mathcal{L}ab}(y) \leq \mathcal{MM}_{\mathcal{L}ab}(Y)$. Let Y' be an arbitrary `out` labelled attacker of Z. Let X be the `in` labelled attacker of Y (line 22 of the current loop iteration) and let X' be the `in` labelled attacker of Y' (line 22 of the current or a previous loop iteration). We distinguish two cases:

   - $X' = X$
     In that case, from the fact that $\mathcal{MM}_{\mathcal{L}ab}(Y) = \mathcal{MM}_{\mathcal{L}ab}(X) + 1$ (line 26) and $\mathcal{MM}_{\mathcal{L}ab}(Y') = \mathcal{MM}_{\mathcal{L}ab}(X') + 1$ (line 26) it follows that $\mathcal{MM}_{\mathcal{L}ab}(Y') = \mathcal{MM}_{\mathcal{L}ab}(Y)$ so (trivially) also that $\mathcal{MM}_{\mathcal{L}ab}(Y') \leq \mathcal{MM}_{\mathcal{L}ab}(Y)$.
   - $X' \neq X$
     As X was removed from `unproc_in` during the current loop iteration, it follows that X' was removed from `unproc_in` during one of the previous loop iterations. This means that X' was removed from `unproc_in` *before* X was removed from `unproc_in`, which implies (Lemma 5) that $\mathcal{MM}_{\mathcal{L}ab}(X') \leq \mathcal{MM}_{\mathcal{L}ab}(X)$. From the fact that $\mathcal{MM}_{\mathcal{L}ab}(Y) = \mathcal{MM}_{\mathcal{L}ab}(X)+1$ (line 26) and $\mathcal{MM}_{\mathcal{L}ab}(Y') = \mathcal{MM}_{\mathcal{L}ab}(X') + 1$ (line 26) it follows that $\mathcal{MM}_{\mathcal{L}ab}(Y') \leq \mathcal{MM}_{\mathcal{L}ab}(Y)$.

   As we now observed that $\mathcal{MM}_{\mathcal{L}ab}(x)$ is the correct min-max number of x at the moment it was assigned (line 32) we can use similar reasoning as at the previous point (point (a)) to obtain that it is still the correct min-max number at the end of the current loop iteration.

2. if $\mathcal{L}ab(x) = \texttt{out}$ then $\mathcal{MM}_{\mathcal{L}ab}(x) = min(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks x and $\mathcal{L}ab(y) = \texttt{in}\})+1$
We distinguish two cases:

   (a) x was already labelled `out` at the start of the current loop iteration. In that case, our induction hypothesis that the min-max numbers are correct at the start of the current loop iteration implies that $\mathcal{MM}_{\mathcal{L}ab}(x) = min(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks x and $\mathcal{L}ab(y) = \texttt{in}\}) + 1$ at the start of the current loop iteration. As the current loop iteraton does not change the value of $\mathcal{MM}_{\mathcal{L}ab}(x)$ (once a value for $\mathcal{MM}_{\mathcal{L}ab}(x)$ is assigned, the algorithm never changes it) this value will still be the same at the end of the current loop iteration. We therefore only need to verify that this value is still correct at the end of the current loop iteration. For this, we need to be sure that any newly `in` labelled argument (that is, an argument that became labelled `in` during the current loop iteration) does not change the value of $min(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks x and $\mathcal{L}ab(y) = \texttt{in}\})$. Let Z be a newly `in` labelled attacker of x (line 31). Then Z was added to the rear of `unproc_in` (line 30). Let Z' be an arbitrary `in` labelled attacker of x. We distinguish two cases:

   - $Z' = Z$
     In that case, it directly follows that $\mathcal{MM}_{\mathcal{L}ab}(Z') = \mathcal{MM}_{\mathcal{L}ab}(Z)$ so (trivially) also that $\mathcal{MM}_{\mathcal{L}ab}(Z') \leq \mathcal{MM}_{\mathcal{L}ab}(Z)$.
   - $Z' \neq Z$
     In that case, it follows that Z' was added to `unproc_in` *before* Z was added to `unproc_in`. Lemma 4 then implies that $\mathcal{MM}_{\mathcal{L}ab}(Z') \leq \mathcal{MM}_{\mathcal{L}ab}(Z)$.

   In both cases, we obtain that $\mathcal{MM}_{\mathcal{L}ab}(Z') \leq \mathcal{MM}_{\mathcal{L}ab}(Z)$. This means that whenever x gets a new `in` labelled attacker $min(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks x and $Lab(y) = \texttt{in}\})$ does not change. Therefore, the value of $\mathcal{MM}_{\mathcal{L}ab}(x)$ is still the correct min-max number of x at the end of the current loop iteration.

   (b) x became labelled `out` during the current loop iteration. This can only have happened at line 25, so $x = Y$. $\mathcal{MM}_{\mathcal{L}ab}(Y)$ is then assigned $\mathcal{MM}_{\mathcal{L}ab}(X) + 1$ at line 26. In order for $\mathcal{MM}_{\mathcal{L}ab}(Y)$ to be a correct min-max number, we need to verify that $\mathcal{MM}_{\mathcal{L}ab}(X) = min(\{\mathcal{MM}_{\mathcal{L}ab}(y) | y$ attacks Y and $\mathcal{L}ab(y) = \texttt{in}\})$. This is the case because at line 25, X is the only `in` labelled attacker of Y (otherwise Y would have been labelled `out` before). As we have observed that $\mathcal{MM}_{\mathcal{L}ab}(Y)$ is the correct min-max value at the moment it was assigned, we can use similar reasoning as at the previous point (point (a)) to obtain that it is still the correct min-max number at the end of the current loop iteration.

□

In order for a labelling to be strongly admissible, its min-max numbering has to contain natural numbers only (no ∞). We therefore proceed to show the absence of ∞ in an inductive way. That is, we show the absence of ∞ at the start of each iteration of the while loop. We then later need to do a bit of additional work to show the absence of ∞ at the moment we jump out of the while loop using the return statement.

Lemma 7. *Let $AF = (Ar, att)$ be an argumentation framework and let A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. At the start of each iteration of the while loop at lines 21-37, it holds that for each* `in` *or* `out` *labelled argument $x \in Ar$, $\mathcal{MM}_{\mathcal{L}ab}(x)$ is a natural number (no ∞)*

Proof. We prove this by induction over the number of iterations of the while loop at lines 21-37.
As for the basis of induction(n=1), let us consider the first loop iteration. This is just after the for loop at lines 9-18 has finished. We need to prove that for each `in` or `out` labelled argument $x \in Ar$, $\mathcal{MM}_{\mathcal{L}ab}(x)$ is a natural number. We therefore need to prove that:

1. if $\mathcal{L}ab(x) = \texttt{in}$ then $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$
   Let x be labelled `in` by the for loop at lines 9-18. This can only have happened at line 14. According to line 15, it then follows that $\mathcal{MM}_{\mathcal{L}ab}(x) = 1$. Hence $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$.

2. if $\mathcal{L}ab(x) = \texttt{out}$ then $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$
   This is trivially the case as the end of the for loop at lines 9-18, no argument is labelled `out`.

As for the induction step, suppose that at the start of a particular loop iteration, for each `in` or `out` labelled argument $x \in Ar$, $\mathcal{MM}_{\mathcal{L}ab}(x)$ is a natural number. We therefore need to prove that by the end of the iteration (and therefore also at the start of the next loop iteration) it holds that:

1. if $\mathcal{L}ab(x) = \texttt{in}$ then $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$
   We distinguish two cases:

   - x was already labelled `in` at the start of the current loop iteration. From the induction hypothesis it follows that $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$ at the start of the current iteration. As Algorithm 1 does not change any values of $\mathcal{MM}_{\mathcal{L}ab}$ once these have been assigned, it follows that $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$ will hold at the end of the current loop iteration.
   - x became labelled `in` during the current loop iteration. In the case x was labelled `in` at line 31 (with $Z = x$). Following line 32, $\mathcal{MM}_{\mathcal{L}ab}(X) = \mathcal{MM}_{\mathcal{L}ab}(Y) + 1$. According to line 26, $\mathcal{MM}_{\mathcal{L}ab}(Y) = \mathcal{MM}_{\mathcal{L}ab}(X) + 1$ with X being an attacker of Y that became labelled `in` during a previous iteration of the while loop. From our induction hypothesis it follows that $\mathcal{MM}_{\mathcal{L}ab}(X) \neq \infty$. As $\mathcal{MM}_{\mathcal{L}ab}(Y) = \mathcal{MM}_{\mathcal{L}ab}(X)+1$ it follows that $\mathcal{MM}_{\mathcal{L}ab}(Y) \neq \infty$. As $\mathcal{MM}_{\mathcal{L}ab}(Z) = \mathcal{MM}_{\mathcal{L}ab}(Y)+1$ it follows that $\mathcal{MM}_{\mathcal{L}ab}(Z) \neq \infty$. That is (as $x = Z$) $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$.

2. if $\mathcal{L}ab(x) = \texttt{out}$ them $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$
   We distinguish two cases:

   - x was already labelled `out` at the start of the current loop iteration. From the induction hypothesis it follows that $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$ at the start of the current iteration. As Algorithm 1 does not change any values of $\mathcal{MM}_{\mathcal{L}ab}$ once these have been assigned, it follows that $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$ will hold at the end of the current loop iteration.
   - x became labelled `out` during the current loop iteration. This can only have happened at line 25 (with $x = Y$). $\mathcal{MM}_{\mathcal{L}ab}(Y)$ is then assigned $\mathcal{MM}_{\mathcal{L}ab}(X) + 1$ at line 26, with X being an attacker of Y that became labeled `in` during a previous iteration of the while loop. From the induction hypothesis, it follows that $\mathcal{MM}_{\mathcal{L}ab}(X) \neq \infty$. As $\mathcal{MM}_{\mathcal{L}ab}(Y) = \mathcal{MM}_{\mathcal{L}ab}(X) + 1$ (line 26) it follows that $\mathcal{MM}_{\mathcal{L}ab}(Y) \neq \infty$. That is (as $x = Y$) $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$.

□

Although most of our results so far are about the algorithm itself, we also need an additional theoretical property of grounded semantics, stated in the following lemma.

Lemma 8. *Let $AF = (Ar, att)$ be an argumentation framework. It holds that the grounded labelling of AF is the only argument labelling that is both strongly admissible and complete.*

Proof. First of all, it has been observed that the grounded labelling is both strongly admissible [7] and complete (Definition 7). We proceed to prove that it is also the *only* argument labelling that is both strongly admissible and complete. Let $\mathcal{L}ab$ be an argument labelling that is both strongly admissible and complete. From the fact that the grounded labelling ($\mathcal{L}ab_{gr}$) is the unique biggest strongly admissible labelling [7] it follows that $\mathcal{L}ab \sqsubseteq \mathcal{L}ab_{gr}$. From the fact that the grounded labelling is the unique smallest complete labelling (Definition 7) it follows that $\mathcal{L}ab_{gr} \sqsubseteq \mathcal{L}ab$. Together, this implies that $\mathcal{L}ab = \mathcal{L}ab_{gr}$. □

If we would not finish the algorithm after hitting the main argument and instead continue to execute the algorithm until `unproc_in` is empty, we would be computing the grounded labelling with its associated min-max numbering as stated by the following lemma.

