FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

A Randomized Approach to Reasoning in Argumentation Frameworks Based on Random Walks

Bachelor's Thesis

in partial fulfillment of the requirements for
the degree of Bachelor of Science (B.Sc.)
in Computer Science

submitted by
Dominik Hillmann

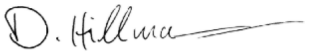
First examiner:	Prof. Dr. Matthias Thimm Artificial Intelligence Group
Advisor:	Prof. Dr. Matthias Thimm Artificial Intelligence Group

Statement

I hereby certify that this thesis has been composed by me and is based on my own work, that I did not use any further resources than specified – in particular no references unmentioned in the reference section – and that I did not submit this thesis to another examination before. The paper submission is identical to the submitted electronic version.

	Yes	No
I agree to have this thesis published in the library.	☒	☐
I agree to have this thesis published on the webpage of the artificial intelligence group.	☒	☐
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	☒	☐
The source code is available under a GNU General Public License (GPLv3).	☒	☐
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	☒	☐

Magdeburg, October 24, 2022



Zusammenfassung

Die Abschlussarbeit folgt der Implementierung und Evaluation eines Algorithmus zum Finden einer zulässigen Menge von Argumenten in einem Argumentationssystem. Dieser Algorithmus basiert auf einem Random Walker, der den Argumenten Label zuordnet. Die Zuordnung ist dabei abhängig vom Pfad, den der Random Walker zum aktuellen Argument nahm als auch der Elternknoten dieses Arguments. Zusätzlich beschreibt die Abschlussarbeit einen Vergleich mit anderen Solvern, eine Analyse von Zeit- und Raumkomplexität und einen Korrektheitsbeweis des Algorithmus.

Abstract

This thesis will follow the implementation and analysis of an algorithm for finding an admissible set in an argumentation framework. The algorithm is based on a random walker assigning labels to arguments. The assignment is dependent on the path the walker took towards its current location as well as the neighborhood of the current argument. Additionally, the thesis provides a comparison of different solvers, analyses of space and time complexity and a proof of correctness.

Contents

1 Introduction and motivation

The following sections give a short overview of formal argumentation and continue with the motivation of this thesis.

1.1 Introduction to formal argumentation

In contrast to currently more popular approaches to artificial intelligence, like artificial neural networks and various other machine learning methods, formal argumentation tries to explicitly model knowledge. Because if this explicitly encoded knowledge, formal argumentation belongs to the category of knowledge-based systems [6]. If explainability is an important requirement, formal argumentation can be a better choice than systems with implicitly encoded knowledge due to a human's ability to better understand the arguments and how they relate to each other [8].

In formal argumentation, one possible knowledge base is called an argumentation framework. An argumentation framework is a directed graph $F = (A, R)$ where the vertices A represent the arguments themselves, and the edges R between them portray the relationship of one argument attacking another. Intuitively, Dung describes the attack mechanism as "[t]he one who has the last word laughs best" [14, p. 1]. This means that a statement can be considered as valid as long as there are no other arguments attacking it, or all of those arguments can be dismantled by attacks on themselves.

Argumentation frameworks also have a graphical representation. An edge pointing from node a to node b means "argument a attacks argument b". For example, the

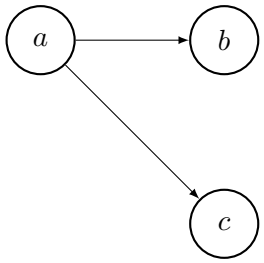


Figure 1: a attacks b and c.

argumentation framework $F = (A, R)$ in figure 1 is made of three nodes $A = \{a, b, c\}$ and two directed edges $R = \{(a, b), (a, c)\}$ using the notation $(x, y) \in R \Leftrightarrow x \rightarrow y$.

From this point of view, one can identify arguments which are acceptable. An acceptable argument is either not attacked or it is defended against every attack on itself. A set of acceptable arguments is an admissible set. Thus, an acceptable argument is one which is not attacked by any unattacked arguments. Based on this approach one can calculate admissible sets of arguments for an argumentation framework F which, according to Dung [14], must fulfil two conditions listed in [4, p. 23]:

1. The set is conflict-free. No arguments that are contained in the set attacks another argument in the set.

2. For every argument within the set that is attacked from the outside, the set must contain an argument that attacks the attacking argument (i.e. every attacked argument gets defended from within the set).

Taking a look at figure 1, both b and c are attacked by argument a. Therefore, b and c are not acceptable arguments and an admissible set $E \subseteq A$ does not contain them. In contrast to that, a does not have any attackers. Hence, a set of admissible arguments is $E = \{a\}$.

Based on the concept of the admissible set, one can calculate the extension of an argumentation framework. To decide whether a single argument or a set of arguments is acceptable, Dung [14, pp. 2] describes multiple ways of computing sets of "arguments that are justifiable as a whole" [30, p. 1] (extensions) based on an understanding of what attributes this group is supposed to have (argumentation semantics). Examples of such extensions and their semantics are:

- E is a **complete extension** iff it is an admissible set and every argument defended by the set is contained in the set [4, p. 178].

- E is a **preferred extension** iff it is the largest possible admissible set in the argumentation framework [4, p. 183].

- E is a **stable extension** iff it attacks any argument that does not belong to E [14, p. 2].

1.2 Motivation

These extensions have to be computed, and there are two common approaches to do so. One approach is reduction-based [7, pp. 2626] and the other is a direct implementation [7, p. 2638]. Reduction-based methods try to translate the argumentation problem into another problem class for which there are already advanced solvers (e.g. constraint-satisfaction problems CSP). The direct approach creates an algorithm to solve an argumentation problem under a specific semantics. Labeling-based approaches are a common implementation. Here, each argument from A gets a assigned a labeling $\mathtt{x} \in \Lambda = \{\mathtt{in}, \mathtt{out}, \mathtt{undec}\}$ meaning that the assignee is accepted, rejected or undecided. Labels get assigned via a total function $\mathcal{L}ab : A \mapsto \Lambda$. Using the expression $\mathtt{in}(\mathcal{L}ab), \mathtt{out}(\mathcal{L}ab), \mathtt{undec}(\mathcal{L}ab) \subseteq A$ one can express the sets of all arguments labeled `in`, `out` or `undec`. To show which arguments belong to which labeling, one can also use the notation $\mathcal{L}ab = (\mathtt{in}(\mathcal{L}ab), \mathtt{out}(\mathcal{L}ab), \mathtt{undec}(\mathcal{L}ab))$. Also, depending on the algorithm, the number and and kinds of labels can vary.

Such algorithms can be fully deterministic (e.g. enumeration [7, pp. 2638]) or make use of randomization. [30] introduces such algorithm in the form of stochastic

local search to find a stable extension. In terms of labels a stable extension is given if the argumentation framework is labeled such that $\texttt{undec}(\mathcal{L}ab) = \emptyset$ [30, p. 3]. Similarly, this thesis will try to formulate, implement and analyze an algorithm that tries to find an admissible set to base an extension on in a given argumentation framework.

2 The algorithm

This section starts with an intuitive description of the algorithm, continues a with a formalization and ends with on an example to apply the algorithm to.

2.1 Algorithm outline

The algorithm's basic idea for deciding whether an argument a belongs to an admissible set is as follows:

- **Input:** An argumentation framework $F = (A, R)$ and an initial argument $a \in A$ where the first random walker will start.
- **Output:** An admissible set $E \subseteq A$ with $a \in E$ or NO if no such admissible set was found.

In the beginning, every argument gets labelled `undec` and the set of random walkers is empty $W = \emptyset$. A random walker W_0 [22] spawns at the initial argument $a \in A$ which gets labeled $\mathcal{L}ab(a) = \texttt{in} \in \Lambda$. Also, the walker is added to the set of all random walkers $W := W \cup \{W_0\}$. From this point on, the algorithm cycles through all available random walkers $W_i \in W$, each of which will behave in the following manner when arriving at or being set to an argument $b \in A$. First, b's parent nodes $P_b := \{p \in A \mid p \to b\}$ have to be considered.

- In case $\mathcal{L}ab(b) = \texttt{in}$, a random walker W_j will be set onto any $p \in P_b$ which is not labeled `out`. Label these $p \in P_b$ `out`. Terminate the random walker W_i that currently occupies b which means that the set of random walkers should be $W := (W \setminus \{W_i\}) \cup \{W_j\}$ for every newly created W_j.
- If $\mathcal{L}ab(b) = \texttt{out}$ and if there is any parent $p \in P_b$ such that $\mathcal{L}ab(p) = \texttt{in}$, then W_i is terminated. Otherwise, W_i moves to a randomly drawn $p \in P_b$ by means of a uniform distribution and labels it `in`. If there is no parent available, the walker will also be terminated.
- The case $\mathcal{L}ab(b) = \texttt{undec}$ does not have to be considered because everytime a walker is set onto an argument the argument gets labeled `in` or `out` beforehand as the random walkers moves there or is created.

These steps repeat as long as W contains a random walker. If the set W is empty, the algorithm will return $\texttt{in}(\mathcal{L}ab)$.

Since the random walkers only move in the direction of the parent arguments, there still remains the problem of how to transmit the label change information to its child arguments. An argument can either be labeled `in` or `out`.

- If an argument gets labeled `out`, then all of its children will be `in` dependent on the children's other parents also being labeled `out`.

- If an argument gets labeled `in`, then all children will be labeled `out` with the condition that all of the child's parents are labeled `in`.

If an argument has its label changed, its child arguments will be labeled the opposite label (only considering `in` and `out`) and their child arguments will, in turn, be labeled with the opposite labels of their parents with the same condition that all other parents need to have the same label as well. This cascade flows along labeling the arguments alternatingly until the conditions for child labeling are not met or the cascade encounters a which is not supposed to be relabeled.

