UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

WeST People and Knowledge Networks
Institute for Web Science and Technologies

Heuristic Search Algorithms for Abstract Argumentation

Masterarbeit

zur Erlangung des Grades eines Master of Science (M.Sc.)
im Studiengang Informatik

vorgelegt von

Nils Geilen

Erstgutachter:	Prof. Dr. Steffen Staab Institute for Web Science and Technologies
Zweitgutachter:	Dr. Matthias Thimm Institute for Web Science and Technologies

Koblenz, im August 2017

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐
Der Text dieser Arbeit ist unter einer Creative Commons Lizenz verfügbar.	☐	☐
Der Quellcode ist unter einer Creative Commons Lizenz verfügbar.	☐	☐
Die erhobenen Daten sind unter einer Creative Commons Lizenz verfügbar.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Im Rahmen dieser Masterarbeit sollen die Möglichkeiten heuristischer Suchverfahren für die Abstrakte Argumentation erforscht werden. Hierzu werden spezifische Backtracking-Suchalgorithmen vorgestellt, die den Einsatz von Heuristiken unterstützen. Anschließend werden eine Reihe Heuristiken definiert, welche im Zuge dieser Masterarbeit implementiert wurden. Diese werden dann untereinander und mit anderen Ansätzen zur Abstrakten Argumentation experimentell verglichen. Für unterschiedliche Problemstellungen der Abstrakten Argumentation wird jeweils eine passende Heuristik empfohlen. Als nützlich haben sich beispielsweise Heuristiken erwiesen, die Pfade innerhalb der Graphstruktur eines Argumentationssystems analysieren.

Abstract

This thesis aims at exploring the potential of heuristic search algorithms for Abstract Argumentation. Therefore specific backtracking search algorithms are presented, which support the use of heuristics. Thereafter several heuristics are defined which have been implemented as part of this thesis. These will then be experimentally compared among each other and with other approaches to Abstract Argumentation. For different problems in Abstract Argumentation a suitable heuristic is suggested. For example heuristics which analyse paths inside the graph structure of an abstract argumentation framework have proven useful.

Contents

Ευρηκα! Ευρηκα! – *I have found it! I have found it!*

– Archimedes of Syracuse (287 – 212 BC)

1 Introduction

An important topic in artificial intelligence is reasoning in situations where conflicting information is present. Classical logics, such as propositional logic or first-order logic, cannot handle such situations. Therefore non-monotonic logics have been developed, that allow defeasible inferences. In contrast to a classical logic, a non-monotonic logic allows the enlargement of the knowledge base to invalidate conclusions which hold true in the original knowledge base. *Abstract Argumentation* is a form of non-monotonic reasoning, that models conflicting information as an argumentation system and then tries to infer plausible information from that system. Other non-monotonic logics include for example *Reiter's Default Logic* [24].

An argumentation system represents assumptions and contradictions between assumptions as arguments and attacks between arguments respectively. A very basic class of argumentation systems are abstract argumentation frameworks as proposed by Dung in 1995 [13]. An abstract argumentation framework does neither consider the statements of arguments nor the rationale behind attacks, but only the structure of the network of arguments and attacks. Lots of research into abstract argumentation frameworks has been conducted over the years and many argumentation systems have been developed which expand on the idea of abstract argumentation. For example *Assumption-Based Argumentation* [14] combines abstract argumentation with default reasoning. Dung originally defined four different semantics for reasoning in abstract argumentation systems, which are grounded, complete, preferred and stable semantics. Under different semantics, different arguments are considered justified in the same abstract argumentation framework. Later more semantics have been introduced to complement the original four semantics, for example semi-stable semantics [6].

There are two main computational approaches to reasoning in abstract argumentation frameworks: direct reasoning and reduction. Backtracking algorithms fall into the first category as they analyse argumentation frameworks directly, while the latter category includes solvers which first reduce an argumentation problem into a problem from another domain. The reduced problem is then solved by an external solver which is specific to that domain. Most commonly argumentation frameworks are reduced to answer set programming (ASP) or satisfiability (SAT) problems, as many efficient solvers are available for these kinds of problems. The fastest solvers for abstract argumentation which are currently available are reduction-based [9].

This thesis explores the heuristic backtracking approach to abstract argumentation. For most abstract argumentation problems, by far the most cost-intensive part of reasoning in abstract argumentation frameworks is the search for extensions, which are sets of arguments that form a consistent argumentation. In an abstract ar-

gumentation framework with n arguments, a search space of 2^n possible extensions has to be navigated. This task can be accomplished using a backtracking search algorithm. Backtracking search algorithms find solutions to a given problem by gradually narrowing down a potential solution. When there are several paths which could lead to the solution, a backtracking algorithm has to take a decision and chooses one of them. But if at any point it becomes apparent that the chosen path will not lead to a valid solution, it reverts all changes up to the last decision and tries another path. Backtracking algorithms for abstract argumentation have been proposed by Nofal et al. [22].

Every time a backtracking search algorithm takes a decision, it guesses which path will lead towards a solution in the shortest amount of time. Here heuristics could help to improve its guess by guiding the algorithm along a specific path through the search space. This process is rather simplistic in current backtracking approaches like [22]. The word "heuristic" is a 19^{th} century neologism which was derived from a Greek verb which is most often translated as "to find". A heuristic is a method to infer satisfactory information in situations of incomplete knowledge. This means that a well-chosen heuristic should improve the result in most cases, but will probably worsen the result in some cases. The heuristic method therefore constitutes a compromise between performance and accuracy. The only information available in abstract argumentation is the set of arguments and the relation of defeat between those arguments. Especially the semantic meaning of arguments is ignored. Therefore heuristics in abstract argumentation can only analyse the structure fo the defeat network between the arguments of an abstract argumentation framework. Examples of heuristics for abstract argumentation have been suggested for example in [11], but no systematic comparison of heuristics has been conducted yet. This thesis tries to improve on this underdeveloped area of reasoning in abstract argumentation by defining and evaluating several new heuristics for abstract argumentation.

An abstract argumentation solver called HEUREKA was implemented to demonstrate the impact on the performance of the reasoning process which can be achieved with the heuristics that will be presented in this thesis. HEUREKA complies to the specifications of *The Second International Competition on Computational Models of Argumentation* (ICCMA'17) [17] and an early version of the solver takes part in several tracks of this competition. This competition as well as its predecessor which was held in 2015 lets different abstract argumentation solvers compete against each other by tasking them to solve several abstract argumentation problems on a number of randomly generated argumentation frameworks as fast as possible. For every problem a heuristic will be determined which the HEUREKA solver will employ by default, but HEUREKA will also allow for custom heuristics to be used which can be specified with the help of a specific notation.

First in Section 2 abstract argumentation frameworks and their semantics will be defined. The different backtracking algorithms and heuristics for abstract argumentation will then be presented in Sections 3 and 4 respectively. Section 5

will detail on the implementation of these algorithms and heuristics as part of the HEUREKA solver. In this section the notation for the definition of custom heuristics for HEUREKA will also be described. In Section 6 the performance of HEUREKA will finally be evaluated and compared to the performance of other abstract argumentation solvers. To accomplish this the different heuristics will be first compared among each other to determine a good heuristics for different scenarios.

2 Abstract Argumentation

Abstract argumentation is a method to infer information from a set of conflicting statements (arguments) by constructing valid sets of statements, which can be accepted together (extensions). Several problems in artificial intelligence can be modelled by an abstract argumentation framework. This section starts with introduction to abstract argumentation frameworks as a way to formalise abstract argumentation problems. It will then define semantics and extensions which are required for reasoning in abstract argumentation, and will finally give an overview of labellings as an alternative way to express semantics.

2.1 Argumentation Frameworks

An abstract argumentation framework as defined by Dung [13] represents a network of arguments and attacks between those.

Definition 2.1. An **abstract argumentation framework** (AAF) is a tuple $\Gamma = (\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ is a set of arguments and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ is a binary attack relation between arguments. If $(a, b) \in \mathcal{R}$ it is said that argument a attacks argument b, which is also denoted as $a \rightarrow b$. The attack relation can contain circles. Circles and other paths can be denoted as $a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n$ with $a_1, a_2, \ldots, a_n \in \mathcal{A}$ and $a_i \rightarrow a_{i+1}$ for $i = 1, 2, \ldots, n-1$. Further it is also allowed that an argument attacks itself and that two arguments mutually attack each other. A mutual attack between two arguments $a, b \in \mathcal{A}$ is written as $a \rightleftarrows b$.

An argumentation framework can be visualised as a simple, directed graph, where every node stands for an argument while every edge stands for an attack.

Example 2.2. A graph representation of an AAF is shown in Figure 1. The depicted AAF consists of four arguments and six attacks. The argument d attacks itself ($d \rightarrow d$), while c and d mutually attack each other ($c \rightleftarrows d$). A circle is for example formed by the arguments a, b and c ($a \rightarrow b \rightarrow c \rightarrow a$).

Before abstract argumentation semantics can be defined which allow reasoning in argumentation frameworks, several concepts and notions must be introduced which describe relations between argument sets, such as the defence of arguments. Let $E \subseteq \mathcal{A}$ be a set of arguments in an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$. It is said that E attacks an argument $b \in \mathcal{A}$ iff there is an argument $a \in E$ such that $a \rightarrow b$. Likewise it is said that an argument $a \in \mathcal{A}$ attacks E iff there is an argument $b \in E$ such that $a \rightarrow b$.

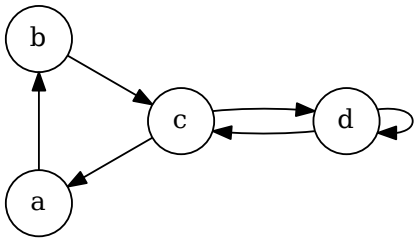


Figure 1: An abstract argumentation framework $\Gamma = (\mathcal{A}, \mathcal{R})$ with $\mathcal{A} = \{a, b, c, d\}$ and $\mathcal{R} = \{(a, b), (b, c), (c, a), (c, d), (d, c), (d, d)\}$ visualized as a directed graph.

Definition 2.3. For any argument set $E \subseteq \mathcal{A}$ in an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$, let E^+_Γ be the set of arguments attacked by E and E^-_Γ the set of arguments which attack at least one element of E. Let $E^\pm_\Gamma$ be the union of E^+_Γ and E^-_Γ. To improve readability E^+_Γ, E^-_Γ and $E^\pm_\Gamma$ will from now on be shortened to E^+, E^- and $E^\pm$ where the AAF Γ is implicit.

$$E^+ = \{a \in \mathcal{A} \mid \exists e \in E : e \to a\}$$

$$E^- = \{a \in \mathcal{A} \mid \exists e \in E : a \to e\}$$

$$E^\pm = E^+ \cup E^-$$

From this it follows that for any argument set $E \subseteq \mathcal{A}$ and any argument $a \in \mathcal{A}$

$$E \text{ attacks } a :\Longleftrightarrow a \in E^+$$

Similarly it is said that an argument set E defends an argument a iff for all arguments $b \in \mathcal{A}$ it holds that if b attacks a there is an $e \in E$ such that e attacks b. An argument set can defend arguments inside of it as well as arguments outside of it.

$$E \text{ defends } a :\Longleftrightarrow \{a\}^- \subseteq E^+$$

The notion of attack and defence can be extended to sets of argument. An argument set A attacks another argument set B iff A attacks at least one element of B and A defends B iff A defends all elements of B.

Let us finally introduce the characteristic function of abstract argumentation which will help to define several abstract argumentation semantics in the next section.

Definition 2.4. Let the **power set** $\mathcal{P}(S)$ of a set S be the set which consists of exactly all subsets of S. Especially for every set S it holds that $S \in \mathcal{P}(S)$ and $\emptyset \in \mathcal{P}(S)$.

$$\mathcal{P}(S) = \{E \mid E \subseteq S\}$$

Definition 2.5. The **characteristic function** $\mathfrak{F} : \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{A})$ of an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ maps a set of arguments to all arguments which it defends.

$$\mathfrak{F}(E) = \{a \in \mathcal{A} \mid \{a\}^- \subseteq E^+\}$$

Example 2.6. Consider the argument c in the AAF given in Figure 1. The set of attackers of c is $\{c\}^- = \{b, d\}$, while its attackee set is $\{c\}^+ = \{a, d\}$. The set of arguments which are defended by $\{c\}$ is $\mathfrak{F}(\{c\}) = \{b\}$ as c defeats a which in turn is the only attacker of b.