Lemma 9. *If in Algorithm 1 one would comment out line 16 and line 33 and add the following line (line 41) at the end:*
return $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$
then the output of the thus modified algorithm would be the grounded labelling $\mathcal{L}ab$ of AF, together with its min-max numbering $\mathcal{MM}_{\mathcal{L}ab}$.

Proof. We first observe that $\mathcal{L}ab$ is a strongly admissible labelling. This follows from the facts that

1. $\mathcal{L}ab$ is an admissible labelling
   This can be proved in a similar way as Lemma 2.

2. $\mathcal{MM}_{\mathcal{L}ab}$ is a correct min-max numbering of $\mathcal{L}ab$
   This can be proved in a similar way as Lemma 6.

3. $\mathcal{MM}_{\mathcal{L}ab}$ does not contain ∞ (natural numbers only)
   This can be proved in a similar way as 7.

We proceed to show that $\mathcal{L}ab$ is also a complete labelling. For this, we first show the following two properties:

1. if $\mathcal{L}ab(y) =$ `out` for each attacker y of x then $\mathcal{L}ab(x) =$ `in`
   Suppose $\mathcal{L}ab(y) =$ `out` for each attacker of x. This means that at the end of the algorithm, it holds that `undec_pre`$(x) = 0$ which implies that x became labelled `in` (either at line 14 or at line 31) at the moment when `undec_pre`(x) became 0 (at either line 11 or line 28)

2. if $\mathcal{L}ab(y) =$ `in` for some attacker y of x then $\mathcal{L}ab(x) =$ `out`
   Suppose $\mathcal{L}ab(y) =$ `in` for some attacker y of x. At the end of the algorithm, it holds that `unproc_in` is empty. As each `in` labelled argument in $\mathcal{L}ab$ (such as y) was added to `unproc_in` when it became labelled `in`, this implies that each `in` labelled argument

in $\mathcal{L}ab$ (in particular y) was subsequently removed from `unproc_in`. This removal can only have happened at line 23, which implies (line 24 and 25) that each argument that is attacked by y (in particular x) is labelled `out`.

Suppose $\mathcal{L}ab(x) = \texttt{undec}$. From point 1 and the fact that $\mathcal{L}ab(x) \neq \texttt{in}$ we obtain that (3) there is an attacker y of x such that $\mathcal{L}ab(y) \neq \texttt{out}$. From point 2 and the fact that $\mathcal{L}ab(x) \neq \texttt{out}$ we obtain that (4) there is no attacker y of x such that $\mathcal{L}ab(y) = \texttt{in}$. From point (3) and (4) it follows that

- if $\mathcal{L}ab(x) = \texttt{undec}$ then there is a y that attacks x such that $\mathcal{L}ab(y) = \texttt{undec}$ and for each y that attacks x such that $\mathcal{L}ab(y) \neq \texttt{undec}$ it holds that $\mathcal{L}ab(y) = \texttt{out}$

This, together with the fact that $\mathcal{L}ab$ is an admissible labelling implies that $\mathcal{L}ab$ is a complete labelling (Definition 5).

From the thus obtained facts that $\mathcal{L}ab$ is both a strongly admissible labelling and a complete labelling it follows (Lemma 8) that $\mathcal{L}ab$ is the grounded labelling. □

Using the above lemmas, we now proceed to show the correctness of the algorithm.

Theorem 10. *Let $AF = (Ar, att)$ be an argumentation framework and let A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. Let $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$ be the output of the algorithm. It holds that $\mathcal{L}ab$ is a strongly admissible labelling that labels A `in` and has $\mathcal{MM}_{\mathcal{L}ab}$ as its min-max numbering.*

Proof. We first observe that as A is in the grounded extension of AF, the modified algorithm of Lemma 9 would have produced the grounded labelling, which labels A in. This implies that at some moment in Algorithm 1, line 16 or line 33 is trigered, meaning that $\mathcal{L}ab$ as returned by Algorithm 1 labels A `in`. At the moment the return statement of line 16 or 33 is triggered, it holds that:

1. $\mathcal{L}ab$ is an admissible labelling.
   This follows directly from Lemma 2.

2. $\mathcal{MM}_{\mathcal{L}ab}$ is a correct min-max numbering of $\mathcal{L}ab$.
   To see that this is the case, we distinguish two cases:

   (a) The return statement that was triggered was the one at line 16. In that case, $\mathcal{MM}_{\mathcal{L}ab}$ is the correct min-max numbering of $\mathcal{L}ab$. The proof is similar to the first half of the proof of Lemma 6.

   (b) The return statement that was triggered was the one at line 33. In that case, Lemma 6 tells us that the value of $\mathcal{MM}_{\mathcal{L}ab}$ at the start of the last iteration of the while loop was a correct min-max numbering of the value of $\mathcal{L}ab$ at the start of the last iteration of the while loop. We then need to show that the value of $\mathcal{MM}_{\mathcal{L}ab}$ at the time of the return statement (line 33) is still a correct min-max numbering of the value of $\mathcal{L}ab$ at the time of the return statement (line 33). This can be proved in a similar way as is done in the second half of the proof of Lemma 6 (instead of going until the end of the loop iteration, one goes until the moment the return statement of line 33 is triggered).

3. $\mathcal{MM}_{\mathcal{L}ab}$ numbers each `in` or `out` labelled argument with a natural number (no ∞).
   To see that this is the case, we distinguish two cases:

(a) The return statement that was triggered is the one at line 16. In that case, for each `in` or `out` labelled argument x it holds that $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$. The proof is similar to the first half (the basis) of the proof of Lemma 7.

(b) The return statement that was triggered is the one at line 33. In that case, Lemma 7 tells us that at the start of the last iteration of the while loop, $\mathcal{MM}_{\mathcal{L}ab}(x) \neq \infty$ for each argument x that was labelled `in` or `out`. We need to show that this is still the case at the time of the return statement (line 33). This can be proved in a similar was as is done in the second half of the proof of Lemma 7 (instead of going until the end of the loop iteration, the idea is to go until the return statement of line 33 is triggered).

□

It turns out that the algorithm runs in polynomical time (more specific, in cubic time).

Theorem 11. *Let $AF = (Ar, att)$ be an argumentation framework and let A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 1. Let $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$ be the output of the algorithm. It holds that Algorithm 1 computes $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$ in $O(n)^3$ time*

Proof. Let n be the number of arguments in AF (that is, $n = |Ar|$). The for loop (lines 9-18) can have at most n iterations. The while loop (lines 21-37) can also have at most n iterations. This is because each iteration of the while loop removes an argument from `unproc_in`, which can be done n times at most, given that no argument can be added to `unproc_in` more than once (this follows from line 31 and line 27). For each iteration of the while loop, the outer for loop (lines 24-36) will run at most n times. Also, for each iteration of the outer for loop, the inner for loop (lines 27-35) will run at most n times. This means that the total number of instructions executed by the while loop is of the order n^3 at most. This, combined with the earlier observed fact that the for loop of lines 9-18 runs at most n times brings the total complexity of Algorithm 1 to $O(n + n^3) = O(n^3)$. □

3.2 Algorithm 2

The basic idea of Algorithm 2 is to prune the part of the strongly admissible labelling that is not needed, by identifying the part that actually is needed. This is done in a top-down way, starting by including the main argument (which is labelled `in`), then including all its attackers (which are labelled `out`), for each of which a minimally numbered `in` labelled attacker is included, etc. The idea is to keep doing this until reaching the (`in` labelled) arguments that have no attackers. Each argument that has not been included by this process is unnecessary for the strongly admissible labelling and can be made `undec`, resulting in a labelling that is smaller or equal to the strongly labelling labelling one started with.

To see how the algorithm works, consider again the argumentation framework of Figure 1. Let C be the main argument. Suppose the input labelling $\mathcal{L}ab_I$ is $(\{A, C, D\}, \{B\}, \{E, F, G, H\})$ and its associated input labelling numbering $\mathcal{MM}_{\mathcal{L}abO}$ is $\{(A : 1), (B : 2), (C : 3), (D : 1)\}$.[2] At the start of the first iteration of the while loop, it holds that $\mathcal{L}ab_O = (\{C\}, \emptyset, \{A, B, D, E, F, G, H\})$,

[2]The reader may have noticed that this was the output of Algorithm 1 for the example that was given in Section 3.1.

Algorithm 2 Prune a strongly admissible labelling that labels A `in` and its associated min-max numbering.

Input: An argumentation framework $AF = (Ar, att)$,
an argument $A \in Ar$ that is in the grounded extension of AF, A strongly admissible labelling $\mathcal{L}ab_I$ where $A \in \texttt{in}(\mathcal{L}ab_I)$ and the associated min-max numbering $\mathcal{MM}_{\mathcal{L}abI}$.
Output: A strongly admissible labelling $\mathcal{L}ab_O \sqsubseteq \mathcal{L}ab_I$ where $A \in \texttt{in}(\mathcal{L}ab_O)$,
the associated min-max numbering $\mathcal{MM}_{\mathcal{L}abO}$.

