THEIA: A labeling based backtracking solver for Abstract Argumentation

Bachelor's Project Thesis

Lukas Kinder, s3686566, l.m.kinder@student.rug.nl,
Supervisor: Prof Dr Verheij

Abstract: THEIA is a labeling based algorithm to find complete sets of a Dung argumentation framework. State of the art backtracking solvers do this by repeatedly choosing an argument and label it until either a contradiction with respect to the labels is reached or a solution is found. The main idea of THEIA is to reduce the number of backtracking steps by using propagation techniques that keep track of arguments that cannot be defeated or undefeated. To assess the performance, the program was tested on the data-set of the ICCMA 2019 and was in general faster than the backtracking solvers HEUREKA and DREDD. This result shows that backtracking solvers can be improved by using a bigger set of labels which enable more powerful propagation techniques.

1 Introduction

Computational argumentation is a multidisciplinary area used in Artificial Intelligence, Philosophy, Law and Linguistics [16]. In recent years argumentation has been extensively studied as a subject of Artificial Intelligence [4]. A general introduction to computational argumentation can be found in [3].

The argumentation framework introduced by Dung [6] is a simple but powerful method used to abstract argumentation. It represents arguments as nodes in a graph which can have attack relations to other arguments. Interpreting these graphs by finding meaningful sets of arguments is a non-trivial task that may require great computational time and effort.

State of the art programs for this purpose are for example backtracking algorithms. These algorithms gradually label arguments to analyse the graph and backtrack whenever contradicting labels are found. In this paper, the backtracking solvers HEUREKA [8] and DREDD [15] are discussed and compared to THEIA, the algorithm posed here. Other backtracking solvers working similarly are for example discussed in [13], [12] or [11].

The program THEIA is different from HEUREKA or DREDD because it uses more selective techniques to propagate labels. The basic idea is to keep track of arguments that cannot be defeated or undefeated at some state during the search. This can reduce the number of backtracking steps during the search as conflicting labels can be discovered earlier.

To asses the performance of THEIA the program was compared to HEUREKA and DREDD. To do this, the data-set of the ICCMA 2019 [7] was used which contains 326 argumentation frameworks with a variety of characteristics. The different programs were compared in terms of processing time to find complete sets and correct results given.

The result of THEIA is relevant as it suggests that combining a powerful heuristic like the one of HEUREKA with more efficient propagation techniques may give a further improvement to existing backtracking solvers.

This paper will explain how existing backtracking solvers can work to interpret Dung's argumentation frameworks and how THEIA is able to extend them. Afterwards, the performance of THEIA is assessed compared to other solvers, and strengths and weaknesses of THEIA are analysed. A section about future work describes label propagation techniques that were not used by THEIA but may be implemented by future backtracking solvers.

2 Background

2.1 Dungs Argumentation frameworks

The argumentation framework introduced by Dung [6] consists of a tuple $AF = \langle \mathcal{A}, \mathcal{R} \rangle$. Here $\mathcal{A}$ is the set of arguments and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ is the set of attacks. An attack relation $(a, b) \in \mathcal{R}$ can be interpreted as argument a *attacks* argument b. An example for such an argumentation framework can be seen in Figure 2.1.

For an argument $a \in \mathcal{A}$ let's define a^+ as the set of arguments which are attacked by a and a^- as the set of arguments attacking a. Likewise, given a set of arguments $Args \subseteq \mathcal{A}$ let's define $Args^+ = \{b \mid \exists a \in Args : b \in a^+\}$ and $Args^- = \{b \mid \exists a \in Args : b \in a^-\}$.

An argumentation framework can be interpreted by finding meaningful sets of arguments. A set $E \subseteq \mathcal{A}$ is conflict-free if there is no $a, b \in E$ such that a attacks b. A set E is admissible if E is conflict-free and $E^- \subseteq E^+$. Further, a set E is complete if E is admissible and there is no argument a such that $a \notin E$ and $a^- \subseteq E^+$. A set E is grounded if E is complete and there is no complete set E' such that $E' \subseteq E$. Finally, a set E is stable if E is conflict-free and $E \cup E^+ = \mathcal{A}$. More about relations and properties of these sets can be found in [6].

The grounded set E_{GR} can be found by repetitious addition of all arguments in the set which are not attacked by other arguments or are only attacked by arguments which are themselves attacked by arguments which are already in the set [9]. A straight forward pseudo-code that uses this idea to find the grounded set E_{GR} is shown in Algorithm 1.

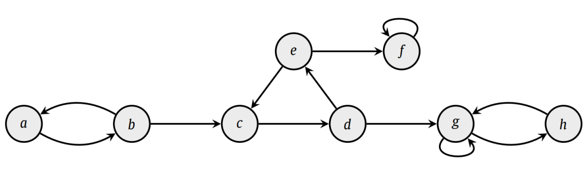


Figure 2.1: An argumentation framework as pictured in the design and result paper of the ICCMA 2019 [7]. Arguments are written down as nodes in the graph and the attack relations are shown as arrows. Note that arguments *f* and *g* attack themselves.

Algorithm 1 The pseudo-code to find the grounded set E_{GR} of an argumentation framework.

```
1:  procedure findGroundedSet($AF$)
2:      input: An argumentation framework $AF = \langle \mathcal{A}, \mathcal{R} \rangle$ with $\mathcal{A}$ being the set of arguments and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ being the set of attacks.
3:      output: The grounded set $E_{GR}$ of the argumentation framework.
4:      $E \leftarrow \{ a \mid a^- = \emptyset \}$
5:      $E' \leftarrow \emptyset$
6:      while $E' \neq E$: do
7:          $E' \leftarrow E$
8:          for all $a \in \mathcal{A}$ do
9:              if $a^- \subseteq E^+$ then
10:                 add $a$ to $E$
11:     return $E$
```

2.2 Basic backtracking solvers for complete sets

In this paper, we focus on algorithms which internally represent sets with a labeling function $\mathcal{L}$ that maps arguments to labels. These label can be for instance IN, OUT, $UNDEC$ or $BLANK$ [5].

$$\mathcal{L} : \mathcal{A} \longmapsto \{IN, OUT, UNDEC, BLANK\} \quad (2.1)$$

Let's define $in(\mathcal{L}) = \{x|\mathcal{L}(x) = IN\}$; $out(\mathcal{L}) = \{x|\mathcal{L}(x) = OUT\}$; $undec(\mathcal{L}) = \{x|\mathcal{L}(x) = UNDEC\}$; $blank(\mathcal{L}) = \{x|\mathcal{L}(x) = BLANK\}$.

