A New Labelling Algorithm for Generating Preferred Extensions of Abstract Argumentation Frameworks

Samer Nofal¹, Katie Atkinson², Paul E. Dunne², Ismail Hababeh¹

¹Department of Computer Science, German Jordanian University, Jordan
²Department of Computer Science, University of Liverpool, United Kingdom

Keywords: abstract argumentation, argumentation semantics, labelling semantics, argumentation algorithm, labelling algorithm, preferred semantics

Abstract: The field of computational models of argument aims to provide support for automated reasoning through algorithms that operate on arguments and attack relations between them. In this paper we present a new labelling algorithm that lists all preferred extensions of an abstract argumentation framework. The new algorithm is enhanced by a new pruning strategy. We verified our new labelling algorithm and showed that it enumerates preferred extensions faster than the old labelling algorithm.

1 Introduction

The study of computational argumentation is a major field of artificial intelligence, see for example (Atkinson et al., 2017; Modgil et al., 2013). Abstract argumentation frameworks of (Dung, 1995) (AFs for short) are directed graphs with nodes representing abstract arguments while directed edges denote attacks between arguments. In spite of their simplicity, AFs are an effective mechanism for decision making in different domains, see for example (Longo and Dondio, 2014; Bench-Capon et al., 2015; Tamani et al., 2015).

A fundamental issue arises in the context of AFs concerning identifying which arguments are collectively accepted in a given AF. To this end, an argumentation semantics defines rules under which one can compute sets of accepted arguments, what are so-called extensions. In the literature we find several proposals motivating different argumentation semantics. These varied semantics give a wide range of selection among which one can choose what best fit the needs of the target application, see (Baroni et al., 2011) for a comprehensive review of argumentation semantics. In this paper we are concerned with the problem of listing all extensions of a given AF under preferred argumentation semantics, which is one of the most studied semantics. We give a precise definition for preferred semantics in section 2.

A labelling algorithm for listing preferred extensions of a given AF is basically a search algorithm that expands an abstract binary tree typically in a depth-first manner. The nodes of the tree represent different states of the input AF, what are so-called labellings. Labellings of a given AF are defined by a total function that maps arguments from the AF to elements from a predefined set of statuses, what are so-called argument labels.

The objective of this paper is to present a new, more efficient labelling algorithm that lists all preferred extensions of a given AF. The current state-of-the-art labelling algorithm for preferred extension enumeration is presented in (Nofal et al., 2016). However, in this paper we enhance the algorithm of (Nofal et al., 2016) by a more efficient pruning strategy to speed up preferred extension generation. We implemented our new algorithm and verified that the new pruning strategy resulted in a faster preferred extension enumeration. Although we focus on the problem of listing preferred extensions, we believe that the notions of the new pruning strategy are transferable (with appropriate adjustments) to other computational problems in the context of abstract argumentation frameworks.

In section 2 we give a necessary background material. In section 3 we recall the state-of-the-art labelling algorithm for generating all preferred extensions. In section 4 we present our new labelling algorithm for listing preferred extensions. We verify the efficiency of the new algorithm in section 5. We conclude the paper in section 6.
2 Preliminaries

In the following definition we recall the notion of abstract argumentation frameworks as introduced in the seminal work of Dung, 1995.

Definition 1 (abstract argumentation frameworks). An abstract argumentation framework $AF$ is a pair $(A, R)$ where $A$ is a set of abstract arguments and $R \subseteq A \times A$ is called the attack relation.

We refer to $(x, y) \in R$ as $x$ attacks $y$ (or $y$ is attacked by $x$). We denote by $\{x\}^-$ respectively $\{x\}^+$ the subset of $A$ containing those arguments that attack (respectively are attacked by) the argument $x$, and so we use $\{x\}^\pm$ to represent the set $\{x\}^- \cup \{x\}^+$. For a set of arguments $S \subseteq A$, we define

$$S^- \equiv \{ y \in A \mid \exists x \in S \text{ s.t. } y \in \{x\}^- \}$$
$$S^+ \equiv \{ y \in A \mid \exists x \in S \text{ s.t. } y \in \{x\}^+ \}$$

We denote by $S^\pm$ the union $S^+ \cup S^-$. We say $S \subseteq A$ attacks $T \subseteq A$ (or $T$ is attacked by $S$) if and only if $S^+ \cap T \neq \emptyset$. $S \subseteq A$ attacks $x \in A$ (or $x$ is attacked by $S$) if and only if $x \in S^+$. Given a subset $S \subseteq A$, then

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

In this paper we are concerned with the following problem: given an $AF$ $H = (A, R)$, enumerate the preferred extensions of $H$.