2.2 Formal description of the algorithm

In order to better understand the whole algorithm and make parts of it reusable, it is broken up into four parts. Algorithm 1 describes the main framework of the overall algorithm. It is easy to overview that it iterates over the random walkers present in W and behaves differently according to the argument label the walker is currently placed on. This behavior is characterized in algorithms 2 and 3. Both, in turn, take advantage of the children labeling algorithm 4, which is encapsulated independently to be reusable.

Algorithm 1 The algorithm for finding an admissible set from an argumentation framework $F = (A, R)$ and a starting argument $a \in A$.

```
1:  procedure FINDADMISSIBLESET(F, a)
2:      W ← {W_0}
3:      place W_0 onto a
4:      Lab(a) ← in
5:
6:      for W_i ∈ W do
7:          x ← the node W_i is placed on
8:
9:          if Lab(x) = in then
10:             HANDLEWALKERSETONINARGUMENT(x, W_i)        ▷ See algorithm 2.
11:
12:         else if Lab(x) = out then
13:             HANDLEWALKERSETONOUTARGUMENT(x, W_i)       ▷ See algorithm 3.
14:
15:     return in(Lab)
```

2.3 An example

The argumentation framework shown in figure 2 has multiple features to better understand the algorithm in an example. First, there is a cycle which consists of the

Algorithm 2 Algorithm that defines the behavior of a walker that is set onto an argument a labeled `in`.

procedure HANDLEWALKERSETONINARGUMENT($a \in A$, $W_i \in W$)
    $P_a \leftarrow \{p \in A \mid p \rightarrow a\}$

    for $p \in \{p \in P_a \mid \mathcal{L}ab(p) \neq \texttt{out}\}$ **do**
        $\mathcal{L}ab(p) \leftarrow \texttt{out}$
        LABELCHILDREN(p, `out`) ▷ See algorithm 4
        create a new W_j and place it onto p.
        $W \leftarrow W \cup \{W_j\}$

    terminate W_i
    $W \leftarrow W \setminus \{W_i\}$

Algorithm 3 Algorithm that defines the behavior of a walker that is set onto an argument a labeled `out`.

procedure HANDLEWALKERSETONOUTARGUMENT($a \in A$, $W_i \in W$)
    $P_a \leftarrow \{p \in A \mid p \rightarrow a\}$

    for $p \in P_a$ **do**
        if $\mathcal{L}ab(p) = \texttt{in}$ **then**
            terminate W_i
            $W \leftarrow W \setminus \{W_i\}$
            return

    $p_{\text{random}} \leftarrow$ randomly select with equal distribution from P_a
    $\mathcal{L}ab(p_{\text{random}}) \leftarrow \texttt{in}$
    LABELCHILDREN(p_{random}, `in`) ▷ See algorithm 4.
    move W_i to p_{random}

Algorithm 4 Algorithm for labeling the children of $a \in A$ with the a's label $\texttt{label} \in (\Lambda \setminus \{\texttt{undec}\})$ and also appropriatly label the following children.

```
1:  procedure LABELCHILDREN(a, label)    ▷ Not recursive due to memory limit.
2:      C_a ← {c ∈ A | a → c}
3:      toBeLabeledQueue ← initialize queue with C_a
4:      dispatchingParentsQueue ← initialize queue with |C_a| times a
5:
6:      while toBeLabeledQueue is not empty do
7:          x ← dequeue argument from toBeLabeledQueue
8:          p ← dequeue argument from dispatchingParentsQueue
9:
10:         if all of x's parents have the same label Lab(p) then
11:             if Lab(p) = in then
12:                 Lab(x) ← out
13:
14:             else if Lab(p) = out then
15:                 Lab(x) ← in
16:
17:             C_x ← {c ∈ A | x → c ∧ Lab(c) = undec}
18:             enqueue C_x to toBeLabeledQueue
19:             enqueue |C_x| times x to dispatchingParentsQueue
```

arguments $a \to b \to c \to d \to a$. Another important aspect of the graph are arguments which are only connected to either their children arguments or their parent arguments. For example, the nodes f and g are only connected to children and the nodes i and k are connected to the rest of the argumentation framework through h and j.

The starting state is illustrated in figure 2. The argument $a \in A$ was passed as the start argument into the algorithm which is why the random walker W_0 spawns there. a's parent arguments e and d are both labeled `undec` while a is labeled `in`. Hence, two random walkers $W_1, W_2 \in W$ are set onto e and d labeling them `out` while W_0 is terminated. a also has child arguments h and b which must be informed about a's label change. Because of that, child arguments and their children alternatingly get labeled `out` and `in` following their children relationships. None of the arguments which get labeled `out` have `undec`-labeled parents which is why this labeling is permitted. h and k get the label `out` and j as well as i get the label `in`. The same happens to b and c which are labeled `in` and `out` respectively. The process is stopped by d since it has f as a parent still labeled `undec`.

In the next iteration displayed figure 3, the random walker W_2 is executed first. Since the walker is situated on an argument labeled `out` and the `in`-labeled argument c is attacking it, W_2 is terminated. Next, W_1 is executed again. It has no `in`-labeled arguments attacking it, which is why a parent node is randomly chosen.

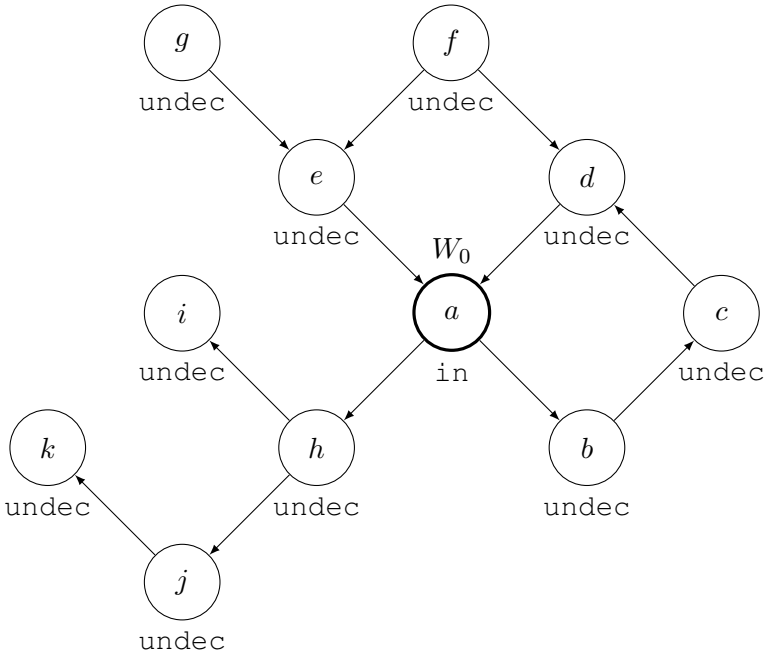


Figure 2: State at the start of the algorithm. The current labels are placed below every argument and the random walker, if present, above.

In this case, W_1 moves to argument f and labels it `in`.

Afterwards, W_1 cannot reach any parent arguments and gets terminated, which causes W to be empty. Finally, the algorithm outputs $\texttt{in}(\mathcal{L}ab) = \{a, c, f, i, j\}$ as an admissible set.

2.4 Limitations

During unit and integration testing, two major shortcomings of the program become apparent. It is unable to process graphs where the random walkers enter unlabeled uneven cycles on the one hand. On the other hand, child labeling can obstruct random walkers and lead finding smaller than possible admissible sets.

2.4.1 Uneven cycles

A small example of an argumentation framework with an uneven cycle is shown in figure 5 to demonstrate how the algorithm does not terminate in case of an uneven cycle. On the left hand side, one can see the state at the start of the algorithm. The first random walker is placed onto the start argument a which gets labeled `in`. This causes a's child arguments to be labeled according to algorithm 4 which is why b gets labeled `out` and c gets labeled `in`. In the first iteration, W_0 is terminated and spawns new random walkers onto any parent not labeled `out`. In this case, the only

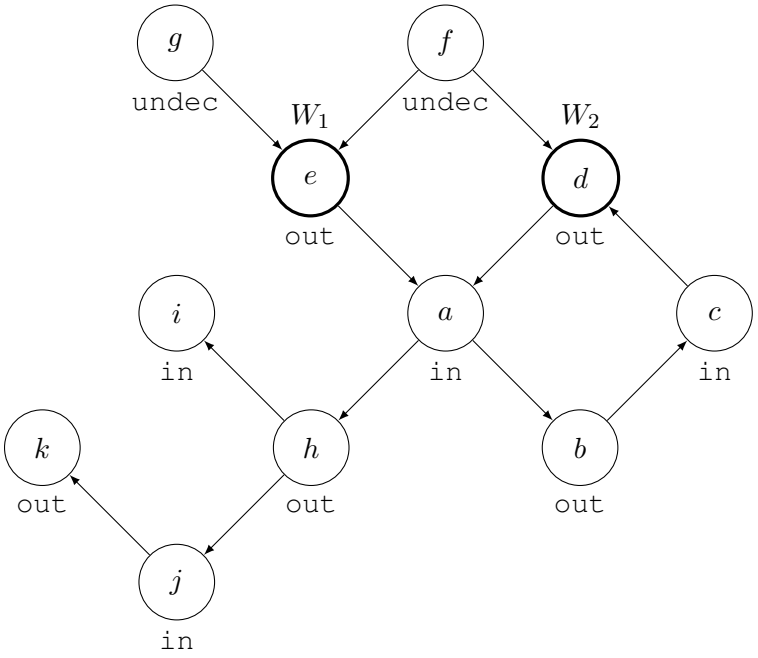


Figure 3: State at the beginning of the second iteration after W_0 was set onto a.

argument that fits the description is b. It is labeled `out` and the child labeling does not change anything because a is already labeled `in`. The state of the graph at this stage can be seen in figure 5 on the right hand side. The next iteration starts and according to algorithm 3, the random walker W_1 moves to b and labels it `in`. This state is displayed in figure 6 on the left hand side. The following state, which can be found in the same figure on the right hand side, emerges from W_1 being terminated and spawning W_2 onto a. Now, one can see that the labeling is in a similar situation to the on in figure 5 on the right hand side. a does not have a parent labeled `in` which is why the walkers will continue to cycle through the graph. That causes the algorithm to never terminate because at any time a random walker will be present in W.

