A Labelling-based Solver for Computing Complete Extensions of Abstract Argumentation Frameworks

Lukas Kinder[1], Matthias Thimm[2] and Bart Verheij[1,*]

[1]Bernoulli Institute of Mathematics, Computer Science and Artificial Intelligence, Nijenborgh 9, 9747 AG Groningen, The Netherlands
[2]Artificial Intelligence Group, University of Hagen, Universitätsstr. 11, 58097 Hagen, Germany

Abstract
THEIA is a labeling-based system computing the complete extensions of an abstract argumentation framework. Like other backtracking solvers, THEIA does this by repeatedly choosing an argument and labelling it until either a contradiction with respect to the labels is reached or a complete extension is found. THEIA reduces the number of backtracking steps needed by using propagation techniques that use a larger set of labels. These labels keep track of arguments that cannot be labeled IN, OUT or UNDEC during a state in the search. THEIA is also using an extensive look-ahead strategy to prune branches. It is shown that THEIA outperforms related labeling-based backtracking solvers by the use of more sophisticated propagation and pruning rules and forward-looking strategy.

Keywords
Abstract argumentation, backtracking solvers

1. Introduction

Argumentation theory is a multidisciplinary area connected to philosophy, law and linguistics [1, 2, 3, 4]. In recent years, argumentation has been extensively studied formally and computationally, especially also since Phan Minh Dung presented abstract argumentation frameworks [5] (see also [6]). Abstract argumentation frameworks are a simple but powerful approach to formal argumentation. They represent arguments as nodes in a graph which attack other arguments. Interpreting these graphs by finding meaningful sets of arguments is a non-trivial task that may require significant computational time and effort. For instance, so-called backtracking algorithms gradually label arguments to analyse the graph and backtrack whenever contradicting labels are found. Examples of backtracking solvers are for example [7, 8, 9, 10, 11, 12].

THEIA, the system presented here, enumerates the set of complete extensions of an abstract argumentation framework. THEIA is in particular inspired by HEUREKA [7] and DREDD [8]. The system THEIA differs from HEUREKA and DREDD due to its use of more sophisticated rules during propagation and pruning and by a forward looking strategy. The basic idea is to keep

SAFA'22: Fourth International Workshop on Systems and Algorithms for Formal Argumentation 2022, September 13, 2022, Cardiff, Wales, United Kingdom

*The research reported in this paper is based on Lukas Kinder's Bachelor thesis 'THEIA: A labeling based backtracking solver for abstract argumentation' supervised by Bart Verheij (`fse.studenttheses.ub.rug.nl/25428/`), and extended in collaboration with Matthias Thimm.

$$\begin{tikzcd}
A \arrow[r, leftrightarrow] \arrow[d] & B \arrow[dr] & C \arrow[loop above] \\
D \arrow[r] & E \arrow[r] & F \arrow[u]
\end{tikzcd}$$

Figure 1: An argumentation framework. Note that argument C is attacking itself.

track of arguments that cannot be labeled IN, OUT, or $UNDEC$ while searching for complete extensions. This can reduce the number of backtracking steps during the search as conflicting labels can be discovered early.

This paper gives an overview of the relevant background and related work in Section 2. Section 3 gives an overview of the system's design, which elaborates on propagation techniques, the look-ahead strategy and heuristics used. This is followed by an experimental evaluation in Section 4 and a discussion in Section 5. The code of `THEIA` is available at `github.com/LukasKinder/THEIA`.

2. Background

An abstract argumentation framework [5] is 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 attack relation. A pair $(a, b) \in \mathcal{R}$ can be interpreted as argument a `attacking` argument b. An example for an argumentation framework is shown in Figure 1.

For an argument $arg \in \mathcal{A}$, define arg^+ as the set of arguments that are `attacked` by arg and arg^- as the set of arguments `attacking` arg. Likewise, given a set of arguments $Args \subseteq \mathcal{A}$, 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 do not exist any $arg_1, arg_2 \in E$ such that arg_1 `attacks` arg_2. 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 does not exist an argument arg such that $arg \notin E$ and $arg^- \subseteq E^+$. The grounded set is the smallest (wrt. set inclusion) complete set. More about relations and properties of these sets can be found in [5].

A variant of the above set-based interpretation of argumentation frameworks, is the labelling-based interpretation [13]. Here, each argument is assigned a label IN, OUT or $UNDEC$ using a labeling function $\mathcal{L}$.

$$\mathcal{L} : \mathcal{A} \longmapsto \{IN, OUT, UNDEC\}$$

Define $in(\mathcal{L}) = \{x | \mathcal{L}(x) = IN\}$; $out(\mathcal{L}) = \{x | \mathcal{L}(x) = OUT\}$; $undec(\mathcal{L}) = \{x | \mathcal{L}(x) = UNDEC\}$.