We give now a general account of a labelling algorithm that generates all preferred extensions. As said earlier, a labelling algorithm expands a conceptual binary search tree in a depth-first way. The algorithm forks to a left node if it decides to include an argument in a current under-construction extension. On the other hand the algorithm forks to a right node if it decides to exclude an argument from the current under-construction extension. Once no argument is left un-included or un-excluded, the algorithm backtrack to find another extension. Take the AF of figure 1 then a labelling algorithm would generate the search tree visualized in figure 2.

In the next section we recall a precise definition of the state-of-the-art labelling algorithm for listing preferred extensions.

3 The State-of-the-art Labelling Algorithm for Preferred Extensions

We recall the state-of-the-art labelling algorithm, presented in Nofal et al., 2016, for listing all preferred extensions of a given AF. In section 2 we presented an extension-based definition for preferred argumentation semantics. Alternatively, preferred semantics can be described in terms of labellings, which are mappings that relate every argument in a given AF to a label in $\{\text{in, out, undec}\}$. For example, let $(A, R)$ be an AF and $S \subseteq A$ be an admissible set then the equivalent labelling for $S$ is described by a total mapping $Lab : A \rightarrow \{\text{in, out, undec}\}$ where $S = \{x \mid Lab(x) = \text{in}\}$, $S^+ = \{x \mid Lab(x) = \text{out}\}$ and $A \setminus (S \cup S^-) = \{x \mid Lab(x) = \text{undec}\}$. For a thorough presentation on labelling semantics see Caminada and Gabbay, 2009. Although a 3-label mapping is probably sufficient for characterizing extensions, additional labels have been found useful for enhancing the efficiency of extension enumeration. Thereby the labelling algorithms of Nofal et al., 2016 and Caminada and Gabbay, 2009 use instead a 5-label total function that maps arguments to labels from $\{\text{in, out, undec, blank, must out}\}$.

As noted earlier, a labelling algorithm for preferred extension enumeration is basically a depth-first search that explores a conceptual binary tree. At the root node of the search tree, all arguments of the given AF are initially labelled according to the following specification.

Definition 2 (initial labelling). Let $H = (A, R)$ be an AF and $S \subseteq A$ be the set of self-attacking arguments. Then the initial labelling of $H$ is defined by the union: $\{(x, \text{blank}) \mid x \in A \setminus S\} \cup \{(y, \text{undec}) \mid y \in S\}$.

At any node of the search tree the algorithm transitions to a left node by selecting an argument $x$ with $Lab(x) = \text{blank}$ and subsequently modify argument labels as specified in the following definition.

Definition 3 (in transitions). Let $H = (A, R)$ be an AF. $Lab : A \rightarrow \{\text{in, out, undec, blank, must out}\}$ be a total mapping, and $x$ be an argument with $Lab(x) = \text{blank}$ then in transitions $Lab(x, H, Lab)$ is defined by the following ac-
Figure 2: A search tree that would be expanded by a basic labelling algorithm for listing the preferred extensions of \( AF_1 \), which is depicted in figure 1.

Definitions:
1. \( \text{Lab}' \leftarrow \text{Lab} \).
2. \( \text{Lab}'(x) \leftarrow \text{in} \).
3. for each \( y \in \{x\}^+ \), \( \text{Lab}'(y) \leftarrow \text{out} \).
4. for each \( y \in \{x\}^- \) with \( \text{Lab}'(y) \neq \text{out} \), \( \text{Lab}'(y) \leftarrow \text{must out} \).
5. return \( \text{Lab}' \).

After the algorithm finished exploring the left sub-tree, that is induced by an \( \text{in} \) transition, it expands a right node by an \( \text{undec} \) transition as described in the following definition.

Definition 4 (\( \text{undec} \) transitions). Let \( H = (A,R) \) be an \( AF \), \( \text{Lab} : A \rightarrow \{\text{in}, \text{out}, \text{undec}, \text{blank}, \text{must out}\} \) be a total mapping and \( x \) be an argument with \( \text{Lab}(x) = \text{blank} \). Then \( \text{und}_\text{trans}(x,H, \text{Lab}) \) is defined by the following actions:
1. \( \text{Lab}' \leftarrow \text{Lab} \).
2. \( \text{Lab}'(x) \leftarrow \text{undec} \).
3. return \( \text{Lab}' \).

A labelling algorithm would reach a leaf node if there are no \text{blank} arguments, we call such leaf nodes terminal labellings.