This is the case for all uneven cycles because uneven cycles necessitate that two neighboring arguments are labeled the same if the only labels available are `in` and `out`.

2.4.2 Premature child labeling in even cycles

An example of this happening in the implementation can be found in the `prematureChildLabeling` test in the integration tests. Figure 7 shows how any walker being placed in an even cycle, for example on a, causes b and c to be labeled `out` and `in` respectively. This situation can be seen on the left side of graphs shown in figure 7. It follows that W_0 spawns a new walker W_1 on argument d and labels the argu-

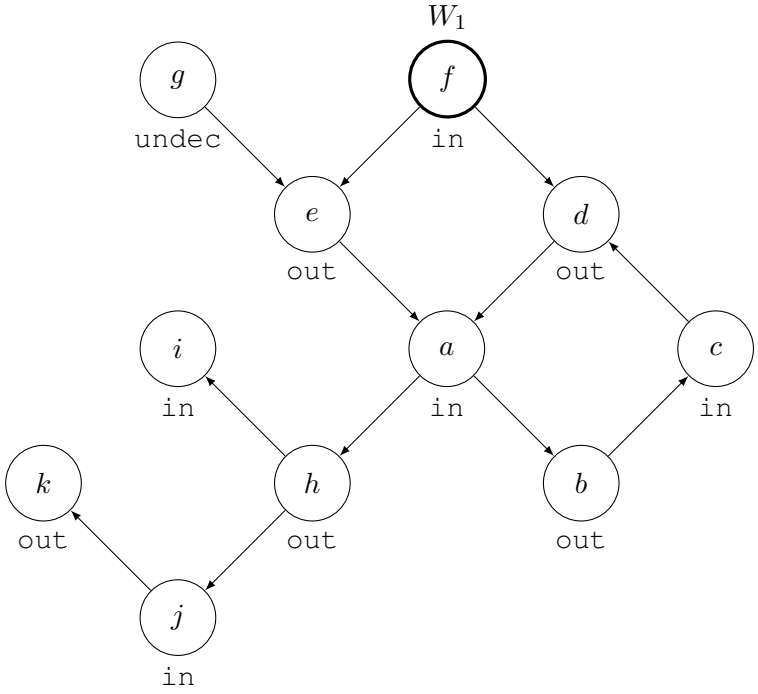


Figure 4: State at the beginning of iteration 2.

Figure 5: The state at the beginning of the algorithm is displayed on the left side and the state at the end of the first iteration on the right. It shows how the algorithm cannot terminate due to two same-labeled arguments always being placed next to each other.

ment `out`. In turn, this situation can be observed in figure 7. W_1 then terminates according to algorithm 3 because c is labeled `in`. Without child labeling, the walker W_1 would have the possibility to visit argument e and thus label it `in`. Because of this, e will always be excluded from being part of an admissible set and limits the possible admissible sets that the algorithm can return.

Figure 6: The images show the cycling of the algorithm because of an uneven cycle. The state at the end of iteration three is displayed on the left and the end of iteration four on the right.

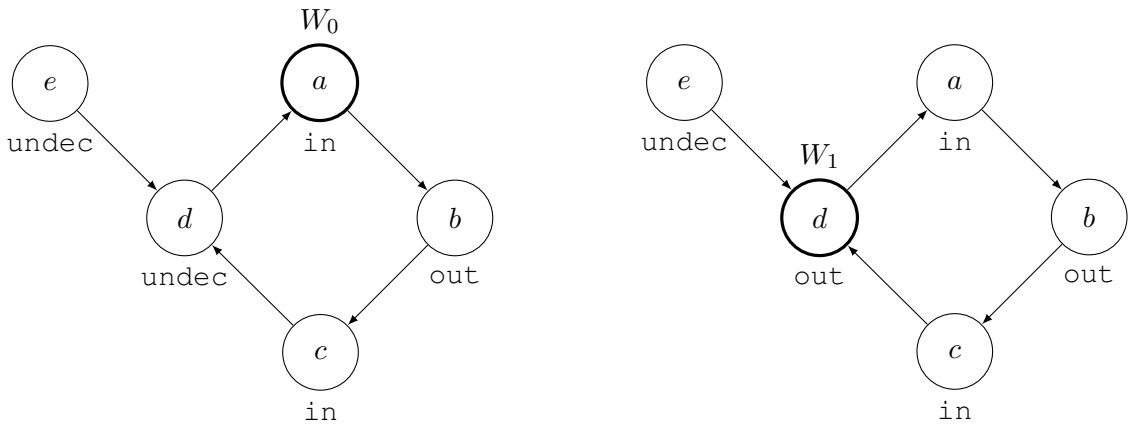


Figure 7: The left graph shows how the arguments will be labeled once an argument is placed on a. The right side shows how d will be labeled in the following iteration of the algorithm.

3 Evaluation and analysis

Within this section, the evaluation of the solver is done in comparison to other solvers. Moreover, it provides analyses of the performance and correctness of the algorithm. All graphs within this section were created using the Python library Matplotlib [32, 19], Pandas [23] and NumPy [18].

3.1 Comparison to different solvers

The following sections give an overview of the setup and results of the comparison to other solvers.

3.1.1 Comparison solver setup

Three other solvers are included in the comparison. The first one is Pyglaf, which was submitted to ICCMA'19 [1]. It is implemented via "Python and uses Circumscriptino" [1, p. 1] which is a SAT solver. This means that the solver reduces the graph and problem in a way that it can be solved by mapping it to a SAT problem. Pyglaf supports "[a]ll problems from ICCMA'17" [1, p. 1] and accepts graphs encoded as TGF as well as APX. In the comparison, Pyglaf is used solving the problems of whether an argument credulously belongs to a complete extension (`DC-CO`) and whether it belongs to a preferred extension (`DC-PR`). Both can be compared to a solver meant for solving the same task for admissible sets because every complete or preferred extension is also an admissible set. This is possible since the definitions of both extensions are the definitions of an admissible set with additional requirements ([14, p. 178] and [4, p. 85]). Hence, every argument that Pyglaf points out as belonging to a complete or preferred extensions must also belong to an admissible set.

The next solver is μ-toksia submitted to ICCMA'19 [26]. μ-toksia is able to solve "all dynamic tasks as well as all classical tasks" [26, p. 2] and accepts TGF and APX files. Although it is able to solve more problems than `DC-CO`, only `DC-CO` is analyzed. This solver is also implemented by reducing the tasks to SAT problems. Both, Pyglaf and μ-toksia use Glucose [3] as the core engine for solving SAT problems.

The last solver is Heureka, which was submitted to ICCMA'17 [17]. It is able to solve problems related to "complete, grounded, preferred and stable" [17, p. 1] extensions. It is written in C++ and solves problems by "dynamically (re-)order[ing] the arguments in order to minimize backtracking steps". In the evaluation, it solved the `DC-CO` problem.

It is important to keep in mind that performance does not only rely on the implementation but also the programming language the solver is implemented in. Many programming languages are inherently faster than others if tested on the same tasks [28, p. 37]. This means that performance differences in speed may not purely stem from the difference in the approach to solving the problems.

The test was executed by using the provided Docker images [24] or packaging a given solver into a Docker image if none are available online.

From each of the graphs, all argument names were extracted. Each of those arguments were evaluated whether they belong to an admissible set (complete extension, preferred extension) and the result (either `YES`, `NO` or `timeout`) is recorded. Additionally to recording the result, the run time length is also saved by wrapping the Docker entrypoint in the Bash `time` command. This command provides information on how long a program ran in real, user and system time. All statistics to do with run time length use the real time that the command returns. To prevent outliers in single runs to dominate the data, every argument in every graph is used as an input to each solver three times. This way, run time lengths can be averaged and timeouts which occured in single runs may be mitigated.

The machine all tests are evaluated on has an Intel Core i7-6600U CPU with a clock speed of 2.60 GHz and four cores as well as 16 Gigabytes of RAM. The operating system is Ubuntu 20.04. Timeouts are enforced with the Bash `timeout` command as well as periodically polling the currently running Docker containers and terminating every container that reached the timeout limit of five minutes.

3.1.2 Example data setup

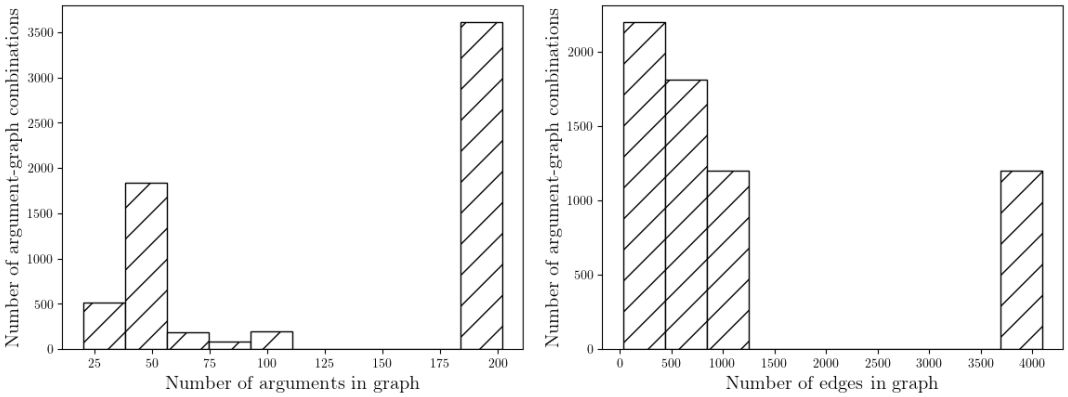


Figure 8: The image shows the histogram of the number of arguments (left) and edges (right) of a graph-argument combinations i.e. how many arguments there are in a graph of a certain size. Each of these combinations was evaluated three times by each solver.