1: // We start with the type definitions
2: $\mathcal{L}ab_O : Ar \rightarrow \{\texttt{in}, \texttt{out}, \texttt{undec}\}$
3: $\mathcal{MM}_{\mathcal{L}abO} : \texttt{in}(\mathcal{L}ab) \cup \texttt{out}(\mathcal{L}ab) \rightarrow \mathbb{N} \cup \{\infty\}$
4: $\texttt{unproc\_in} : [X_1, ...X_n]$ ($X_i \in Ar$ for each $1 \leq i \leq n$) // list of arguments
5: // Initialize $\mathcal{L}ab_O$ and include the main argument
6: $\mathcal{L}ab_O \leftarrow (\emptyset, \emptyset, Ar)$ // $\mathcal{L}ab_O$ becomes the all-`undec` labelling
7: $\texttt{unproc\_in} \leftarrow [A]$
8: $\mathcal{L}ab_O(A) \leftarrow \texttt{in}$
9: $\mathcal{MM}_{\mathcal{L}abO}(A) \leftarrow \mathcal{MM}_{\mathcal{L}abI}(A)$
10:
11: // Next, process the other arguments in a top-down way
12: **while** `unproc_in` is not empty **do**
13: let X be the argument at the front of `unproc_in`
14: remove X from `unproc_in`
15: **for** each attacker Y of X **do**
16: $\mathcal{L}ab_O(Y) \leftarrow \texttt{out}$
17: $\mathcal{MM}_{\mathcal{L}abO}(Y) \leftarrow \mathcal{MM}_{\mathcal{L}abI}(Y)$
18: **if** there is no minimal (w.r.t $\mathcal{MM}_{\mathcal{L}abI}$) `in` labelled (w.r.t $\mathcal{L}ab_I$) attacker of Y that is also labelled `in` by Lab_O **then**
19: Let Z be a minimal (w.r.t $\mathcal{MM}_{\mathcal{L}abI}$) in labelled (w.r.t Lab_I) attacker of Y
20: Add Z to the rear of `unproc_in`
21: $\mathcal{L}ab_O(Z) \leftarrow \texttt{in}$
22: $\mathcal{MM}_{\mathcal{L}abO}(Z) \leftarrow \mathcal{MM}_{\mathcal{L}abI}(Z)$
23: **end if**
24: **end for**
25: **end while**

$\mathcal{MM}_{\mathcal{L}abO} = \{(C : 1)\}$ and `unproc_in` $= [C]$. The first iteration of the while loop then removes C from `unproc_in` (line 14), labels its attacker B `out` (line 16), numbers B with 2 (line 17), adds A to `unproc_in` (line 20), labels A `in` (line 21) and numbers A with 1 (line 22). The second iteration of the while loop then removes A from `unproc_in` (line 14). However, as A does not have any attackers, the for loop (lines 15-24) is skipped. As `unproc_in` is now empty, the while loop is finished and the algorithm terminates, with $\mathcal{L}ab_O = (\{A, C\}, \{B\}, \{D, E, F, G, H\})$ and $\mathcal{MM}_{\mathcal{L}abO} = \{(A : 1), (B : 2), (C : 3)\}$ being its results.

We now proceed to prove some of the formal properties of the algorithm. The first property to be proved is termination.

Theorem 12. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF, $\mathcal{L}ab_I$ be a strongly admissible labelling where A is labelled in and $\mathcal{MM}_{\mathcal{L}abI}$ be the associated min-max numbering. Let AF, A, $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be given as input to Algorithm 2. It holds that the algorithm terminates.*

Proof. At the while loop of lines 12-25, we observe that only a finite number of arguments can be added to `unproc_in`. This is because there are only a finite number of arguments in the argumentation framework, and because no argument can be added to `unproc_in` more than once. The latter can be seen as follows. Following line 18, only arguments that are not already labelled `in` by $\mathcal{L}ab_O$ can be added to `unproc_in`. Also, if an argument is labelled `in` by $\mathcal{L}ab_O$, it will stay labelled `in` by $\mathcal{L}ab_O$ as there is nothing in the algorithm that will change it. Following line 14, at each iteration of the while loop, an argument is removed from `unproc_in`. From the fact that only a finite number of arguments can be added to `unproc_in`, it directly follows that only a finite number of arguments can be removed from `unproc_in`. Hence, the while loop can run only a finite number of times before `unproc_in` is empty. Hence, Algorithm 2 terminates. □

Next, we prove that the labelling that is yielded by the algorithm is smaller or equal to the labelling the algorithm started with.

Theorem 13. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF, $\mathcal{L}ab_I$ be a strongly admissible labelling where A is labelled* `in` *and $\mathcal{MM}_{\mathcal{L}abI}$ be the associated min-max numbering. Let AF, A, $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be given as input to Algorithm 2. Let $\mathcal{L}ab_O$ and $\mathcal{MM}_{\mathcal{L}abO}$ be the output of Algorithm 2. It holds that $\mathcal{L}ab_O \sqsubseteq \mathcal{L}ab_I$*

Proof. In order to prove that $\mathcal{L}ab_O \sqsubseteq \mathcal{L}ab_I$, we must show:

1. $\texttt{in}(\mathcal{L}ab_O) \subseteq \texttt{in}(\mathcal{L}ab_I)$
   Let x be an arbitrary argument that is labelled `in` by $\mathcal{L}ab_O$. We distinguish two cases:

   - x became labelled `in` at line 8. Therefore, it follows x is the argument in question (with $x = A$). Therefore, A is also labelled `in` by $\mathcal{L}ab_I$. That is, x is also labelled in by $\mathcal{L}ab_I$
   - x became labelled `in` at line 21. According to line 19, x is a minimal `in` labelled attacker of some `out` labelled argument y w.r.t $\mathcal{L}ab_I$. Therefore, x is also labelled `in` within $\mathcal{L}ab_I$.

2. $\texttt{out}(\mathcal{L}ab_O) \subseteq \texttt{out}(\mathcal{L}ab_I)$
   Let y be an arbitrary `out` labelled argument within $\mathcal{L}ab_O$. It follows that y must have been labelled `out` at line 16 (so $y = Y$). From line 15, it follows that Y attacks X, which was removed from `unproc_in` at line 14. This means that X at some point was added to `unproc_in`, which could only have happened at line 7 or line 20. In either case, it holds that $\mathcal{L}ab_O(X) = \texttt{in}$ (line 8 or 21, respectively). From point 1 above, we infer that $\mathcal{L}ab_I(X) = \texttt{in}$. As $\mathcal{L}ab_I$ is an admissible labelling, it follows that each attacker of X (such as Y) is labelled `out` by $\mathcal{L}ab_I$. As $y = Y$ it directly follows that y is labelled `out` by $\mathcal{L}ab_I$.

□

Next, we prove that the output of the algorithm is at least admissible (the fact that it is also *strongly* admissible is proved further on).

Theorem 14. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF, $\mathcal{L}ab_I$ be a strongly admissible labelling where A is labelled* `in` *and $\mathcal{MM}_{\mathcal{L}abI}$ be the associated min-max numbering. Let AF, A, $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be given as input to Algorithm 2. Let $\mathcal{L}ab_O$ and $\mathcal{MM}_{\mathcal{L}abO}$ be the output of Algorithm 2. It holds that $\mathcal{L}ab_O$ is an admissible labelling that labels A* `in`.

Proof. The fact that $\mathcal{L}ab_O$ labels A `in` follows from line 8. In order to prove that $\mathcal{L}ab_O$ is an admissible labelling, we must show that it satisfies the following two properties (Definition 5):

1. if $\mathcal{L}ab_O(x) = \texttt{in}$, then for each y that attacks x it holds that $\mathcal{L}ab_O(y) = \texttt{out}$
   Let x be an arbitrary `in` labelled argument within $\mathcal{L}ab_O$. This means that x became `in` at line 8 or line 21. In either case, x has been added to `unproc_in` (at line 7 or line 20, respectively). Once the algorithm is terminated, `unproc_in` has to be empty. This means that at some point, x must have been removed from `unproc_in`. This can only have happened at line 14, which implies that (lines 15 and 16) each attacker y of x is labelled `out` by $\mathcal{L}ab_O$.

2. if $\mathcal{L}ab_O(x) = \texttt{out}$, then there exists a y that attacks x such that $\mathcal{L}ab_O(y) = \texttt{in}$
   Let x be an arbitrary `out` labelled argument within $\mathcal{L}ab_O$. It follows that x has been labelled `out` at line 16 ($x = Y$). According to Theorem 13, Y is also labelled `out` by $\mathcal{L}ab_I$. Since $\mathcal{L}ab_I$ is an admissible labelling of AF, at least one of Y's attackers is labelled `in` by $\mathcal{L}ab_I$. Following lines 18-21, a minimal (w.r.t $\mathcal{MM}_{\mathcal{L}abI}$) `in` labelled attacker of Y (w.r.t $\mathcal{L}ab_I$), y has been labelled `in` by $\mathcal{L}ab_O$. That is, there exists a y that attacks x such that $\mathcal{L}ab_O(y) = \texttt{in}$

□

Next, we prove that the algorithm does not change the min-max values of the arguments it labels `in` or `out`.

Lemma 15. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF, $\mathcal{L}ab_I$ be a strongly admissible labelling where A is labelled* `in` *and $\mathcal{MM}_{\mathcal{L}abI}$ be the associated min-max numbering. Let AF, A, $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be given as input to Algorithm 2. Let $\mathcal{L}ab_O$ and $\mathcal{MM}_{\mathcal{L}abO}$ be the output of Algorithm 2. It holds that for each argument x that is labelled* `in` *or* `out` *by $\mathcal{L}ab_O$, $\mathcal{MM}_{\mathcal{L}abO}(x) = \mathcal{MM}_{\mathcal{L}abI}(x)$.*

Proof. This follows from Theorem 13 and lines 9, 17 and 22 of Algorithm 2. □

Next, we prove that the output numbering is actually the correct min-max numbering of the output labelling.

Theorem 16. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF, $\mathcal{L}ab_I$ be a strongly admissible labelling where A is labelled* `in` *and $\mathcal{MM}_{\mathcal{L}abI}$ be the associated min-max numbering. Let AF, A, $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be given as input to Algorithm 2. Let $\mathcal{L}ab_O$ and $\mathcal{MM}_{\mathcal{L}abO}$ be the output of Algorithm 2. It holds that $\mathcal{MM}_{\mathcal{L}abO}$ is the correct min-max numbering of $\mathcal{L}ab_O$.*

Proof. Since $\mathcal{L}ab_O$ has been shown to be admissible (Theorem 14), we need to show that (Definition 8):

1. if $\mathcal{L}ab_O(x) = \texttt{in}$ then $\mathcal{MM}_{\mathcal{L}abO}(x) = max(\{\mathcal{MM}_{\mathcal{L}abO}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_O(y) = \texttt{out}\}) + 1$
   Let x be an arbitrary `in` labelled argument within $\mathcal{L}ab_O$. According to Lemma 15, $\mathcal{MM}_{\mathcal{L}abO}(x) = \mathcal{MM}_{\mathcal{L}abI}(x)$. Since $\mathcal{MM}_{\mathcal{L}abI}$ is the correct min-max numbering of $\mathcal{L}ab_I$, $\mathcal{MM}_{\mathcal{L}abI}(x) = max(\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{out}\}) + 1$. It follows that $\mathcal{MM}_{\mathcal{L}abO}(x) = max(\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{out}\}) + 1$. From the fact that $\mathcal{L}ab_O(x) = \texttt{in}$ and that $\mathcal{L}ab_O \sqsubseteq \mathcal{L}ab_I$ (Theorem 13) it follows that $\mathcal{L}ab_I(x) = \texttt{in}$. As both $\mathcal{L}ab_I$ and $\mathcal{L}ab_O$ are admissible labellings, it holds that in both labellings, all attackers of x are labelled `out`. It follows that $\{y \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{out}\} = \{y \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_O(y) = \texttt{out}\}$. From Lemma 15, it then follows that $\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{out}\} = \{\mathcal{MM}_{\mathcal{L}abO}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_O(y) = \texttt{out}\}$. Therefore, from the earlier observed fact that $\mathcal{MM}_{\mathcal{L}abO}(x) = max(\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{out}\})+1$ we obtain that $\mathcal{MM}_{\mathcal{L}abO}(x) = max(\{\mathcal{MM}_{\mathcal{L}abO}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_O(y) = \texttt{out}\}) + 1$.