Definition 1. ***A labeling is is a complete labeling $\mathcal{L}_{complete}$ if and only if:***

- *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 $b \in in(\mathcal{L}_{complete})$.*

Note that this implicitly defines that 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 $c \in in(\mathcal{L}_{complete})$. Note that $\mathcal{L}_{complete}$ is a complete labelling only if $in(\mathcal{L}_{complete})$ is a complete extension [13].

2.1. Existing Solvers

In this work, we are interested in the problem of enumerating all complete extensions of a given abstract argumentation framework AF, which is highly intractable [14]. This motivates the need to develop sophisticated algorithms. A method that can be applied for this task is the DPLL-algorithm (Davis-Putnam-Logemann-Loveland) [15]. The structure of a basic backtracking algorithm is shown in Algorithm 1 (where the procedure to select the next argument and the procedure *propagateLabels* are left unspecified, these are defined by the concrete system). Every iteration an argument is chosen and each possible label for this argument is tried out. The respective new label is then propagated to find labels of other arguments. This may cause a contradiction if it is necessary to re-label an argument. A solution is reached once all arguments got assigned a label. An intuitive way to think about this algorithm is that a search tree is created in which new branches are generated whenever different labels of an argument are tried out. This method is for example used by `ArgTools` [10], HEUREKA [7] and DREDD [8].

The above-mentioned systems are referred to as direct solvers, which directly work within the formalism of abstract argumentation to solve problems [16]. However, the state of the art in solvers for abstract argumentation typically rely on reductions to other problem-solving paradigms such as SAT-solving or answer set programming. For example, μ-toksia [17] is a solver that makes iterative calls to a SAT-solver to enumerate all complete extensions.

Algorithm 1 The pseudo-code of a basic backtracking algorithm.

```
procedure FINDCOMPLETE($AF$)
    if all arguments are labeled then
        print(Solution($AF$))
        return
    $a \leftarrow$ choose an unlabeled argument
    for all labels do
        label $a$ with the next label
        if propagateLabels($AF$) != CONTRADICTION then
            findComplete($AF$)
```

3. Algorithm Design

Our algorithm is following the basic structure of Algorithm 1. However, THEIA differs from other direct solvers insofar as it uses a significantly extended set of propagation techniques, a look-ahead strategy to find contradicting labels earlier, and chooses an argument to split the search based on the exact amount of labels that can be propagated. These three components are discussed in the following sections.

3.1. Additional Propagation Techniques

The procedure *propagateLabels* corresponds to unit propagation in SAT-solving [15]. It sets the labels of further arguments, whose status is already determined by the set labels. For example, if an argument has been labelled IN, it is clear that all arguments attacked by this argument must be labelled OUT. Existing solvers [10, 7, 8] use this and similar rules to implement the procedure *propagateLabels*.

In the following, we significantly extend the set of existing propagation rules by introducing additional labels. The labels $NOTIN$, $NOTOUT$ and $NOTUNDEC$ can be assigned for arguments for which it is known that they cannot be labeled IN, OUT or $UNDEC$ respectively. We refer to the labels IN, OUT and $UNDEC$ as *final* labels, and to the respective opposite labels $NOTIN$, $NOTOUT$ and $NOTUNDEC$ as *intermediate* labels. We use the label $BLANK$ for unlabeled arguments. During the search, arguments with the label $BLANK$ may be relabeled to intermediate or final labels and arguments with intermediate labels may still be relabeled to final labels.

We distinguish the propagation rules by the position of the argument that gets assigned a label, relative to the argument that causes the rule to apply:

- **Forward rule:** Applying the propagation rule on an argument arg forces a new label on an argument in arg^+.
- **Backward rule:** Applying the propagation rule on an argument $a$rg forces a new label on an argument in arg^-.
- **Sideward rule:** Applying the propagation rule on an argument arg forces a new label on an argument in $(arg^+)^-$.

Table 1 contains propagation rules for argument labels. The three basecases can be applied to arguments without considering the labels of other arguments. In practice, checking if a propagation rule can be applied to an argument a can be done in $\mathcal{O}(|a^+|)$. This requires each argument to keep track of how many attackers are labeled OUT, $UNDEC$ or $NOTIN$.

Situations in which basecases can be applied are shown in Figure 6 and situations in which the propagation rules can be applied are shown in Figure 7 in the appendix. Note that the propagation rules 7 and 13, 9 and 14, 11 and 15 as well as 12 and 16 are all pairs that essentially express the same. The difference is that the label assignment of the argument that causes the rule to apply, is at a different position.

We omit the explanation of each of these rules, but give two examples:

- **Rule 3:** An argument that is attacked by at least one argument labeled $UNDEC$ and otherwise only by arguments that cannot be relabeled to IN (I.e. OUT, $UNDEC$ and

Table 1
A collection of propagation rules.