The chosen test cases are the first ICCMA'15 benchmark set[1] [31] and the ICCMA'17 example instances [2] [16]. Due to severe time restrictions, all instances with

[1]The download link is placed in the section "Running your solver" on the website `https://argumentationcompetition.org/2015/rules.html`.

[2]The download link can be found in the section "Solver Requirements" on the website `https://argumentationcompetition.org/2017/participation.html`.

1000 or more arguments were not evaluated. That means that the data set consists of 85 graphs which contain a range of 20 and 202 arguments and between 72 and 8040 edges. The figure 8 shows how the graph sizes in terms of arguments and edges are distributed.

3.1.3 Results

This section answers the following core questions:

1. Did the solver return different results for the same argument in the same graph on different runs?
2. Does the solver give correct results when compared to a ground truth?
3. How much time does the solver need on average to solve a problem?

Question 1 is important to ask since the random walker implementation contains elements where the further progress of the program is determined by random elements. Specifally, the randomness in this implementation can be found in algorithm 3 in line 10 where the walker moves to random not `in`-labeled parent argument. This can cause different runs to result in different outputs. Another reason for differing outputs over multiple runs might be timeouts.

For the random walk reasoner, 97.3 % of all runs returned the same result. From the remaining 2.7 % that did not, 2.5 % returned a difference in `YES/NO` outputs over different runs and the last 0.2 % were different because of timeouts. Pyglaf's outputs are always consistent over all runs. For both problems, `DC-CO` and `DC-PR`, all three runs return the same values. In μ-toksia's case, 99.6 % of all cases yield the same output. The other 0.04 % were different due to timeouts. Heureka is in a similar situation to μ-toksia: 98.9 % of all runs yield the same result. The rest did not because of timeouts.

For question 2, one has to specify a ground truth first. μ-toksia is chosen to be the ground truth because the solver "consistently outperformed" all problems of the main track of ICCMA 2019 [27, p. 803]. Also, this analysis looks at the aggregate over all three runs which is why all the cases were these runs did not result in the same outputs have to be handled. As a rule, if the runs do not agree on whether an argument belongs to an admissible set it results in `YES`, if any of the three results are `YES`, otherwise it is `NO`.

A good way to display how well a solver performed, taking μ-toksia as ground truth, are confusion maticres. The matrix for the random walk reasoning solver is shown in table 1 on the left side. 7.9 % of predictions are false negatives and none of the predictions are false positives. But overall, the solver performs far worse than the other solvers. Heureka (table 1, right side) and Pyglaf (`DC-CO` and `DC-PR`, table 2) are both able to keep their mispredictions below 1 %. The only metric in which this solver performs better is the number of false positives. Heureka (table

		Prediction outcome	
		YES	NO
Actual value	YES	785 12.2 %	504 7.9 %
	NO	0 0.0 %	5122 79.9 %

		Prediction outcome	
		YES	NO
Actual value	YES	1243 19.4 %	46 0.7 %
	NO	10 0.0 %	5112 79.9 %

Table 1: The confusion matrices for the random walk reasoning solver on the left and Heureka on the right. For both, μ-toksia's results serve as ground truth.

1, right side) mispredicts 10 instances whereas the random walk reasoning solver mispredicted none.

Question 3 is concerned with how long it takes each solver to decide whether an argument belongs to an extension or not. Since solvers can have ways to cut short their run time by finding conditions that make it impossible for an argument to belong to an extension, only instances should be compared where all solvers agree on whether the argument belongs to an extension. Therefore, figure 9 shows the run time lengths for all instances where the solvers agree that the arguments belong to an admissible set. Similarly, figure 10 shows the lengths where solvers agree that the arguments do not belong to admissible set. Due to technical difficulty, Heureka cannot join this comparison because the `time` command in the image only delivers the value `0.00` for all but very few instances. Since this is unrealistic, Heureka is excluded from this comparison.

Both figures show a similar situation: μ-toksia is the fastest solver while Pyglaf's median run time length is about 0.05 seconds longer. Pyglaf performs about the same for both problems `DC-CO` and `DC-PR` at a median speed of about 0.1 seconds. The slowest solver is the random walk reasoning solver. In the `YES` and the NO case its median lags behind by about 0.15 to 0.2 seconds.

3.1.4 Evaluation limitations

There are three reasons why this solver comparison has limited informative value:

1. The overall number of example instances is small and those instances which are included have a relatively low number of arguments and edges themselves. The decision to limit the number and size of example instances is made because of time restrictions.

		Prediction outcome	
		YES	NO
Actual value	YES	1289 20.1 %	0 0.0 %
	NO	0 0.0 %	5122 79.9 %

		Prediction outcome	
		YES	NO
Actual value	YES	1243 19.4 %	46 0.7 %
	NO	0 0.0 %	5112 79.9 %

Table 2: The confusion matrices for the Pyglaf solver solving the problem `DC-CO` on the left and Pyglaf solving `DC-PR` on the right. For both, μ-toksia's results serve as ground truth.

2. It is difficult to compare differences in run time length if the algorithms are implemented in different programming languages due to performance differences [28, p. 37].

3. Due to the fact that the Bash `time` command cannot consistently measure run time length within the Heureka Docker image, it could not be included in the comparison of run time lengths.

3.2 Analysis of time and space complexity

This section examines time and space complexity of the overall algorithm by analyzing its components. It starts with the analysis of time complexity and later continues with the space complexity analysis. Here, A denotes the set of arguments contained in a graph $F = (A, R)$.

To start with the analysis, one should start at the most essential building blocks of the algorithm. That is, the behavior based on an argument's label that a walker currently resides on. But these parts are both also based on algorithm 4, which is responsible for labeling information about label changes in the direction of the children. Therefore, the first procedure to be analyzed can be found in algorithm 4. Here, lines 3 to 4 add constant time $c_1 + c_2$ but line 2 promises that all arguments possibly have to be iterated over in the worst case, which is why it adds $|A|$ to the run time. For the loop starting in line 6, one has to assume the worst case which would be that all $|A|$ arguments have to be labeled starting at the argument a. An example of an instance that represents such a worst case would be $a \to a_1 \to a_2 \to \cdots \to a_{|A|-1}$ where labeling a would lead to labeling the rest of the graph using the child labeling algorithm. Lines 7 to 19 lead to the worst case scenario that the loop will iterate $|A|$ times, each time executing constant time $c_2 + \cdots + c_8$. In the end, the

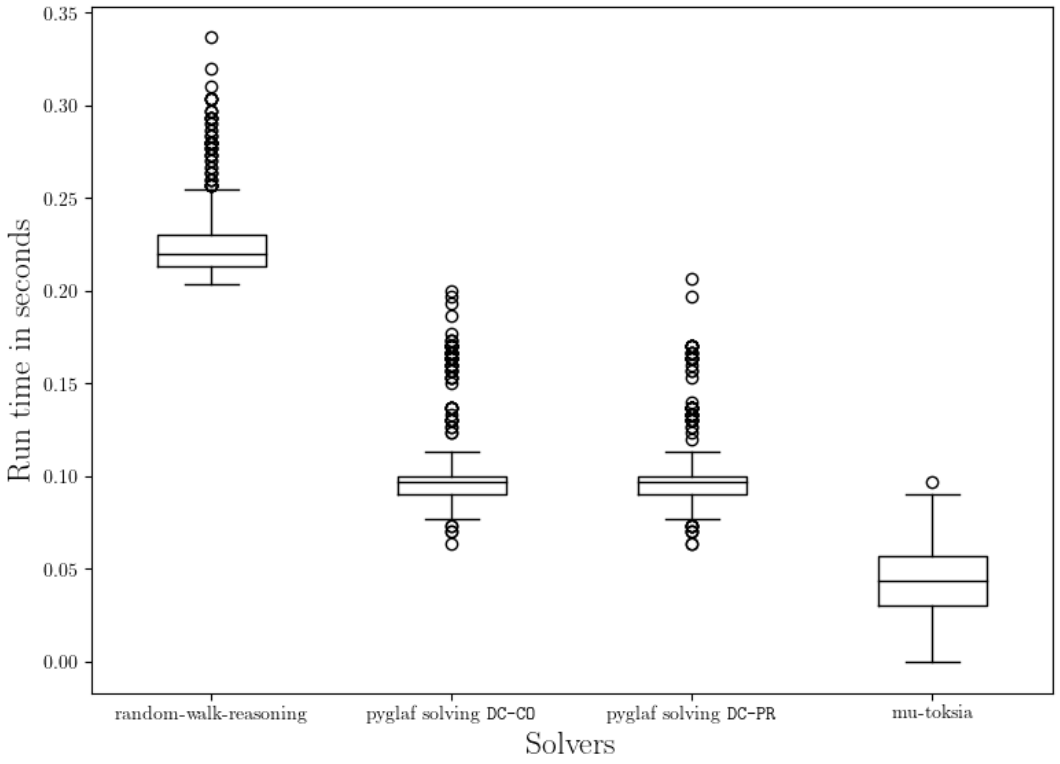


Figure 9: Boxplot of run times for cases with all solvers returning `YES`. Each of the 785 data points is an average of three runs.

algorithm has a run time complexity of $O(|A| + c_1 + c_2 + (c_2 + \cdots + c_8)|A|)$ which is equal to $O(c_1 + c_2 + 2(c_2 + \cdots + c_8)|A|)$. Dropping all constant summands and factors leads to a run time of $O(|A|)$.

Algorithm 3 describes the behavior of a walker placed on an `out`-labeled argument. The expression in line 2 can lead to iterating through all arguments in the worst case, i.e. $|A|$ times. Also, the loop spanning lines 4 to 8 can maximally iterate $|A|$ times. The loop body contains a check for an if-else-statement and therefore adds constant time c_1 to each of these iterations. Lines 10 to 13 add constant time statements $c_2 + c_3 + c_4$ together with the time of the children labeling algorithm. In the end, this means that this sub-algorithm has a worst case run time of $O(|A| + c_1|A| + c_2 + |A| + c_3 + c_4) = O(2c_1|A| + c_2 + c_3 + c_4) = O(|A|)$.