2. if $\mathcal{L}ab_O(x) = \texttt{out}$ then $\mathcal{MM}_{\mathcal{L}abO}(x) = min(\{\mathcal{MM}_{\mathcal{L}abO}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_O(y) = \texttt{in}\}) + 1$
   Let x be an arbitrary `out` labelled argument within $\mathcal{L}ab_O$. As $\mathcal{MM}_{\mathcal{L}abI}$ is the correct min-max numbering of $\mathcal{L}ab_I$ it holds that $\mathcal{MM}_{\mathcal{L}abI}(x) = min(\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{in}\}) + 1$. As $\mathcal{MM}_{\mathcal{L}abO}(x) = \mathcal{MM}_{\mathcal{L}abI}(x)$ (Lemma 15), it follows that $\mathcal{MM}_{\mathcal{L}abO}(x) = min(\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{in}\}) + 1$. The fact that $\mathcal{L}ab_O(x) = \texttt{out}$ means that x must have become labelled out at line 16. From lines 18-22, it follows that $\mathcal{L}ab_O$ will also contain a minimal (w.r.t. $\mathcal{MM}_{\mathcal{L}abI}$) `in` labelled attacker (w.r.t. $\mathcal{L}ab_I$). This implies that $min(\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{in}\}) = min(\{\mathcal{MM}_{\mathcal{L}abO}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_O(y) = \texttt{in}\})$. So from the earlier obtained fact that $\mathcal{MM}_{\mathcal{L}abO}(x) = min(\{\mathcal{MM}_{\mathcal{L}abI}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_I(y) = \texttt{in}\}) + 1$, it follows that $\mathcal{MM}_{\mathcal{L}abO}(x) = min(\{\mathcal{MM}_{\mathcal{L}abO}(y) \mid y \text{ attacks } x \text{ and } \mathcal{L}ab_O(y) = \texttt{in}\}) + 1$.

□

We are now ready to state one of the main results of the current section: the output labelling is strongly admissible.

Theorem 17. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF, $\mathcal{L}ab_I$ be a strongly admissible labelling where A is labelled* `in` *and*

$\mathcal{MM}_{\mathcal{L}abI}$ be the associated min-max numbering. Let AF, A, $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be given as input to Algorithm 2. Let $\mathcal{L}ab_O$ and $\mathcal{MM}_{\mathcal{L}abO}$ be the output of Algorithm 2. It holds that $\mathcal{L}ab_O$ is a strongly admissible labelling of AF.

Proof. In order to show that $\mathcal{L}ab_O$ is strongly admissible, we need to show that $\mathcal{L}ab_O$ is an admissible labelling for which the min-max numbering does not contain any ∞ (Definition 4). First, we observe that $\mathcal{L}ab_O$ is an admissible labelling of AF (Theorem 14) with $\mathcal{MM}_{\mathcal{L}abO}$ as its correct min-max numbering (Theorem 16). As $\mathcal{L}ab_I$ is a strongly admissible labelling of AF, its min-max numbering does not contain ∞. This, together with the fact that $\mathcal{L}ab_O \sqsubseteq \mathcal{L}ab_I$ (Theorem 13) and the fact that for each `in` or `out` labelled argument x by $\mathcal{L}ab_O$, x is assigned the same min-max numbering by $\mathcal{MM}_{\mathcal{L}abO}$ as by $\mathcal{MM}_{\mathcal{L}abI}$ (Lemma 15) implies that $\mathcal{MM}_{\mathcal{L}abO}$ does not contain any ∞. Hence, we observe that $\mathcal{L}ab_O$ is an admissible labelling whose min-max numbering $\mathcal{MM}_{\mathcal{L}abO}$ does not contain ∞. That is, $\mathcal{L}ab_O$ is a strongly admissible labelling of AF. □

It turns out that the algorithm runs in polynomial time (more specific, in cubic time).

Theorem 18. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF, $\mathcal{L}ab_I$ be a strongly admissible labelling where A is labelled in and $\mathcal{MM}_{\mathcal{L}abI}$ be the correct min-max numbering of $\mathcal{L}ab_I$. Let AF, A, $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be given as input to Algorithm 2. Let $\mathcal{L}ab_O$ and $\mathcal{MM}_{\mathcal{L}abO}$ be the output of Algorithm 2. It holds that Algorithm 2 computes $\mathcal{L}ab_O$ and $\mathcal{MM}_{\mathcal{L}abO}$ in $O(n)^3$ time.*

Proof. Let n be the number of arguments in AF (that is, $n = |Ar|$). The while loop (lines 12-25) can have at most n iterations. This is because each iteration of the while loop removes an argument from `unproc_in`, which can be done n times at most, given that no argument can be added to `unproc_in` more than once (this follows from lines 18-21). For each iteration of the while loop, the for loop (lines 15-24) will run at most n times. In addition, for each iteration of the for loop, a sequential search (lines 18-19) will run at most n times. This means that the total number of instructions executed by the while loop is of the order n^3 at most. Therefore, Algorithm 2 computes $\mathcal{L}ab_O$ in $\mathrm{O}(n)^3$ time. □

3.3 Algorithm 3

The idea of Algorithm 3 is to combine Algorithm 1 and Algorithm 2, by running them in sequence. That is, the output of Algorithm 1 is used as input for Algorithm 2.

As an example, consider again the argumentation framework of Figure 1. Let C be the main argument. Running Algorithm 1 yields a labelling $(\{A, C, D\}, \{B\}, \{E, F, H, H\})$ with associated numbering $\{(A : 1), (B : 2), (C : 3), (D : 1)\}$ (as explained in Section 3.1). Feeding this labelling and numbering into Algorithm 2 then yields an output labelling $(\{A, C\}, \{B\}, \{D, E, F, G, H\})$ with associated output numbering $\{(A : 1), (B : 2), (C : 3)\}$ (as explained in Section 3.2).

Given the properties of Algorithm 1 and Algorithm 2, we can prove that Algorithm 3 terminates, correctly computes a strongly admissible labelling and its associated min-max numbering, and runs in polynomial time (more specific, in cubic time).

Theorem 19. *Let $AF = (Ar, att)$ be an argumentation framework and A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 3. It holds that the algorithm terminates.*

Algorithm 3 Construct a relatively small strongly admissible labelling that labels A `in` and its associated min-max numbering.

Input: An argumentation framework $AF = (Ar, att)$,
an argument $A \in Ar$ that is in the grounded extension of AF.
Output: A strongly admissible labelling $\mathcal{L}ab$ where $A \in \texttt{in}(\mathcal{L}ab)$, the associted min-max numbering $\mathcal{MM}_{\mathcal{L}ab}$.

1: run Algorithm 1
2: $\mathcal{L}ab_I \leftarrow \mathcal{L}ab$
3: $\mathcal{MM}_{\mathcal{L}abI} \leftarrow \mathcal{MM}_{\mathcal{L}ab}$
4: run Algorithm 2
5: $\mathcal{L}ab \leftarrow \mathcal{L}ab_O$
6: $\mathcal{MM}_{\mathcal{L}ab} \leftarrow \mathcal{MM}_{\mathcal{L}abO}$

Proof. This follows from Theorem 1 and Theorem 12. □

Theorem 20. *Let $AF = (Ar, att)$ be an argumentation framework and let A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 3. Let $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$ be the output of the algorithm. It holds that $\mathcal{L}ab$ is a strongly admissible labelling that labels A `in` and has $\mathcal{MM}_{\mathcal{L}ab}$ as its min-max numbering.*

Proof. This follows from Theorem 10, Theorem 14, Theorem 16 and Theorem 17. □

Theorem 21. *Let $AF = (Ar, att)$ be an argumentation framework and let A be an argument in the grounded extension of AF. Let both AF and A be given as input to Algorithm 3. Let $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$ be the output of the algorithm. It holds that Algorithm 3 computes $\mathcal{L}ab$ and $\mathcal{MM}_{\mathcal{L}ab}$ in $O(n^3)$ time.*

Proof. This follows from Theorem 11 and Theorem 18. □

Theorem 22. *Let $AF = (Ar, att)$ be an argumentation framework, A be an argument in the grounded extension of AF.Let AF and A be given as input to Algorithm 1 and Algorithm 3. Let $\mathcal{L}ab_I$ and $\mathcal{MM}_{\mathcal{L}abI}$ be the output of Algorithm 1 and let $\mathcal{L}ab_3$ and $\mathcal{MM}_{\mathcal{L}ab3}$ be the output of Algorithm 3. It holds that $\mathcal{L}ab_3 \sqsubseteq \mathcal{L}ab_I$*

Proof. This follows from Theorem 13, together with Theorem 10 and the way Algorithm 3 is defined (by successively applying Algorithm 1 and Algorithm 2) □

4 Empirical Results

Now that the correctness of our algorithms has been proved and their computational complexity has been stated, the next step is to empirically evaluate their performance. For this, we compare both their runtime and output with that of other computational approaches.

4.1 Minimality

Although Algorithm 3 aims to find a relatively small strongly admissible labelling, it is not guaranteed to find an absolute smallest. This is because the problem of finding the absolute

smallest admissible labelling is coNP-complete, whereas Algorithm 3 is polynomial (Theorem 21). In essence, we have given up absolute minimality in order to achieve tractability. The question, therefore, is how much we had to compromise on minimality. That is, how does the outcome of Algorithm 3 compare with what would have been an absolute minimal outcome? In order to make the comparison, we will apply the ASPARTIX ASP encodings of [12] to determine the absolute minimal strongly admissible labelling.

Apart from comparing the strongly admissible labelling yielded by our algorithm with an absolute *minimal* strongly admissible labelling, we will also compare it with the absolute *maximal* strongly admissible labelling. That is, we will compare it with the grounded labelling. The reason for doing so is that the grounded semantics algorithms (e.g. [13, 14]) are to the best of our knowledge currently the only polynomial algorithms for computing a strongly admissible labelling (in particular, for the *maximal* strongly admissible labelling) that have been stated in the literature. As Algorithm 3 is also polynomial (Theorem 21) this raises the question of how much improvement is made regarding minimality.

For queries, we considered the argumentation frameworks in the benchmark sets of ICCMA'17 and ICCMA'19. For each of the argumentation frameworks we generated a query argument that is within the grounded extension (provided the grounded extension is not empty). We used the queried argument when one was provided by the competition (for instance, when considering the benchmark examples of the *Admbuster* class, we took 'a' to be the queried argument as this was suggested by the authors of this class). After considering 514 argumentation frameworks, we found that 277 argumentation frameworks yielded a grounded extension that is not empty (meaning they could used for current purposes).