This paper is focusing on algorithms to find a complete labeling function $\mathcal{L}_{complete}$ as discussed in [2]:

Definition 2.1. For a complete labeling function $\mathcal{L}_{complete}$ it holds that:

- $in(\mathcal{L}_{complete}) \cup out(\mathcal{L}_{complete}) \cup undec(\mathcal{L}_{complete}) = A$
- For all arguments $a \in in(\mathcal{L}_{complete})$ it holds that there does not exist an argument b such that $(b, a) \in \mathcal{R}$ and $b \in in(\mathcal{L}_{complete})$ or $b \in undec(\mathcal{L}_{complete})$
- For all arguments $a \in out(\mathcal{L}_{complete})$ it holds that there exists an argument b such that $(b, a) \in \mathcal{R}$ and $b \in in(\mathcal{L}_{complete})$
- For all arguments $a \in undec(\mathcal{L}_{complete})$ it holds that there exists an argument b such

that $(b, a) \in \mathcal{R}$ and $b \in undec(\mathcal{L}_{complete})$ and there does not exist an argument c such that $(c, a) \in \mathcal{R}$ and $c \in in(\mathcal{L}_{complete})$

The intuition behind this definition is that arguments labeled IN are only attacked by arguments which are attacked by arguments labeled IN. Therefore, $in(\mathcal{L}_{complete})$ is a complete set. The label $BLANK$ is used during the search process for arguments which do not have another label yet. Note that for a given argumentation framework AF there may be multiple complete labeling functions.

A backtracking algorithm can use the grounded labeling $\mathcal{L}_{grounded}$ as a starting point. The grounded labeling can be derived from the grounded set E_{GR}.

Definition 2.2. A grounded labeling $\mathcal{L}_{grounded}$ is a function mapping arguments to labels such that:

$$\mathcal{L}_{grounded}(a) = \begin{cases} IN & \text{if } a \in E_{GR} \\ OUT & \text{if } a \in E_{GR}^{+} \\ BLANK & \text{otherwise} \end{cases} \tag{2.2}$$

A basic backtracking algorithm to find all complete labeling functions $\mathcal{L}$ of an argumentation framework may start with the grounded labeling before starting the search. This uses the fact that for a given argumentation framework every complete labeling function $\mathcal{L}_{complete}$ has the property that $in(\mathcal{L}_{grounded}) \subseteq in(\mathcal{L}_{complete})$ and $out(\mathcal{L}_{grounded}) \subseteq out(\mathcal{L}_{complete})$. The task of the backtracking solver is then to label all arguments in $blank(\mathcal{L}_{grounded})$ either IN, OUT or $UNDEC$.

This can be done by repeatedly picking an argument a with $\mathcal{L}(a) = BLANK$ and label it IN, OUT or $UNDEC$. This step is called *"splitting the search over an argument"*. The respective new label of the argument can then be used to find labels of other arguments. This may cause a contradiction if it is necessary to re-label an argument that is already labeled IN, OUT or $UNDEC$. This process is repeated until either a contradiction is found when propagating the labels or $\mathcal{L}$ is complete. If this is the case, the algorithm backtracks to the last point in which a label was chosen and tries a different one.

The base structure of the algorithm is similar to a constrain satisfaction solver in which each argument is a variable with the domain $\{IN, OUT, UNDEC\}$ and constrains are posed by the attack relations between arguments. An overview of how finding sets for an argumentation framework can be rephrased as a constrain satisfaction problem can be seen in [1].

The pseudo-code of a backtracking solver can be seen in Algorithm 2. Many existing backtracking solvers, including THEIA, HEUREKA and DREDD follow this pseudo-code. Differences between this class of algorithms come from the way they choose arguments (line 8) and the propagation techniques (line 11).

Algorithm 2 The pseudo-code of a basic backtracking algorithm to find complete labeling functions.

```
1:  procedure findCompleteRec($AF, \mathcal{L}$)
2:      input: An argumentation framework $AF = \langle \mathcal{A}, \mathcal{R} \rangle$ with $\mathcal{A}$ being the set of arguments and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ being the set of attacks. A function $\mathcal{L}$ mapping each argument in $\mathcal{A}$ to a label.
3:      result: Prints all complete sets.
4:
5:      if $\forall a[\mathcal{L}(a) \in \{IN, OUT, UNDEC\}]$ then
6:          printSolution($\mathcal{L}$)
7:          return
8:      $a \leftarrow$ an argument from $\mathcal{A}$ such that $\mathcal{L}(a) \notin \{IN, OUT, UNDEC\}$
9:      for all $l \in \{IN, OUT, UNDEC\}$ do
10:         $\mathcal{L}_{new} \leftarrow \mathcal{L}$ with $\mathcal{L}(a) = l$
11:         if propagateLabels($AF, L_{new}$) = SUCCESSFUL then
12:             findCompleteRec($AF, L_{new}$)
13:
14: $\mathcal{L} \leftarrow \mathcal{L}_{grounded}$
15: findCompleteRec($AF, \mathcal{L}$)
```

The following changes can be made to this algorithm for variations of the task:

- If the task is to find all stable sets, then the algorithm should not label the argument $UNDEC$ when splitting the search over an argument. Otherwise, the procedure stays the same.
- If the task is to check if an argument a is in at least one complete set it can initially be labeled IN and afterwards the algorithm runs until one complete set is found.

Let's call a labeling function $\mathcal{L}$ for an AF *illegal* if there does not exist a complete labeling function $\mathcal{L}_{complete}$ for this AF such that $in(\mathcal{L}) \subseteq in(\mathcal{L}_{complete})$, $out(\mathcal{L}) \subseteq out(\mathcal{L}_{complete})$ and $undec(\mathcal{L}) \subseteq undec(\mathcal{L}_{complete})$. Likewise, let's call a labeling function $\mathcal{L}$ *legal* if there exists a complete labeling function $\mathcal{L}_{complete}$ for this AF such that $in(\mathcal{L}) \subseteq in(\mathcal{L}_{complete})$, $out(\mathcal{L}) \subseteq out(\mathcal{L}_{complete})$ and $undec(\mathcal{L}) \subseteq undec(\mathcal{L}_{complete})$.

The speed of the algorithm is largely dependent on how often it is necessary to backtrack because an *illegal* labeling function is detected. The two key methods to reduce the number of backtracking steps are a heuristic to choose the argument over which the search is split and the mechanism to propagate labels.

2.3 Choosing an argument to split the search

As shown in Algorithm 2 line 8, the solver's architecture requires that an argument is chosen repeatedly to split the search. From a computational point of view an argument should be chosen such that:

1. If the labeling function is *legal*, the program should choose an argument a such that giving it a label creates an *illegal* labeling functions for which a contradiction is quickly found or a *legal* labeling function for which many new labels can be propagated.

2. If the labeling function is *illegal*, the program should choose an argument a such that propagating the new label of a quickly results in a contradiction for each new label of a.