The next algorithm, algorithm 2, handles the walker behavior in case it is placed on an argument labeled `in`. As in the algorithm before, the expression in line 2 cycles through all arguments available in A in the worst case scenario. This means it adds a run time of $|A|$. Then, there is a loop that spans lines 4 to 8. Lines 5, 7, and 8 add constant time $c_1 + c_2 + c_3$ but line 6 calls the children labeling algorithm

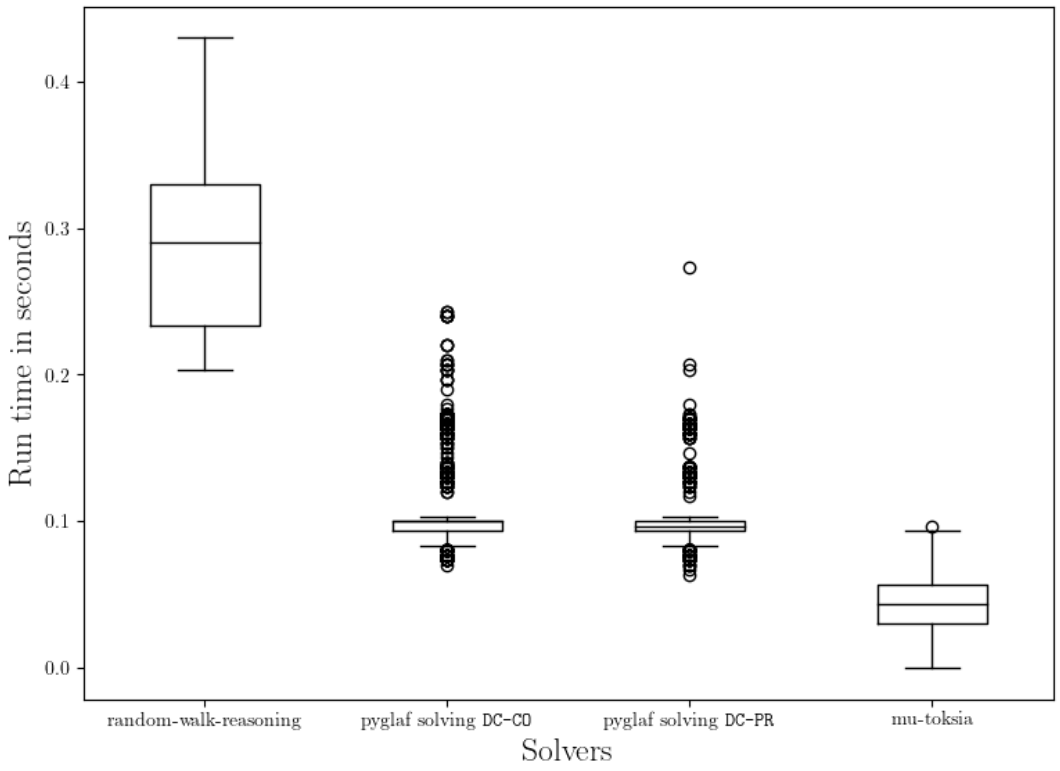


Figure 10: Boxplot of run times for cases with all solvers returning `NO`, averaged over three runs for each of the 3668 data points.

with a run time complexity of $O(|A|)$. This means that the loop creates a run time complexity of $|A|(c_1 + c_2 + c_3 + |A|)$. And again, lines 10 and 11 follow with the constant time of $c_4 + c_5$. This means that this sub-algorithm runs with a time of $O(|A| + |A|(c_1 + c_2 + c_3 + |A|) + c_4 + c_5) = O(2|A|(c_1 + c_2 + c_3 + |A|) + c_4 + c_5)$, which, by dropping constant summands and factors, leads to an overall run time of $O(|A|^2)$.

With all sub-algorithms already evaluated, the main algorithm 1 can now be analyzed. Here, the algorithm starts with three terms in constant time $c_1 + c_2 + c_3$ from lines 2 to 4. In line 6, a loop starts that first executes another statement which takes up constant time c_4 in line 7. The loop itself could possibly run through all available walkers which may be placed on all available $|A|$ arguments. In line 6 to 13, a decision has to be made about whether to handle an argument labeled `in` or `out`. Since the analysis requires considering the worst case scenario, each iteration always evaluates line 10 since HANDLEWALKERONINARGUMENT, with a complexity of $O(|A|^2)$, performs worse than HANDLEWALKERONOUTARGUMENT, with a complexity of $O(|A|)$. In the end, line 15 adds a constant time of c_5. The overall run time

complexity of the algorithm is therefore $O(c_1 + \cdots + c_3|A|(c_4 + |A|^2) + c_4)$, which, by dropping all constant summands and constant factors, becomes $O(|A||A|^2) = O(|A|^3)$. Hence, the overall run time complexity of the whole algorithm is $O(|A|^3)$.

The analysis of space complexity follows the same principle of first analyzing the most basic building blocks of the overall algorithm 1. Therefore the analysis starts at algorithm 4. Here, assuming the worst case, line 1 requires memory space of $|A|$, if all arguments have to be saved. The following lines 2 and 3 also add a space complexity of $2|A|$ because they are dependent on the worst case scenario of line 1. The loop starting in line 6 can cause the queues to become as large as $|A| - 1$. The total space complexity of this procedure is $O(|A|+|A|+|A|+(|A|-1)) = O(4|A|-1)$ yielding $O(|A|)$ by dropping the constants.

Algorithm 3 starts by saving all of the parent arguments P_a of the input argument a. In the worst case, this term can become as large as the entirety of all arguments A. The following loop does not cause occupation of any memory but rather potentially frees it. Nevertheless, the loop is assumed to not free any memory because of the worst case assumption. The last lines do not cause occupation of any memory with the exception the call of the procedure LABELCHILDREN that occupies memory on the stack as well as on the heap with a complexity of $O(|A|)$. Overall, this procedure has a space complexity of $O(|A| + |A|) = O(|A|)$.

The procedure responsible for handling walkers set on an `in`-labeled argument is algorithm 2. It occupies memory space of $O(|A|)$ in the second line similar to terms found at the beginnings of the other procedures. Due to the call to the child labeling function in the loop from line 4 to 8, it can occupy up to $|A||A|c_1$ because of the loop iterating $|A|$ times through all arguments in the worst case. Line 10 will actually free space c_2 which results in a space complexity of $O(|A| + |A||A|c_1 + c_2) = O(|A| + c_1|A|^2 + c_2) = O(|A|^2)$.

All functions are merged in algorithm 1 which starts by adding taking constant space of c_1 in line 2. Afterwards, the worst case branch within the loop (line 10) is repeatedly executed. Since these functions occupy space temporarily and this space occupation does not compound with every loop iteration, the total space complexity of the algorithm stays $O(c_1 + |A|^2) = O(|A|^2)$.

3.3 Proof of correctness

In order to understand the section it is important to know that the algorithm may be started using a start argument $a_{\texttt{YES}} \in A$ that is actually part of an admissible set $E \subseteq A$, or a start argument $a_{\texttt{NO}} \in A$ that is actually not part of of any admissible set. Futhermore, one has to know that in the actual implementation, it is acceptable for the core algorithm 1 to return a non-admissible set since the `Admissibility-Checker` (see section 4.2.1) filters these proposals for admissibility. Hence, there is the need to show that if the start argument is assumed to be an element of an admissible set, the algorithm would return a set $\texttt{in}(\mathcal{L}ab)$ that is admissible and otherwise, if start argument $a_{\texttt{NO}}$ is assumed to not take part in any admissible set, then the

returned in-labeled arguments $\texttt{in}(\mathcal{L}ab)$ cannot possibly be admissible. One more assumption is that the graph $F = (A, R)$ does not contain any cycles with an uneven number of arguments and therefore does not cause the algorithm to cycle (see section 2.4.1).

The following paragraph assumes that the start argument $a_{\texttt{YES}}$ is part of an admissible set. If the algorithm terminates at the beginning of the algorithm and outputs a set $\texttt{in}(\mathcal{L}ab) = \{a_{\texttt{YES}}\}$, then the algorithm stopped at a point where $a_{\texttt{YES}}$ either does not have any parents or the child labeling caused an uneven number of child arguments to be labeled such that $a_{\texttt{YES}}$'s parents are attacked. Both scenarios are coherent with $a_{\texttt{YES}} \in \texttt{in}(\mathcal{L}ab)$ being admissible. If the algorithm does not stop at this point, W is not empty yet and there are walkers placed on $a_{\texttt{YES}}$'s parents. These get labeled out in the next iteration. If any walker terminates being placed on some of these arguments, it means they must be attacked by an in-labeled argument. If any walker stopped at this point without meeting this condition, then $a_{\texttt{YES}}$ could not be part of an admissible set. That cannot be possible because in order for $a_{\texttt{YES}}$ to be assumed admissible there cannot be any out-labeled parent arguments of $a_{\texttt{YES}}$ which remain unattacked. Therefore, at this stage, the algorithm would still have to return a set $\texttt{in}(\mathcal{L}ab)$ which is admissible. Now, the walkers which did not terminate would move to one parent argument of $a_{\texttt{YES}}$'s parent arguments and label them in. These walkers are in the same situation as $a_{\texttt{YES}}$ was at the beginning of the algorithm, hence the same logic can be applied: If a walker terminates being placed on these arguments, it would be in accordance with the assumption because this means that $a_{\texttt{YES}}$'s attackers are attacked because the parent arguments of $a_{\texttt{YES}}$'s parent argument are now labeled in. Overall that means that if the start argument $a_{\texttt{YES}}$ is assumend to be part of an admissible set, the algorithm must return a set of arguments $\texttt{in}(\mathcal{L}ab)$ that is necessarily admissible.