We conducted our experiments on a MacBook Pro 2020 with 8GB of memory and an Intel Core i5 processor. To run the ASPARTIX system we used clingo v5.5.1. We set a timeout limit of 1000 seconds and a memory limit of 8GB per query.

For each of the selected benchmark examples, we have assessed the following:

1. the size of the grounded labelling (determined using the modified version of Algorithm 1 as described in Lemma 9)
2. the size of the strongly admissible labelling yielded by Algorithm 1
3. the size of the strongly admissible labelling yielded by Algorithm 3
4. the size of the absolute minimal strongly admissible labelling (yielded by the approach of [12])

We start our analysis with comparing the output of Algorithm 1 and Algorithm 3 with the grounded labelling regarding the size of the respective labellings. We found that the size of the strongly admissible labelling yielded by Algorithm 1 tends to be smaller than the size of the grounded labelling. More specifically, the strongly admissible labelling yielded by Algorithm 1 is smaller than the size of the grounded labelling in 63% of the 277 examples we tested for. In the remaining 37% of the examples, their sizes are the same.

Figure 2 provides a more detailed overview of our findings, in the form of a bar graph. The rightmost bar represents the 37% of the cases where the output of Algorithm 1 has the same size as the grounded labelling (that is, where the size of the output of Algorithm 1 is 100% of the size of the grounded labelling). The bars on the left of this are for the cases where the size of the output of Algorithm 1 is less than the size of the grounded labelling. For instance, it was found that in 10% of the examples, the size of output of Algorithm 1 is

80% to 89% of the size of the grounded labelling. On average, we found that the size of the output of Algorithm 1 is 76% of the size of the grounded labelling.

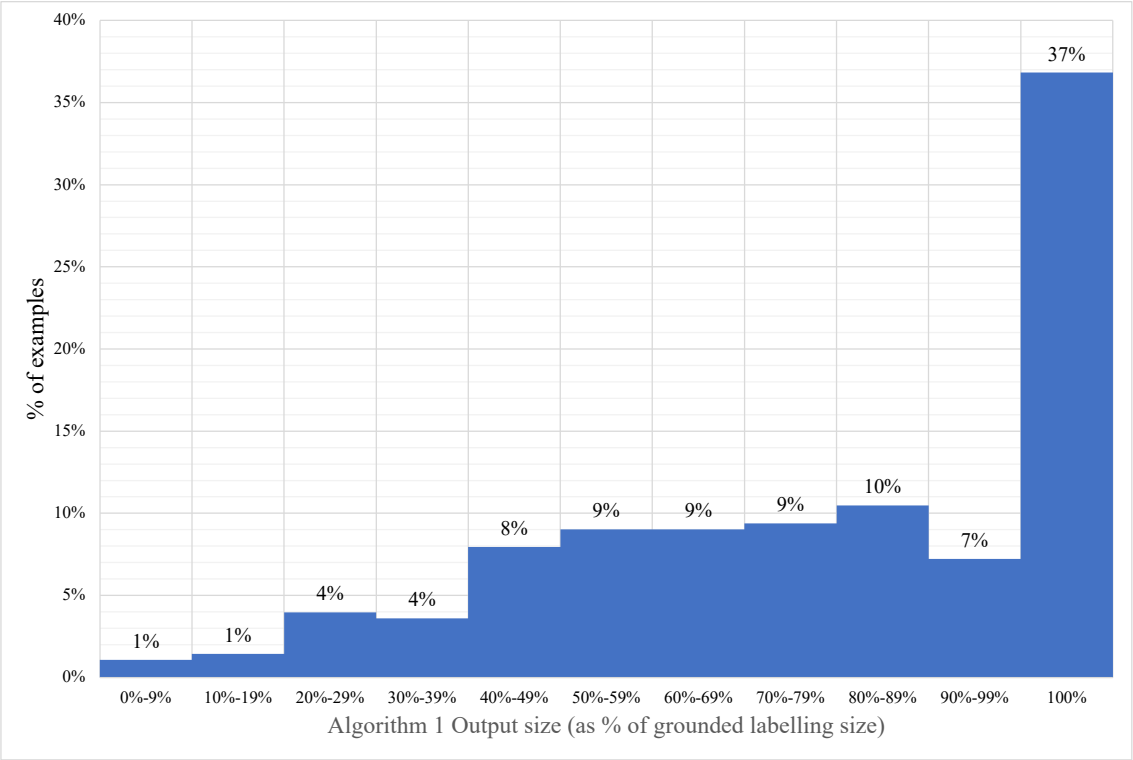

Figure 2: The size of output of Algorithm 1 (as a percentage of the grounded labelling).

As for Algorithm 3, we found an even bigger improvement in the size of it's output labelling compared to the grounded labelling. More specifically, the size of the strongly admissible labelling yielded by Algorithm 3 is smaller than the grounded labelling in 88% of the 277 examples we tested for. Figure 3 provides a more detailed overview of our findings in a similar way as we previously did for Algorithm 1. On average, we found that the output of Algorithm 3 has a size that is 25% of the size of the grounded labelling.

Apart from comparing Algorithm 1 and Algorithm 3 with the grounded labelling, it can also be insightful to compare the two algorithms with each other. In figure 4, each dot represents one of the 277 examples.[3] The horizontal axis represents the size of the output of Algorithm 1, as a percentage of the size of the grounded labelling. The vertical axis represents the size of the output of Algorithm 3 as a percentage of the size of the grounded labelling. For easy reference, we have included a dashed line indicating the situation where the output of Algorithm 1 has the same size as the output of Algorithm 3. Any dots below the dashed line represent the cases where Algorithm 3 outperforms Algorithm 1, in that it yields a smaller strongly admissible labelling. Any dots above the dashed line represents the cases where Algorithm 3 under performs Algorithm 1 in that it yields a bigger strongly admissible labelling. Unsurprisingly, there no such cases as Theorem 22 states that the output of Algorithm 3 cannot be bigger than the output of Algorithm 1.

[3]Please be aware that some of the dots overlap each other.

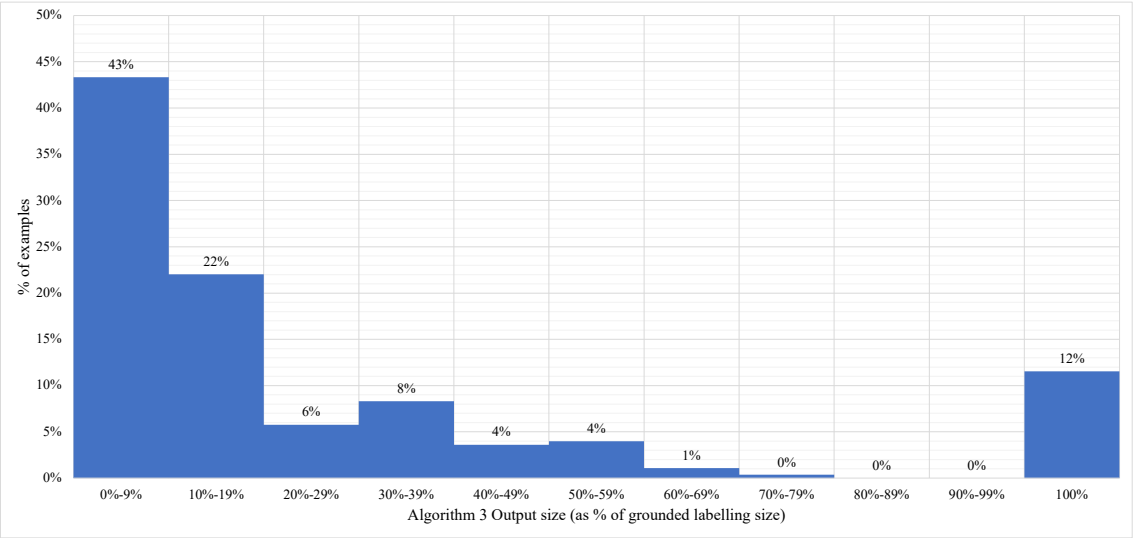

Figure 3: The size of output of Algorithm 3 (as a percentage of the grounded labelling).

We found that for 95% of the examples, Algorithm 3 produces a smaller labelling than Algorithm 1. Moreover, we found that on average, the output of Algorithm 3 is 32% smaller than output of Algorithm 1.

The next question is how the output of our best performing algorithm (Algorithm 3) compares with what would have been the ideal output. That is, we compare the size of the output of Algorithm 3 with the size of an minimal strongly admissible labelling for the main argument in question, as computed using the ASPARTIX encodings of [12]. The results are shown in Figure 5.

We found that in 91% of the 277 examples, the output of Algorithm 3 is of the same size as the smallest strongly admissible labelling for the output of the main argument in question. For the other 9% of the examples, the output of Algorithm 3 has a bigger size. On average, we found that the output of Algorithm 3 is 3% bigger than the smallest strongly admissible labelling for the main argument in question. Figure 6, provides a more detailed overview of how much bigger the output of Algorithm 3 is compared to the smallest strongly admissible labelling for the main argument in question.

4.2 Runtime

The next thing to study is how the runtime of our algorithms compares with the runtime of some of the existing computational approaches. In particular, we compare the runtime of Algorithm 1 and Algorithm 3 with the runtime of the ASPARTIX-based approach of [12].

We first compare the runtime of Algorithm 3 to the runtime of the modified version of Algorithm 3 of [14] for computing the grounded labelling. It turns out that the runtimes of these algorithms are very similar. On average, Algorithm 3 of [14] took 0.02(3%) seconds more than Algorithm 3 to solve the test instances. These runtime results of Algorithm 3 and Algorithm 3 of [14] are illustrated within Figure 7.