In order to choose an argument it is possible to assign heuristic values to arguments and the argument with the highest heuristic is chosen to split the search. Such a heuristic can be static or dynamic. A static heuristic initially assigns a heuristic value to each argument which does not change during the search. A dynamic heuristic, on the other hand, updates the heuristic of the argument, based on the labels that were assigned so far. An example for a dynamic heuristic for an argument a would be $H(a) = |a^+| + |\{b | b \in a^- \wedge \mathcal{L}(b) \neq OUT\}|$ because the heuristic value changes if an attacker of a gets labeled OUT.

A static heuristic only needs to be computed once in the beginning. Contrarily, a dynamic heuristic requires time to reevaluate the heuristic values during the search, but may compensate this by being able to pick better arguments, based on the current state of $\mathcal{L}$.

The paper about HEUREKA [8] provides an in-depth analysis of a dynamic heuristic that can reduce backtracking steps. The authors provide a list of criteria which were beneficial to use for the heuristic. For each argument $a \in \mathcal{A}$ this includes:

1. The number of arguments attacking a which are not labeled OUT (This value should have a negative weight for the heuristic).

2. The ratio between the number of arguments attacked by a and the number of arguments attacking a which is calculated by $\frac{|a^+|+\epsilon}{|a^-|+\epsilon}$ with $\epsilon \in \mathbb{R}$.

3. The number of outgoing and ingoing paths for a with a *path* being a chain of arguments $[a_1, a_2, ..., a_n]$ such that $(a_k, a_{k+1}) \in \mathcal{R}$ for all $k \in \{1, 2, ..., n-1\}$.

For the version of HEUREKA submitted in the ICCMA 2017, the parameters and weights for this heuristic have been fitted, based on an experimental evaluation. Compared to other solvers participating in the ICCMA 2017, HEUREKA was able to achieve medium good results.

2.4 Propagating the labels through the graph

The architecture of the basic solver shown in Algorithm 2 also repeatedly requires propagating labels through the graph. This means that labels that are newly assigned may recursively enforce labels of other arguments. If a point is reached in which an argument that is already labeled IN, OUT or $UNDEC$ gets re-labeled, a contradiction is found and the algorithm can backtrack. Increasing the efficiency of the program can be achieved by propagating the labels such that:

1. If there is an *illegal* $\mathcal{L}$, then propagating the label should ideally detect a contradiction. This prevents the algorithm from splitting the search over an argument of an *illegal* $\mathcal{L}$ which

would result in an increasing number of search branches with *illegal* $\mathcal{L}$ which need to be backtracked later.

2. If the labeling function is *legal*, then propagating the label should assign new labels to as many arguments as possible. This allows to reach a complete $\mathcal{L}$ as soon as possible without having to split up the search into multiple directions as often.

To find ways to propagate labels Table 2.1 can be used. This table lists the requirements of an argument labeled IN, OUT or $UNDEC$ for a complete labeling function $\mathcal{L}$. The rows of the table can be proven as follows:

1. An argument labeled IN can only be attacked by arguments labeled OUT. This is required, based on the definition of a complete labeling function.
2. An argument a labeled OUT should be attacked by at least one argument labeled IN. Otherwise, there are no restrictions for the labels of other arguments attacking a.
3. An argument labeled $UNDEC$ should not be attacked by an argument labeled IN and should be attacked by at least one argument labeled $UNDEC$. This is enforced by the previous two rows of the table. An argument should be labeled OUT if it is attacked by an argument labeled IN and IN if it's only attacked by argument labeled OUT.

Argument label	Can be attacked by	Can not be attacked by	Is attacked by at least one
IN	$\{OUT\}$	$\{IN, UNDEC\}$	$\emptyset$
OUT	$\{IN, OUT, UNDEC\}$	$\emptyset$	$\{IN\}$
$UNDEC$	$\{OUT, UNDEC\}$	$\{IN\}$	$\{UNDEC\}$

Table 2.1: Restrictions for argument labels of a complete $\mathcal{L}$.

Finding rules to propagate labels can use *forward* relations, *backward* relations and *sideward* relations. If an argument a was assigned a new label a *forward* relation can give an argument in a^+ a new label, a *backward* relation can give an argument in a^- a new label and a *sideward* relation can give an argument in $(a^+)^-$ a new label.

Here are examples for a *forward*, *backward* and *sideward* relation which can be used to assign a new label:

- An example using a **forward** relation: Argument a got labeled OUT and is attacking an argument b that is now only attacked by arguments labeled OUT. Therefore, argument b needs to have the label IN. This uses the fact that an argument labeled IN is only attacked by arguments labeled OUT.
- An example using a **backward** relation: Argument a got labeled $UNDEC$. All arguments attacking a except of argument b are labeled OUT. Therefore, argument b needs to have the label $UNDEC$. This uses the fact that an argument labeled $UNDEC$ is attacked by at least one argument labeled $UNDEC$.
- An example using a **sideward** relation: Argument a got labeled OUT and is attacking an argument b labeled OUT. After a got labeled OUT, all arguments attacking b are labeled OUT or $UNDEC$ except argument c. Therefore, argument c needs to have the label IN. This uses the fact that an argument labeled OUT must be attacked by at least one argument labeled IN.

The algorithm DREDD [15] is using all possible *forward*, *backward* and *sideward* relations which can be derived from Table 2.1. All of these propagation rules can be seen in Table A.1 in the Appendix. DREDD was submitted to the ICCMA 2019 but performed badly because of cases in which the program did not respond or gave wrong outputs.

3 THEIA implementation

The program THEIA is a backtracking algorithm that can be used to find all complete sets. Similar to HEUREKA or DREDD, the arguments of the argumentation framework are gradually labeled and backtracked whenever a contradiction is found. The basic architecture can be seen in Algorithm 2.

THEIA's main focus is to explore new efficient techniques to propagate labels through the argumentation framework. This section will explain the propagation techniques THEIA uses, a new way to assign labels to arguments and the heuristic THEIA uses to choose arguments.

3.1 Improved propagation techniques

Let's call the labels IN, OUT and $UNDEC$ *final* labels and any other labels *temporal* labels. During the search *temporal* labels may still be re-labeled to *final* labels.

To improve the propagation of labels through an argumentation framework, it is possible to keep track of arguments for which it is known that they cannot be labeled IN or OUT. This can be done by giving these arguments the *temporal* label $NOTIN$ or $NOTOUT$. Given a labeling function $\mathcal{L}$ an argument a should be labeled $NOTIN$ if it is known that there does not exist a complete labeling function $\mathcal{L}_{complete}$ with $\mathcal{L}_{complete}(a) = IN$ that can be reached by relabeling the *temporal* labels of $\mathcal{L}$. Similar, given a labeling function $\mathcal{L}$ an argument a should be labeled $NOTOUT$ if it is known that there does not exist a complete labeling function $\mathcal{L}_{complete}$ with $\mathcal{L}_{complete}(a) = OUT$ that can be reached by relabeling the *temporal* labels of $\mathcal{L}$.