2.2 Semantics

Argumentation semantics are required to draw conclusions about the validity of arguments from an AAF. Different results can be obtained when utilising different semantics. All semantics are defined with the help of extensions, that is sets of arguments which form a coherent argumentation. Considering an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$, sets of arguments might have the following properties.

Definition 2.7. An argument $a \in \mathcal{A}$ is called **acceptable** with respect to a set of arguments $E \subseteq \mathcal{A}$ iff E defends a, i.e. $a \in \mathfrak{F}(E)$.

Therefore the characteristic function $\mathfrak{F}$ maps an argument set to the set of all arguments which are acceptable with respect to that set.

Definition 2.8. A set of arguments $E \subseteq \mathcal{A}$ is called **conflict-free** iff there are no two arguments $a, b \in E$ such that a attacks b. An argument which attacks itself can therefore never be part of a conflict-free set.

Definition 2.9. A conflict-free argument set is called **admissible** iff it defends all of its elements. With other words, all elements of an admissible set are acceptable with respect to that set.

$$E \text{ admissible} :\Longleftrightarrow \nexists a, b \in E : a \rightarrow b \text{ and } E \subseteq \mathfrak{F}(E)$$

For example the empty set is admissible in every argumentation framework [2]. To deduce which arguments are to be believed, one has to find valid sets of arguments which are acceptable together, such a set is called an extension. An extension is a set of compatible arguments, that can be jointly believed and form a consistent reasoning. Dung establishes four alternative definitions to describe extensions: complete, stable, preferred and grounded semantics [13]. Additional semantics have been defined over time, e.g. semi-stable, ideal and stage semantics [6, 5, 26].

Definition 2.10. In an argumentation framework $\Gamma = (\mathcal{A}, \mathcal{R})$, the set of extensions which is valid under specific semantics σ is denoted by $\mathcal{E}_\sigma(\Gamma) \subseteq \mathcal{P}(\mathcal{A})$. An admissible set of arguments is called

- **complete extension** $E \in \mathcal{E}_{CO}(\Gamma)$ iff every argument that is defended by E is part of E, i.e. $E = \mathfrak{F}(E)$
- **stable extension** $E \in \mathcal{E}_{ST}(\Gamma)$ iff E attacks every argument $a \in \mathcal{A} \backslash E$
- **preferred extension** $E \in \mathcal{E}_{PR}(\Gamma)$ iff it is maximally admissible, i.e. $E \in \mathcal{E}_{CO}(\Gamma)$ and there is no $E' \in \mathcal{E}_{CO}(\Gamma)$ with $E \subset E'$

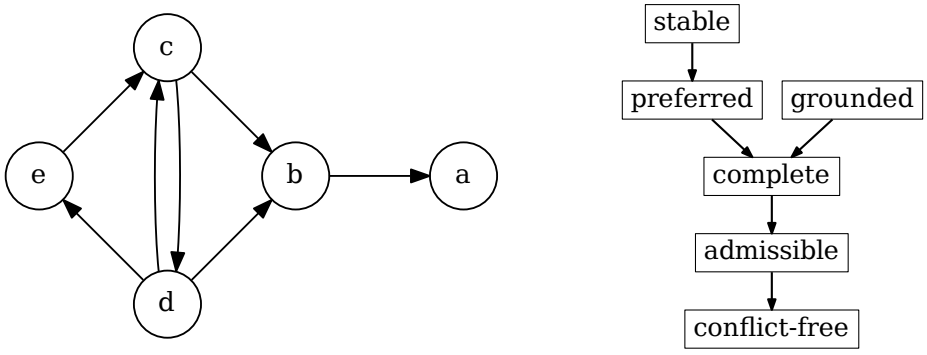


Figure 2: *Left:* The AAF from examples 2.11 and 3.15. *Right:* Hierarchy of semantics with respect to extension set inclusion. $\sigma \rightarrow \sigma'$ means that every extension under semantics σ is also an extension under σ' [10, 5]. This for example means that all admissible sets are at the same time also conflict-free.

- **grounded extension** $E \in \mathcal{E}_{GR}(\Gamma)$ iff it is the single smallest complete extension, i.e. $E \in \mathcal{E}_{CO}(\Gamma)$ and there is no $E' \in \mathcal{E}_{CO}(\Gamma)$ with $E' \subset E$.

Example 2.11. An AAF as shown by the graph on the left of Figure 2 has an empty grounded extension $E_{GR} = \emptyset$. It further has two complete extensions $\{a, d\}$ and $\emptyset$. Additionally $\{a, d\}$ is preferred and stable while $\emptyset$ is neither preferred because it is a subset of $\{a, d\}$ nor stable because it does not attack any of the remaining arguments.

Theorem 2.12. Every AAF Γ has exactly one grounded extension $E_{GR} = \bigcap \mathcal{E}_{CO}(\Gamma)$ which is the intersection of all complete extensions of Γ [20].

Theorem 2.13. Every stable extension is maximal. That means for every stable extension $E \in \mathcal{E}_{ST}(\Gamma)$ in any AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ it holds that E cannot have a strict superset $E' \supset E$ such that E' is also a stable extension.

Proof 2.14. Any superset $E' \supset E$ would need to include at least one argument $a \in \mathcal{A} \setminus E$ from the complement of E, but the inclusion of this argument a would cause internal conflict because the stable extension by definition attacks a. Thus there cannot be a stable extension which is a superset of another stable extension.

Every grounded or preferred extension is also a complete extension by definition. A stable extension is also complete because by attacking all remaining arguments it eliminates all possible attackers and thus defends all of its elements. Additionally it is preferred as maximality is ensured because no additional argument can be added to a stable extension without causing conflict. Therefore for example

complete semantics are more permissive than grounded, preferred, and stable semantics. This yields the following partial order on semantics which is also shown by the graph on the right of Figure 2 [10, 5]:

$$\mathcal{E}_{ST}(\Gamma) \subseteq \mathcal{E}_{PR}(\Gamma) \subseteq \mathcal{E}_{CO}(\Gamma)$$

$$\mathcal{E}_{GR}(\Gamma) \subseteq \mathcal{E}_{CO}(\Gamma)$$

Theorem 2.15. The grounded extension is a subset of every complete extension as it is the intersection of all complete extensions. It is therefore also part of every stable and preferred extension because those are also complete.

A short description of the Stable Marriage Problem follows as an example of an application of abstract argumentation and stable argumentation semantics.

Example 2.16. The problem where a group of men and women have to be arranged into pairs of spouses while considering each persons preferences is known as the **Stable Marriage Problem** (SMP). Each instance of an SMP has a stable solution. Let M be a set of men and W be a set of women and let both sets be of equal size. For every man $m \in M$ there is an ordering $>_m$ over all women which indicates personal preference, analogously there if an ordering $>_w$ over all men for every woman $w \in W$. For example $m_1 >_w m_2$ indicates that w prefers m_1 over m_2. A solution for the SMP is a spousal relation, which can be given by a bijective function $S : M \to W$ where $S(m)$ denotes the wife of m and $S^{-1}(w)$ denotes the husband of w. S is a valid solution if there is no couple $(m, w) \in M \times W$ where m prefers w to his wife $S(m)$ and w prefers m to her husband $S^{-1}(w)$.

$$\nexists m \in M : \exists w \in W : w >_m S(m) \text{ and } w >_w S^{-1}(w)$$

The SMP can be modelled as an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ where $\mathcal{A} = M \times W$ is the set of all possible marriages and $\mathcal{R}$ is an attack relation on $\mathcal{A}$, where a coupling defeats another coupling iff one of the spouses is the same in both coupling and that spouse prefers the partner in the first coupling over the partner in the second one [13].

$$\begin{aligned}(m_1, w_1) \rightarrow (m_2, w_2) \text{ iff } & m_1 = m_2 \text{ and } w_1 >_{m_1} w_2 \\ \text{or } & w_1 = w_2 \text{ and } m_1 >_{w_1} w_2\end{aligned}$$

A stable extension $E \in \mathcal{E}_{ST}(\Gamma)$ is a solution for the SMP. The spousal function S can be constructed in the following way:

$$S(m) = w \text{ iff } (m, w) \in E$$

Consider for example a simple but greatly tragic SMP with two men, Joe and Tom, and two women, Ann and Mary. While Joe prefers Ann and Tom prefers Mary, Ann prefers Tom and Mary prefers Joe.

$$\mathrm{Ann} >_{\mathrm{Joe}} \mathrm{Mary},\ \mathrm{Mary} >_{\mathrm{Tom}} \mathrm{Ann},\ \mathrm{Tom} >_{\mathrm{Ann}} \mathrm{Joe},\ \mathrm{Joe} >_{\mathrm{Mary}} \mathrm{Tom}$$

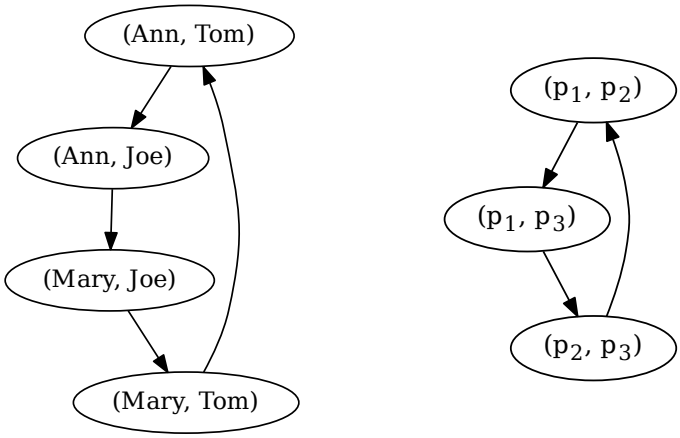


Figure 3: *Left:* An AAF that models an SMP. *Right:* An AAF that models a triangular relation in a SMPG

This SMP is modelled by the AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ with four arguments and four attacks which is visualized on the left in Figure 3.

$$\mathcal{A} = \{\mathrm{Joe}, \mathrm{Tom}\} \times \{\mathrm{Ann}, \mathrm{Mary}\}$$

This AAF has two stable extensions, $E_1 = \{(\mathrm{Mary}, \mathrm{Joe}), (\mathrm{Ann}, \mathrm{Tom})\}$ and $E_2 = \{(\mathrm{Mary}, \mathrm{Tom}), (\mathrm{Ann}, \mathrm{Joe})\}$, which are both solutions to the SMP. In the first extension E_1 for example, Joe's and Mary's marriage could be challenged by the fact that Joe would prefer Ann over Mary, but this is not relevant as Ann is grouped with Tom whom she prefers over Joe. Formally the coupling $(\mathrm{Mary}, \mathrm{Joe})$ is attacked by the coupling $(\mathrm{Ann}, \mathrm{Joe})$, but defended by the coupling $(\mathrm{Ann}, \mathrm{Tom})$, which defeats $(\mathrm{Ann}, \mathrm{Joe})$.

When SMP is expanded into the Stable Marriage Problem with Gays (SMPG), situations can occur where no stable solution can be found.

Example 2.17. When the possibility of same-sex coupling is added to the SMP and thereby triangle relations can exist, it can be observed that in some instances no stable solutions are possible [13].

Let $P = \{p_1, p_2, p_3\}$ be a set of three persons, where p_1 prefers p_2, p_2 prefers p_3 and p_3 prefers p_1. The task is now to find a stable solution to the SMPG, i.e. a set of couplings which defeat all other couplings. Therefore this SMPG is modelled as an AAF, which is depicted on the right in Figure 3. Under stable semantics there is no solution to this AAF, thus there is no solution to this SMPG. The same holds for grounded, complete and preferred semantics. When considering that (p_1, p_2)

is unstable because p_2 prefers (p_2, p_3) and (p_1, p_3) is unstable because p_1 prefers (p_1, p_2) and (p_2, p_3) is unstable because p_3 prefers (p_1, p_3), it becomes apparent that there cannot be a stable solution, which is supported by the model.

There are two different ways to draw conclusions about the validity of an argument from the extension set which corresponds to specific argumentation semantics. These are called credulous and sceptical justification.