The next question is how does the runtime of computing Algorithm 3 compare to the runtime of the ASPARTIX encoding for minimal strongly admissibility. It was observed that the runtime of ASPARTIX encoding is significantly longer than the runtime of Algorith 3. A

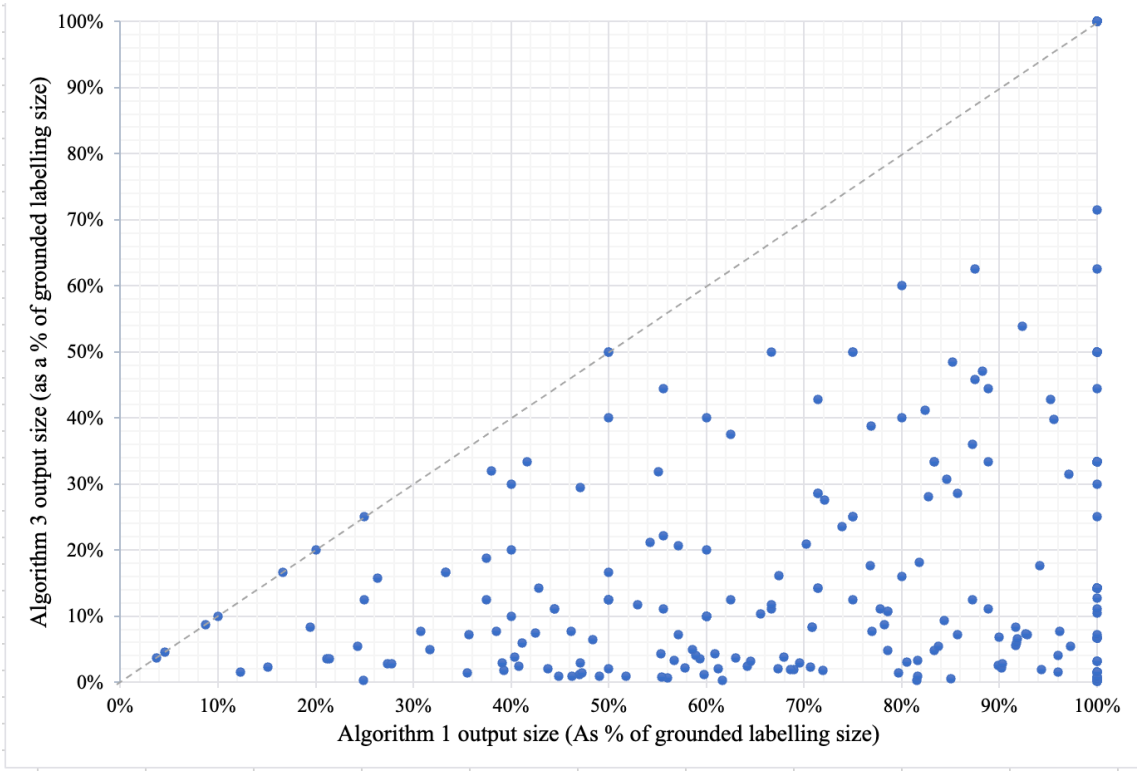

Figure 4: The size of output of Algorithm 1 compared to the output Algorithm 3 (as a percentage of the size of the grounded labelling).

detailed overview of the difference in runtimes of Algorithm 3 and the ASPARTIX encoding on minimal strong admissibility is shown in Figure 8. On average, the ASPARTIX framework took 12.5 seconds (907%) more than Algorithm 3 to solve the test instances.

5 Discussion

In the current paper, we provided two algorithms (Algorithm 1 and Algorithm 3) for computing a relatively small strongly admissible labelling for an argument that is in the grounded extension. We proved that both algorithms are correct in the sense that each of them returns a strongly admissible labelling (with associated min-max numbering) that labels the main argument in question `in` (Theorem 10 and 20). Moreover, each algorithm runs in polynomial (cubic) time (Theorem 11 and Theorem 21). It was also shown that the strongly admissible labelling yielded by Algorithm 3 is smaller than or equal to the strongly admissible labelling yielded by Algorithm 1 (Theorem 13).

The next question we examined is how small the output of Algorithm 1 and Algorithm 3 is compared to the smallest strongly admissible for the main argument in question. Unfortunately, previous findings make it difficult to provide formal theoretical results on this. This is because the k-approximation problem for strong admissibility is NP-hard, meaning that a polynomial algorithm (such as Algorithm 1 and Algorithm 3) cannot provide any guarantees of yielding a result within a fixed parameter k from the size of the absolute smallest strongly admissible labelling for the main argument in question.

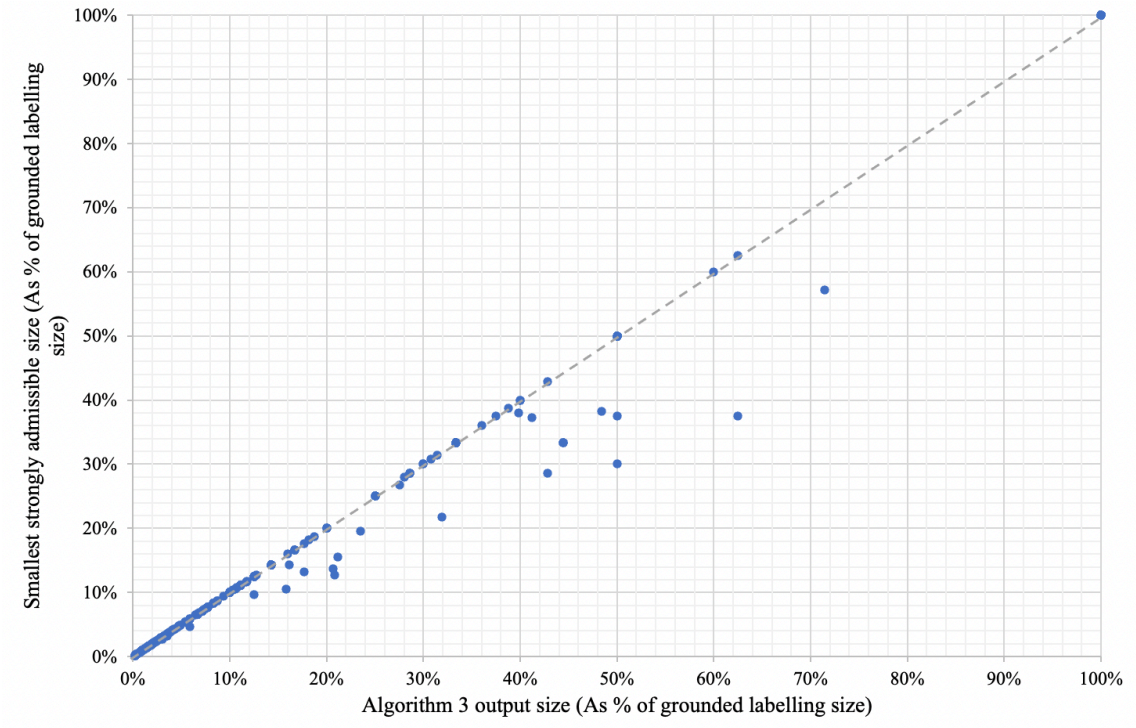

Figure 5: The size of output of Algorithm 3 compared to the smallest strongly admissible labelling (as a percentage of the size of the grounded labelling).

Hence, instead of developing theoretical results, we decided to approach the issue of minimality in an empirical way, using a number of experiments. These experiments were based on the benchmark examples that were submitted to ICCMA'17 and ICCMA'19. We compared the output of Algorithm 1 and Algorithm 3 with both the biggest and the smallest strongly admissible set for the main argument in question (the biggest was computed using Algorithm 3 described in [14] and the smallest was computed using the ASPARTIX based approach on computing minimal strong admissibility in [12]). Overall, we found that Algorithm 3 yields results that are only marginally bigger than the smallest strongly admissible labelling, with a run-time that is a fraction of the time that would be required to find this smallest strongly admissible labelling. The outputs of both or algorithms return a strongly admissible labelling that is significantly smaller than the biggest strongly admissible labelling (the grounded labelling), with the output of Algorithm 3 on average being only 25% of the output of the biggest strongly admissible labelling.

The research of the current paper fits into our long-term research agenda of using argumentation theory to provide explainable formal inference. In our view, it is not enough for a knowledge-based system to simply provide an answer regarding what to do or what to believe. There should also be a way for this answer to be explained. One way of doing so is by means of (formal) discussion. Here, the idea is that the knowledge-based system should provide the argument that is at the basis of its advice. The user is then allowed to raise objections (counterarguments) which the system then replies to (using counter-counter-arguments), etc. In general, we would like such a discussion to be (1) sound and complete for the underlying argumentation semantics, (2) not be unnecessarily long, and (3) be close enough to human

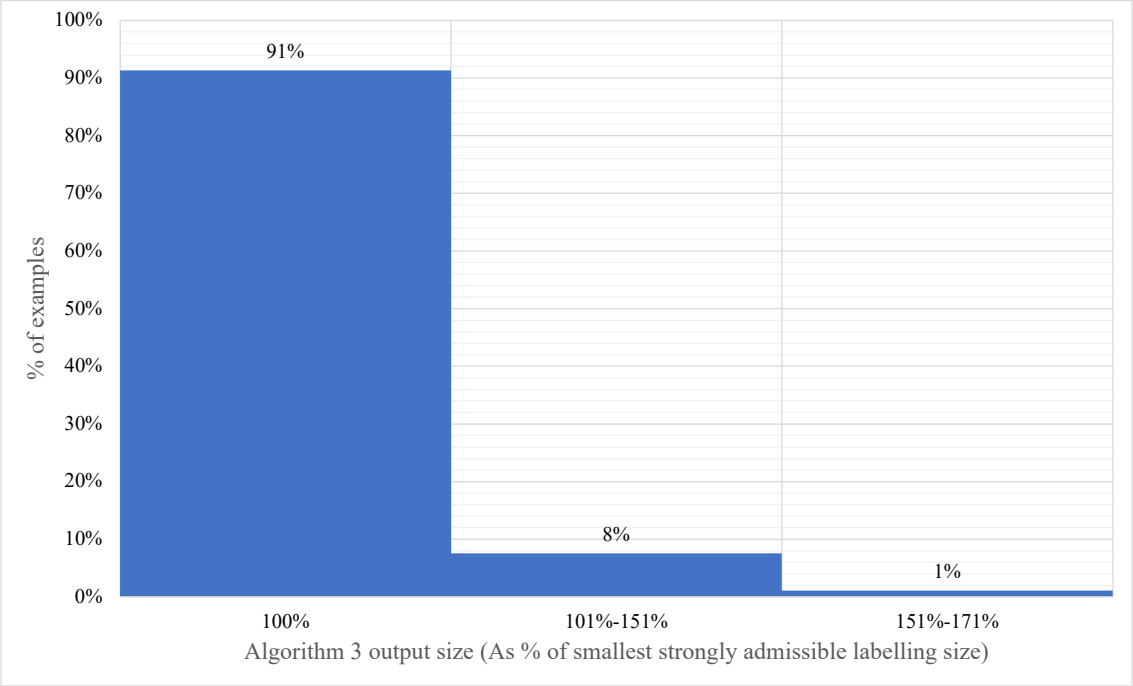

Figure 6: The size of output of Algorithm 3 compared to the smaller strongly admissible labelling (as a percentage of the size of the grounded labelling).

discussion to be perceived as natural and convincing

As for point (1), sound and complete discussion games have been identified for grounded, preferred, stable and ideal semantics [5]. As for point (2), this is what we studied in the current paper, as well as in [4, 7]. As for point (3), this is something that we are aiming to report on in future work.

For future research, it is possible to conduct a similar sort of analysis (as in this paper) on minimal admissible labellings. It was reported obtaining an absolute minimal admissible labelling for a main argument in question is also of coNP-complete complexity [4, 7] therefore, it would be interesting to look into developing an algorithm that generates a small admissible labelling in polynomial time complexity. Similarly, it would also be interesting to look at the complexity and empirical results on generating minimal ideal sets.