Definition 5 (terminal labellings). Let \( H = (A,R) \) be an \( AF \) and \( \text{Lab} : A \rightarrow \{\text{in}, \text{out}, \text{undec}, \text{blank}, \text{must out}\} \) be a total mapping. Then \( \text{Lab} \) is a terminal labelling of \( H \) if and only if for each \( x \in A \), \( \text{Lab}(x) \neq \text{blank} \).

We call terminal labellings with \( \{x \mid \text{Lab}(x) = \text{in}\} \) being admissible by admissible labellings. It follows directly from the definition of admissible sets that if a terminal labelling, for a given \( AF \), does not map any argument to \text{must out} then the set \( \{x \mid \text{Lab}(x) = \text{in}\} \) is admissible.

Definition 6 (admissible labellings). Let \( H = (A,R) \) be an \( AF \) and \( \text{Lab} : A \rightarrow \{\text{in}, \text{out}, \text{undec}, \text{blank}, \text{must out}\} \) be a total mapping. Then \( \text{Lab} \) is an admissible labelling of \( H \) if and only if \( \text{Lab} \) is terminal and there is no \( x \in A \) with \( \text{Lab}(x) = \text{must out} \).

Conversely we denote by rejected labellings (or occasionally dead-end labellings) the terminal labellings with \( \{x \mid \text{Lab}(x) = \text{in}\} \) being not admissible. It follows directly from the definition of admissible sets that if a terminal labelling, for a given \( AF \), maps an argument to \text{must out} then the set \( \{x \mid \text{Lab}(x) = \text{in}\} \) is not admissible.

Definition 7 (rejected labellings). Let \( H = (A,R) \) be an \( AF \) and \( \text{Lab} : A \rightarrow \{\text{in}, \text{out}, \text{undec}, \text{blank}, \text{must out}\} \) be a total mapping. Then \( \text{Lab} \) is rejected if and only if \( \text{Lab} \) is terminal and there is no \( x \in A \) with \( \text{Lab}(x) = \text{must out} \).

We denote by preferred labellings the admissible labellings with \( \{x \mid \text{Lab}(x) = \text{in}\} \) being inclusion-wise maximal among all admissible labellings.

Definition 8 (preferred labellings). Let \( H = (A,R) \) be
an AF and Lab : A → \{in, out, undec, must_out, blank\} be a total mapping. Then Lab is a preferred labelling of H if and only if Lab is admissible and \{x \mid Lab(x) = in\} is maximal (w.r.t. \subseteq) among all admissible labellings.

Now we recall the pruning strategy used in the labelling algorithm of Nofal et al. (2016). Note that the pruning strategy of (Nofal et al., 2016) is centered around two notions: labelling propagation and hopeless labellings, which both improved preferred extension enumeration. Labelling propagation is about inferring argument labels by analyzing the current labelling while hopeless labelling are those labellings that never grow to a preferred labelling.

Definition 9 (labelling propagation). Let H = (A, R) be an AF and Lab : A → \{in, out, undec, must_out, blank\} be a total mapping. Then propagate(H, Lab) is defined by the following actions:
1. while \exists x \text{ Lab}(x) = \text{blank} s.t. \forall y \in \{x\}^- \text{ Lab}(y) \in \{out, must\_out\} do
   1.1. Lab(x) ← in
   1.2. for each y \in \{x\}^+ do Lab(y) ← out

Definition 10 (hopeless labellings). Let H = (A, R) be an AF and Lab : A → \{in, out, undec, must_out, blank\} be a total mapping. Then Lab is a hopeless labelling of H if and only if there is x \in A with Lab(x) = must_out such that for all y \in \{x\}^- Lab(y) \in \{out, must_out, undec\}.

Now we give algorithm 1 that lists all preferred extensions. If algorithm 1 is invoked on a given AF H, the initial labelling of H and an empty set E, then it will return E containing all preferred extensions. Throughout the paper we assume that a call by reference has to be made to invoke an algorithm or a procedure. Referring to line 4 one can check the maximality of a given labelling Lab by ensuring that for each preferred extension S \in E generated so far it is the case that \{x \mid Lab(x) = in\} \subseteq S. This is true because algorithm 1 builds admissible sets in a descending order with respect to set inclusion, which means maximal sets are visited first.

In the following section we develop an improved algorithm for listing preferred extensions.