Definition 2.18. Under grounded semantics an argument is considered justified iff it is part of the only grounded extension. Under complete, stable, and preferred semantics an argument is

- **credulously justified** iff it is contained in at least one extension.
- **sceptically justified** iff it is contained in all extensions.

Sceptical justification is generally stricter than credulous justification. Under stable semantics it can occur that an AAF has no valid extension at all, in this case every element is sceptically justified while no element is credulously justified. The highest acceptance is achieved when an argument is credulously as well as sceptically justified. The two following examples illustrate the differences between credulous and sceptical justification.

Example 2.19. Given the AAF shown by the graph on the left of Figure 2. Nothing is justified under grounded semantics because the grounded extension is empty. Under complete semantics a and d are credulously justified because both are part of the union of all complete extensions $\bigcup \mathcal{E}_{CO} = \{a, d\}$ but nothing is sceptically justified because the intersection of all complete extensions $\bigcap \mathcal{E}_{CO} = \emptyset$ is empty. Under preferred and stable semantics there is only one extension $\{a, d\}$, so a and d are both sceptically and credulously justified.

Example 2.20. The **Nixon Diamond** is a problem of the following form: Nixon is a Quaker and Quakers are pacifists, but Nixon is also a Republican and Republicans are not Pacifists. This problem can be expressed in first-order logic as a set of terms, which would be considered inconsistent:

$$\begin{aligned} &quaker \\ &quaker \Rightarrow pacifist \\ &republican \\ &republican \Rightarrow \neg pacifist \end{aligned}$$

The Nixon Diamond can be alternatively modelled as the following simple AAF:

$$pacifist \rightleftarrows \neg pacifist$$

While the grounded extension is empty, $\{pacifist\}$ and $\{\neg pacifist\}$ are both stable, preferred as well as complete extensions. Therefore under stable, preferred and complete semantics, both $pacifist$ and $\neg pacifist$ are credulously justified while nothing is sceptically justified [3].

An abstract argumentation problem consists of a task in combination with semantics. ICCMA'17 [17] defines the following tasks for each semantics $\sigma \in \{\mathsf{CO}, \mathsf{PR}, \mathsf{ST}, \mathsf{GR}\}$ and an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$:

- Find a single extension $E \in \mathcal{E}_\sigma(\Gamma)$
- Enumerate all extensions $\mathcal{E}_\sigma(\Gamma)$
- Decide whether $a \in \mathcal{A}$ is credulously justified under σ, i.e. there is an $E \in \mathcal{E}_\sigma(\Gamma)$ with $a \in E$
- Decide whether $a \in \mathcal{A}$ is sceptically justified under σ, i.e. for all $E \in \mathcal{E}_\sigma(\Gamma)$ it holds that $a \in E$

To be able to easily identify problems and distinguish them among each other, a problem may be denoted as a pair TASK-SEMANTICS, where the TASK is any of EE (enumerate extensions), SE (find a single extension), DS (decide/justify sceptically) or DC (decide/justify credulously), while SEMANTICS refers to either CO (complete), PR (preferred), GR (grounded) or ST (stable). Problems are for example enumeration of all complete extensions (EE-CO) or sceptical justification under stable semantics (DS-ST). This thesis aims at providing methods to answer these problems with respect to a specific AAF.

Under grounded semantics the task of finding a single extension is equal to the task of enumerating all extensions as there is always exactly one extension. Therefore an argument is credulously justified under grounded semantics exactly if it is sceptically justified under grounded semantics. Additionally sceptical justification under complete semantics is equal to justification under grounded semantics as the grounded extension is the intersection of all complete extensions.

$$\text{SE-GR} = \text{EE-GR}$$

$$\text{DC-GR} = \text{DS-GR} = \text{DS-CO}$$

Before the complexity of these problems can be classified, several complexity classes need to be defined. These definitions rely on Turing machines, for a description of deterministic and non-deterministic Turing machines see for example [15].

Definition 2.21. A complexity class is a set of problems which can be solved by spending a maximal amount of a certain resource. This maximal amount depends on the problem size. Let the following all complexity classes concerning time.

- **P** is the set of all problems which can be decided by a deterministic Turing machine in polynomial time
- **NP** is the set of all problems which can be decided by a non-deterministic Turing machine in polynomial time

task	semantics			
	GR	**CO**	**ST**	**PR**
credulous	P-complete	NP-complete	NP-complete	NP-complete
sceptical	P-complete	P-complete	coNP-complete	Π_2^P-complete

Figure 4: This table specifies the complexity classes of different justification problems in abstract argumentation according to [15].

P-complete	**(co-)NP-complete**	**Π_2^P-complete**
GR	DC-CO	DS-PR
DS-CO	ST	
	DC-PR	

Figure 5: Ordering of justification problems according to their complexity class. A problem is not harder than a problem in a column to its right.

- **coNP** is the set of all problems whose negation can be decided by a non-deterministic Turing machine in polynomial time

Theorem 2.22. Every problem which is part of P is proven to be also part of NP as well as coNP [15]. Π_2^P is an additional complexity class which is potentially even harder than NP. For a definition of Π_2^P, see [15].

$$P \subseteq NP \subseteq \Pi_2^P$$

$$P \subseteq coNP \subseteq \Pi_2^P$$

Definition 2.23. For any complexity class $\mathcal{C}$, a problem is considered $\mathcal{C}$-hard if any problem in $\mathcal{C}$ can be reduced to it. A problem which is part of $\mathcal{C}$ and $\mathcal{C}$-hard is called $\mathcal{C}$-complete. Therefore $\mathcal{C}$-complete problems are the hardest problems inside the complexity class $\mathcal{C}$.

Figure 4 lists complexity classes for different justification problems in abstract argumentation. None of the problems is trivial, but they vary greatly in complexity. The easiest problems are sceptical and credulous justification under grounded semantics and sceptical justification under complete semantics, which are all P-complete. Those three semantics are equal, therefore they have equal complexity. The potentially hardest problem is sceptical justification under preferred semantics, as this problem possibly lies neither in NP nor in coNP. This ordering is visualized in Figure 5.

2.3 Labellings

Legal sets of arguments under a specific semantics can alternatively be described by argument labellings. Labellings extend the idea of extensions, but are more suitable

for keeping the state of an algorithm as a labelling can be represented as one simple integer array.

Definition 2.24. A **Λ-labelling** $\mathcal{L} \in \mathfrak{L}(\Lambda, \Gamma)$ for an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ is a function $\mathcal{L} : \mathcal{A} \to \Lambda$ where Λ is a set of labels with $|\Lambda| = n$. $\mathcal{L}$ can also be seen as a partition of $\mathcal{A}$, i.e. an n-tuple of disjoint subsets of $\mathcal{A}$ whose union is $\mathcal{A}$. Then every element of that tuple refers to one label and contains all arguments that were assigned that label. For every label $\lambda \in \Lambda$ the corresponding element is defined as the argument set

$$\lambda(\mathcal{L}) = \mathcal{L}^{-1}(\lambda) = \{a \in \mathcal{A} \mid \mathcal{L}(a) = \lambda\}$$

Definition 2.25. The most common instance of a class of labellings maps arguments to $\Lambda_3 = \{\mathrm{IN}, \mathrm{OUT}, \mathrm{UNDEC}\}$. From now on this class of labellings is written as just $\mathfrak{L}(\Gamma) = \mathfrak{L}(\Lambda_3, \Gamma)$ as all labelling-based semantics are defined with a three-label mappings of the type

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

Such a labelling can be alternatively denoted as the triple

$$\mathcal{L} = (\mathrm{IN}(\mathcal{L}), \mathrm{OUT}(\mathcal{L}), \mathrm{UNDEC}(\mathcal{L}))$$

with for example $\mathrm{IN}(\mathcal{L}) = \{a \in \mathcal{A} | \mathcal{L}(a) = \mathrm{IN}\}$ [2, 22]. Additionally to the two labelling notations which have been introduced in Definition 2.24, a third way to describe a labelling is to represent it as a set of tuples $(a, \mathcal{L}(a))$ which consist of an argument and the corresponding label, which is illustrated by the following example.

Example 2.26. Let there be three arguments $a, b, c \in \mathcal{A}$ and a labelling $\mathcal{L} : \mathcal{A} \to \Lambda_3$ with $\mathcal{L}(a) = \mathrm{IN}$, $\mathcal{L}(b) = \mathrm{OUT}$ and $\mathcal{L}(c) = \mathrm{UNDEC}$. This labelling can be alternatively denoted as a tuple of sets

$$\mathcal{L} = (\{a\}, \{b\}, \{c\})$$

or a set of tuples

$$\mathcal{L} = \{(a, \mathrm{IN}), (b, \mathrm{OUT}), (c, \mathrm{UNDEC})\}$$

The backtracking algorithms which will be introduced in Section 3 rely on a four-label mapping which includes an additional label BLANK which is used by the algorithms to mark arguments which it has not yet visited.

Definition 2.27. Let $\mathfrak{L}(\Lambda_4, \Gamma)$ be the set of all labelling functions of the type $\mathcal{L} : \mathcal{A} \to \Lambda_4$ with $\Lambda_4 = \{\mathrm{IN}, \mathrm{OUT}, \mathrm{UNDEC}, \mathrm{BLANK}\}$.

Extension semantics can alternatively expressed as labellings. Generally the label IN means that an argument is included into the extension, an argument labelled OUT is defeated by the extension and an argument labelled UNDEC is neither included into the extension nor defeated by the extension.

Definition 2.28. Let a be an argument and let $\lambda \in \{IN, OUT, UNDEC\}$ be a label. Further $\mathcal{L}$ is a labelling and it holds that $\mathcal{L}(a) = \lambda$. The argument label assignment $a \mapsto \lambda$ is then either considered legal or otherwise considered illegal. a is then called legally or illegally λ respectively. Whether the assignment $a \mapsto \lambda$ is legal depends on the labels of the attackers of a [7].

- An argument $a \in IN(\mathcal{L})$ is considered legally IN iff every argument which attacks a is labelled OUT, i.e. $\{a\}^- \subseteq OUT(\mathcal{L})$
- An argument $a \in OUT(\mathcal{L})$ is considered legally OUT iff there is an argument b which is labelled IN and attacks a, i.e. $\{a\}^- \cap IN(\mathcal{L}) \neq \emptyset$
- An argument $a \in UNDEC(\mathcal{L})$ is considered legally UNDEC iff it is not attacked by any argument from $IN(\mathcal{L})$ but not every argument that attacks a is labelled OUT, i.e. $\{a\}^- \cap IN(\mathcal{L}) = \emptyset$ and $\{a\}^- \nsubseteq OUT(\mathcal{L})$

Definition 2.29. In any AAF, a labelling $\mathcal{L}$ is called an **admissible labelling** iff no argument is either illegally IN or illegally OUT.

The different extension-based semantics defined in Section 2.2 can now be redefined with the help of labellings as labelling-based semantics.

Definition 2.30. Given an abstract argumentation framework Γ, a labelling $\mathcal{L} \in \mathfrak{L}(\Gamma)$ is now called

- **complete labelling** iff it is admissible and additionally has no illegally UNDEC arguments, i.e. both of the following statements hold
  i) $\mathcal{L}(a) = IN$ iff all all of its attackers are labelled OUT
  ii) $\mathcal{L}(a) = OUT$ iff there is an argument $b \in \{a\}^-$ with $\mathcal{L}(b) = IN$
- **stable labelling** iff it is complete and $UNDEC(\mathcal{L}) = \emptyset$
- **grounded labelling** iff it is complete and $IN(\mathcal{L})$ is minimal, i.e. for every complete labelling $\mathcal{L}'$ it holds that $IN(\mathcal{L}') \subseteq IN(\mathcal{L})$
- **preferred labelling** iff it is complete and $IN(\mathcal{L})$ is maximal, i.e. there is no complete labelling $\mathcal{L}'$ with $IN(\mathcal{L}) \subseteq IN(\mathcal{L}')$

Every labelling-based semantics corresponds to extension based-semantics in the following way.

Theorem 2.31. For any extension semantics σ, the argument set $E = IN(\mathcal{L})$ which consists of all arguments labelled IN by $\mathcal{L}$ is a σ-extension iff $\mathcal{L}$ is a σ-labelling [7].