6 Epilogue

Although the main topic of the current paper is how to construct a relatively small strongly admissible labelling(for a particular argument) in a time-efficient way, our results also allow us to provide an analysis of two adjacent questions: what is the additional cost of computing the min-max numbering compared to only computing the strongly admissible labelling itself and what is the fastest approach for computing *any* strongly admissible labelling (for a particular argument) if the size of the labelling does not matter. In the following two sections, we study the questions in more detail.

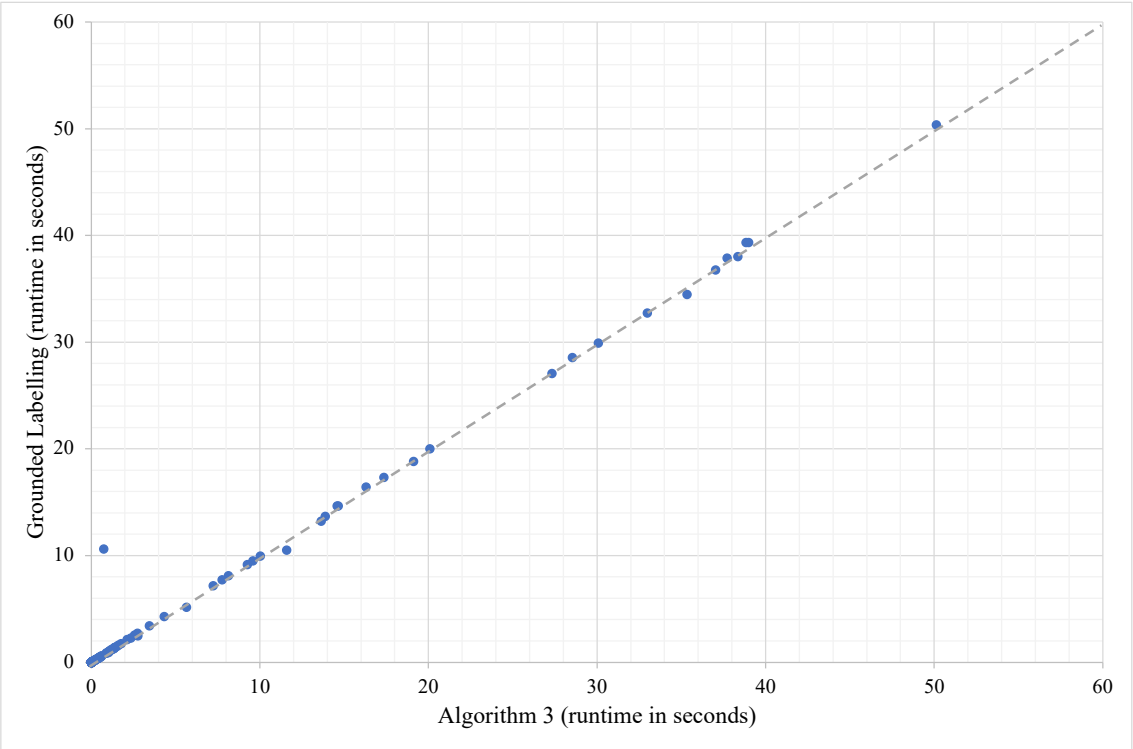

Figure 7: The runtime of computing Algorithm 3 compared to the runtime of computing the grounded labelling).

6.1 The additional costs of computing the min-max numbering

As we mentioned earlier, our approach (in particular Algorithm 1) is based on the work of [14]. However, where the works of [14] only computes a strongly admissible labelling (the biggest strongly admissible labelling, to be precise) our approach additionally computes the associated min-max numbering. This raises the question of what is the additional runtime needed to compute this min-max numbering.

In order to a like-for-like comparison, we compare the runtime of Algorithm 3 of [14] with the runtime of our own Algorithm for computing the grounded labelling and it's associated min-max numbering, as described by Lemma 9. Each of the two algorithms was run on 277 examples of the earlier mentioned testset. The results are provided in Figure 9. On average, the runtime of the algorithm of Lemma 9 is 0.0004% longer than the runtime of Algorithm 3 of [14].

It is also possible to do a like-for-like comparison w.r.t Algorithm 1, by comparing the runtime of the algorithm itself with the runtime of the algorithm after commenting out lines 15, 26 and 32 (which are used to compute the min-max numbering). The results are provided in Figure 10. On average, we found that the runtime of Algorithm 1 is 3%(0.067 seconds) longer than the runtime of Algorithm 1 with lines 15, 26 and 32 commented out.

Overall, we observe that the additional runtime for computing the min-max numbering is only marginally higher than the runtime for computing only the strongly admissible labelling itself. As an aside, the reader might wonder why we did not carry out a similar like-for-like comparison in the context of Algorithm 3. That is, why did we not compare the runtime of

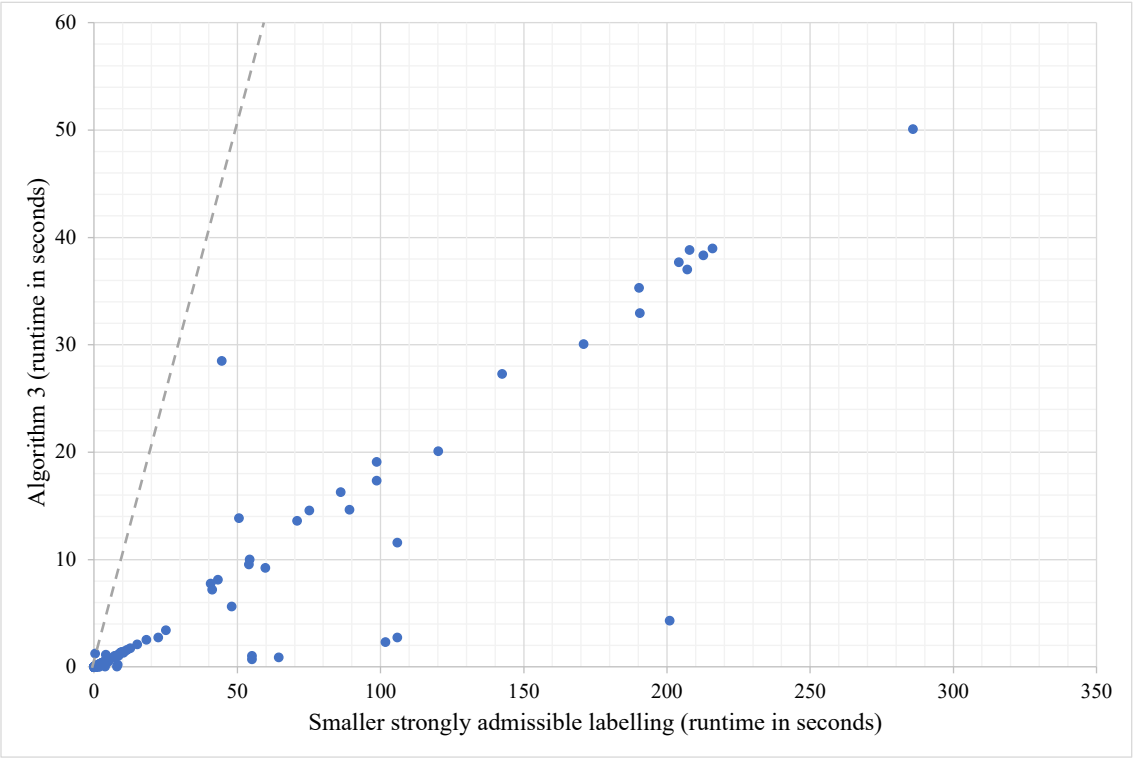

Figure 8: The runtime of computing Algorithm 3 compared to the runtime of computing the ASPARTIX encoding on minimal strong admissibility.

Algorithm 3 with the runtime of a modified version of Algorithm 3 in which all computation of the min-max numbering has been commented out? The reason for not doing so is that Algorithm 3 contains Algorithm 2 whose correctness critically depends on the presence of a min-max numbering. To illustrate this, consider the argumentation framework of Figure 11. Suppose E is the main argument in question. Algorithm 1 in its unmodified form will yield the strongly admissible labelling $(\{A, C, E\}, \{B, D\}, \emptyset)$ and it's associated min-max numbering $\{(A : 1), (B : 2), (C : 3), (D : 4), (E, 5)\}$. Algorithm 2, in it's unmodified form will use this min-max numbering once it arrives at argument B for selecting a *minimally numbered* in-labelled attacker of B (which is A). However, without the numbering, Algorithm 2 would not know whether to choose A or E as the attacker of B. In the absence of a min-max numbering, the algorithm could decide that B already has an in-labelled attacker (E), rather than adding the *minimally numbered* in-labelled attacker (A), resulting in the incorrect strongly admissible labelling of $(\{C, E\}, \{B, D\}, \{A\})$. Hence, we cannot compare Algorithm 3 with a modified version of 3 that does not dedicate any resources for computing the min-max numbering, as the latter algorithms would be guaranteed to be correct.

6.2 Computing an *arbitrary* strongly admissible labelling

So far, we have focused our attention on the problem of finding a *small* strongly admissible labelling (for a particular argument) in a time efficient way. We now examine, the question of how to find an *arbitrary* strongly admissible labelling (for a particular argument) in a time efficient way. That is, we are interested in a fast way of constructing a strongly admissible

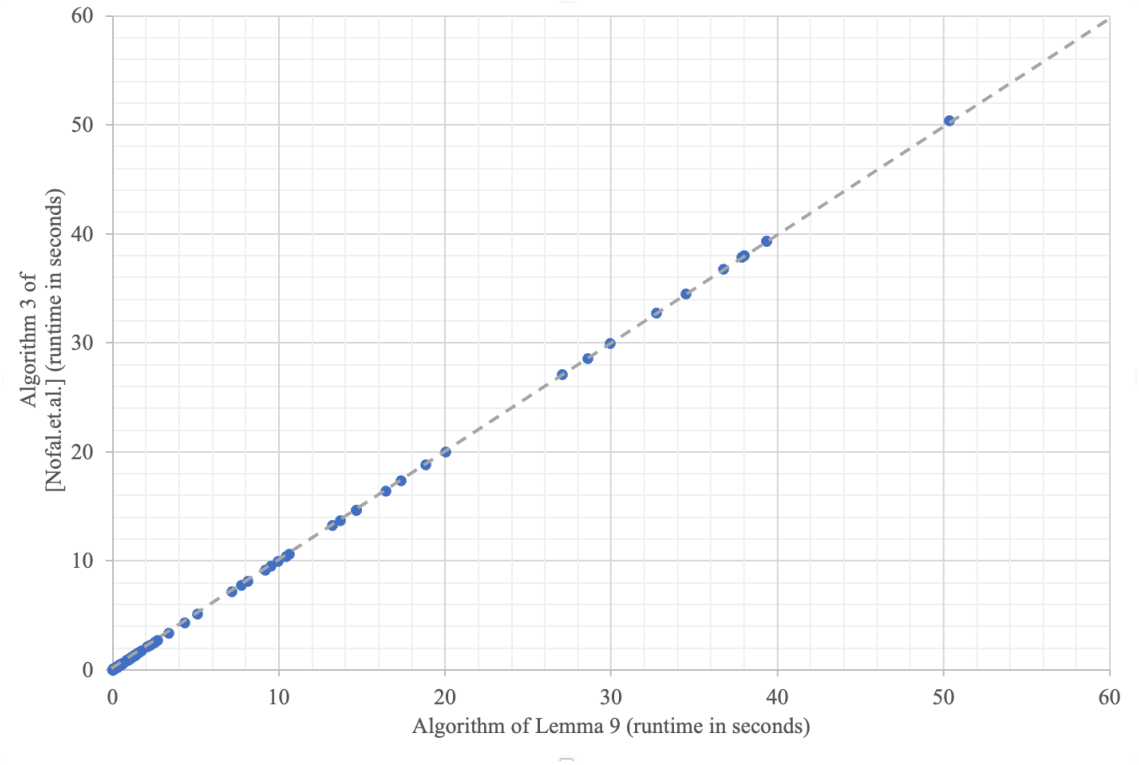

Figure 9: The runtime of computing the Algorthim of Lemma 9 compared to the runtime of computing Algorithm 3 of [14]).