4 A New Labelling Algorithm for Preferred Extension Enumeration

Our new labelling algorithm is enhanced by a new pruning strategy that we present in section 4.2. In section 4.3 we give a new strategy for selecting arguments that induce in and undec transitions. We introduce a new labelling scheme in section 4.1

4.1 A New Labelling Scheme

In section 3 we employed a 5-label total function that maps arguments from a given AF to labels from \{in, out, must_out, undec, blank\}. We add to this scheme two more labels: must_in and must_undec. Hence we use a 7-label total function that maps arguments from a given AF to labels from \{in, out, must_out, undec, blank, must_in, must_undec\}. From now on we refer to this 7-label set as L.

Before we give a precise description for those arguments that are eligible to be labelled with must_in or must_undec, we define first two helpful total mappings that both will streamline the computations around deciding such eligibility. Our total functions are called BLANK and UNDEC. BLANK maps an argument to the number of the attackers that are labeled with either blank, must_in, or must_undec. UNDEC maps an argument to the number of undec attackers. A similar idea to the essence of BLANK and UNDEC has been used in Modgil and Caminada (2009) for computing the grounded extension. We specify BLANK and UNDEC precisely shortly. Observe that we denote the set of nonnegative integers by \(\mathbb{N}_0\).

Definition 11 (BLANK and UNDEC). Let H = (A, R) be an AF and Lab : A → \mathbb{L} be a total mapping, then BLANK : A → \mathbb{N}_0 and UNDEC : A → \mathbb{N}_0 are total mappings such that for every x \in A

- BLANK(x) = |\{ y \in \{x\}^- : \text{Lab}(y) \in \{\text{blank, must_in, must_undec}\}\}|
- UNDEC(x) = |\{ y \in \{x\}^- : \text{Lab}(y) = \text{undec}\}|

Now we are ready to define the conditions under
which an argument becomes eligible to be labelled with must\_undec or must\_in.

A blank argument, say $x$, can be labelled with must\_undec if $x$ has to join (but not yet) the current set of undec arguments because otherwise the under-construction set of in arguments, say $S$, together with $x$ (i.e. $\{x\} \cup S$) will never grow to a preferred extension for one of two reasons as specified in the following definition.

**Definition 12** (must\_undec arguments). Let $H = (A,R)$ be an AF and $\text{Lab} : A \to \mathbb{L}$ be a total mapping, and $x$ be an argument with $\text{Lab}(x) = \text{blank}$ then $x$ is eligible to be labelled with must\_undec if it holds that

$$\exists y \in \{x\}^{-} \text{ with } \text{Lab}(y) \in \{\text{blank, undec, must\_undec}\} \text{ s.t. } \text{BLANK}(y) = 0,$$

or it holds that

$$\exists y \in \{x\}^{+} \text{ with } \text{Lab}(y) = \text{must\_out} \text{ such that } \text{BLANK}(y) = 1.$$

**4.2 A New Pruning Strategy**

We develop a precise description for our new pruning strategy by defining a number of constructs in this section. We start with two important sets MUST\_IN and MUST\_UNDEC that respectively hold must\_in and must\_undec arguments temporarily until they are eventually labeled with in and undec. The advantage of using these sets is that we confine computations to those arguments that truly need further processing, and in consequence we avoid scanning all arguments unnecessarily. One might wonder why we utilize two related notions for apparently the same purpose, for example the set MUST\_IN and the label must\_in seem to denote the same arguments. We note that although the two notions refer to the same set of arguments but they allow for different levels of access: by using must\_in it is computationally easy to check the status of a specific argument at any point of the extension enumeration while the set of MUST\_IN enables an efficient access to the collection of those arguments that need to be finalized with in. Throughout this section we show exactly the usage of MUST\_IN and must\_in as well as the two related structures MUST\_UNDEC nad must\_undec. We refine now the initial labelling and hopeless labellings, taking into account the new labels must\_in and must\_undec.

**Definition 13** (must\_in arguments). Let $H = (A,R)$ be an AF and $\text{Lab} : A \to \mathbb{L}$ be a total mapping, and $x$ be an argument with $\text{Lab}(x) = \text{blank}$ then $x$ is eligible to be labelled with must\_in if it holds that

$$\text{BLANK}(x) = 0 \text{ and undec}(x) \in \{0, |\{y : \text{Lab}(y) = \text{undec and y } \in \{x\}^{\pm}\}|\},$$

or it holds that

$$\exists y \in \{x\}^{+} \text{ with } \text{Lab}(y) = \text{must\_out} \text{ such that } \text{BLANK}(y) = 1.$$

By using the new labels (i.e. must\_in and must\_undec) we identify four cases by which an argument’s label can be deduced from the current labelling: in two cases an argument must be eventually labelled with undec while in the other two cases an argument must end with the in label. The essence of these four cases are similar to the ones introduced in [Doutre and Mening, 2001], but the implementation is totally new as we show throughout the paper.