Argument labellings will be utilised by the search algorithms presented in the following section, which try to find extension under certain semantics. These algorithms attempt to construct labellings which represent valid extensions under specific semantics. The corresponding extensions can then be easily deduced from these labellings.

3 General Backtracking Search Algorithms

This section introduces several algorithms which enumerate extensions of an AAF under certain semantics and deduces their correctness. Except for the relatively trivial algorithm which computes the grounded extension, all algorithms utilise backtracking and can be adopted to use different heuristics. Section 5 gives details on how the HEUREKA solver implements the algorithms which will be defined in this section. Because the different argumentation semantics which were originally defined by Dung all have certain similarities, the different algorithms which are used to compute extensions under these semantics are similar and partly rely on each other. For example the grounded extension is part of every complete extension because the grounded extension is the intersection of all complete extensions. It is also part of every stable and preferred extension as those are complete extensions at the same time. Thus as a first step every algorithm will compute the grounded extension to improve its performance. Therefore when being invoked, all of the backtracking algorithms as a first step call the grounded algorithm and task it with finding the grounded extension.

3.1 Computation of the Grounded Extension

Finding the grounded extension is relatively trivial compared to extensions under other semantics and can be done with smaller effort. Let us recall the definition of the grounded extension. The grounded extension is the minimal complete extension, which is the same as the intersection of all complete extensions. Therefore an argument is part of the grounded extension iff it is either not attacked by another argument or is recursively defended by such an argument. An argument is recursively defended iff every attacker of this argument is either attacked by an unattacked argument or attacked by an argument which is again recursively defended. Every unattacked argument has to be part of every complete extension as it is defended by every extension. The same holds for arguments which are recursively defended by an unattacked argument. Those arguments form a minimal complete extension, which is the grounded extension, as they form a complete extension and are at the same time part of every complete extension.

Example 3.1. The abstract argumentation framework which is given in Figure 6 has the grounded extension $E_{GR} = \{a, c, e\}$. a is not attacked by any argument, c is recursively defended by the unattacked argument a, and d is recursively defended by the recursively defended argument c. The remaining arguments b, d and f cannot be recursively defended and can thus not be part of the grounded extension.

The grounded extension is equal to the set of all unattacked and recursively defended arguments. The set of all unattacked arguments can be described by applying the characteristic function $\mathfrak{F}$ to the empty set. Renewed application of the characteristic function to this set yields a set which contains all arguments defended

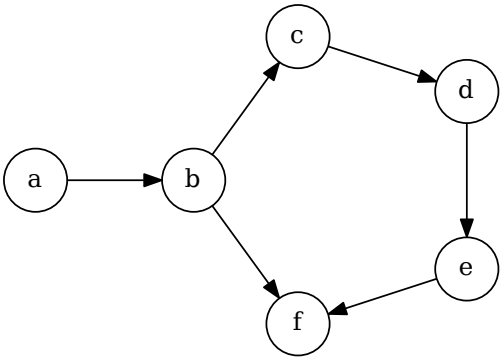


Figure 6: The AAF from Examples 3.1 and 3.3.

by unattacked arguments. It is apparent that the grounded extension can be computed by repeating this process until a fix point is reached, i.e. further application of the characteristic function would not change the set.

Theorem 3.2. The grounded extension E_{GR} is the set of all arguments which are recursively defended by the empty set, i.e. the union of all sets which can be derived from the empty set by repeated application of the characteristic function $\mathfrak{F}$ [20].

$$E_{GR} = \bigcup_i \mathfrak{F}^i(\emptyset) \text{ where } \mathfrak{F}^1 = \mathfrak{F} \text{ and } \mathfrak{F}^n = \mathfrak{F} \circ \mathfrak{F}^{n-1} \text{ for } n > 1$$

Example 3.3. As a continuation of Exmaple 3.1 and considering the abstract argumentation framework given in Figure 6, we want to construct the grounded extension with help of the characteristic function. Therefore the following steps are necessary:

1. $\mathfrak{F}(\emptyset) = \{a\}$
2. $\mathfrak{F}^2(\emptyset) = \mathfrak{F}(\{a\}) = \{a, c\}$
3. $\mathfrak{F}^3(\emptyset) = \mathfrak{F}(\{a, c\}) = \{a, c, e\}$
4. $\mathfrak{F}^4(\emptyset) = \mathfrak{F}(\{a, c, e\}) = \{a, c, e\}$

A fixpoint has been reached at step 4, so we can construct the grounded extension $E_{GR} = \bigcup_{i=1}^{4} \mathfrak{F}^i(\emptyset) = \{a, c, e\}$.

Algorithm 1 uses the method described above to compute the grounded extension [21]. The algorithm is purely iterative and does not require any heuristic guidance. It starts with the empty set as a partial solution and then repeat to include all arguments which are defended by the current partial solution. It does that until it reaches a fixpoint, i.e. there are no more defended arguments. Two different

Algorithm 1 Compute the Grounded Extension [21]

Input: $\Gamma = (\mathcal{A}, \mathcal{R})$ AAF
$\mathcal{L} \in \mathfrak{L}(\Gamma)$ labelling function

Output: $E_{GR} \subseteq \mathcal{A}$ grounded extension

1: **for all** $a \in \mathcal{A}$ **do**
2: $\quad \mathcal{L}(a) \leftarrow \text{UNDEC}$
3: **while** $\mathcal{L}$ changed **do**
4: $\quad$ **for all** $a \in \text{UNDEC}(\mathcal{L})$ **do**
5: $\quad\quad$ **if** $\{a\}^- \setminus \text{OUT}(\mathcal{L}) = \emptyset$ **then**
6: $\quad\quad\quad \mathcal{L}(a) \leftarrow \text{IN}$
7: $\quad\quad\quad$ **for all** $b \in \{a\}^+$ **do**
8: $\quad\quad\quad\quad \mathcal{L}(b) \leftarrow \text{OUT}$
9: **return** $E_{GR} = \text{IN}(\mathcal{L})$

versions of Algorithm 1 have been implemented in HEUREKA, one can be used to find the grounded extension while the other one can be used to justify an argument under grounded semantics. The implementation of Algorithm 1 in HEUREKA for every argument keeps track of the number of attackers of that argument which are not defeated and therefore not yet labelled OUT, and as soon as such an counter reaches zero the corresponding argument is set to the label IN [20]. Algorithm 1 is also used for finding a single complete extension and sceptical justification under complete semantics, as those tasks can be done by finding the grounded extension or deciding justification under grounded semantics respectively.

3.2 Search for Admissible Subsets

Complete, preferred and stable extensions are each enumerated by a backtracking algorithm. To illustrate the basic idea of a backtracking, in this section a backtracking algorithm which searches admissible sets is gradually being developed. A backtracking algorithm is a brute-force algorithm which in a depth-first manner constructs solutions for a problem until it either finds a full solution or reaches a point where it becomes apparent that the current partial solution cannot be part of a full solution. In the second case it backtracks, i.e. it undoes the last decision and picks another route. The backtracking algorithms described in this thesis search extensions, i.e. subsets of an argument set $\mathcal{A}$. There is a binary decision for every argument, as it can be either inside or outside the solution. Therefore there are 2^n possible paths with $n = |\mathcal{A}|$, where every path corresponds to a different subset $E \subseteq \mathcal{A}$. To illustrate the basic idea of backtracking, let us first consider three examples of backtracking algorithms which perform simpler subtasks of extension enumeration. The examples each extend the preceding one and the third example will finally enumerate all admissible sets.

Algorithm 2 Enumerate All Subsets

Input: $\mathcal{A}$ argument set
$\mathcal{L} : \mathcal{A} \to \{\mathsf{IN}, \mathsf{OUT}, \mathsf{UNDEC}\}$ labelling function

Output: $\mathcal{P}(\mathcal{A})$ power set of $\mathcal{A}$

```
1: for all a ∈ A do
2:     L(a) ← UNDEC
3: FIND_SUBSETS(L)
4: procedure FIND_SUBSETS(L)
5:     if there is an argument a with L(a) = UNDEC then
6:         L' ← L
7:         L'(a) ← IN
8:         FIND_SUBSETS(L')
9:         L(a) ← OUT
10:        FIND_SUBSETS(L)
11:    else
12:        add IN(L) to P(A)
```

Example 3.4. As a simplistic example for backtracking, Algorithm 2 enumerates all subsets of an argument set $\mathcal{A}$. This task is equal to constructing the power set $\mathcal{P}(\mathcal{A})$. All the different subsets constructed by Algorithm 2 and the order of their construction are shown in the graph in Figure 7. Initially the algorithm builds a labelling $\mathcal{L}$ with $\mathsf{UNDEC}(\mathcal{L}) = \mathcal{A}$. At every iteration it then does one of two things: If there is still an argument a with $\mathcal{L}(a) = \mathsf{UNDEC}$, it adds that argument to the current partial solution $\mathsf{IN}(\mathcal{L})$ and resumes with the next iteration. This is called a decision as the algorithm has decided to include an argument which could either be inside or outside the solution. If there is no more such argument, the algorithm reports the current partial solution $\mathsf{IN}(\mathcal{L})$ as a subset. After doing so it backtracks by reverting all changes until the last decision. It starts with the argument label which was last changed. If that label is IN, the decision is reverted by setting the label to OUT and the algorithm then does not backtrack further but resumes with another iteration. If that label is OUT, the decision to include that argument has already been reverted, therefore its label is changed to UNDEC again and the algorithm backtracks further. When the algorithm has to backtrack but there is no more decision to revert, the algorithm terminates and reports all found subsets.

This algorithm would at first decide to include every argument. Then it would report a first subset, which is $\mathcal{A}$. Every time it finds a solution it would backtrack and undo the last decision by excluding the corresponding argument from the solution. Then it would repeat either including arguments into the solution or backtracking until all possibilities have been considered. The application of this algorithm to a simple AAF is visualized as a tree structure in Figure 7. Every branch of that tree

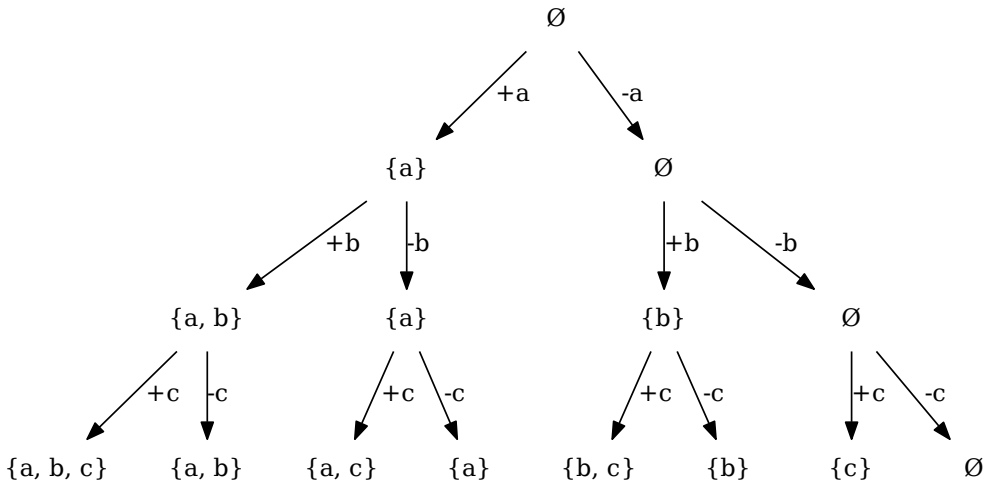


Figure 7: This tree visualizes a backtracking algorithm which enumerates all subsets of the argument set $\{a, b, c\}$. An edge labelled „+x“ represents a decision to include x in the solution, while „-x“ represents a decision to exclude x from the solution [22].

corresponds to one decision and every leaf or maximal path corresponds to one subset.

Example 3.5. As a next step, all conflict free subsets of an argument set shall be enumerated. This is achieved by Algorithm 3, which differs from Algorithm 2 only slightly. It additionally adds all elements of $\{a\}^{\pm}$ to $\mathrm{OUT}(\mathcal{L})$ when adding a to $\mathrm{IN}(\mathcal{L})$. This excludes exactly all subsets which contain conflicts. During every iteration the current partial solution is conflict-free as with the inclusion of an argument all of its possible conflicts are being excluded, and every single conflict-free subset is reached as all ignored solutions would contain conflicts because they each at least contain one argument and second that attacks it.