labelling (that labels the argument in question `in`) without caring how big or small the labelling is.[4] In section 4.2, we compared the runtime of Algorithm 1 and 3 with the runtime of the ASPARTIX-based approach of [12]. The complete our analysis, in the current section we will also compare the runtime of the algorithm of Lemma 9 (for computing the grounded labelling) with the run-time of the ASPARTIX-based approach of [12]. The results are provided in the Figure 12. On average, the runtime of the Algorithm of Lemma 9 is 16% of the runtime of the ASPARTIX-based approach of [12].

To summarise our results, we found that compared to the ASPARTIX-based approach of [12]

1. the runtime of Algorithm 1 is on average 16%,
2. the runtime of Algorithm 3 is on average 16%, and
3. the runtime of Algorithm of Lemma 9 is on average 16%.

Hence, when the aim is to find an arbitrary strongly admissible labelling for a particular argument, our findings confirm the expectation that of above mentioned three algorithms, Algorithm 1 is the most time-efficient approach.[5]

[4]It can be mentioned that ICCMA'17 and ICCMA'19 describe the somewhat similar task of finding an arbitrary extension for a particular semantics.

[5]This is what one would expect to find as Algorithm 1 is part of Algorithm 3, and unlike the Algorithm of Lemma 9, Algorithm 1 terminates when encountering the main argument in question.

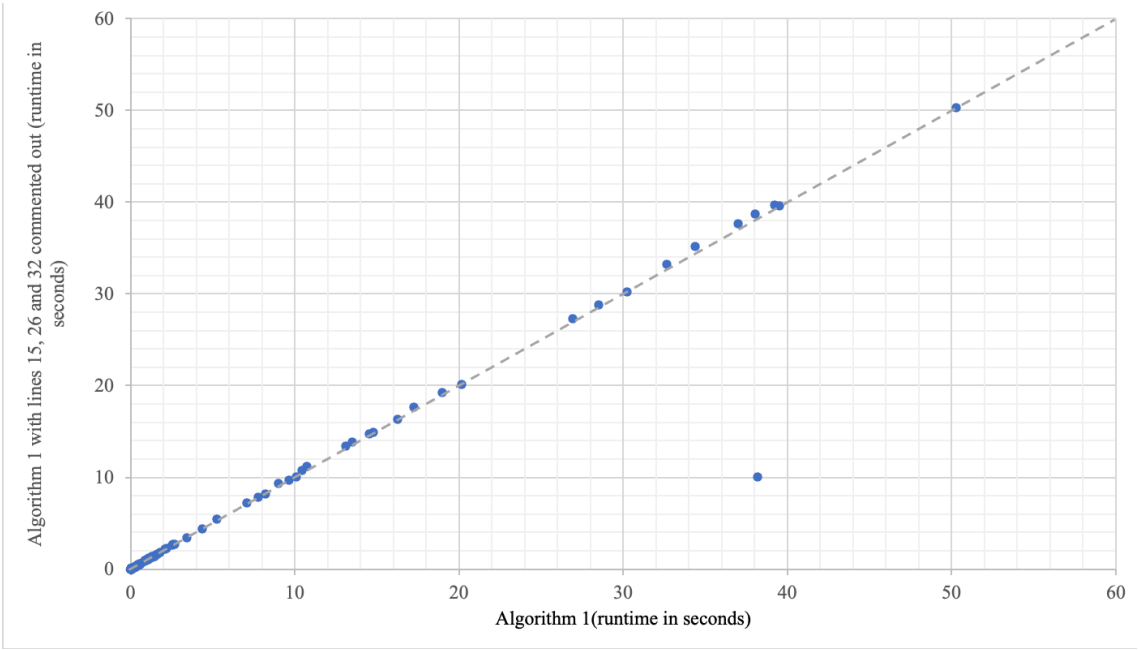

Figure 10: The runtime of computing the Algorthim 1 with the runtime of computing Algorithm 1 with lines 15, 26 and 32 commented out.

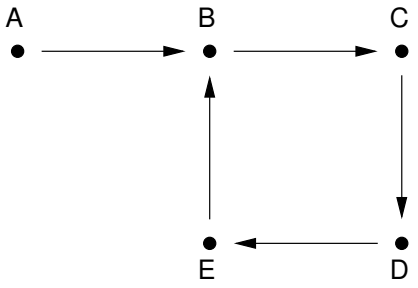

Figure 11: Example argumentation framework

References

[1] P. Baroni and M. Giacomin. On principle-based evaluation of extension-based argumentation semantics. *Artificial Intelligence*, 171(10-15):675–700, 2007.

[2] M.W.A. Caminada. On the issue of reinstatement in argumentation. In M. Fischer, W. van der Hoek, B. Konev, and A. Lisitsa, editors, *Logics in Artificial Intelligence; 10th European Conference, JELIA 2006*, pages 111–123. Springer, 2006. LNAI 4160.

[3] M.W.A. Caminada. On the issue of reinstatement in argumentation. Technical Report UU-CS-2006-023, Institute of Information and Computing Sciences, Utrecht University, 2006.

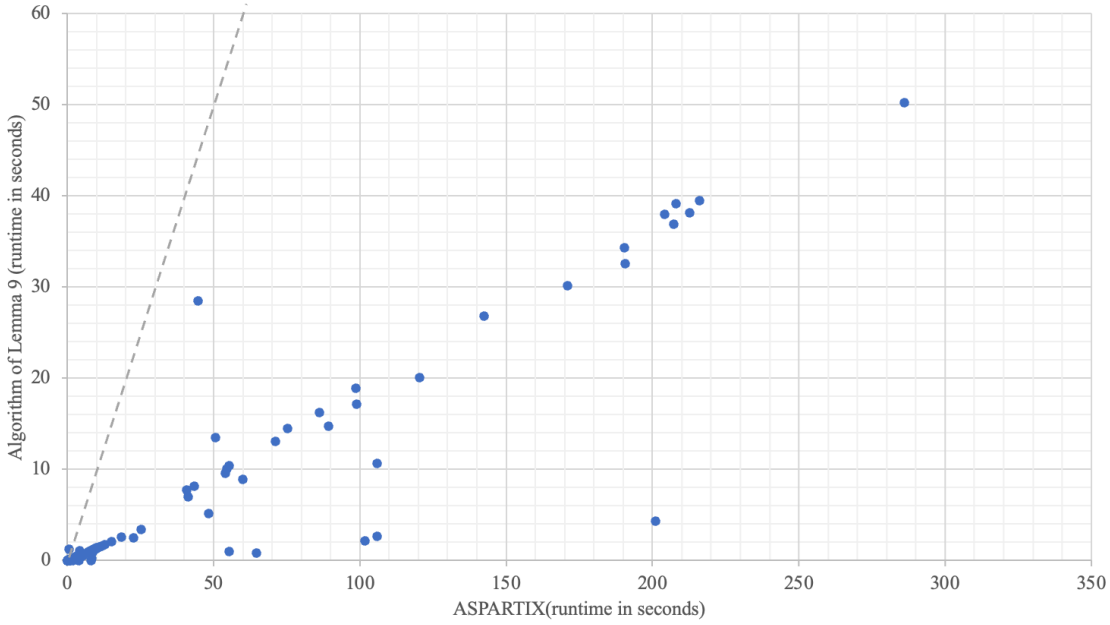

Figure 12: The runtime of computing the Algorithm of Lemma 9 with the ASPARTIX-based approach of [12].

[4] M.W.A. Caminada. Strong admissibility revisited. In S. Parsons, N. Oren, C. Reed, and F. Cerutti, editors, *Computational Models of Argument; Proceedings of COMMA 2014*, pages 197–208. IOS Press, 2014.

[5] M.W.A. Caminada. A discussion game for grounded semantics. In E. Black, S. Modgil, and N. Oren, editors, *Theory and Applications of Formal Argumentation (proceedings TAFA 2015)*, pages 59–73. Springer, 2015.

[6] M.W.A. Caminada, P. Baroni, and M. Giacomin. Abstract argumentation frameworks and their semantics. In *Handbook of Formal Argumentation*, volume 1. College Publications, 2018.

[7] M.W.A. Caminada and P.E. Dunne. Strong admissibility revised: theory and applications. *Argument & Computation*, 10:277–300, 2019.

[8] M.W.A. Caminada and P.E. Dunne. Minimal strong admissibility: a complexity analysis. In H. Prakken, S. Bistarelli, F. Santini, and C. Taticchi, editors, *Proceedings of COMMA 2020*, pages 135–146. IOS Press, 2020.

[9] M.W.A. Caminada and D.M. Gabbay. A logical account of formal argumentation. *Studia Logica*, 93(2-3):109–145, 2009. Special issue: new ideas in argumentation theory.

[10] M.W.A. Caminada and G. Pigozzi. On judgment aggregation in abstract argumentation. *Autonomous Agents and Multi-Agent Systems*, 22(1):64–102, 2011.

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

[12] W. Dvořák and J. Wallner. Computing strongly admissible sets. In H. Prakken, S. Bistarelli, F. Santini, and C. Taticchi, editors, *Proceedings of COMMA 2020*, pages 179–190. IOS Press, 2020.

[13] S. Modgil and M.W.A. Caminada. Proof theories and algorithms for abstract argumentation frameworks. In I. Rahwan and G.R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 105–129. Springer, 2009.

[14] S. Nofal, K. Atkinson, and P.E. Dunne. Computing grounded extensions of abstract argumentation frameworks. *The Computer Journal*, 64:54–63, 2021.