By giving arguments the label $NOTIN$ and $NOTOUT$, new propagation techniques can be used to assign labels. An overview of the propagation rules THEIA uses can be seen in Table A.2 in the Appendix. Each propagation rules can be explained as follows:

1. Results from the definition of a complete labeling function.
2. Results from the definition of a complete labeling function.
3. An argument that is attacked by an argument labeled $UNDEC$ and otherwise by arguments that cannot be re-labeled to IN should get labeled $UNDEC$. This is a consequence of the definition of a complete labeling that enforces an argument labeled $UNDEC$ to be attacked by at least one argument labeled $UNDEC$ and no arguments labeled IN.
4. As stated in the definition of a complete labeling function, an argument labeled OUT is attacked by at least one argument labeled IN. If all of the attackers of an argument cannot be re-labeled to IN, this argument cannot have the label OUT.
5. As stated in the definition of a complete labeling function, an argument labeled IN is only attacked by arguments labeled OUT. If at least one of the attackers of an argument cannot be re-labeled to OUT, this argument cannot have the label IN.
6. Results from the definition of a complete labeling function.
7. As stated in the definition of a complete labeling function, an argument labeled OUT is attacked by at least one argument labeled IN. If all except one of the attackers of an argument labeled OUT cannot be re-labeled to IN, this argument must have the label IN.
8. As stated in the definition of a complete labeling function, an argument labeled $UNDEC$ is attacked by at least one argument labeled $UNDEC$. If all except one of the attackers of an argument labeled $UNDEC$ are labeled OUT, this argument must have the label $UNDEC$.
9. As stated in the definition of a complete labeling function, an argument labeled IN is only attacked by arguments labeled OUT. Therefore, an argument labeled $NOTIN$ cannot only be attacked by arguments with the label OUT. Consequently, if all but one attackers of an argument labeled $NOTIN$ have the label OUT, this one argument cannot have the label OUT and should therefore be labeled $NOTOUT$.
10. As stated in the definition of a complete labeling function, an argument attacked by an argument labeled IN must have the label OUT. Therefore, the attackers of an argument that cannot be re-labeled to OUT cannot have the label IN and should be labeled $NOTIN$.
11. Same logic as for **8.**
12. Same logic as for **9.**
13. Same logic as for **7.**

Using these additional propagation rules, it is possible to modify the algorithm in the following way:

- If there are two rules which can be applied to label an argument $NOTIN$ and IN, a contradiction is found.
- If there are two rules which can be applied to label an argument $NOTOUT$ and OUT, a contradiction is found.
- Initially, all self-attacking arguments are labeled $NOTIN$ (This is because self attacking arguments cannot be labeled IN as they would otherwise be attacked by an argument labeled IN).
- If it is possible to apply a rule to label an argument $NOTIN$ that is already labeled OUT or $UNDEC$, no changes are applied.
- If it is possible to apply a rule to label an argument $NOTOUT$ that is already labeled IN or $UNDEC$, no changes are applied.
- If two rules can be applied to label an argument $NOTIN$ and $NOTOUT$ then this argument will be labeled $UNDEC$.
- It is possible to split the search over an argument labeled $NOTIN$ or $NOTOUT$. In this case, there are only two possible ways to relabel the argument (OUT or $UNDEC$ if it was labeled $NOTIN$ and IN or $UNDEC$ if it was labeled $NOTOUT$).

3.2 A new way to split the search over an argument

If an argument gets assigned the label IN, it is more likely that a propagation rule can be applied to label a new argument than if the argument was assigned the label OUT or $UNDEC$. This is because the propagation rules which can be applied because an argument was labeled IN are less restricted. If an argument a got labeled IN, it is possible to label all arguments OUT that are attacking a or are being attacked by a. On the other hand, if an argument gets labeled OUT or $UNDEC$, more specific criteria must be met to be able to apply a propagation rule. In order to increase the efficiency of a program, there is an alternative way to split the search that labels more arguments IN.

Definition 3.1. An argument a is *unjustified* OUT if $\mathcal{L}(a) = OUT$ and there does not exist an argument b such that $(b, a) \in \mathcal{R}$ and $\mathcal{L}(b) = IN$.

Definition 3.2. An argument a is a *potential defeater* of an argument b if $(a, b) \in \mathcal{R}$ and $\mathcal{L}(a) \in \{BLANK, NOTOUT\}$.

For a complete labeling, at least one attacker of an argument labeled OUT should be labeled IN. It is possible to take the n *potential defeater* of an argument that is *unjustified* OUT and split the search by creating a new labeling function for each possible way such that at least one of the *potential defeaters* is labeled IN. Note that there are $\sum_{k=1}^{n} \binom{n}{k}$ different variations in which at least one of the n potential defeaters is labeled IN. Therefore, it is crucial to choose an argument that is *unjustified* OUT and has a small number of *potential defeaters.*

THEIA can keep track of arguments which are *unjustified* OUT and splits the search with this new method if it finds an *unjustified* OUT argument that only has two *potential defeaters.* The pseudocode for this new method can be seen in Algorithm 3.

An example of an argumentation framework for which this new algorithm can help can be seen in Figure 3.1. For this framework there are three complete sets. One with argument a and c being labeled IN, one with all b-arguments labeled IN and one with all arguments being labeled $UNDEC$. Let's assume a basic backtracking solver as shown in Algorithm 2 is run using the propagating techniques in Table A.1. Let's further assume that the order in which the arguments are chosen to split the search favours the b-arguments. In this case the time complexity would be $\mathcal{O}(2^n)$ with n being the number of b-arguments. This is because if an b-argument gets labeled OUT or $UNDEC$, it is not possible to propagate any labels. Therefore, the next b argument is considered to split the search and only after every of the b-arguments got assigned a label, the solutions are found. By doing this, an exponentially growing search tree is created.