The question now is why we do not label an argument with in and undec immediately instead of must\_in and must\_undec. Note that labelling an argument with in or undec may trigger new changes in the current labelling (e.g. labelling propagation) and in turn these changes may produce further ones and so on. Thus, to process these cascading changes more efficiently we use must\_in and must\_undec. We explain more precisely the computational benefit of must\_in and must\_undec next.

The set $\text{AF}(x)$ is totally new as we show throughout the paper. These four cases are similar to the ones introduced in Definition 12. We explain more precisely the computational benefit of must\_in and must\_undec next.

**Definition 14** (new initial labelling). Let $H = (A,R)$ be an AF, $S$ be the set of self attacking arguments and $T$ be the set of $\{x : |\{x\}^{-} = 0\}$. Then the initial labelling of $H$ is defined by the union of the following sets:

$$\{(x,\text{blank}) | x \in A \setminus (S \cup T)\} \cup$$

$$\{(x,\text{must\_undec}) | x \in S\} \cup$$

$$\{(x,\text{must\_in}) | x \in T\}.$$

**Definition 15** (new hopeless labellings). Let $H = (A,R)$ be an AF and $\text{Lab} : A \to \mathbb{L}$ be a total mapping. Then Lab is a hopeless labelling of $H$ if and only if there is $x$ with $\text{Lab}(x) = \text{must\_out}$ such that $\text{BLANK}(x) = 0$.

We introduce the notion of non-maximal labellings to describe those labellings with an undec (or must\_undec) argument being attacked by only out (or must\_out) arguments. This is because by labelling an argument, say $x$, with undec we mean to find a preferred extension excluding $x$. But if at some point $x$ becomes acceptable to the current set of in arguments, then we better backtrack because the current labelling will never be maximal. Such labellings are hopeless in the sense that they will never grow to a preferred extension although they might be admissible. Nonetheless, we call such labellings non-maximal to
distinguish them from the hopeless labellings that are inevitably not admissible.

**Definition 16** (non-maximal labellings). Let \( H = (A, R) \) be an AF and \( \text{Lab} : A \rightarrow \mathbb{L} \) be a total mapping. Then \( \text{Lab} \) is a non-maximal labelling of \( H \) if and only if there exists \( x \) with \( \text{Lab}(x) \in \{\text{undec}, \text{must}\_\text{undec}\} \) such that \( \text{BLANK}(x) = 0 \) and \( \text{UNDEC}(x) = 0 \).

Now we describe our new labelling propagation. Basically, labelling propagation might be invoked at any point of the search to identify those arguments that are eligible for \text{must} and \text{must}\_\text{undec} labels.

**Definition 17** (new labelling propagation). Let \( H = (A, R) \) be an AF, \( \text{Lab} : A \rightarrow \mathbb{L} \) be a total mapping, \( x \in A \) be an argument, \( \text{BLANK} : A \rightarrow \mathbb{N}_0 \) and \( \text{UNDEC} : A \rightarrow \mathbb{N}_0 \) be total mappings, \( \text{MUST}\_\text{IN} \subseteq A \) and \text{MUST}\_\text{UNDEC} \subseteq A be sets of arguments then propagate\((x, H, \text{Lab}, \text{BLANK}, \text{UNDEC}, \text{MUST}\_\text{IN}, \text{MUST}\_\text{UNDEC})\) is defined by:

1. for each \( y \in \{x\}^+ \) do
2.1. if \( y \) is eligible for \text{must}\_\text{undec} then
2.1.1. \( \text{Lab}(y) \leftarrow \text{must}\_\text{undec} \)
2.1.2. \( \text{MUST}\_\text{UNDEC} \leftarrow \text{MUST}\_\text{UNDEC} \cup \{y\} \)
2.2. if \( y \) is eligible for \text{must} then
2.2.1. \( \text{Lab}(y) \leftarrow \text{must} \)
2.2.2. \( \text{MUST}\_\text{IN} \leftarrow \text{MUST}\_\text{IN} \cup \{y\} \)
3. for each \( z \in \{y\}^+ \) do
3.1. if \( z \) is eligible for \text{must}\_\text{undec} then
3.1.1. \( \text{Lab}(z) \leftarrow \text{must}\_\text{undec} \)
3.1.2. \( \text{MUST}\_\text{UNDEC} \leftarrow \text{MUST}\_\text{UNDEC} \cup \{z\} \)
3.2. if \( z \) is eligible for \text{must} then
3.2.1. \( \text{Lab}(z) \leftarrow \text{must} \)
3.2.2. \( \text{MUST}\_\text{IN} \leftarrow \text{MUST}\_\text{IN} \cup \{z\} \)
4. if \( \text{Lab} \) is hopeless or non-maximal then return false.
5. return true.