	Argument a has label:	Prerequisite	Enforced label
Basecase	-	$a^- = \emptyset$	$\mathcal{L}(a) = IN$
	-	$a \in a^-$	$\mathcal{L}(a) = NOTIN$
	-	$\{a\} = a^-$	$\mathcal{L}(a) = UNDEC$
Forward rules	**1.)**$\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \land \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) = OUT]$	$\mathcal{L}(b) = IN$
	2.)$\mathcal{L}(a) = IN$	$(a,b) \in \mathcal{R}$	$\mathcal{L}(b) = OUT$
	3.)$\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = UNDEC$ **or** $\mathcal{L}(a) = NOTIN$	$(a,b) \in \mathcal{R} \land \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) \in \{OUT, UNDEC, NOTIN\}] \land \exists d[(d,b) \in \mathcal{R} \land \mathcal{L}(d) = UNDEC]$	$\mathcal{L}(b) = UNDEC$
	4.)$\mathcal{L}(a) = NOTOUT$ **or** $\mathcal{L}(a) = UNDEC$	$(a,b) \in \mathcal{R}$	$\mathcal{L}(b) = NOTIN$
	5.)$\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = NOTIN$	$(a,b) \in \mathcal{R} \land \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) \in \{NOTIN, OUT\}]$	$\mathcal{L}(b) = NOTOUT$
	6.)$\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = NOTUNDEC$	$(a,b) \in \mathcal{R} \land \forall c[(c,b) \in \mathcal{R} \longrightarrow \mathcal{L}(c) \in \{OUT, NOTUNDEC\}]$	$\mathcal{L}(b) = NOTUNDEC$
Backward rules	**7.)**$\mathcal{L}(a) = OUT$	$(b,a) \in \mathcal{R} \land \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \lor \mathcal{L}(c) \in \{OUT, UNDEC, NOTIN\})]$	$\mathcal{L}(b) = IN$
	8.)$\mathcal{L}(a) = IN$	$(b,a) \in \mathcal{R}$	$\mathcal{L}(b) = OUT$
	9.)$\mathcal{L}(a) = UNDEC$	$(b,a) \in \mathcal{R} \land \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \lor \mathcal{L}(c) = OUT)]$	$\mathcal{L}(b) = UNDEC$
	10.)$\mathcal{L}(a) = NOTOUT$ **or** $\mathcal{L}(a) = UNDEC$	$(b,a) \in \mathcal{R}$	$\mathcal{L}(b) = NOTIN$
	11.) $\mathcal{L}(a) = NOTIN$	$(b,a) \in \mathcal{R} \land \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \lor \mathcal{L}(c) = OUT)]$	$\mathcal{L}(b) = NOTOUT$
	12.)$\mathcal{L}(a) = NOTUNDEC$	$(b,a) \in \mathcal{R} \land \forall c[(c,a) \in \mathcal{R} \longrightarrow (c = b \lor \mathcal{L}(c) = OUT)]$	$\mathcal{L}(b) = NOTUNDEC$
Sideward rules	**13.)**$\mathcal{L}(a) = OUT$ **or** $\mathcal{L}(a) = UNDEC$ **or** $\mathcal{L}(a) = NOTIN$	$(a,b) \in \mathcal{R} \land \mathcal{L}(b) = OUT \land (c,b) \in \mathcal{R} \land \forall d[(d,b) \in \mathcal{R} \longrightarrow (d = c \lor \mathcal{L}(d) \in \{OUT, UNDEC, NOTIN\})]$	$\mathcal{L}(c) = IN$
	14.)$\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \land \mathcal{L}(b) = UNDEC \land (c,b) \in \mathcal{R} \land \forall d[(d,b) \in \mathcal{R} \longrightarrow (\mathcal{L}(d) = OUT \lor d = c)]$	$\mathcal{L}(c) = UNDEC$
	15.)$\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \land \mathcal{L}(b) = NOTIN \land (c,b) \in \mathcal{R} \land \forall d[(d,b) \in \mathcal{R} \longrightarrow (\mathcal{L}(d) = OUT \lor d = c)]$	$\mathcal{L}(c) = NOTOUT$
	16.)$\mathcal{L}(a) = OUT$	$(a,b) \in \mathcal{R} \land \mathcal{L}(b) = NOTUNDEC \land (c,b) \in \mathcal{R} \land \forall d[(d,b) \in \mathcal{R} \longrightarrow (\mathcal{L}(d) = OUT \lor d = c)]$	$\mathcal{L}(c) = NOTUNDEC$

$NOTIN$) should get labeled $UNDEC$. This is because (in a complete labeling) an argument that is not attacked by an argument labeled IN can not be labeled OUT and argument that is attacked by an argument labeled $UNDEC$ can not be labeled IN, which leaves $UNDEC$ as the only option.