This can be avoided by splitting the search over the *potential defeater* of an argument labeled *unjustified* OUT. This is because once one of the b-arguments gets labeled OUT the argument a and c are potential defeaters of this argument. Therefore, the search can be continued by labeling argument

Algorithm 3 THEIAS new algorithm to split the search.

```
procedure findCompleteRec($AF, \mathcal{L}$)
    input: An argumentation framework $AF = \langle \mathcal{A}, \mathcal{R} \rangle$ with $\mathcal{A}$ being the set of arguments and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ being the set of attacks. A function $\mathcal{L}$ mapping each argument in $\mathcal{A}$ to a label.
    result: Prints all complete sets.

    if $\forall a[\mathcal{L}(a) \in \{IN, OUT, UNDEC\}]$ then
        $printSolution(\mathcal{L})$
        return
    if there exists an argument $a$ that is $UNJUSTIFIED\_OUT$ and it only has two potential defeating arguments $b$ and $c$. then
        $\mathcal{L}_{new} \leftarrow \mathcal{L}$ with $\mathcal{L}(b) = IN$ and $\mathcal{L}(c) = NOTIN$
        if propageLabels($AF, \mathcal{L}_{new}$) = SUCCESSFUL then
            findCompleteRec($AF, \mathcal{L}_{new}$)
        $\mathcal{L}_{new} \leftarrow \mathcal{L}$ with $\mathcal{L}(b) = NOTIN$ and $\mathcal{L}(c) = IN$
        if propageLabels($AF, \mathcal{L}_{new}$) = SUCCESSFUL then
            findCompleteRec($AF, \mathcal{L}_{new}$)
        $\mathcal{L}_{new} \leftarrow \mathcal{L}$ with $\mathcal{L}(b) = IN$ and $\mathcal{L}(c) = IN$
        if propageLabels($AF, \mathcal{L}_{new}$) = SUCCESSFUL then
            findCompleteRec($AF, \mathcal{L}_{new}$)
    else
        $a \leftarrow$ an argument from $\mathcal{A}$ labeled $NOTIN$, $NOTOUT$ or $BLANK$ with a higher heuristic than any other argument
        for all $l \in \{IN, OUT, UNDEC\}$ do
            $\mathcal{L}(a) = l$
            if propagateLabels($AF, L$) = SUCCESSFUL then
                findCompleteRec($AF, L$)
$\mathcal{L} \leftarrow \mathcal{L}_{grounded}$
findCompleteRec($AF, \mathcal{L}$)
```

a and/or b IN which avoids an exponential growth of the search tree.

While testing this new technique, it was discovered that this extension can give great improvements in computational time when searching for all complete sets. However, for some problems this extension did not give an improvement but caused the program to take more time. This can be explained with the fact that keeping track of all arguments that are labeled OUT to see if they are *unjustified* OUT takes additional time.

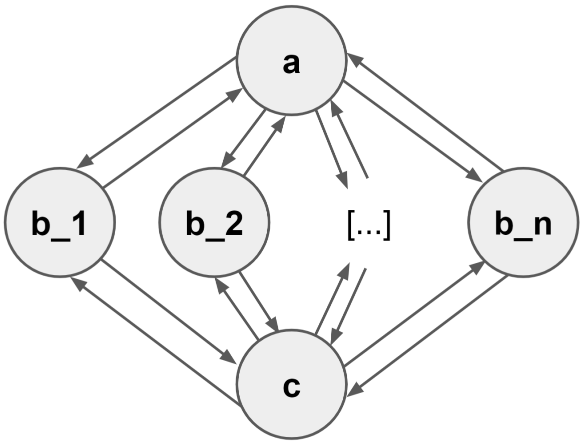


Figure 3.1: An argumentation framework with $\mathcal{A} = \{a, b_k, c\}$ and $\mathcal{R} = \{\langle a, b_k \rangle, \langle b_k, a \rangle, \langle c, b_k \rangle, \langle b_k, c \rangle\}$ for all $k \in \{1, 2, ..., n\}$ with n being a positive number.

Empirically, the extension only gives an improvement in computational time if the average amount of attack relations of arguments is low. This makes sense because if an argument is labeled OUT and the amount of attacking arguments is big, it is more unlikely that it has only two *potential defeaters.*

A simple solution for this is to initially check the average amount of attack relations of the arguments in the argumentation frame. This gives the option to only use the extension if this average is smaller than some certain threshold μ. To investigate this, three different versions of THEIA were created:

- THEIA_BASIC does not keep track of arguments which are *unjustified* OUT and does not split the search over the *potential defeater* of an argument labeled *unjustified* OUT.
- THEIA_EXTENDED always keeps track of arguments which are *unjustified* OUT and splits the search over the *potential defeater* of an argument labeled *unjustified* OUT whenever there are two *potential defeater.*
- THEIA_HYBRID initially calculates the aver-

age number of attack relations of the arguments in the argumentation framework and behaves like THEIA_BASIC if this average is above 50 and like THEIA_EXTENDED if this average is below 50.

For THEIA_HYBRID, $\mu = 50$ was chosen based on experiments with different values. For these experiments, the data-set used are the argumentation frameworks of the ICCMA 2019.

3.3 Heuristic used for THEIA

The dynamic heuristic used to repeatedly choose arguments to split the search is similar but easier to compute than the heuristic used by HEUREKA. The heuristic value of an argument a is $H(a) = n + 10 * isNotin + 8 * isNotOut$. Here n is the number of arguments a is attacking. The variable $isNotin$ is 1 if $\mathcal{L}(a) = NOTIN$ and 0 otherwise. Similarly, the variable $isNotout$ is 1 if $\mathcal{L}(a) = NOTOUT$ and 0 otherwise. The values 8 and 10 were chosen because they let the program perform best in a row of experiments conducted with big argumentation frameworks of the ICCMA 2019.

If there are multiple arguments that are *unjustified OUT* and with two *potential defeaters*, then a random one is chosen to split the search.

4 Results

In order to estimate and compare the performance of THEIA, the data-set of the ICCMA 2019 was used. It contains 326 argumentation frameworks with a variety of different characteristics like the amount of arguments and attacks, size of grounded set, number of complete sets, etc. More information about the argumentation frameworks and how they were generated can be found in [7]. The full data-set can be downloaded under http://argumentationcompetition.org/2019/files.html.

The solver μ-toksida [10] was used to generate the ground truth of the tasks because it did not give any wrong outputs for this data-set, as reported by the ICCMA. The procedure to check correctness of μ-toksida by the ICCMA is elaborated in [7].

The different solvers were tested on this data-set in which they had to find all complete sets. For each argumentation framework they had 60 seconds of CPU time for this task. If they did not find all complete sets by then, they were stopped. #SUCCESS, #TIMEOUT, #ERR, #TIME corresponds to: number of correct answers, number of timeouts, number of crashes and the total time for all problems in seconds, respectively. Each timeout and crash added 60 seconds to #TIME.

First, different versions of THEIA were compared. The results of the different versions of THEIA can be seen in Table 4.1.

Table 4.1: Performance of different versions of THEIA for the 336 argumentation frameworks which were used in the ICCMA 2019.