Now we expand \text{in} transitions to include labelling propagation.

**Definition 18** (new \text{in} transitions). Let \( H = (A, R) \) be an AF and \( \text{Lab} : A \rightarrow \mathbb{L} \) be a total mapping, \( s \) be an argument with \( \text{Lab}(s) \in \{\text{blank}, \text{must}\_\text{in}\} \), \( \text{BLANK} : A \rightarrow \mathbb{N}_0 \) and \( \text{UNDEC} : A \rightarrow \mathbb{N}_0 \) be total mappings, \( \text{MUST}\_\text{IN} \subseteq A \) and \text{MUST}\_\text{UNDEC} \subseteq A be sets of arguments then intrans\((s, H, \text{Lab}, \text{BLANK}, \text{UNDEC}, \text{MUST}\_\text{UNDEC}, \text{MUST}\_\text{IN})\) is defined by:

1. \( \text{Lab}(s) \leftarrow \text{in} \)
2. for each \( x \in \{s\}^+ \) with \( \text{Lab}(x) \neq \text{out} \) do
2.1. for each \( y \in \{x\}^+ \) do
2.1.1. if \( \text{Lab}(x) = \text{undec} \) then \( \text{UNDEC}(y) \leftarrow \text{UNDEC}(y) - 1 \)
2.2. \( \text{BLANK}(x) \leftarrow \text{BLANK}(x) - 1 \)
2.3. \( \text{Lab}(x) \leftarrow \text{out} \)
2.4. if \( \text{propagate}(x, H, \text{Lab}, \text{BLANK}, \text{UNDEC}, \text{MUST}\_\text{IN}, \text{MUST}\_\text{UNDEC}) = \text{false} \) then return false.
3. for each \( x \in \{s\}^- \) with \( \text{Lab}(x) \notin \{\text{out}, \text{must}\_\text{out}\} \) do
3.1. \( \text{Lab}(x) \leftarrow \text{must}\_\text{out} \)
3.2. if \( \text{propagate}(x, H, \text{Lab}, \text{BLANK}, \text{UNDEC}, \text{MUST}\_\text{IN}, \text{MUST}\_\text{UNDEC}) = \text{false} \) then return false.
4. return true.

Similarly, we expand \text{undec} transitions to include labelling propagation.

**Definition 19** (new \text{undec} transitions). Let \( H = (A, R) \) be an AF and \( \text{Lab} : A \rightarrow \mathbb{L} \) be a total mapping, \( x \) be an argument with \( \text{Lab}(x) \in \{\text{blank}, \text{must}\_\text{undec}\} \), \( \text{BLANK} : A \rightarrow \mathbb{N}_0 \) and \( \text{UNDEC} : A \rightarrow \mathbb{N}_0 \) be total mappings, \( \text{MUST}\_\text{IN} \subseteq A \) and \text{MUST}\_\text{UNDEC} \subseteq A be sets of arguments then undtrans\((x, H, \text{Lab}, \text{BLANK}, \text{UNDEC}, \text{MUST}\_\text{UNDEC}, \text{MUST}\_\text{IN})\) is defined by:

1. \( \text{Lab}(x) \leftarrow \text{undec} \)
2. for each \( y \in \{x\}^+ \) do
2.1. \( \text{UNDEC}(y) \leftarrow \text{UNDEC}(y) + 1 \)
2.2. \( \text{BLANK}(y) \leftarrow \text{BLANK}(y) - 1 \)
3. return propagate\((x, H, \text{Lab}, \text{BLANK}, \text{UNDEC}, \text{MUST}\_\text{IN}, \text{MUST}\_\text{UNDEC})\).

Before we make a new branch (by \text{in} and \text{undec} transitions) we finalize the label of \text{must} and \text{must}\_\text{undec} arguments with \text{in} and \text{undec} respectively. Every time we relabel a \text{must} (respectively \text{must}\_\text{undec}) argument with \text{in} (respectively \text{undec}) we may find that some blank arguments have become eligible for either \text{must} or \text{must}\_\text{undec}. Thus, a change in some argument’s label may cause other changes in the current labeling and so forth. We define this recurrent process by labelling broadcasting.