- **Rule 11:** This rule is applied in a situation in which we have an argument for which it is known that it cannot have the label IN. This means that it should not be the case that all attackers are labeled OUT. Consequently, if all attackers except one argument b are labeled OUT, then b should be marked with $NOTOUT$.

It should be quite easy to see that all rules depicted in Table 1 are correct.

3.2. Look-Ahead

In each iteration, before an argument gets chosen to split the search, THEIA is testing every possible label for all arguments with non-final labels (a similar idea had already been suggested in [9]). Note that the number of combinations for this is $6 * n\_blank + 2 * n\_intermediate$ with n_blank being the number of arguments labeled $BLANK$ and $n\_intermediate$ being the number of arguments having intermediate labels. After assigning a label to an argument, this label is propagated and it is checked if this causes a contradiction (A contradiction is caused if a propagation rule forces a label on an argument that can not be relabeled based on its current label). If this is the case, the argument permanently gets labeled the opposite label. For example if labeling an argument $NOTIN$ was leading to a contradiction, then it gets labeled IN. If assigning the opposite label also leads to a contradiction, then the whole branch needs to be backtracked.

3.3. Choosing an Argument to Split the Search

THEIA repeatedly chooses an argument arg that does not have a final label yet to split the search. For this argument two labels are selected and propagated respectively. If arg had the label $BLANK$, then the two labels need to be a final label and the opposite of the the final label (either $IN/NOTIN$, $OUT/NOTOUT$ or $UNDEC/NOTUNDEC$). If arg had an intermediate label, then there are only two distinct labels arg can be relabeled to. For example if arg had the label $NOTOUT$, then l_1 and l_2 need to be IN and $UNDEC$.

During the look-ahead phase, all possible labels of all arguments with nonfinal labels were already tested out and propagated. Therefore, for all arguments Y with nonfinal labels and for any label X such that Y can be relabeled to X we know the amount Z such that when assigning label X to argument Y and propagating this label causes Z other arguments to be assigned a label.

The knowledge about the amount Z for every argument-label combination is used as a heuristic to decide how to split the search, in the sense that an argument-label combination is chosen such that the lowest amount of remaining solving time is expected. An intuitive heuristic would be to choose the argument-label combination such that the sum of propagated labels in both branches is maximal. However, the remaining solving time of a branch usually increases exponentially with the number of unlabeled arguments. Therefore, being able to propagate n labels in both resulting branches is usually much better than propagation $2n$ labels in one branch and 0 in the other. This idea motivated us to experiment with three other heuristics elaborated in the results section.

3.4. Combining All Steps

The pseudocode outlining the system THEIA is shown in Algorithm 2. Given an argumentation framework AF the basecases are used first to find the initial labels of arguments. The propagation rules are recursively used to propagate labels for each option and the algorithm backtracks as soon as a contradiction is found. The look-ahead method is not only used to detect contradictions, but also to determine additional label assignments. A complete set is found as soon as all arguments got assigned final labels.

Note that opposed to Algorithm 1, during label-propagation there will never be a contradiction. This is because the look-ahead method already detects if labeling any argument with any label leads to a contradiction and prunes the branch accordingly.

An example of how the complete extensions of an argumentation framework are found is shown in Figure 2.

Algorithm 2 THEIA

```
procedure findComplete($AF$)
    $\mathcal{L} \leftarrow$ such that $\mathcal{L}(a) = BLANK$ for all $a \in \mathcal{A}$
    for all $a \in \mathcal{A}$ and all basecases $b$ do
        if $b$ enforeces label $l$ on $a$ then
            propagateLabel($AF, \mathcal{L}, b, l$)
    findCompleteRec($AF, \mathcal{L}$)

procedure findCompleteRec($AF, \mathcal{L}$)
    if lookAhead($AF, \mathcal{L}$) = CONTRADICTION then
        return
    if $\mathcal{L}$ is a complete labeling then
        print all labels with label $IN$
        return
    Choose an argument $a$ that can be relabeled to $l_1$ or $l_2$.
    for $l \in \{l_1, l_2\}$ do
        propagateLabel($AF, \mathcal{L}, a, l$)
        findCompleteRec($AF, \mathcal{L}$)
        reverse label assignments by propagateLabels() and findCompleteRec()

procedure propagateLabel($AF, \mathcal{L}, a, l$)
    $\mathcal{L}(a) \leftarrow l$
    for all propagation rules $p$ do
        for all $b, l\_new$ such that applying $p$ on $a$ enforces label $l\_new$ on $b$ do
            propagateLabel($AF, \mathcal{L}, b, l_{new}$)
```

4. Experimental Evaluation

The goal of our experimental evaluation is to compare the runtime behaviour of THEIA with existing solvers on the problem of enumerating complete extensions (i. e., on the track EE-CO of the ICCMA competitions; see argumentationcompetition.org).