	#SUCCESS	#TIMEOUT	#ERR	#TIME
THEIA_BASIC	286	40	0	2518
THEIA_EXTENDED	285	41	0	2560
THEIA_HYBRID	290	36	0	2310

The results of THEIA_HYBRID, DREDD and HEUREKA can be seen in Table 4.2. Overall THEIA_HYBRID needed less CPU time to find all complete sets of the 326 argumentation frameworks than the other programs.

Table 4.2: Performance in finding all complete sets of THEIA, DREDD and HEUREKA for the 336 argumentation frameworks which were used in the ICCMA 2019.

	#SUCCESS	#TIMEOUT	#ERR	#TIME
HEUREKA	280	46	0	2891
DREDD	152	155	19	10807
THEIA_HYBRID	290	36	0	2310

Let's call the processing time for an argumentation framework of program X meaningfully faster than program Y if Y needed more than 0.5 seconds of CPU time and program X was more than twice as fast as Y. The reason that only results that needed more than 0.5 seconds are considered is that it is interesting to learn about how the processing time of the programs scales for large problems.

In this sense, HEUREKA was meaningfully faster than THEIA for 28 of the argumentation frameworks and THEIA_HYBRID was 26 times meaningfully faster than HEUREKA. DREDD, on the other hand, was never meaningfully faster than THEIA_HYBRID for any of the 326 argumentation frameworks but THEIA_HYBRID was 168 times meaningfully faster than DREDD.

Experiments were conducted using a wsl-terminal run on an AMD Ryzen 7 3700U 2.3GHz CPU, with 13.9 GB RAM.

5 Discussion

The results provided by different versions of THEIA show that there can be an improvement by splitting the search over an argument that is *unjustified OUT*. In cases in which THEIA_EXTENDED needed more time than THEIA_BAIC the number of times the programs had to backtrack was about the same. On the other hand, in cases in which THEIA_EXTENDED needed less time than THEIA_BAIC, the number of times THEIA_EXTENDED backtracked was much lower. This shows that THEIA_EXTENDED needs time to track arguments that are *unjustified OUT*, but could sometimes reduce the size of the search tree by doing so. THEIA_EXTENDED checks each time when an argument gets labeled OUT, $UNDEC$ or $NOTIN$ if this argument attacks another argument that is *unjustified OUT* and now only has two *potential defeaters*. To do this, THEIA_EXTENDED needs time. The main problem is that if an argument is *unjustified OUT* but has many attacking arguments, it is unlikely that it only has two *potential defeaters*. Therefore, it makes sense to not keep track of arguments which are *unjustified OUT* if the average number of attack relations of the arguments in an argumentation framework is high. By doing this, THEIA_HYBRID is able to perform better than the other versions of THEIA.

The lower processing time THEIA needed compared to HEUREKA and DREDD shows that keeping track of arguments which cannot be IN or OUT at some point during the search can increase the performance of a backtracking solver. However, HEUREKA was able to be meaningfully faster than THEIA for some argumentation frameworks. This can be explained by the heuristic HEUREKA is using to choose an argument. Apparently, there are some argumentation frameworks, for which it is important to choose some key arguments early on to reduce the size of the search tree which HEUREKA is sometimes able to do better than THEIA. An example for such an argumentation framework can be seen in Figure 3.1. For this framework it is important to choose arguments a and c first to split the search because after arguments a and c are labeled the labels for all b-arguments can be found directly. Choosing the b arguments first to split the search can result in an exponentially growing search tree, as explained in Section 3.2.

Even though THEIA_HYBRID was in general faster than HEUREKA, it was the case that HEUREKA was 28 times meaningfully faster than THEIA_HYBRID and THEIA_HYBRID was only 26 times meaningfully faster than HEUREKA. This is because if HEUREKA was faster, the time difference between the solvers was usually not as big, compared to the time difference if THEIA_HYBRID was faster. This indicates that the time THEIA_HYBRID needs scales better with large problems.

DREDD seemed not to be able to seriously compete with HEUREKA or THEIA. This is probably not only due to errors in its code causing crashes or wrong outputs, but also due to its worse heuristic compared to HEUREKA and weaker label-propagation techniques compared to THEIA.

6 Future work

The propagation techniques THEIA uses which can be seen in Table A.2 are all possible forward, backward and sideward relations using the labels IN, OUT, $UNDEC$, $NOTIN$ and $NOTOUT$. However, there are more propagation techniques possible, using rules that can only be applied in a setup with more involving arguments. For example, triangle relations within an argumentation framework often force labels on arguments. A triangle relation is a set of three arguments a, b and c such that the attack relations between a, b and c are: a attacks b, b attacks c and c attacks a. How this can enforce labels on arguments is illustrated in Figure A.1 in the Appendix.

One more way to increase the ability to propagate labels through a graph may be to introduce a new label $NOTUNDEC$ that can be used for an argument that is known to be either IN or OUT during some state in the search similar to the $NOTIN$ or $NOTOUT$ label. An example of a situation in which an argument may be labeled $NOTUNDEC$ is if there are two *potential defeaters* a and b of an argument c that is *unjustified OUT* and $a^- = \{b\}$. In this case, argument b may be labeled $NOTUNDEC$. This is because if b gets labeled $UNDEC$, then a must be labeled $UNDEC$ as well because all of its attackers would have the label $UNDEC$. However, this would not be possible because at least one of the *potential defeaters* of c needs to have the label IN. Future work may investigate how the $NOTUNDEC$ label could be used to create new propagation rules which may give an improvement over existing backtracking solvers.

There is another propagation technique that promises to improve existing backtracking solver. Let's assume we have arguments a and b and $a^- = b^-$. In this case both arguments must have the same label in any complete labeling. This is because the label of an argument is fully dictated by the labels of the arguments attacking it. For example, all b-arguments in the argumentation frame shown in Figure 3.1 must have the same label. Using this relation alone is not necessarily helpful because it is unlikely that two arguments share the same attacking arguments. However, in some cases it is not necessary that all of the attacking arguments are the same.

Let's assume for example, argument a gets labeled OUT. In this case, all arguments b with $\{c | c \in a^- \wedge \mathcal{L}(c) \notin \{OUT, UNDEC, NOTIN\}\} \subseteq b^-$ should be labeled OUT as well. This is because, based on the definition of a complete labeling, if an argument a is labeled OUT, at least one argument in a^- must be labeled IN. Therefore, the set $\{c | c \in a^- \bigwedge \mathcal{L}(c) \notin \{OUT, UNDEC, NOTIN\}\}$ must contain at least one argument that gets labeled IN to reach a complete labeling.

In general, these propagation techniques can be helpful because labeling based backtracking solvers have the tendency to create an exponentially growing search tree for big argumentation frameworks. Therefore, label-propagation techniques, even if they have a time complexity that is a polynomial of n with n being the number of arguments, can make the algorithm more efficient if they reduce the exponential growth factor.