Example 3.6. As a subsequent example, we consider an algorithm which enumerates admissible subsets. We recall that an admissible set is a conflict-free set which defends all of its elements. Therefore we augment Algorithm 3 from example 3.5 with the following additional feature: Additionally when a conflict-free set is found the algorithm checks if it contains undefended arguments. Only if this is not the case the subset will be reported as an admissible subset. The application of this algorithm to a simple AAF is visualized as a tree structure in Figure 8. The algorithm finds three admissible subsets $\{a, c\}$, $\{a\}$ and $\emptyset$. The two other conflict free sets $\{b\}$ and $\{c\}$ contain undefended arguments and are therefore not admissible.

Because every admissible subset is conflict-free, this algorithm needs only to consider conflict-free subsets. In the AAF given in Figure 8, this reduces the number

Algorithm 3 Enumerate All Conflict-Free Subsets

Input: $\Gamma = (\mathcal{A}, \mathcal{R})$ AAF
$\mathcal{L} \in \mathfrak{L}(\Gamma)$ labelling function

Output: $\mathcal{S}_{CF} \subseteq \mathcal{P}(\mathcal{A})$ conflict-free sets

```
1: for all a ∈ A do
2:     L(a) ← UNDEC
3: FIND_CF(L)
4: procedure FIND_CF(L)
5:     if there is an argument a with L(a) = UNDEC then
6:         L' ← L
7:         L'(a) ← IN
8:         for all b ∈ {a}^± do
9:             L'(b) ← OUT
10:        FIND_CF(L')
11:        L(a) ← OUT
12:        FIND_CF(L)
13:    else
14:        add IN(L) to S_CF
```

of paths corresponding to a subset to five. A backtracking algorithm which would enumerate all subsets and then choose only admissible ones would consider eight paths. Further improvements would be possible if a path could be aborted when it becomes clear that one of the included arguments cannot be defended. The performance of a backtracking algorithm can thus be improved by disregarding fruitless branches which cannot lead to viable solutions.

3.3 Enumeration of Complete Extensions

The HEUREKA solver, that was implemented as part of this thesis, solves tasks under complete and preferred semantics with a single general depth-first backtracking algorithm. This includes enumerating all extensions, finding a single extension and sceptical as well as credulous justification. Problems with stable semantics use Algorithm 7 which extends this algorithm. All tasks can be deduced from enumeration. According to the underlying abstract argumentation problem, as soon as it has provided satisfactory information the algorithm is stopped and a result is reported depending on task and semantics. When for example tasked to enumerate all extensions, the algorithm runs until it has found all extensions and the set of all extensions is returned as a result. When searching a single extension, the algorithm is stopped as soon as it has found an extension and that extension is returned as the result. When solving a sceptical justification problem, the algorithm is stopped as soon as it has reported an extension which does not contain the argument which is

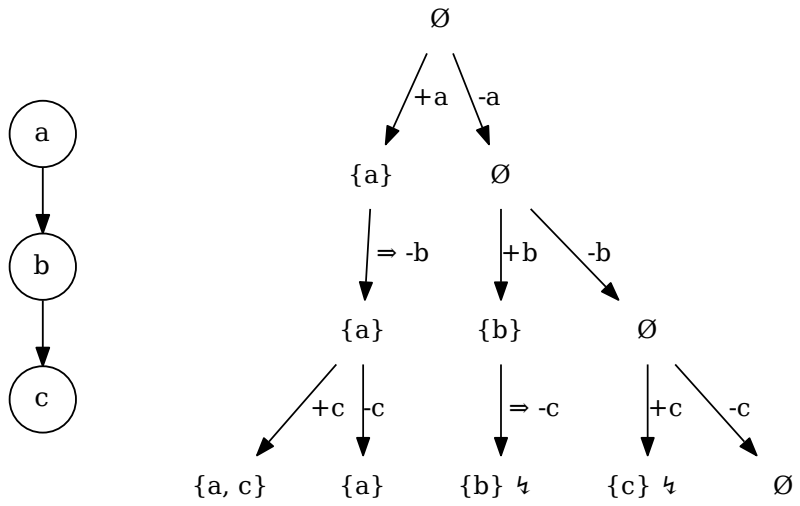


Figure 8: *Left:* The AAF used by the algorithm which is illustrated on the right. *Right:* This tree visualizes a backtracking algorithm which enumerates all admissible subsets of the argument set of the AAF given in the graph on the left. The algorithm is described in example 3.6. An edge labelled „+x“ represents a decision to include x in the solution, while „-x“ represents a decision to exclude x from the solution and „$\Rightarrow -$x“ means that x cannot be part of the solution due to a former decision.

to be justified and the argument is deemed not justified. If the algorithm terminates on the other hand without finding such an extension, the argument is justified. For a credulous justification problem, the algorithm is analogously stopped as soon as it has reported an extension which contains the argument in question and the argument is justified. If the algorithm terminates on the other hand without finding such an extension, the argument is not justified. Therefore a single algorithm can solve all relevant tasks without much overhead when guided correctly.

The general backtracking algorithm which enumerates complete extensions is given by Algorithm 5. It is here shown recursively for the sake of conciseness and readability. This algorithm relies on a heuristic to suggest at every step of algorithm execution which argument is handled next and thereby lead the algorithm through the search space. Many different heuristics will be introduced in Section 4 which also gives a definition of heuristics for abstract argumentation. For now a heuristic establishes an order over all arguments and the algorithm will then handle arguments in that order. To keep track of the arguments' state, the general backtracking algorithm employs a four-label mapping with the labels IN, OUT, UNDEC and BLANK. These labels correspond to the following argument states: An argument labelled IN is part of the current solution, an argument labelled OUT is defeated by the current solution and an argument labelled UNDEC attacks the current solution, and can therefore not be part of it, but is not yet defeated. Finally arguments la-

Algorithm 4 Prepare Initial Labelling for a Backtracking Algorithm

Input: $\Gamma = (\mathcal{A}, \mathcal{R})$ AAF
$E_{GR} \subseteq \mathcal{A}$ the grounded extension

Output: $\mathcal{L} \in \mathfrak{L}(\Lambda_4, \Gamma)$ initial labelling

1: **for all** $a \in \mathcal{A}$ **do**
2: $\quad \mathcal{L}(a) \leftarrow$ BLANK
3: **for all** $a \in E_{GR}$ **do**
4: $\quad$ SET_IN$(\mathcal{L}, a)$
5: **for all** $a \in \mathcal{A}$ **do**
6: $\quad$ **if** $a \rightarrow a$ **then**
7: $\quad\quad$ SET_UNDEC$(\mathcal{L}, a)$

belled BLANK fall into none of these categories. Before it can be executed, an initial labelling has to be computed by Algorithm 4. In combination with the specific functions SET_IN and SET_UNDEC for complete semantics from Algorithm 5 this leads to the following results: Every argument which is part of the grounded extension has to be part of every complete extension also and is therefore included in the initial labelling. Every argument which attacks the grounded extension or is attacked by it must on the other hand be outside of every extension. Arguments which attack themselves can be safely considered UNDEC because they cannot be defended and thus cannot be part of any extension. This leads to the labelling which is described in the following definition.

Definition 3.7. For an AAF Γ with the grounded extension E_{GR} let the **initial labelling** $\mathcal{L}_{init} \in \mathfrak{L}(\Lambda_4, \Gamma)$ be defined as

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

Whenever an argument is set to IN or UNDEC by Algorithm 5, consequences are drawn according to [22]. These consequences allow the algorithm to consider only branches which possibly contain extensions and eliminate other branches. Setting an argument to OUT and therefore defeating an argument does not require consequences to be drawn. When an argument a is set IN, these consequences include setting all arguments which attack a to UNDEC while all arguments attacked by a are set to OUT. Additionally if after the addition of a to the current partial solution $\text{IN}(\mathcal{L})$ all attackers of an argument c are attacked by the partial solution, c is set IN. When an argument a is set UNDEC and attacks the partial extension but it is not yet attacked by the partial extension while there is only one candidate c which attacks a

Algorithm 5 Enumerate All Complete Extensions

Input: $\Gamma = (\mathcal{A}, \mathcal{R})$ AAF
$\mathfrak{h}$ heuristic
$\mathcal{L}_{\text{init}} \in \mathfrak{L}(\Lambda_4, \Gamma)$ initial labelling

Output: $\mathcal{E}_{\text{CO}} \subseteq \mathcal{P}(\mathcal{A})$ complete extensions

```
1:  ENUMERATE_EXTENSIONS(L_init)
2:  function SET_IN(L, a)
3:      L(a) ← IN
4:      for all b ∈ {a}^- do
5:          if not SET_UNDEC(L, b) then
6:              return false
7:      for all b ∈ {a}^+ do
8:          L(b) ← OUT
9:      for all c ∈ ({a}^+)^+ do
10:         if {c}^- ⊆ IN(L)^+ then
11:             if L(c) = UNDEC then
12:                 return false
13:             else if not SET_IN(L, c) then
14:                 return false
15:     return true
16: function SET_UNDEC(L, a)
17:     L(a) ← UNDEC
18:     if a ∈ IN(L)^- and |{a}^- \ IN(L)^+| = 1 then
19:         find b ∈ {a}^- \ IN(L)^+
20:         if not SET_IN(L, b) then
21:             return false
22:     return true
23: procedure ENUMERATE_EXTENSIONS(L)
24:     let h choose next argument a
25:     if no argument is left then
26:         if L is complete then
27:             add IN(L) to E_CO
28:     else if L(a) = BLANK then
29:         L' ← L
30:         if SET_IN(L', a) then
31:             ENUMERATE_EXTENSIONS(L')
32:         if SET_UNDEC(L, a) then
33:             ENUMERATE_EXTENSIONS(L)
34:     else ENUMERATE_EXTENSIONS(L)
```

Algorithm 6 Decide Completeness

Input: $\Gamma = (\mathcal{A}, \mathcal{R})$ AAF
$\mathcal{L} \in \mathfrak{L}(\Gamma)$ terminal labelling

1: **if** $|\mathrm{IN}(\mathcal{L})| = |\mathfrak{F}(\mathrm{IN}(\mathcal{L}))|$ **and** $\mathrm{IN}(\mathcal{L})^- \subseteq \mathrm{IN}(\mathcal{L})^+$ **then**
2: **return** true
3: **else**
4: **return** false

and could be added to the extension, the algorithm tries to include c into the partial extension.

Algorithm 5 generally resumes including and excluding arguments from the partial solution by labelling arguments until it reaches a terminal labelling.

Definition 3.8. A labelling $\mathcal{L} \in \mathfrak{L}(\Lambda_4, \Gamma)$ is considered **terminal** when no argument is still labelled BLANK. Thus a labelling is terminal iff it lies within $\mathfrak{L}(\Gamma)$.

When reaching a terminal labelling Algorithm 6 is called to check whether the found solution is a complete extension. Afterwards the algorithm backtracks in every case. Considering the performance of Algorithm 6 which checks the completeness of a labelling, the following information is stored to support this process: For every argument the number of elements of the partial solution which attack that argument is stored in an array which is updated at every step of the algorithm. The same is done for the number of elements of the partial solution which are attacked by an argument. This allows the algorithm to check the completeness of a terminal labelling in linear time. This method has worked better than the more intuitive approach of letting counters keep track of the extension size and the number of defended arguments. Continuously updating the counters has been more expensive than checking the legality of all labels for terminal labels as these checks tend to fail after examining a few argument labels.

As a next step the correctness of the output of Algorithm 5 shall be proven, which also relies on Algorithms 4 and 6.

Definition 3.9. For an algorithm to be **correct**, the following two statements have to hold true about that algorithm

- it is **sound**, i.e. it does not return any wrong solutions
- it is **complete**, i.e. it returns all valid solutions

Let us begin to decide the first one of both statement.

Theorem 3.10. Algorithm 5 is sound, that is every argument set returned by Algorithm 5 is a valid extension under complete semantics.