4.1. Experimental Setup

We evaluate the runtime performance on the same benchmark data as used in the ICCMA15 (192 instances) and ICCMA19 (326 instances) competitions (we did not use the benchmark data from the ICCMA17 and ICCMA21 competitions, due to the size of the solutions for these data sets and our resource restrictions).

THEIA is written in the C programming language and is not using any external libraries. We used four different heuristics for selecting the next argument during search, giving rise to four different variants of our THEIA solver:

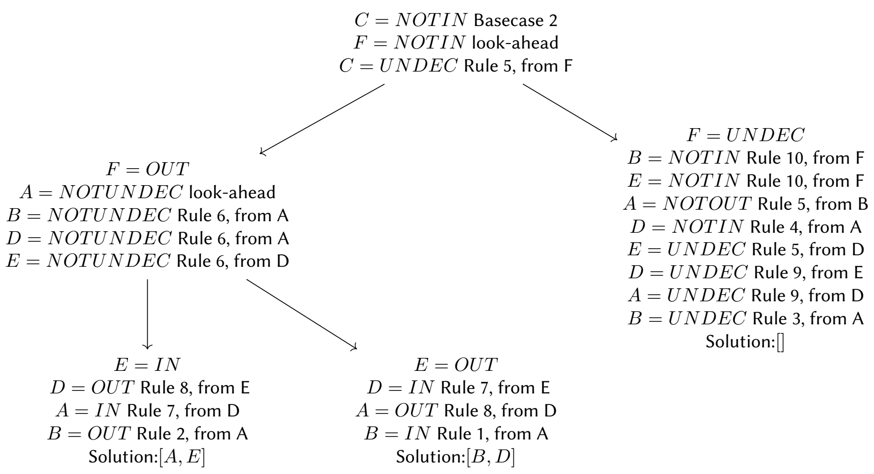

$C = NOTIN$ Basecase 2
$F = NOTIN$ look-ahead
$C = UNDEC$ Rule 5, from F
$F = OUT$
$A = NOTUNDEC$ look-ahead
$B = NOTUNDEC$ Rule 6, from A
$D = NOTUNDEC$ Rule 6, from A
$E = NOTUNDEC$ Rule 6, from D
$F = UNDEC$
$B = NOTIN$ Rule 10, from F
$E = NOTIN$ Rule 10, from F
$A = NOTOUT$ Rule 5, from B
$D = NOTIN$ Rule 4, from A
$E = UNDEC$ Rule 5, from D
$D = UNDEC$ Rule 9, from E
$A = UNDEC$ Rule 9, from D
$B = UNDEC$ Rule 3, from A
Solution:[]
$E = IN$
$D = OUT$ Rule 8, from E
$A = IN$ Rule 7, from D
$B = OUT$ Rule 2, from A
Solution:$[A, E]$
$E = OUT$
$D = IN$ Rule 7, from E
$A = OUT$ Rule 8, from D
$B = IN$ Rule 1, from A
Solution:$[B, D]$

Figure 2: An example of a search graph produced by THEIA when solving the argumentation framework in Figure 1. Labeling argument F to IN would force B and E to be labeled OUT, which would consequently force A and D to be labeled IN. However, if A is labeled IN, then D should instead be labeled OUT. This contradiction is found using the look-ahead strategy, which causes F to be labeled $NOTIN$.

- THEIA$_{\text{min}}$: Given an argument-label combination that can split the search, lets call the "short branch" the branch that results in a lower amount of labels propagated than the other branch. The min-heuristic is choosing the argument-label combination with the longest short branch.
- THEIA$_{\text{sum}}$: The argument-label combination is chosen that maximises the sum of propagated labels in both resulting branches.
- THEIA$_{\text{exp}}$: The solving time of a branch is estimated using $3^{\frac{1}{20}*n_{BLANK}} * 2^{\frac{1}{20}*n_{INTERMEDIATE}}$ with n_{BLANK} being the number of arguments labeled $BLANK$ and $n_{INTERMEDIATE}$ being the number of arguments with intermediate labels. The argument-label combination is chosen that minimizes the cumulative estimated solving time of both resulting branches.
- THEIA$_{\text{adp}}$: The idea of this heuristic is that it adapts itself to the given argumentation framework. The solving time of a branch is estimated using $(3 - 3 * avgErr)^{\frac{n_{BLANK}}{1+decBlank}} * (2 - 2 * avgErr)^{\frac{n_{INTERMEDIATE}}{1+decIntermediate}}$ with n_{BLANK} being the number of arguments labeled $BLANK$ and $n_{INTERMEDIATE}$ being the number of arguments with intermediate labels. $avgErr$ is the proportion of branches that are backtracked and $decBlank$ and $decIntermediate$ are the average number of arguments with a $BLANK$ or intermediate label respectively, that are relabeled each iteration. The values $avgErr$, $decBlank$ and $decIntermediate$ are initially 0, but these values get approximated while the solver is exploring the search tree of a given argumentation framework.