7 Conclusion

This paper discussed two new techniques for backtracking solvers in abstract argumentation. The main technique is to use advanced propagation techniques to be able to detect contradictions in the labeling function earlier. The second is a new way of generating new branches in the search tree by looking at the *potential defeaters* of an argument labeled *unjustified* OUT.

The advanced propagation techniques proved to be very useful and TEHEIA_HYBRID was able to outperform DREDD and HEUREKA in the task of finding all complete sets.

Secondly, the new way of splitting the search over an argument labeled *unjustified* OUT gave an improvement for argumentation frames for which the probability of an argument attacking some other argument is low. This is interesting because other solvers often perform better on argumentation frameworks with a high degree of attack relations among arguments (see for example [14]). Therefore, the new techniques to generate new search branches may be useful as a tool for specific argumentation frameworks.

Currently, THEIA can only find all complete sets, but it should be possible to adapt it to find only stable sets or to check if an argument is *sceptically* or *credulously* accepted in a similarly as existing backtracking solvers, as described in section 2.2.

Interestingly, even though the solver THEIA gave better overall results than the solver HEUREKA, it was still the case that HEUREKA outperformed THEIA for some argumentation frameworks. HEUREKA seemed to be able to do this by compensating its more limited propagation techniques with its heuristic to dynamically re-order arguments to reduce the number of backtracking steps. For future work, it would be interesting to see if it is possible to combine the heuristic of HEUREKA and the propagation techniques of THEIA to create an efficient solver, benefiting from both sides.

The **C** code of THEIA is available under https://github.com/LukasKinder/THEIA.

References

[1] L. Amgoud and C. Devred. Argumentation frameworks as constraint satisfaction problems. In *International Conference on Scalable Uncertainty Management*, pages 110–122. Springer, Berlin, 2011.

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

[3] P. Baroni, F. Toni, and B. Verheij. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games: 25 years later. *Argument Comput.*, 11(1-2):1–14, 2020.

[4] T. J. Bench-Capon and P. E. Dunne. Argumentation in artificial intelligence. *Artificial intelligence*, 171(10–15):619–641, 2007.

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

[6] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial intelligence*, 77(2):321–357, 1995.

[7] S. A. Gaggl, T. Linsbichler, M. Maratea, and S. Woltran. Design and results of the second international competition on computational models of argumentation. *Artificial Intelligence*, 279, 2020.

[8] N. Geilen and M. Thimm. Heureka: a general heuristic backtracking solver for abstract argumentation. In *Proceedings of the 2017 International Workshop on Theory and Applications of Formal Argument (TAFA'17)*, pages 143–149, 2017.

[9] S. Modgil and M. Caminada. Proof theories and algorithms for abstract argumentation frameworks. In *Rahwan Y, Simar G (eds.) Argumentation in artificial intelligence*, pages 105–129. Springer, Heidelberg, 2009.

[10] A. Niskanen and M. Järvisalo. μ-toksia: An efficient abstract argumentation reasoner. In *Calvanese D, Erdem E, Thielscher M (eds.) Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 800–804. KR 2020, Rhodes, Greece, 2020.

[11] S. Nofal, K. Atkinson, and P. E. Dunne. Algorithms for argumentation semantics: labeling attacks as a generalization of labeling arguments. *Journal of Artificial Intelligence Research*, 49:635–668, 2014.

[12] S. Nofal, K. Atkinson, and P. E. Dunne. Algorithms for decision problems in argument systems under preferred semantics. *Artificial Intelligence*, 207:23–51, 2014.

[13] S. Nofal, K. Atkinson, and P. E. Dunne. Looking-ahead in backtracking algorithms for abstract argumentation. *International Journal of Approximate Reasoning*, 78:265–282, 2016.

[14] O. Rodrigues. A forward propagation algorithm for the computation of the semantics of argumentation frameworks. In *Black E, Modgil S, Oren N (eds.) International Workshop on Theorie and Applications of Formal Argumentation*, pages 120–136. Springer, Cham, 2017.

[15] M. Thimm. Dredd-a heuristics-guided backtracking solver with information propagation for abstract argumentation. *The Third International Competition on Computational Models of Argumentation (ICCMA'19)*, page url: https://www.iccma2019.dmi.unipg.it/submissions.html, 2019.

[16] D. Walton and D. M. Godden. The impact of argumentation on artificial intelligence. *Considering Pragma-Dialectics*, pages 287–299, 2006.

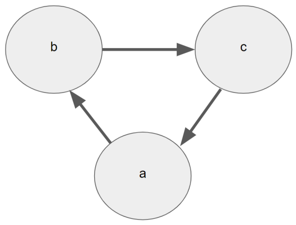
A Appendix

Relation	Argument a got labeled:	prerequisite	New resulting label
Forward relations	$\mathcal{L}(a) = IN$	$(a,b) \in \mathcal{R}$	$\mathcal{L}(b) = OUT$
	$\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \wedge \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) = OUT]$	$\mathcal{L}(b) = IN$
	$\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = UNDEC$	$(a,b) \in \mathcal{R} \wedge \forall c[(c,b) \in \mathcal{R} \longrightarrow (\mathcal{L}(c) = OUT \vee \mathcal{L}(c) = UNDEC)] \wedge \exists d[(d,b) \in \mathcal{R} \wedge \mathcal{L}(d) = UNDEC]$	$\mathcal{L}(b) = UNDEC$
Backward relations	$\mathcal{L}(a) = IN$	$(b,a) \in \mathcal{R}$	$\mathcal{L}(b) = OUT$
	$\mathcal{L}(a) = OUT$	$(b,a) \in \mathcal{R} \wedge \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \vee \mathcal{L}(c) = OUT \vee \mathcal{L}(c) = UNDEC)]$	$\mathcal{L}(b) = IN$
	$\mathcal{L}(a) = UNDEC$	$(b,a) \in \mathcal{R} \wedge \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \vee \mathcal{L}(c) = OUT)]$	$\mathcal{L}(b) = UNDEC$
Sideward relations	$\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \wedge \mathcal{L}(b) = UNDEC \wedge \exists c[(c,b) \in \mathcal{R} \wedge \forall d[(d,b) \in \mathcal{R} \longrightarrow (\mathcal{L}(d) = OUT \vee d = c)]]$	$\mathcal{L}(c) = UNDEC$
	$\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = UNDEC$	$(a,b) \in \mathcal{R} \wedge \mathcal{L}(b) = OUT \wedge \exists c[(c,b) \in \mathcal{R} \wedge \forall d[(d,b) \in \mathcal{R} \longrightarrow (\mathcal{L}(d) = OUT \vee \mathcal{L}(d) = UNDEC \vee d = c)]]$	$\mathcal{L}(c) = IN$

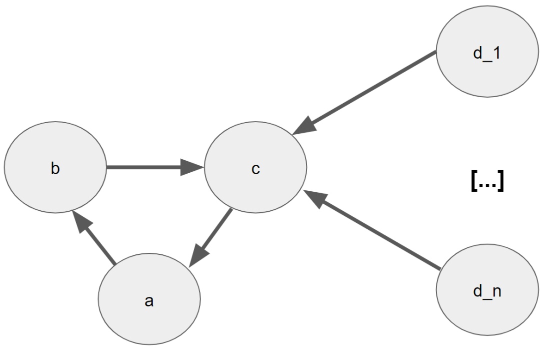
Table A.1: All propagation rules which can be derived from Table 2.1 and are used by DREDD.