Proof 3.11. As a reminder, an extension is complete iff it is admissible and contains all arguments which it defends, and an admissible set is one that defends all of its

elements. At every step during its execution the algorithm ensures the following property of the partial solution $E = IN(\mathcal{L})$

i) the partial solution E is conflict free

$$E \cap E^{\pm} = \emptyset$$

This holds at every step of the algorithm execution as every time an argument is included into the solution every argument which attacks it or is attacked by it is excluded from the solution.

To decide whether the found solution is a complete extension, when reaching a terminal labelling Algorithm 6 is called to check the following constraints

ii) the number of arguments which are labelled IN is equal to the number of defended arguments

$$|E| = |\mathfrak{F}(E)|$$

iii) there is no argument which attacks E but is not attacked by E

$$E^{-} \subseteq E^{+}$$

From constraints i) and iii) it follows that

iv) there is no element of E which is not defended by E

$$E \subseteq \mathfrak{F}(E)$$

because an element of the solution cannot be attacked from inside the solution as it is conflict-free due to i) and the solution can defend itself against all attacks from the outside due to iii). If we now combine constraints ii) and iv) we can conclude that

v) an element is defended by E iff it is part of E

$$E = \mathfrak{F}(E)$$

Because every element of E is also an element of $\mathfrak{F}(E)$ and E has as many elements as $\mathfrak{F}(E)$ it follows that every element of $\mathfrak{F}(E)$ is also an element of E and therefore both sets are equal. Therefore Algorithm 5 is sound, i.e. it only yields complete extensions, as an extension E is complete iff it holds that $E = \mathfrak{F}(E)$. So every extension accepted by the algorithm is admissible because it is conflict-free (i) and defends all of its elements (iv). Because it contains all elements which it defends (v), it is also complete.

The soundness of Algorithm 5 can alternatively be derived from the labelling based definition of complete semantics.

Proof 3.12. From Definition 2.30 we remember that a labelling $\mathcal{L}$ is complete iff for every argument a it holds that

i) $\mathcal{L}(a) = \text{IN}$ iff all of its attackers are labelled OUT

ii) $\mathcal{L}(a) = \text{OUT}$ iff there is an argument b with $\mathcal{L}(b) = \text{IN}$ which attacks a

Every labelling produced at any step by Algorithm 5 fulfils condition ii) because every arguments is labelled OUT when one of its attackers is set IN and this is the only occasion that an argument is set OUT. Condition i) is tested by Algorithm 6 every time a terminal labelling is reached.

After showing its soundness, to ensure the correctness of Algorithm 5, its completeness has to be derived.

Theorem 3.13. Algorithm 5 is complete, that is it enumerates all complete arguments.

Proof 3.14. The algorithm basically enumerates all subsets of the argument set, but excludes certain subsets. A subset is excluded either from the start due to a label assigned to an argument by the initial labelling $\mathcal{L}_{\text{init}}$ or whenever an argument is assigned a label as consequence of a decision concerning another argument. The initial labelling which is constructed by Algorithm 4 assigns a label other than BLANK to an argument in the following cases:

- $\mathcal{L}_{\text{init}}(a) = \text{IN}$ iff a is part of the grounded extension, all extensions which do not contain a are thereby avoided, which is correct as the grounded extension is part of every complete extension
- $\mathcal{L}_{\text{init}}(a) = \text{OUT}$ iff a is attacked by the grounded extension, all extensions which contain a are thereby avoided, which is correct as the grounded extension is part of every complete extension
- $\mathcal{L}_{\text{init}}(a) = \text{UNDEC}$ iff a attacks itself, all extensions which contain a are thereby avoided, which is correct as any extension including a would contain inner conflicts

Therefore all argument sets excluded by the initial labelling cannot be complete extensions. When an argument a is included, consequences can be drawn in the following lines of the algorithm:

5: arguments that attack a are excluded from the extension

8: arguments that are attacked by a are excluded from the extension

13: an argument is included when all of its attackers have been defeated by the partial solution

20: an argument is included when it becomes the only argument which could possibly defeat an argument that attacks the partial solution

All extensions that the algorithm prevents as a consequence of line 5 or 8 contain conflict and can be savely avoided, as they would contain conflicts. Line 13 prevents an extension where an argument is defended by the extension but not included in it, this extension can thus not be complete. Lastly if an extension is not reached by the algorithm as a consequence of line 20, it cannot defend itself and therefore not be complete either. Therefore Algorithm 5 reaches all complete extensions and is complete. The algorithm is correct, as it is sound and complete.

3.4 Enumeration of Preferred Extensions

Algorithm 5 can be reused to solve problems under preferred semantics. A preferred extension E is maximally complete, i.e. E is complete and there is no complete extension E', such that $E \subset E'$. Therefore the set of preferred extensions can be enumerated by the same algorithm that enumerates complete extensions with an additional maximality criterion, which filters out extensions which are part of other extensions. All problems under preferred semantics are thus equal to the corresponding problem with complete semantics when the reported extensions are additionally filtered in such a way that only maximally complete extensions are reported.

$$\mathcal{E}_{PR} = \{E \in \mathcal{E}_{CO} \mid \nexists E' \in \mathcal{E}_{CO} : E \subset E'\}$$

Such a maximality filter can be easily implemented as the complete enumeration algorithm will try to include every argument before considering extensions which exclude that argument. Therefore if extension E' includes extension E, E' will always be found before E, and thus the algorithm only has to compare a newly found extension to preferred extensions which have previously been found.

3.5 Enumeration of Stable Extensions

Because the process of enumerating stable extensions has several significant differences from the enumeration of complete extensions, a second backtracking algorithm based on Algorithm 5 is developed to fulfil this task. Algorithm 7 which is specifically designed to enumerate stable extensions differs from the aforementioned algorithm in two points: the function SET_UNDEC uses different criteria for consequences and after every execution of the function SET_IN the algorithm checks whether the current partial extension is stable. Algorithm 7 supports the same tasks as the complete enumeration algorithm and performs them in roughly the same way.

Like Algorithm 5, Algorithm 7 relies on a heuristic to to guide it through the search space. Before the execution of Algorithm 7 again Algorithm 4 is called to compute the initial labelling which is described in Definition 3.7. Although different functions SET_IN and SET_UNDEC are used, the initial labelling is equal to

Algorithm 7 Enumerate All Stable Extensions

Input: $\Gamma = (\mathcal{A}, \mathcal{R})$ AAF
$\mathfrak{h}$ heuristic
$\mathcal{L}_{\text{init}} \in \mathfrak{L}(\Lambda_4, \Gamma)$ initial labelling

Output: $\mathcal{E}_{ST} \subseteq \mathcal{P}(\mathcal{A})$ stable extensions

ENUMERATE_EXTENSIONS($\mathcal{L}_{\text{init}}$)

function SET_IN($\mathcal{L}$, a)
    $\mathcal{L}(a) \leftarrow$ IN
    for all $b \in \{a\}^-$ **do**
        if not SET_UNDEC($\mathcal{L}$, b) **then**
            return false
    for all $b \in \{a\}^+$ **do**
        $\mathcal{L}(b) \leftarrow$ OUT
    for all $c \in (\{a\}^+)^+$ **do**
        if $\{c\}^- \subseteq \text{IN}(\mathcal{L})^+$ **then**
            if $\mathcal{L}(c) =$ UNDEC **then**
                return false
            else if not SET_IN($\mathcal{L}$, c) **then**
                return false
    if IS_STABLE($\mathcal{L}$) **then**
        add $\text{IN}(\mathcal{L})$ to $\mathcal{E}_{ST}$
        return false
    else return true

function SET_UNDEC($\mathcal{L}$, a)
    $\mathcal{L}(a) \leftarrow$ UNDEC
    if $|\{a\}^- \setminus \text{IN}(\mathcal{L})^+| = 1$ **then**
        find $b \in \{a\}^- \setminus \text{IN}(\mathcal{L})^+$
        if not SET_IN($\mathcal{L}$, b) **then**
            return false
    return true

procedure ENUMERATE_EXTENSIONS($\mathcal{L}$)
    let $\mathfrak{h}$ choose next argument a, if there is none, **stop**
    if $\mathcal{L}(a) =$ BLANK **then**
        $\mathcal{L}' \leftarrow \mathcal{L}$
        if SET_IN($\mathcal{L}'$, a) **then**
            ENUMERATE_EXTENSIONS($\mathcal{L}'$)
        if SET_UNDEC($\mathcal{L}$, a) **then**
            ENUMERATE_EXTENSIONS($\mathcal{L}$)
    else ENUMERATE_EXTENSIONS($\mathcal{L}$)

Algorithm 8 Decide Stableness

Input: $\Gamma = (\mathcal{A}, \mathcal{R})$ AAF
$\mathcal{L} \in \mathfrak{L}(\Lambda_4, \Gamma)$ labelling function

1: **if** $\mathrm{IN}(\mathcal{L}) \subseteq \mathfrak{F}(\mathrm{IN}(\mathcal{L}))$ **and** $\mathcal{A} \setminus \mathrm{IN}(\mathcal{L}) \subseteq \mathrm{IN}(\mathcal{L})^+$ **then**
2: **return** true
3: **else**
4: **return** false

the one computed for the complete enumeration algorithm, as none of the specific differences between those functions in both algorithms apply in this initial stage. The implementation of the function SET_UNDEC from the stable enumeration Algorithm 7 checks whether there is only one possible attacker for every argument it sets UNDEC, as a stable extension has to defeat all remaining arguments [22]. It then tries to include the single attacker into the partial solution and backtracks if it fails. The corresponding function in the complete enumeration Algorithm 5 in contrast does this only if the argument which is set to UNDEC attacks the partial solution.

To increase performance, Algorithm 7 calls Algorithm 8 to check whether the current partial extension is stable every time it adds a new argument to the partial extension. This is a great difference to Algorithm 5 which includes as many arguments as possible before checking whether the currently found solution is a valid extension. Stability can be checked in linear time in a similar way to completeness by storing the relevant information in vectors which are continuously updated. This procedure increases the performance of the stable algorithm as all stable extensions are maximal and the algorithm can therefore immediately backtrack after finding an extension. Let the following example show the application of Algorithm 7 on a simple AAF.

Example 3.15. Consider the AAF Γ depicted in Figure 10. We want to demonstrate the execution of Algorithm 7 and thereby find one stable extension. Assume a heuristic h that determines the following order of arguments and is used in combination with the algorithm.

$$a >_h b >_h c >_h d >_h e$$

As a first step, we determine that the grounded extension is empty and that there is no self-attacking argument, so we start with an empty labelling, where all arguments are blank. Algorithm 7 then proceeds in the following way:

1. *decision:* a is picked and set IN as it is the blank argument with the highest heuristic priority
2. as a consequence of step 1, all attackees of a are set to OUT while all attackers of a are set to UNDEC. The resulting solution $\{a\}$ is not stable, therefore the algorithm proceeds

step	labelling				
	a	**b**	**c**	**d**	**e**
1.	IN	-	-	-	-
2.	IN	UNDEC	-	-	-
3.	IN	UNDEC	IN	-	-
4.	IN	OUT	IN	OUT	UNDEC
5.	IN	UNDEC	UNDEC	-	-
6.	IN	UNDEC	UNDEC	IN	-
7.	IN	OUT	OUT	IN	OUT

Figure 9: The labelling of all arguments in Γ from Example 3.15 after every step of the execution of the stable enumeration algorithm. Here "-" stands for BLANK.

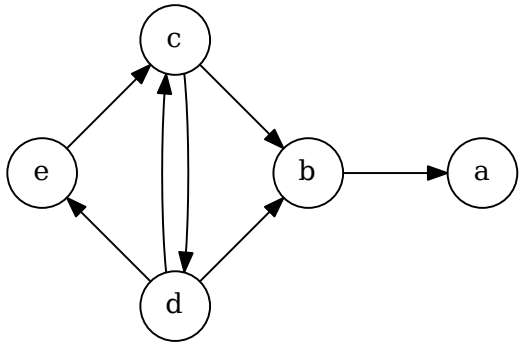


Figure 10: The AAF Γ used in Example 3.15

3. *decision:* c is picked and set IN

4. as a consequence of step 3, all attackees of c are set to OUT while all attackers of c are set to UNDEC, $\{a, c\}$ is not stable

5. there are no more arguments which are still blank, so the algorithm backtracks to the last decision which happened in step 3. It reverts all changes from step 3 on and then sets c to UNDEC

6. *decision:* d is picked and set IN

7. as a consequence of step 6, all attackees of d are set to OUT while all attackers of d are set to UNDEC, $\{a, d\}$ is stable as $\text{UNDEC}(\mathcal{L}) \cup \text{BLANK}(\mathcal{L}) = \emptyset$, therefore the algorithm stops as it has been tasked to find a single stable extension

The algorithm reports the found solution $\{a, d\}$ as a stable extension.