We compared the runtime performance of these four THEIA solvers with μ-toksia version 2019.04.07 [17] and HEUREKA version 0.2 [7]. The μ-toksia solver is based on iterative SAT calls and can be regarded as the state-of-the-art solver for that problem, since that version won the EE-CO track in ICCMA 2019 and the competition of 2021 did not feature the EE-CO track. The solver HEUREKA can be regarded as a baseline for direct approaches to solve EE-CO.

We used `probo2` (available at `github.com/aig-hagen/probo2`) as evaluation environment and conducted the experiments on a dedicated server with Intel Xeon CPUs (3.4 GHz, only a single CPU was used) with 192 GB RAM and running Ubuntu 20.04.4. We set a 600s CPU-time cutoff time, as used in the ICCMA competitions.

4.2. Results

Table 3 and Table 2 show the results for the two benchmark data sets ICCMA15 and ICCMA19, respectively. For each solver we report the number of correctly solved instances (Corr), the number of times the solver could not give the correct answer within the given time (TO) and the cumulative runtime over solved instances (RT). The performances of the solvers are further compared in Figures 3 and 4. We also provide scatter plots for both benchmark data sets comparing THEIA$_{\text{exp}}$ and μ-toksia which are shown in Figure 5a and 5b, respectively.

Table 2
Results for ICCMA 2015

Solver	Corr	TO	RT
μ-toksia	192	0	1790
THEIA$_{\text{exp}}$	189	3	8545
THEIA$_{\text{sum}}$	188	4	9962
THEIA$_{\text{adp}}$	187	5	9989
THEIA$_{\text{min}}$	145	47	1364
HEUREKA	139	53	1298

Table 3
Results for ICCMA 2019

Solver	Corr	TO	RT
μ-toksia	326	0	601
THEIA$_{\text{exp}}$	326	0	3061
THEIA$_{\text{sum}}$	326	0	4146
THEIA$_{\text{adp}}$	326	0	4214
THEIA$_{\text{min}}$	303	23	3808
HEUREKA	285	41	943

We observe that all versions of THEIA timed out a significantly smaller number of times than HEUREKA. Except for THEIA$_{\text{min}}$ the amount of timeouts was almost as good as μ-toksia, but μ-toksia still dominated in regard to the cumulative runtime over solved instances. However, as the scatter plots 5a and 5b show, the runtime differences between THEIA$_{\text{exp}}$ and μ-toksia are rarely larger than one order of magnitude (those are instances outside of the green bar) and there is also a number of instances where THEIA$_{\text{exp}}$ outperformed μ-toksia. Although the overall performance of THEIA is still below that of μ-toksia we see the results as a significant step forward in the development of direct argumentation solvers.

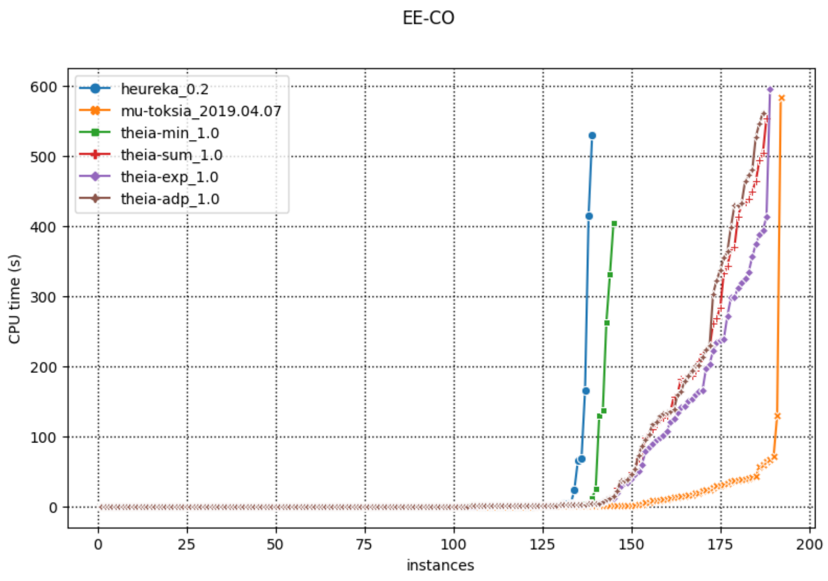


Figure 3: Cactus plot of runtimes for the ICCMA 2015 benchmark.