The following paragraph assumes that the start argument $a_{\texttt{NO}}$ is not part of any admissible set. Here, the algorithm cannot possibly stop at the beginning and return $\texttt{in}(\mathcal{L}ab) = \{a_{\texttt{NO}}\}$ because there needs to be at least one parent attacking the start argument $a_{\texttt{NO}}$. This is because the assumption of $a_{\texttt{NO}}$ does not hold if there are not any parents attacking it. The first walker being set onto the start argument will spawn walkers on its parent arguments which will be labeled out. If walkers terminate at this point, it would not violate the assumption of $a_{\texttt{NO}}$ not being part of any admissible set because the set $\texttt{in}(\mathcal{L}ab)$ would contain an attacked argument, $a_{\texttt{NO}}$, and $\texttt{in}(\mathcal{L}ab)$ could not possibly be admissible. If a walker does not terminate, it will move to the parent argument of $a_{\texttt{NO}}$'s parent argument. These arguments are labeled in. At this moment, the walkers are in the same situation as if they were placed on the start argument $a_{\texttt{NO}}$. This means that the walker should not be able to terminate at this point in the algorithm unless this argument is attacked by an argument with a parent argument. If this were not the case, then $a_{\texttt{NO}}$'s parent argument would not be attacked which means that the assumption of $a_{\texttt{NO}}$ not being part of an admissible set would not hold. From this point on, one can reason the same way on in-labeled arguments as the reasoning started with the start argument $a_{\texttt{NO}}$. Therefore, if the

algorithm starts with a start argument which is not part of any admissible set, it will return a $\texttt{in}(\mathcal{L}ab)$ which cannot possibly be admissible.

4 Implementation

The following sections describe how to use the software and how it is built and tested.

4.1 Usage

The solver implements the interface described in Bistarelli et al. [5, pp. 7] using Java [2] and that every other solver used in the evaluation used as well. Executing the JAR file that includes all dependencies will print the author's name, the program's name and its version to the command line. Also, appending `--formats` to the command, as in `java -jar random-walk-reasoning.jar --formats` will yield all accepted input formats. Currently, the program only accepts TGF files which is why `[tgf]` will be printed. Furthermore, the program gives information about which problems it can solve.

Since the software is developed to find admissible sets and the solver requirements do not provide any abbreviation for admissible sets, the interface will use `AD` in its place. Therefore, if the program is called using the `--problems` flag, it will return the tasks it is able to solve. These are the tasks `[DC-AD, SE-AD, EE-AD]`. `DC-AD` will take a graph and an argument name through different parameters and decide whether the argument is credulously accepted [13, p. 2] with respect to the admissible sets that the program was able to find in the run. If the argument is credulously accepted, the program outputs `YES`, otherwise `NO`. The task `SE-AD` will also require a start argument to be passed into the program via another parameter. Instead of outputting `YES` in case of credulous acceptance, it will output the admissible set the start argument was accepted into in the form of `[a1, a2, ..., an]`. The task `EE-AD` will return a list of all unique admissible sets that were found while cycling through all arguments as start arguments. It does not require passing a start argument as parameter.

The parameter `--file` takes the path to the file that contains the graph. Currently, the program can only be applied to TGF files which is why the parameter `-fo` is optional. This parameter is used to identify the file type of the file that the `--file` path points to.

`-a` or `--start-arg` has to be provided in case the problems `DC-AD` or `SE-AD` are to be solved using the `-p` flag. Internally, passing and argument name through `-a` does not only mean that it should be checked for credulous acceptance, but also that it serves as the starting argument for the algorithm. This guarantees that an admissible set containing the start argument is not only found by chance, but that any possible admissible set must contain the start argument. If the algorithm still returns `NO`, this argument does not belong to any possible admissible set. The argument passed through `-a` has to be included in the graph contained in the `--file` graph.

Additionally, the programm provides the `--help` flag which will output an explanation of all other options to the command line.

To build the JAR file, the user first has to navigate into the source code root directory. Then, execute `mvn clean package`. This will generate a `target/` directory with the artifact `java-implementation-1.0.0-jar-with-dependencies.jar` in it. It can then be executed on an example graph that is included with the source code like this:

```
java -jar \
  target/java-implementation-1.0.0-jar-with-dependencies.jar
  -p DC-AD -fo tgf \
  --file src/main/resources/expose-example.tgf -a a
```

Because of the fact that the solver is wrapped in a Docker [24] image, there are some patterns that have to be taken into consideration when executing the image as a container. The program reads the graphs as TGF files from hard disk which it is isolated from when running it within a Docker container. Therefore, the directory that contains the graph files needs to be mounted onto a directory inside the container. In order to do so, the `--volume` flag has to be used with the path of the host machine and the path of the container separated by a double colon `:`. All parameters that would be accepted by the JAR file now need to be appended to the name of the Docker image to be executed. So, a valid command executing using Docker might be:

```
docker run \
  --volume $(pwd)/path/to/graphs/directory/:/app/resources \
  dominikhillmann/random-walk-reasoning:1.3.0 \
  --file /app/resources/graph.tgf -fo tgf -p DC-AD -a a1
```

The directory `/app/resources/` is chosen as an example, but it is important that the same directory is used in the `--file parameter`. It is important to know that the `docker run` command requires an absolute path on the host system for the `--volume` parameter. This can be achieved by prepending the path with `$(pwd)` in the Bash shell, for example. The image can be downloaded using the shell command `docker pull dominikhillmann/random-walk-reasoning`.

In order to execute the unit tests, one has to use Maven [25]. After navigating into the root directory of the project, type `mvn clean test` and the unit tests results are displayed.

4.2 Architecture

The following sections describe the architecture of the software.

4.2.1 Software architecture

The software tries to follow the hexagonal architecture pattern [15, pp. 19]. This is due to the fact that this pattern makes it easy to separate domain logic and communication to and from the application and to develop them independently. The

program is therefore separated into three large packages: `domain`, `application` and `adapter`. The adapter layer is again divided into `inbound` and `outbound` defines how communication to and from the program works. The `adapter` layer classes make use of the actual program logic by using the classes of the `application` layer which encapsulate `domain` logic into separate use cases. At last, the `domain` layer contains all classes related to the actual logic of the program. All of these make heavy use of the Java library Tweety [29] to provide essential building blocks like `Graph`s, `Labeling`s and `Argument`s.

Figure 11[3] shows the composition of the package and it consists of the following classes:

- `Walker` and `RandomWalkerImpl`: `RandomWalkerImpl` as well as its interface `Walker` do not contain any logic about what path the walker should take. These classes only exist to indicate the current position in the graph and, if needed, change it using the method `moveTo`. To find out which argument it occupies, the method `getOccupiedArgument` can be used.

  All classes that take advantage of any walkers will use the interface. This way, any other walker with different behavior to `RandomWalkerImpl` can be easily inserted into the program.

- `AdmissibilityChecker`: The program can start with a `--start-arg` parameter that cannot possibly part of an admissible set. In that case, the algorithm encapsulated in `SearchOrchestrator` will still return a set of arguments labeled `in`. This is because the start argument has to be necessarily labeled `in` at the start of the algorithm. Therefore, there needs to be a manner of checking for actual admissibility of the returned `in`($\mathcal{L}ab$) from the `SearchOrchestrator`. This happens within the `AdmissibleSetFinder` class discussed later.

  To check for admissibility, the `GraphState` and is passed into the constructor and the currently `in`-labeled set of arguments can be tested for admissibility with respect to the state by calling `isAdmissible`.

- `WalkerDecider` and its implementations `WalkerDeciderOnLabelIn` and `WalkerDeciderOnLabelOut`: The `SearchOrchestrator` expects the logic of the `Walker`'s behavior to be inserted into its constructor via the `WalkerDecider` parameters. This way, the core behavior of the program can be tested independently.

  `WalkerDecider` has to two implementations. The first one is `WalkerDeciderOnLabelIn` and it realizes what to do with any `Walker` if it resides on an `in`-labeled argument specified in algorithm 2. The second one, `WalkerDeciderOnLabelOut`, realizes the algorithm specified in algorithm 3. Both

[3]The tool used to create the graphics is Draw.io, which can be found at `https://app.diagrams.net/`.

WalkerDecider implementations gain access to the graph's state by passing it as a parameter to their constructors. The method orchestrate contains the logic about what happens to a specific Walker in the context of the current state. Changes to the walker itself are applied immediatly using the walker's methods like moveTo. Whether or not a new Walker is added to the list of all is communicated using the returned UpdateCollectors.

- SearchOrchestrator: In the SearchOrchestrator, all domain classes with the exception of the AdmissibilityChecker come together to implement the entirity of the domain logic. The SearchOrchestrator requires the GraphState, an initial argument to perform the initial labeling and the desired behavior regarding the Walkers through accepting the WalkerDeciders as parameters. The only method of the class, solve, returns a set of arguments labeled in which will be checked for admissibility in the application layer.

- UpdateCollector: This is a helper class that is needed because updates to the walkers list within SearchOrchestrator have to be concentrated at one point. The class will collect any Walker to be added to the list or removed from it by using the methods addLaterAddition and addLaterRemoval respectively. These additions and removals can then be accessed at a later point in time. The class is parameterized, so it can be used in different contexts other than the Walker updates.

- GraphState: This class serves as the central state of the graph, both for the Graph itself but also for its Labeling. It does not contain any logic about itself, but provides a convenient interface for all classes that read and write changes to the labeling. For example, methods like getParentsWithLabel solves the problem of sorting through arguments based on the label which would have been implemented in several other classes otherwise. Similarly, setLabelAndChildrenLabel offers the functionality of changing a label only in way that everytime a label does get changed, the information automatically flows to the children as described in algorithm 4.

Figure 12 contains the UML class diagram of the application layer. This layer is the only one that consists of a single class called AdmissibleSetFinder. AdmissibleSetFinder represents the use case of a finding an admissible set that may be be used by any kind of adapter of the adapter layer. Figure 12 shows that the use cases use both the domain SearchOrchestrator, which bundles the algorithm logic and the AdmissibilityChecker for checking an in-labeled proposal for actual admissibility. Also, this class provides two ways of searching for admissible sets. First, the method solveForStartArg takes an argument as a starting argument and returns and Optional of a possibly admissible set which is only filled if it passed the AdmissibilityChecker tests. Second, there is the solveForAllArgs method which iterates through all arguments of the graph and takes

each of the graph arguments as the algorithm starting argument. It then outputs all unique admissible sets found throughout the iteration of all arguments.