Lastly the correctness of the output of Algorithm 7 has to be proven, which also relies on Algorithms 4 and 8. Please recall Definition 3.9 about the correctness of an algorithm.

Theorem 3.16. Algorithm 7 is sound, that is every argument set returned by Algorithm 7 is a valid extension under stable semantics.

As a reminder concerning the soundness of the algorithm, an extension is stable iff it is admissible and attacks every argument which is not part of it. Therefore every stable extension is maximal, i.e. it cannot be a subset of another stable extension. Because it attacks all remaining arguments, no additional argument could be included.

Proof 3.17. At every step of the execution of Algorithm 7 it holds that

i) the partial solution $E = \text{IN}(\mathcal{L})$ is conflict free

$$E \cap E^{\pm} = \emptyset$$

This constraint is ensured by the fact that every time an argument is added to the solution, all of its attackers and attackees are excluded.

Every time a new element is included into the solution, Algorithm 8 is called to check whether the following two constraints hold.

ii) there is no element of E which is not defended by E

$$E \subseteq \mathfrak{F}(E)$$

iii) every argument which is not part of E is attacked by E

$$\mathcal{A} \setminus E(\mathcal{L}) \subseteq E^{+}$$

The partial solution E is reported as a stable extension exactly if those two statements hold. So every extension accepted by Algorithm 7 is admissible because it is conflict-free (i) and defends all of its elements (ii). Because it attacks all remaining elements (iii), it is also stable, as stability of an extension follows out of admissibility and the defeat of every element of its complement. Algorithm 7 is therefore sound as it only returns stable extensions.

Theorem 3.18. Algorithm 7 is complete, that is it enumerates all valid extensions under stable semantics.

Proof 3.19. Regarding the completeness of the algorithm three aspects have to be considered. First, when it has found a stable extension, Algorithm 7 backtracks, disregarding all supersets of the newly found extension. After finding a stable extension, the algorithm can safely backtrack because further search on the current branch could only yield extensions which include the current extension. Such an extension

cannot exist as a stable extension attacks all remaining arguments, so nothing could be further included in it. This aspect does therefore not affect the completeness of the algorithm. Second the initial labelling which is again constructed by Algorithm 4 excludes certain argument sets. The proof that these argument sets cannot be stable extensions works analogously to the proof for complete extensions given in Proof 3.14. Third we have to analyse all subsets which are excluded due to consequences to the labelling which are drawn as a result of a decision regarding another argument. These consequences occur in the following lines:

5: arguments that attack a are excluded from the extension

8: arguments that are attacked by a are excluded from the extension

13: an argument is included when all of its attackers have been defeated by the partial solution

23: an argument is included when it becomes the only argument which could possibly defeat an argument outside the partial solution

All extensions that the algorithm prevents as a consequence of line 5 or 8 contain conflicting arguments and can be savely avoided, as they would contain conflicts. Line 13 prevents an extension where an argument is defended by the extension but not included in it, this extension can thus not be stable. Lastly if an extension is not reached by the algorithm as a consequence of line 23, it cannot defeat all remaining arguments and therefore not be stable either. Therefore Algorithm 7 reaches all stable extensions and is complete.

Algorithm 7 is therefore correct, as it is sound and complete. Three algorithms which have been defined in the section are sufficient to solve all abstract argumentation problems which have been described in Section 2. One of these algorithms finds the grounded extension, one enumerates all complete extensions and one enumerates all stable extension. All other problems can be derived with slight tweeky from these primary tasks. The next section will define several heuristics to be used with the latter two of those algorithms. The performance of the algorithm will be measured and compared to other methods in Section 6.

4 Heuristics

The different backtracking algorithms which were introduced in Section 3 utilise heuristics to navigate the decision space more efficient and thereby reduce their runtime. This section will first give a definition of heuristics for the purpose of abstract argumentation and then define several heuristics which are used in HEUREKA, while the general implementation of heuristics in HEUREKA is described in Section 5. Then in Section 5.3, the possibilities and the notation of combining heuristics in HEUREKA are explained. Finally in Section 6.2 experiments will be conducted to compare the performance of different heuristics.

4.1 Static and Dynamic Heuristics

A heuristic is a method to infer satisfactory information in situations of incomplete knowledge. For the purpose of this thesis let a heuristic be defined in the following way.

Definition 4.1. A **heuristic** is a function $h : \mathcal{P}(\mathcal{A}) \times \mathcal{A} \to \mathbb{R}$ which maps an argument $a \in \mathcal{A}$ to a numerical value $h(E, a)$. Some heuristics also consider the current partial solution $E \in \mathcal{P}(\mathcal{A})$, those are called **dynamic heuristics**. Other heuristics which do not rely on E are called **static heuristics**.

Any heuristics will give precedence to the argument which is assigned the highest values $h(E, a)$ and this argument will thus be handled first by the algorithm. Afterwards the other arguments follow according to their heuristical values in descending order. All heuristics, static and dynamic, analyse the structure of the defeat network between arguments, as this is the only information available in abstract argumentation. In this section several classes of static as well as dynamic heuristics will be defined. Generally it is considered beneficial if heuristics choose arguments that increase the potential effectiveness of the extension and avoid those that increase its vulnerability. We consider a set of arguments effective if it defeats many arguments, while undefeated attackers make a set of arguments vulnerable.

While static heuristics are only computed once before the invocation of the algorithm, dynamic heuristics need to be recomputed every time an algorithm has to pick an argument to be handled next. As this is very cost-intensive it should not be done for all arguments at once. When using a dynamic heuristic, arguments should be preordered by a static heuristic once at the beginning. For any dynamic heuristic a window size k has to be specified and when invoked, the dynamic heuristic will compare the k unhandled arguments which are most promising according to the static heuristic. When the argument set has for example been presorted with an SCC-based heuristic (see Section 4.6), it would not make sense for the dynamic heuristic to frequently mix the different strongly connected components again, and the window size should therefore not be greater than the size of an average component.

The algorithms presented by Nofal et al. in [22] will always give precedence to the unhandled, that is blank, argument with most neighbours, i.e.ãttackers and attackees. Nofal et al. call such an argument influential. This method could be interpreted as a static heuristic h_{inf}. Heuristics similar to this heuristic will be presented in Section 4.2, where degree-based heuristics are defined.

$$h_{inf}(E, a) = \left|\{a\}^{\pm}\right| = deg^{-}(a) + deg^{+}(a)$$

In the remaining parts of this section, various heuristics will be defined, all of which rely on analysing different aspects of the graph structure of an AAF.

4.2 Degree-Based Heuristics

In this section several heuristics will be defined which analyse the direct neighbourhood of an argument, that is its attackers and attackees, as direct attacks are probably the most influential relations between arguments. Direct neighbours of an argument a are arguments which either attack a or are attacked by a.

Definition 4.2. The number of direct neighbours of an argument a is called the **degree** of a which is denoted as $\deg(a)$. The **indegree** $\deg^-(a)$ is equal to the number of attackers of a while the **outdegree** $\deg^+(a)$ is equal to the number of attackees of a.

$$\deg^-(a) = \left|\{a\}^-\right|$$
$$\deg^+(a) = \left|\{a\}^+\right|$$

Either $\deg^+$ or $\deg^-$ as well as combinations thereof can be used as static heuristics. For example indegree and outdegree can be combined by addition. h^{Σ}_{deg} is the sum of indegree and outdegree weighted with the parameters β and γ. This heuristic could be useful as high-degree nodes are generally more influential than low-degree nodes as every out- and ingoing attack potentially causes another argument to be excluded from the extension. Further arguments with many outgoing attacks could help to secure the extension against outside attacks by defeating attackers while arguments with many ingoing nodes make the extension more vulnerable to outside attacks and therefore could allow the algorithm to backtrack earlier.

$$h^{\Sigma}_{deg}(E, a) = \beta \deg^-(a) + \gamma \deg^+(a)$$

But the inclusion of arguments with a high indegree could also have a negative effect on the algorithm efficiency as it would guide away the algorithm from valid extensions. Thus indegree and outdegree are alternatively combined in such a way that the indegree is weighted negatively. The degree difference heuristic h^{Δ}_{deg} can be derived from the degree sum heuristic h^{Σ}_{deg} by assigning a negative value to the parameter β.

$$h^{\Delta}_{deg}(E, a) = \gamma \deg^+(a) - \beta \deg^-(a)$$

The degree product heuristic h^{Π}_{deg} combines indegree and outdegree in a similar way to the degree sum heuristic h^{Σ}_{deg}. Here high indegrees and outdegrees reinforce each other. A parameter $\epsilon \in \mathbb{R}^+$ is added to both values to allow the differentiation between arguments where either indegree or outdegree is 0. To understand the rationale behind this, compare Example 4.3, which illustrates an analogous effect which occurs for the division of indegree and outdegree instead of their multiplication.

$$h^{\Pi}_{deg}(E, a) = (\deg^-(a) + \epsilon) \cdot (\deg^+(a) + \epsilon)$$

Another way to combine indegree and outdegree is to compute their ratio. This could be a valuable heuristic as it shows how much new arguments could be possibly defeated by the addition of an argument to the partial solution in relation to

this arguments vulnerability to attackers. This idea is implemented by the degree ratio heuristic $h^{\div}_{deg}$ which maps an argument to the ratio between its outdegree and its indegree. A value $\epsilon > 0$ is added to both dividend and divisor of the outdegree-to-indegree-ratio. Therefore arguments with either an indegree or an outdegree of zero are still distinguishable, as is shown in Example 4.3. Additionally this helps to avoid division by zero, as the sum of the indegree $deg^-(a) \geq 0$ and $\epsilon > 0$ is always greater zero for any argument a.

$$h^{\div}_{deg}(E, a) = \frac{deg^+(a) + \epsilon}{deg^-(a) + \epsilon}$$

Example 4.3. Imagine an argument a with $deg^-(a) = 1$ and $deg^+(a) = 0$ as well as an argument b with $deg^-(b) = 2$ and $deg^+(b) = 0$. If ϵ would be set to 0, the degree ration heuristic would assign the same value 0 to both a and b. Therefore no preference could be established among them.

$$\frac{deg^+(a)}{deg^-(a)} = 0 = \frac{deg^+(b)}{deg^-(b)}$$

If $\epsilon = 1$ would be chosen, the heuristic would prefer a over b because the degree ratio would be greater for that argument.

$$\frac{deg^+(a) + 1}{deg^-(a) + 1} = \frac{1}{2} > \frac{deg^+(b) + 1}{deg^-(b) + 1} = \frac{1}{3}$$

It makes sense to prefer a over b as the additional attack leaves b more vulnerable compared to a.

This idea can be extended by considering the degree ratio of all of an argument's neighbours, as for example the defeat of an argument with many outgoing but few ingoing paths is especially beneficial. Let the resulting heuristic be called the second-order degree ratio heuristic $h^{\div *}_{deg}$. Again a value ϵ is added to each sum to distinguish between arguments with no ingoing or outgoing attacks and avoid division by zero.

$$h^{\div *}_{deg}(E, a) = \frac{\sum_{b \in \{a\}^+} h^{\div}_{deg}(E, b) + \epsilon}{\sum_{b \in \{a\}^-} h^{\div}_{deg}(E, b) + \epsilon}$$

A dynamic heuristic which can be derived from degree is based on aggressors. When adding an argument to a partial solution, any argument which attacks that argument but is not defeated by the partial solution is considered an aggressor. Let the aggressor count heuristic h_{aggr} be defined as the number of undefeated aggressors. This heuristic could be useful as it approximates how much vulnerability could potentially be added to the partial solution. In HEUREKA this is implemented in way, such that arguments are first pre-sorted with help of a static heuristic and afterwards

every time the heuristic is invoked, it only compares the k most promising, but unhandeled candidates. The window size k is a specific parameter for the aggressor heuristic. This heuristic should contribute negatively to a combined heuristic.

$$h_{aggr}(E, a) = \left|\{a\}^{-} \setminus E^{+}\right|$$

Similar to the aggressor count heuristic h_{aggr} is the defensor count heuristic h_{defor}. It counts how many of the undefeated aggressors of the current partial solution an argument defeats. Again the argument should be presorted before the execution of the algorithm by a static heuristic. When invoked by the algorithm, the defensor count heuristic then selects the most promising of the next k arguments where k is the specified window size.

$$h_{defor}(E, a) = \left|\{a\}^{+} \cap (E^{-} \setminus E^{+})\right|$$

4.3 Path-Based Heuristics

Path-based heuristics rely on the number of paths starting or ending in an argument. In a graph, a path or walk is a sequence of edges where every edge starts in the same node where its predecessor ends.