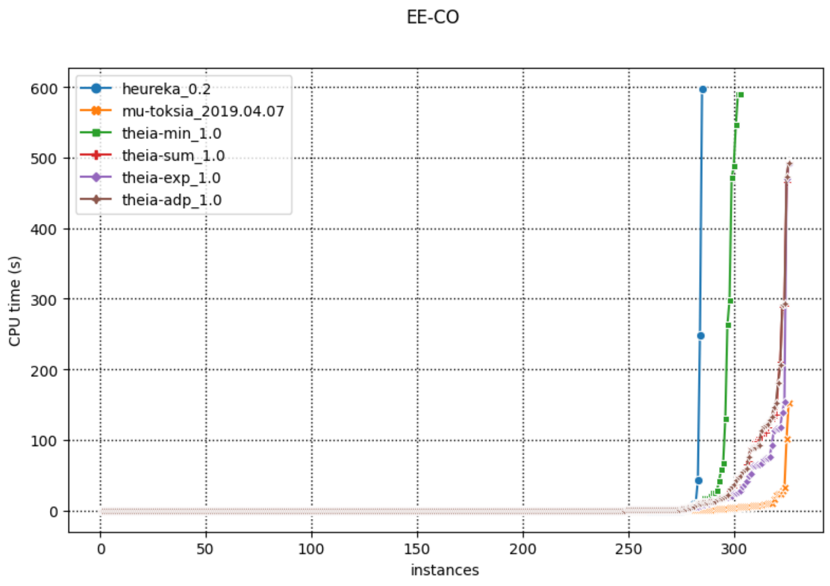
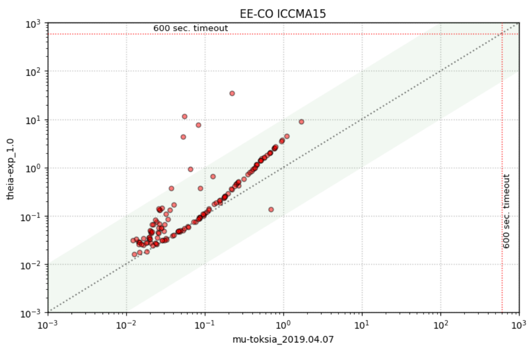


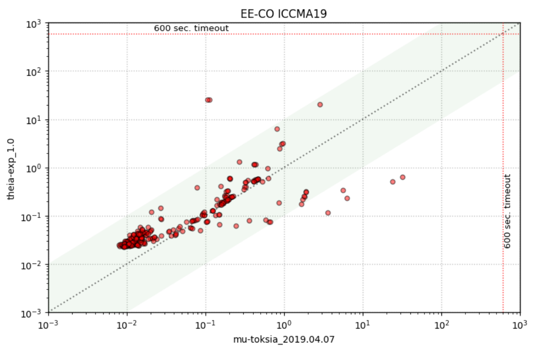
Figure 4: Cactus plot of runtimes for the ICCMA 2019 benchmark.

5. Summary and Conclusion

In this paper we presented the system THEIA that extends current backtracking solvers for abstract argumentation by means of more refined propagation techniques and look-ahead strategies to find complete sets. The results show a clear improvement over the backtracking



(a) ICCMA 2015



(b) ICCMA 2019

Figure 5: Scatter plot comparing the performance in runtime between μ-toksia (x-axis) and THEIA$_{\text{exp}}$ (y-axis). The task was to enumerate the complete extensions of the argumentation frameworks of the ICCMA 2015 benchmark (a) and the ICCMA 2019 benchmark (b).

solver HEUREKA. We were also able to reduce the performance gap between SAT-based solvers like μ-toksia and direct solvers.

In future work, it would be interesting to investigate which types of argumentation frameworks THEIA is better or worse than other solvers, as well as to analyse how much the different implemented mechanisms contribute to the performance.

References

[1] T. J. Bench-Capon, P. E. Dunne, Argumentation in artificial intelligence, Artificial intelligence 171 (2007) 619–641.

[2] P. Baroni, D. Gabbay, M. Giacomin, L. van der Torre, Handbook of Formal Argumentation, London, England: College Publications, 2018.

[3] K. Atkinson, P. Baroni, M. Giacomin, A. Hunter, H. Prakken, C. Reed, G. Simari, M. Thimm, S. Villata, Towards artificial argumentation, AI magazine 38 (2017) 25–36.

[4] F. H. van Eemeren, B. Garssen, E. C. W. Krabbe, A. F. Snoeck Henkemans, B. Verheij, J. H. M. Wagemans, Handbook of Argumentation Theory, Springer, Berlin, 2014.

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

[6] P. Baroni, F. Toni, B. Verheij, Introduction to the special issue 'On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games: 25 years later', Argument & Computation 11 (2020) 1–14.