Relation	Argument a got labeled:	prerequisite	New resulting label
Forward relations	**1.)** $\mathcal{L}(a) = IN$	$(a,b) \in \mathcal{R}$	$\mathcal{L}(b) = OUT$
	2.) $\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \wedge \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) = OUT]$	$\mathcal{L}(b) = IN$
	3.) $\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = UNDEC$ **or** $\mathcal{L}(a) = NOTIN$	$(a,b) \in \mathcal{R} \wedge \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) \in \{OUT, UNDEC, IN\}]$ $\wedge \exists d[(d,b) \in \mathcal{R} \wedge \mathcal{L}(d) = UNDEC]$	$\mathcal{L}(b) = UNDEC$
	4.) $\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = NOTIN$	$(a,b) \in \mathcal{R} \wedge \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) \in \{NOTIN, OUT\}]$	$\mathcal{L}(b) = NOTOUT$
	5.) $\mathcal{L}(a) = NOTOUT$ **or** $\mathcal{L}(a) = UNDEC$	$(a,b) \in \mathcal{R}$	$\mathcal{L}(b) = NOTIN$
Backward relations	**6.)** $\mathcal{L}(a) = IN$	$(b,a) \in \mathcal{R}$	$\mathcal{L}(b) = OUT$
	7.) $\mathcal{L}(a) = OUT$	$(b,a) \in \mathcal{R} \wedge \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b$ $\vee \mathcal{L}(c) \in \{OUT, UNDEC, NOTIN\})]$	$\mathcal{L}(b) = IN$
	8.) $\mathcal{L}(a) = UNDEC$	$(b,a) \in \mathcal{R} \wedge \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \vee \mathcal{L}(c) = OUT)]$	$\mathcal{L}(b) = UNDEC$
	9.) $\mathcal{L}(a) = NOTIN$	$(b,a) \in \mathcal{R} \wedge \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \vee \mathcal{L}(c) = OUT)]$	$\mathcal{L}(b) = NOTOUT$
	10.) $\mathcal{L}(a) = NOTOUT$ **or** $\mathcal{L}(a) = UNDEC$	$(b,a) \in \mathcal{R}$	$\mathcal{L}(b) = NOTIN$
Sideward relations	**11.)** $\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \wedge \mathcal{L}(b) = UNDEC \wedge (c,b) \in \mathcal{R} \wedge \forall d[(d,b) \in \mathcal{R}$ $\longrightarrow (\mathcal{L}(d) = OUT \vee d = c)]$	$\mathcal{L}(c) = UNDEC$
	12.) $\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \wedge \mathcal{L}(b) = NOTIN \wedge (c,b) \in \mathcal{R} \wedge \forall d[(d,b) \in \mathcal{R}$ $\longrightarrow (\mathcal{L}(d) = OUT \vee d = c)]$	$\mathcal{L}(c) = NOTOUT$
	13.) $\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = UNDEC$ **or** $\mathcal{L}(a) = NOTIN$	$(a,b) \in \mathcal{R} \wedge \mathcal{L}(b) = OUT \wedge (c,b) \in \mathcal{R} \wedge \forall d[(d,b) \in \mathcal{R}$ $\longrightarrow (d = c \vee \mathcal{L}(a) \in \{OUT, UNDEC, NOTIN\})]$	$\mathcal{L}(c) = IN$

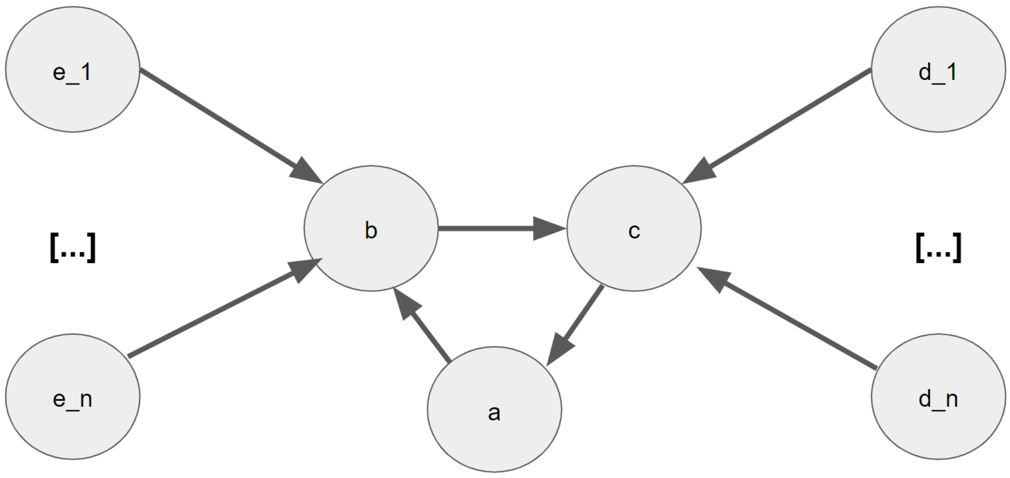
Table A.2: A table summarising all propagation rules THEIA is using. The rules in blue rows are also used by DREDD, the rules in purple are used by DREDD without the $NOTIN$ and $NOTOUT$ label and the red rules are only used by THEIA.



(a) A triangle relation. There may be attack relations from a, b and c to some other argument but no external argument is attacking a, b or c.



(b) A triangle relation. There may be attack relations from a, b and c to some other argument but no external argument is attacking a or b.



(c) A triangle relation. There may be attack relations from a, b and c to some other argument but no external argument is attacking a.

Figure A.1: Three triangle relations: In (a) none of the arguments can have the label IN or OUT. In (b) c can not have the label IN, a can not have the label OUT and b can not have the label IN. And in (c) b can not have the label IN. This is a consequence of propagation rules 1) and 2) in Table A.2. For example, if argument b gets labeled IN, then argument c must be labeled OUT and argument a would get labeled IN because it is only attacked by c. This would mean that argument b is attacked by an argument labeled IN which is not possible because it has the label IN.