**Definition 20** (labelling broadcasting). Let \( H = (A, R) \) be an AF and \( \text{Lab} : A \rightarrow \mathbb{L} \) be a total mapping, \( \text{BLANK} : A \rightarrow \mathbb{N}_0 \) and \( \text{UNDEC} : A \rightarrow \mathbb{N}_0 \) be total mappings, \( \text{MUST}\_\text{IN} \subseteq A \) and \text{MUST}\_\text{UNDEC} \subseteq A be sets of arguments then broadcast\((\text{Lab}, H, \text{BLANK}, \text{UNDEC}, \text{MUST}\_\text{IN}, \text{MUST}\_\text{UNDEC})\) is defined by:

1. while \( \text{MUST}\_\text{IN} \neq \emptyset \) or \( \text{MUST}\_\text{UNDEC} \neq \emptyset \) do
1.1. while \( \text{MUST}\_\text{IN} \neq \emptyset \) do
1.1.1. remove an argument \( x \) from \text{MUST}\_\text{IN}. 

1.2. while MUST_UNDEC ≠ ∅ do
1.2.1. remove an argument x from MUST_UNDEC.
1.2.2. if und-trans(x, H, Lab, BLANK, UNDEC, MUST_IN, MUST_UNDEC)=false then return false.

2. return true.

Using the new in transition, undec transition, and labelling broadcasting, we present algorithm 2. Let $H = (\mathcal{A}, \mathcal{R})$ be an AF, Lab be the initial labelling of $H$, and for each $x \in \mathcal{A}$ let BLANK($x$) be equal to $|\{x\}|$ while UNDEC($x$) be equal to 0, and let MUST_IN be the set \{ $x : |\{x\}| = 0$, MUST_UNDEC be the set \{ $x : |\{x(x)e \in R\}$, and E be an empty set then if we call algorithm 2 on H, Lab, BLANK, UNDEC, MUST_IN, MUST_UNDEC, and E, then the algorithm returns E containing all preferred extensions of $H$. Figure 3 shows a running of algorithm 2. In the next section we add a further enhancement, which is a new argument selection strategy.

4.3 A New Argument Selection Strategy

Referring to line 6 of algorithm 2 there are many possible selection strategies. One possibility is to pick arguments randomly. Another strategy may depend on some heuristic measures, such as the number of adjacent arguments, which is the strategy used in (Nozal et al., 2016). See (Geilen and Thimm, 2017) for more discussions on heuristic-based selection strategies that depend on the current AF labelling and/or its underlying graph structure properties. Here we introduce a different selection strategy that relies on argument history profile as we explain next.

Our selection strategy is simple. Every time we reach a hopeless labelling because of a must_out argument, say $x$, we mark such $x$ as a failure point and later we give priority to the blank attackers of $x$ to be selected for inducing a transition. In other words, our selection strategy prioritizes those arguments that might soon produce a hopeless labelling, instead of delaying the awareness of hopeless labelling possibly until a very late point of the search.

Now we come to the specifications of our selection strategy. We note that a minor modification has to be introduced to algorithm 2, such that every time we detect a hopeless labelling because of a must_out argument, say $x$, the algorithm has to push $x$ on top of a stack structure denoted by $\mathcal{S}$. Algorithm 3 implements our selection strategy. Note that the algorithm either returns a selected argument or it returns -1 to indicate that the current labelling is either hopeless or terminal.

5 Verifying the Efficiency of the New Algorithm

We implemented our new algorithm using the C++ programming language. The source code of the implementation can be found at https://sourceforge.net/projects/argtools. We evaluated the new algorithm using benchmark A of the second international competition of computational models of argumentation 2017 (ICCMA17) (ICC). Benchmark A includes 350 AFs, for more details see (ICC). We carried out the evaluation on a system with an intel-core-i7 processor and four gigabytes of system memory. For each problem instance we limit the memory space to one gigabyte. In contrast, ICCMA17 uses a more powerful environment with an intel-xeon processor and with four gigabytes of memory being allocated for each problem instance. However, with respect to the running time we follow ICCMA17, and hence, set a timeout of 10 minutes for each problem instance.

The objective of this evaluation is to verify that the new algorithm enumerates preferred extensions faster than the old algorithm. We found that the new algorithm is able to solve successfully 233 AFs out of benchmark A. Note that the implementation of the old algorithm is included in ArgTools (first version), which is a labelling-based solver that was submitted to the first version of the competition ICCMA15 (Thimm and Villata, 2017). In fact, the old algorithm did not solve any AF of benchmark A. In its second version, submitted to ICCMA17, ArgTools includes some (not all) aspects of the new algorithm. ArgTools (version 2) enumerated all preferred extensions successfully for 157 AFs (ICC). Another labelling-based solver is Heureka (Geilen and Thimm, 2017), which also participated in ICCMA17. Heureka implements the old algorithm but with a profound heuristic-based argument selection strategy. Heureka enumerated all preferred extensions for 178 AFs of benchmark A.