The last layer is the `adapter` layer. It is responsible for communication to and from the program. This package makes heavy use of the library Picoli [12] to describe which command line arguments are expected by the compiled program. It consists of the following classes for the case of `inbound` data:

- `PathToFileConverter` is responsible for turning the path passed into the program via the command line parameters into a file and then reading the specific lines as `String`s and providing these.
- `TrivialGraphFormatReader`: This class interprets the content of the TGF file given to it as a list of strings and outputs a `Graph` instance that fits the description of the TGF file.
- `CommandLineAdapter`: The adapter contains everything related to interpreting parameters given over the command line and coordinating all classes intended for converting them into processable instances. Also, this class will pass the usable instances the use cases of the `application` and in turn passes any solution to any of the `outbound` adapters.
- `InvalidTgfFileException` is a helper class to easily recognize program failure due to an invalid TGF file.

There is only one interface in the `outbound` subpackage along with its implementation. `SolutionWriter` exists as an interface so that any other way of writing the solution can be implemented using it and all classes dependent on a `SolutionWriter` do not need to change any of their internals because of it. `CommandLineSolutionWriter` takes any solutions, formats them and writes it to `stdout`.

4.2.2 Docker image composition

The Docker image [24] is split into two stages. The first one is the build image and the second is the regular image. The build image includes Maven [25] and the Java runtime. Here, the source code is copied into the image and compiled using familiar Maven commands. The build image can be downloaded using `docker pull maven:3.8.6-amazoncorretto-17:latest`.

The second stages uses Amazon Corretto [20] as a base image. It copies the compiled JAR file from the builder stage and then provides access to it via the `ENTRYPOINT` keyword. This Amazon Corretto image can be downloaded using `docker pull amazoncorretto:17-alpine3.13`.

4.3 Unit and integration tests

The unit and integration tests were implemented using JUnit [10], AssertJ [9] and Mockito [11]. JUnit is a basic unit test framework which can be enhanced by AssertJ

because it adds better understandable assertions. Mockito is used for mimicing injected objects such that their behavior can be defined in place. Additionally, Mockito enables developers to inspect if and how often mocked methods were called. Reports about the test coverage can be generated using JaCoCo [21].

The unit tests cover 97 % of the code if measured by instructions and 98 % if measured by branch coverage. Overall, 61 tests are implemented which are distributed over 13 test classes. One test is left ignored because it checks for the condition that causes the algorithm to cycle and not terminate described in section 2.4.1.

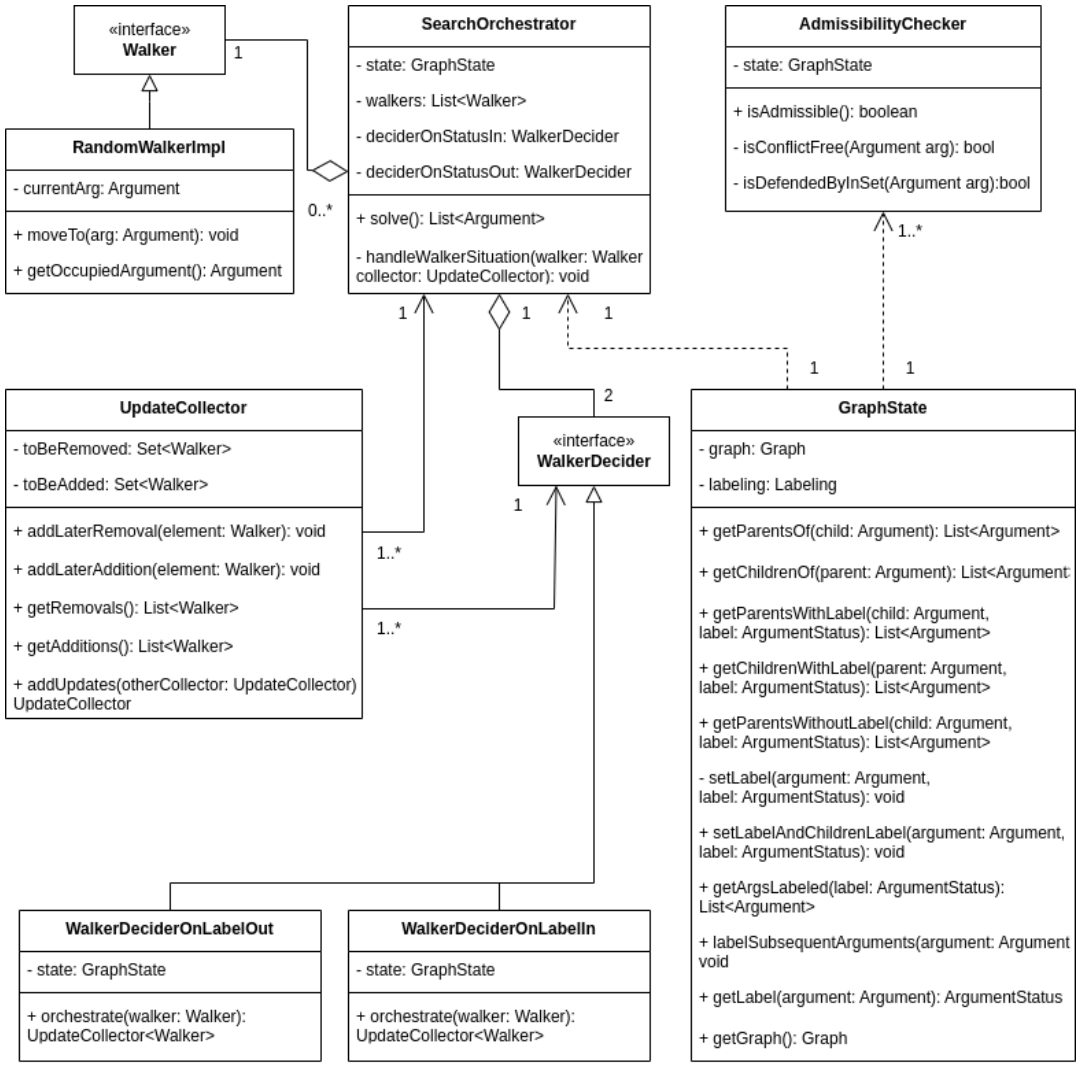


Figure 11: The UML class diagram of the domain classes of the program. This diagram does not offer any information about how it relates to the `application` and `adapter` layers which will be better shown in later illustrations. The central class is the `SearchOrchestrator`. It combines information about the state of graph and labeling with the behavior definition of the `WalkerDecider`s and the `Walker`s as indicators into a implementation of the core domain logic as described in algorithm 1.

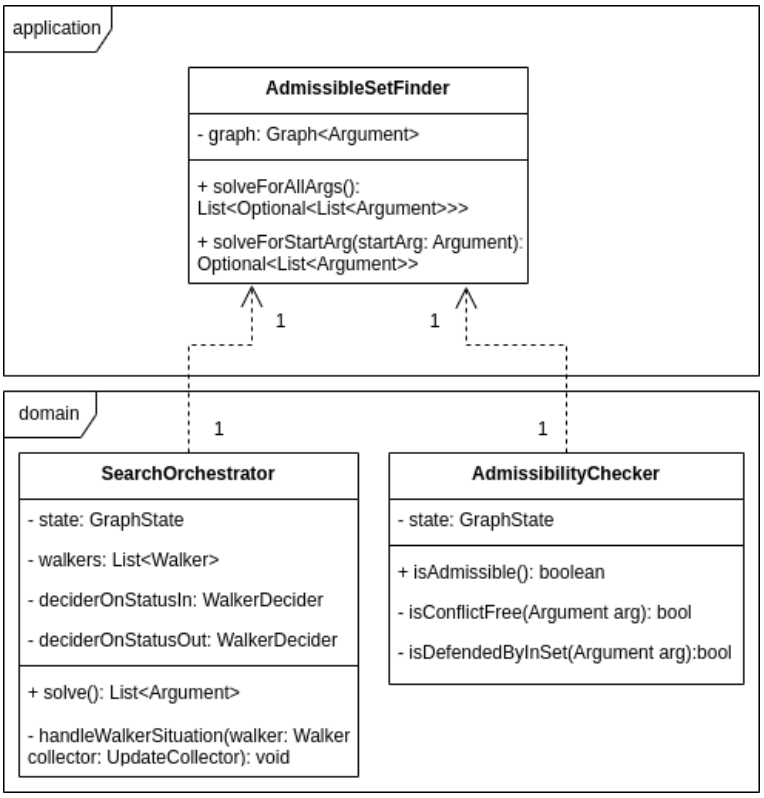


Figure 12: The `application` layer only consists of a single class. To better understand how this class relates to the `domain` classes, all relevant ones are included in the image within the `domain` package.