[7] N. Geilen, 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), 2017, pp. 143–149.

[8] M. Thimm, Dredd-a heuristics-guided backtracking solver with information propagation

for abstract argumentation, in: The Third International Competition on Computational Models of Argumentation (ICCMA'19), 2019.

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

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

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

[12] S. Nofal, K. Atkinson, P. E. Dunne, I. O. Hababeh, A new labelling algorithm for generating preferred extensions of abstract argumentation frameworks, in: ICEIS, 2019.

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

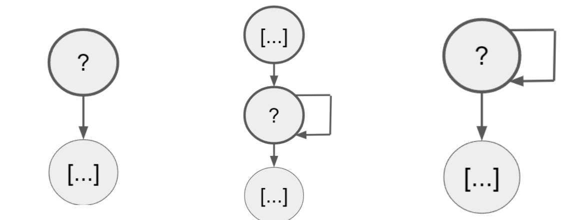
[14] M. Kröll, R. Pichler, S. Woltran, On the complexity of enumerating the extensions of abstract argumentation frameworks, in: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17), 2017.

[15] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009.

[16] F. Cerutti, S. A. Gaggl, M. Thimm, J. P. Wallner, Foundations of implementations for formal argumentation, in: Handbook of Formal Argumentation, College Publications, 2018.

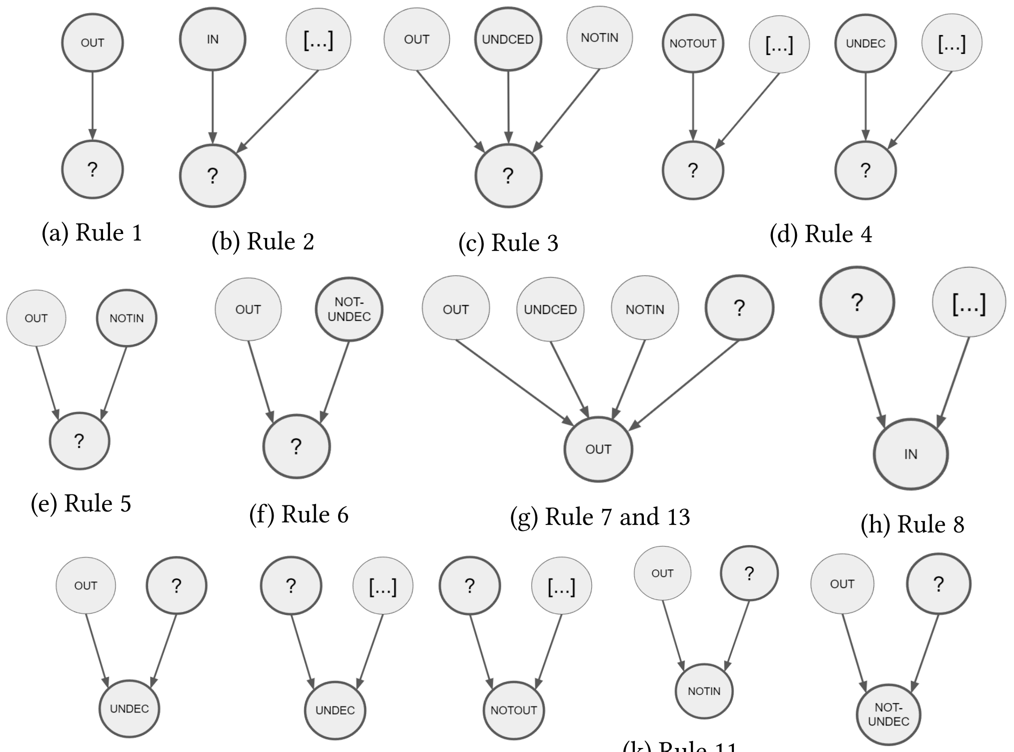
[17] A. Niskanen, M. Järvisalo, μ-toksia: An efficient abstract argumentation reasoner, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, volume 17, 2020, pp. 800–804.

Appendix



(a) Basecase 1 (b) Basecase 2 (c) Basecase 3

Figure 6: A collection of structures of arguments that enforce a label on the node marked with "?". Nodes with a thick border are necessary, nodes with a thin boarder are optional.



(a) Rule 1 (b) Rule 2 (c) Rule 3 (d) Rule 4
(e) Rule 5 (f) Rule 6 (g) Rule 7 and 13 (h) Rule 8
(i) Rule 9 and 14 (j) Rule 10 (k) Rule 11 and 15 (l) Rule 12 and 16

Figure 7: A collection of structures of arguments that enforce a label on the node marked with "?". A Nodes with a thick border represent one or more arguments with a specific label. Nodes with a thin boarder mark zero or more arguments with a specific label. The "[...]" marks arguments that can have any label.