6 Conclusion

We presented a new labelling algorithm that lists all preferred extensions of a given AF. We evaluated the new algorithm and our findings verified that the new algorithm enumerates preferred extensions sig-
Algorithm 2: New list-preferred-extensions

\begin{algorithm}
\begin{algorithmic}
\State \textbf{input} : $H = (A,R)$, $Lab : A \rightarrow \{\text{MUST, UNDEC}\}$, $S$ is a stack of arguments.
\State \textbf{output}: $x \in A, S$.
\State \textbf{while} $S \neq \emptyset$ \textbf{do}
\State \hspace{1em} $y \leftarrow$ pop an argument from top of $S$;
\State \hspace{1em} \textbf{if} $Lab(y) = \text{must out}$ \textbf{then}
\State \hspace{2em} return $x \in \{y\}$ \textbf{with} $Lab(x) = \text{blank else}$ return $-1$;
\State \hspace{1em} \textbf{foreach} $y \in A$ \textbf{with} $Lab(y) = \text{must out}$ \textbf{do}
\State \hspace{2em} \textbf{if} $Lab(y) > 0$ \textbf{then} return some $x \in \{y\}$ \textbf{with} $Lab(x) = \text{blank else}$ return $-1$;
\State \hspace{1em} \textbf{if} $\exists x$ \textbf{with} $Lab(x) = \text{blank such that}$ UNDEC($x$) $> 0$
\State \hspace{2em} return $x$;
\State \hspace{1em} \textbf{if} $\exists x$ \textbf{with} $Lab(x) = \text{blank then}$ return $x$ \textbf{else} return $-1$;
\end{algorithmic}
\end{algorithm}

significantly faster than the old algorithm. The obtained speedup is due to the new pruning strategy of the new algorithm. We plan to study the impact of our new pruning strategy in the context of other computational problems in the field of abstract argumentation.

Lastly we note that the labelling algorithm of (Nofal et al., 2014b) for preferred extension enumeration enhanced the previous version of (Doutre and Mengin, 2001) (Caminada, 2007). Nevertheless, the algorithm of (Nofal et al., 2014b) has been improved further in (Nofal et al., 2016) by a look-ahead strategy. In this work we build on the algorithm of (Nofal et al., 2016) as we explained throughout the paper. Another mainstream research concerns building reduction-based solvers, see some examples in (Thimm and Villata, 2017). For computational complexity of abstract argumentation see for example (Dunne and Wooldridge, 2009). For a survey on methods for solving different computational problems of AFSs see the article of (Charwat et al., 2015).

Algorithm 3: New selecting an argument for transitions

\begin{algorithm}
\begin{algorithmic}
\State \textbf{input} : $H = (A,R)$, $Lab : A \rightarrow \{\text{MUST, UNDEC}\}$, $S$ is a stack of arguments.
\State \textbf{output}: $x \in A, S$.
\State \textbf{while} $S \neq \emptyset$ \textbf{do}
\State \hspace{1em} $y \leftarrow$ pop an argument from top of $S$;
\State \hspace{1em} \textbf{if} $Lab(y) = \text{must out}$ \textbf{then}
\State \hspace{2em} return $x \in \{y\}$ \textbf{with} $Lab(x) = \text{blank else}$ return $-1$;
\State \hspace{1em} \textbf{foreach} $y \in A$ \textbf{with} $Lab(y) = \text{must out}$ \textbf{do}
\State \hspace{2em} \textbf{if} $Lab(y) > 0$ \textbf{then} return some $x \in \{y\}$ \textbf{with} $Lab(x) = \text{blank else}$ return $-1$;
\State \hspace{1em} \textbf{if} $\exists x$ \textbf{with} $Lab(x) = \text{blank such that}$ UNDEC($x$) $> 0$
\State \hspace{2em} return $x$;
\State \hspace{1em} \textbf{if} $\exists x$ \textbf{with} $Lab(x) = \text{blank then}$ return $x$ \textbf{else} return $-1$;
\end{algorithmic}
\end{algorithm}

REFERENCES

Organized by: Sarah A. Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran.


Figure 3: The search tree that is expanded by algorithm $A_2$ in listing the preferred extensions of $AF_1$ depicted in figure $A_1$.