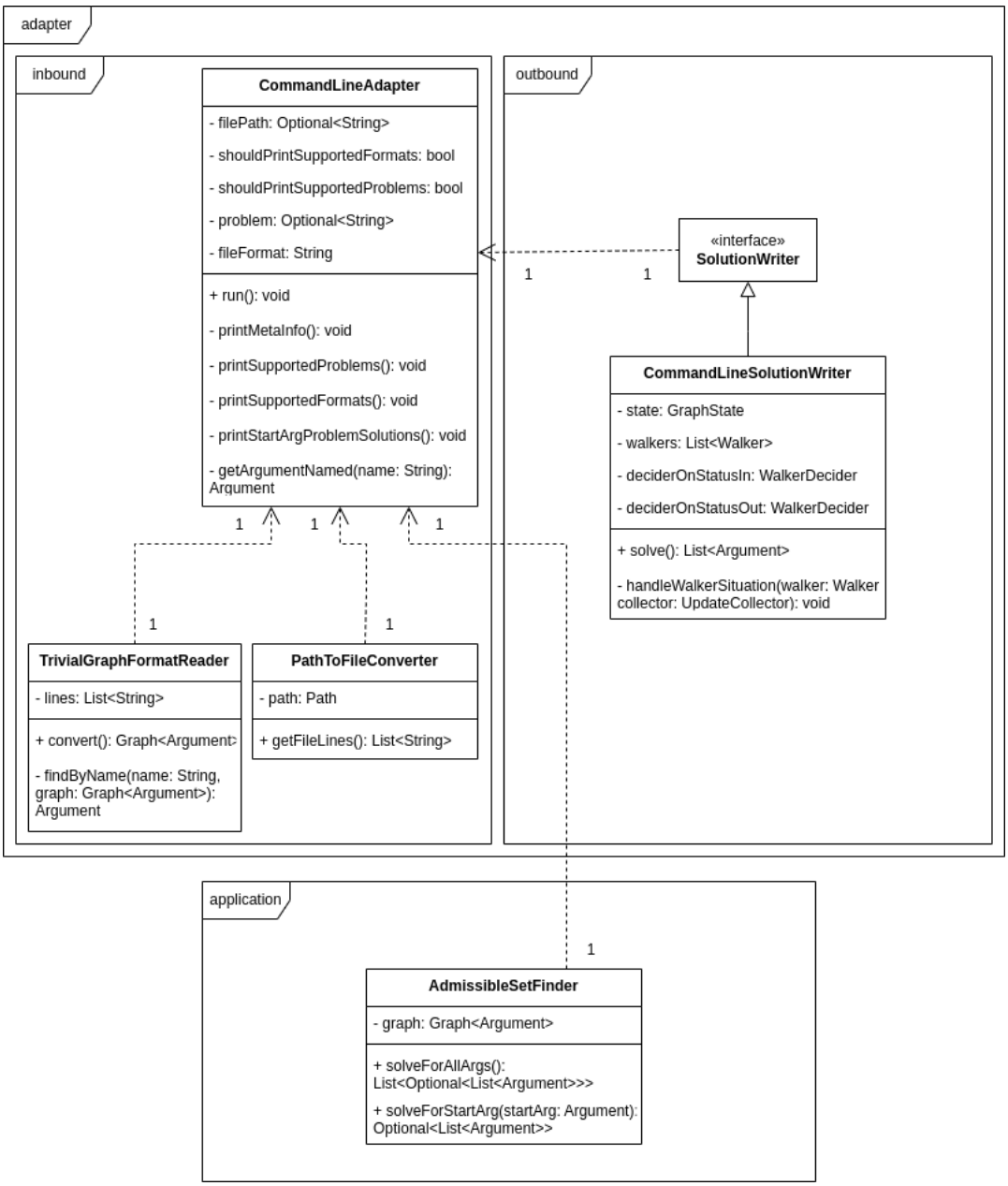


Figure 13: This figure shows the UML class diagram of the `adapter` layer. The `adapter` layer is included to better understand how it relates to the the rest of the program.

5 Conclusion

The thesis followed the conceptualization, formalization, implementation and evaluation of an algorithm that is based on random walkers assigning labels to arguments. It is implemented by three separated layers, which are each responsible for the domain logic, encapsulating use cases and organizing communication to from the program. The program continues to have two weaknesses discussed in the thesis: It is prone to cycling if the graph contains cycles with an uneven number of arguments, and even cycles may cause the algorithm to find smaller than possible admissible sets that the start argument belongs to. The evaluation answered three questions: Does the solver produce different results between runs based on the same graphs with the same start argument? How precise does the solver perform, if μ-toksia is taken as ground truth? And how fast is the solver able to solve the instances? The random walk solver does output different results between different runs, which are not due to timeouts and can be traced back to the random parts of the algorithm. Also, the random walk solver performs the worst if compared to the other solvers on the basis of μ-toksia's results as ground truth. In the speed comparison between solvers, the implemented solver also scores the worst. The evaluation is limited because the solvers did not involve the larger graphs that would have been available for testing and Heureka's run time length could not be measured and therefore compared.

Hence, there remain many possibilities to improve upon the solver in terms of speed and precision. As far as speed is concerned, one could take advantage of multithreading, for example for solving the same graph with multiple start arguments at the same time. Moreover, the graph can be checked for being bipartite before the core algorithm is applied to a graph in order to prevent the algorithm from entering cycles with an uneven number of arguments. It is also important to find out what the conditions are which lead the solver to find many more false negatives when compared to the other solvers.

References

[1] Mario Alviano. The Pyglaf argumentation reasoner. *The Third International Competition on Computational Models of Argumentation (ICCMA'19)*, 2019. Online, accessed on October 16, 2022: `https://iccma2019.dmi.unipg.it/papers/ICCMA19_paper_6.pdf`.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[3] Gilles Audemard and Laurent Simon. On the Glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(1):1–25, 2018.

[4] Pietro Baroni, Dov Gabbay, Massimiliano Giacomin, and Leendert van der Torre, editors. *Handbook of Formal Argumentation*. College Publications, Nocross, GA, 2018.

[5] Stefano Bistarelli, Lars Kotthoff, Theofrastos Mantadelis, Francesco Santini, and Carlo Taticchi. Solver requirements. In *The Third International Competition on Computational Models of Argumentation (ICCMA'19)*, 1 2019. Online: accessed on September 27, 2022 `https://iccma2019.dmi.unipg.it/res/SolverRequirements.pdf`.

[6] Martin Caminada, Mikołaj Podlaszewski, and Matthew Green. Explaining the outcome of knowledge-based systems; a discussion-based approach. In *Proceedings of the Society for the Study of Artificial Intelligence and the Simulation of Behaviour*. AISB, 2013.

[7] Federico Cerutti, Sarah Gaggl, Matthias Thimm, and Johannes Wallner. Foundations of implementations for formal argumentation. *IfCoLog Journal of Logics and their Applications*, 4(8):2623–2705, 2017.

[8] Federico Cerutti, Nava Tintarev, and Nir Oren. Formal argumentation: A human-centric perspective. In *Eleventh International Workshop on Argumentation in Multi-Agent Systems (ArgMAS 2014)*, 2014.

[9] The AssertJ contributors. AssertJ, 2014. Online, accessed on October 21, 2022: `https://assertj.github.io/doc/`.

[10] The JUnit contributors. JUnit 5, 2002. Online, accessed on October 21, 2022: `https://junit.org/`.

[11] The Mockito contributors. Mockito, 2010. Online, accessed on October 21, 2022: `https://site.mockito.org/`.

[12] The Picoli contributors. Picoli - a mighty tiny command line interface, 2017. Online: `https://picocli.info/`.

[13] Sylvie Doutre and Jérôme Mengin. On sceptical versus credulous acceptance for abstract argument systems. In *European Workshop on Logics in Artificial Intelligence*, pages 462–473. Springer, 2004.

[14] Phan Minh Dung. An argumentation-theoretic foundation for logic programming. *The Journal of logic programming*, 22(2):151–177, 1995.

[15] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[16] Sarah Alice Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran. Summary report of the second international competition on computational models of argumentation. *AI Magazine*, 39(4):77–79, December 2018.

[17] Nils Geilen and Matthias Thimm. Heureka: A general heuristic backtracking solver for abstract argumentation. In *International Workshop on Theory and Applications of Formal Argumentation*, pages 143–149. Springer, 2017. Online, accessed September 24, 2022: `https://argumentationcompetition.org/2017/heureka.pdf`.

[18] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[19] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[20] Amazon Web Services Inc. Amazon Corretto – a no-cost, multiplatform, production-ready distribution of the Open Java Development Kit (OpenJDK). online, 2022. Online: accessed September 10, 2022: `https://docs.aws.amazon.com/corretto/`.

[21] Kohsuke Kawaguchi and other contributors. JaCoCo – Java code coverage library, 2009. Online, accessed on October 21, 2022: `https://www.jacoco.org/jacoco/`.

[22] László Lovász. Random walks on graphs: A survey. *Combinatorics*, 2:1–46, 1993.

[23] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[24] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[25] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Apache Maven*. Alpha Press, 2010.

[26] Andreas Niskanen and Matti Järvisalo. μ-toksia participating in ICCMA 2019. *The Third International Competition on Computational Models of Argumentation (ICCMA'19)*, 2019. Online: `https://iccma2019.dmi.unipg.it/papers/ICCMA19_paper_11.pdf`.

[27] Andreas Niskanen and Matti Järvisalo. μ-toksia: An efficient abstract argumentation reasoner. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 800–804, 9 2020. Online, accessed September 24, 2022: `https://proceedings.kr.org/2020/82/kr2020-0082-niskanen-et-al.pdf`.

[28] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205, 2021.

[29] Matthias Thimm. Tweety – A comprehensive collection of Java libraries for logical aspects of artificial intelligence and knowledge representation. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, 2014.

[30] Matthias Thimm. Stochastic local search algorithms for abstract argumentation under stable semantics. In Sanjay Modgil, Katarzyna Budzynska, and John Lawrence, editors, *Proceedings of the Seventh International Conference on Computational Models of Argumentation (COMMA'18)*, volume 305 of *Frontiers in Artificial Intelligence and Applications*, pages 169–180, Warsaw, Poland, September 2018.

[31] Matthias Thimm and Serena Villata. The first international competition on computational models of argumentation: Results and analysis. *Artificial Intelligence*, 252:267–294, 2017.

[32] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

FernUniversität in Hagen

Selbstständigkeitserklärung

Name: Dominik Fred Hillmann

Matrikel-Nr.: 6764860

Fach: Bachelorstudiengang Informatik

Modul: Bachelorarbeit

Thema: A Randomized Approach to Reasoning in Argumentation Frameworks Based on Random Walks

Ich erkläre, dass ich die Bachelorarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Datum: 22.10.2022 **Unterschrift:** D. Hill