Definition 4.4. In an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$, a sequence $\pi \in \mathcal{R}^n$ of n attacks is called a **path** iff it holds that $\pi_{i-1} = (a, b)$ and $\pi_i = (b, c)$ for every $i = 2, \ldots, n$ and $a, b, c \in \mathcal{A}$. It is then said that π is a path of length n. An argument can occur repeatedly in a path, i.e. a path can run through the same argument several times.

To simplify its denotation, a path of length n is written as a sequence of the $n+1$ arguments it links. For example $((a, b), (b, c)) \in \mathcal{R}^n$ is given as $(a, b, c) \in \mathcal{A}^{n+1}$, or alternatively shortened to $a \to b \to c$.

Definition 4.5. Let Π_n be the set of paths of length $n > 0$ in an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ and let Π be the set of all paths regardless of their length in Γ.

$$\Pi_n = \{\pi \in \mathcal{A}^{n+1} \mid \pi_i \to \pi_{i+1} \text{ for } i = 1 \ldots n\}$$

$$\Pi = \bigcup_{i \in \mathbb{N}} \Pi_i$$

Definition 4.6. Let the argument $\alpha(\pi) = \pi_1$ be the start node and $\omega(\pi) = \pi_{n+1}$ be the end node of a path $\pi \in \Pi_n$ of length $n \in \mathbb{N}$.

Definition 4.7. For any number $n > 0$, let $\delta_n^+(a)$ be the set of paths of length n starting in an argument a and $\delta_n^-(a)$ be the set of paths of length n ending in a.

$$\delta_n^+(a) = \{\pi \in \Pi_n \mid \alpha(\pi) = a\}$$

$$\delta_n^-(a) = \{\pi \in \Pi_n \mid \omega(\pi) = a\}$$

Now $d_n^+(a)$ which is the number of paths of length n starting in an argument a and $d_n^-(a)$ which is the number of paths of length n ending in a can be defined.

$$d_n^+(a) = |\delta_n^+(a)|$$

$$d_n^-(a) = |\delta_n^-(a)|$$

The value $d_1^+(a)$ is thus equal to the outdegree of a and $d_1^-(a)$ to the indegree of a. As a direct attack constitutes a path of length one, the different degree-based heuristics which have been explored in Section 4.2 could be interpreted as path-based heuristics, which consider only paths of length one.

Let us first focus on outgoing paths, i.e. paths that start in a specific argument, and which influence they have on the eligibility of that argument. An argument either attacks or supports another argument through every outgoing paths. Consider a path π which starts in the argument $\alpha(\pi)$ and ends in the argument $\omega(\pi)$. If π is of length one, $\alpha(\pi)$ directly attacks $\omega(\pi)$. For longer paths we have to distinguish two cases: In the first case $\alpha(\pi)$ directly or indirectly attacks one of the attacker of $\omega(\pi)$, $\alpha(\pi)$ thus supports $\omega(\pi)$. The second case is that $\alpha(\pi)$ supports an attacker of $\omega(\pi)$ and thus indirectly attacks $\omega(\pi)$. The addition of arguments with many outgoing paths therefore increase the potential of the extension to defeat and defend arguments. Both cases are beneficial as the defeat of an argument could either eliminate an aggressor or make the extension impossible to defend when the defeated argument is part of the extension. The algorithm could then backtrack earlier. Defending an argument is beneficial as this argument can then be safely considered part of the extension. A heuristic which is purely based on outgoing paths is defined first to profit from these advantages. It combines the number of outgoing paths of different lengths with different weights. The heuristic h_{path}^+ relies on two parameters, the depth $k \in \mathbb{N}$ and the weight $\gamma \in [0, 1]$. The parameter k limits the maximal length of paths which contribute to the score while γ allows direct attacks and short paths to be weighted higher than longer paths.

$$h_{path}^+(E, a) = \sum_{i=1}^{k} \gamma^i d_i^+(a)$$

Ingoing paths on the other hand determine the vulnerability of an argument, and could therefore be used to define a heuristic which chooses an argument with minimal vulnerability. In a heuristic based on outgoing paths, all paths can be weighted positively as they represent either direct attacks, support relations or support of attackers. To define a heuristic based on ingoing paths on the other hand one has to further discriminate between paths of odd length on one hand and paths of even length on the other hand. A path π of length one between two arguments $\alpha(\pi)$ and $\omega(\pi)$ is a direct attack from $\alpha(\pi)$ to $\omega(\pi)$. Paths of length one should therefore contribute negatively to the heuristic score of $\omega(\pi)$. A path π of length two on the other hand is an attack on an attacker of $\omega(\pi)$ and therefore helps to defeat an attacker of

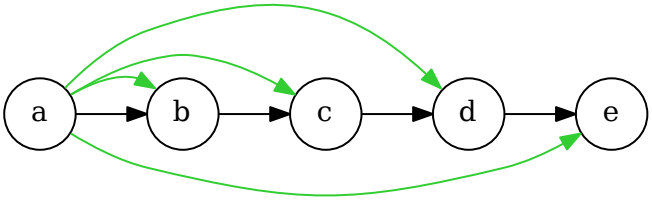


Figure 11: This graph visualises how outgoing paths of various lengths affect the eligibility of the argument a. Attacks are shown in black, and paths which count positively in green.

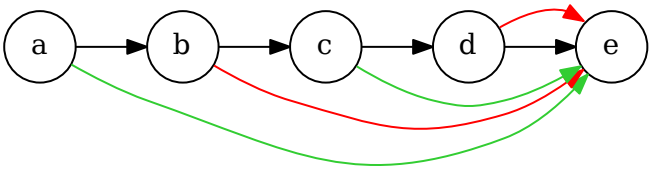


Figure 12: This graph visualises how ingoing paths of various lengths affect the eligibility of the argument e. Attacks are shown in black, paths which count positively in green, and paths which count negatively in red.

$\omega(\pi)$ and decreases the vulnerability of $\omega(\pi)$. It should thus contribute positively to the heuristic score of $\omega(\pi)$. This notion can be generalised to all paths of odd and even length. Every path π with a length greater than one contains a subpath between $\alpha(\pi)$ and an attacker of $\omega(\pi)$. When this subpath constitutes a direct or indirect attack, π supports $\omega(\pi)$, and when this subpath constitutes a support relation, π indirectly attacks $\omega(\pi)$. Therefore support and attack relations alternate with path length. Even paths constitute support for their end node while odd paths help to defeat that argument. From this it becomes apparent that ingoing paths should be weighted positively if they are of even length and negatively if they are of odd length as every ingoing path of even length decreases the vulnerability of an argument while an ingoing path of odd length increases its vulnerability [11].

Example 4.8. Without any loss of generality, let us explain the difference between in- and outgoing paths with a simple example. In an AAF let there be five arguments a, b, c, d and e, such that there is a path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$. Let us first consider paths which start in a and how they affect the eligibility of a. The outgoing paths of a are visualised in Figure 11. Through these paths, a attacks b and by doing so supports c. The support for c makes d, which is attacked by c, more vulnerable, this in turn supports e. All of which help to either defeat or defend arguments and thereby help to either exclude arguments from the partial solution or stabilize it by

defending elements of the partial solution. As both possible outcomes are desired, all outgoing paths can be weighted positively.

Let us now consider path ending in e and how they affect the eligibility of e. The ingoing paths of e are visualised in Figure 12. Here we have to discriminate between path of odd length and paths of even length. d attacks e and thereby increases its vulnerability. An ingoing path of length one should therefore decrease eligibility. The same holds true for paths of length three, as b supports d, which attacks e, by attacking its attacker c. c supports e as it attacks its attacker d. An ingoing path of length two should therefore increase eligibility, as should a path of length four as a it supports a supporter of e. Analogously this can be extender to longer odd and even paths respectively. Thus we conclude that ingoing paths of even length should contribute positively to an arguments heuristic score, while ingoing paths of odd length should contribute negatively.

As an additional heuristic based on ingoing paths, h^-_{path} is now defined. Here $k \in \mathbb{N}$ is again the depth and the weight $\beta \in \mathbb{R}$ should lie between -1 and 0, so odd paths subtract from the score.

$$h^-_{path}(E, a) = \sum_{i=1}^{k} \beta^i d_i^-(a)$$

To develop heuristics which exploit the available information effectively, heuristics based on in- and outgoing pths are combined with arithmetic operations analogously to indegree and outdegree. As a combination of both simple path-based heuristics, $h^\Sigma_{path} = h^-_{path} + h^+_{path}$ and $h^\Delta_{path} = h^+_{path} - h^-_{path}$ consider ingoing as well as outgoing paths.

$$h^\Sigma_{path}(E, a) = \sum_{i=1}^{k} \gamma^i d_i^+(a) + \sum_{i=1}^{k} \beta^i d_i^-(a)$$

$$h^\Delta_{path}(E, a) = \sum_{i=1}^{k} \gamma^i d_i^+(a) - \sum_{i=1}^{k} \beta^i d_i^-(a)$$

As β is generally chosen negatively, h^Σ_{path} already tends to give a negative weight to ingoing paths. In contrast to the corresponding degree-based heuristics, the path difference heuristic h^Δ_{path} therefore gives a positive weight to ingoing paths.

Further a path product heuristic $h^\Pi_{path} = (h^-_{path} + \epsilon) \cdot (h^+_{path} + \epsilon)$ is developed analogously to the degree product heuristic h^Π_{deg}. The product is multiplied with (-1) as h^-_{path} tends to yield negative results for the majority of arguments.

$$h^\Pi_{path}(E, a) = (h^+_{path}(E, a) + \epsilon) \cdot (h^-_{path}(E, a) + \epsilon) \cdot (-1)$$

Because both heuristics based on paths as well as heuristics based on node degree are promising approaches, heuristics are developed which combine both ideas.

The outpath-to-indegree-ratio heuristic $h^{\div}_{path}$ computes an argument's score by dividing a static heuristic which combines outgoing paths of different lengths through the argument's indegree. This is equal to the simple degree ratio heuristic $h^{\div}_{deg}$ with the outdegree replaced with the path-based heuristic component. Here indegree way chosen over an ingoing-path-based component, as those components yield scores with unpredictable sign which could leave to undesirable results, like division through zero in the worst case. Dividend and divisor are again increased by values ϵ and δ respectively for the reasons explained in Example 4.3. The rationale behind the outpath-to-indegree-ratio heuristic is to improve upon the degree ratio heuristic $h^{\div}_{deg}$ by giving more weight to attacks on nodes if those in turn have many outgoing attacks.

$$h^{\div}_{path}(E, a) = \frac{h^{+}_{path}(E, a) + \epsilon}{deg^{-}(a) + \delta}$$

Based on the static outpath-to-indegree-ratio heuristic $h^{\div}_{path}$, a dynamic heuristic is developed, which only consider ingoing attacks from undefeated arguments. The outpath-to-aggressor-count-ratio heuristic $h^{\div}_{aggr}$ is computed by dividing a heuristic based on outgoing paths through the aggressor count heuristic h_{aggr}. This is thought to improve upon the outpath-to-indegree-ratio heuristic $h^{\div}_{path}$ because the number of its undefeated aggressors reflects the vulnerability of an argument more accurately than its indegree.

$$h^{\div}_{aggr}(E, a) = \frac{h^{+}_{path}(E, a) + \epsilon}{h_{aggr}(E, a) + \delta}$$

4.4 Matrix Exponential

The paths inside an AAF can alternatively be computed as the matrix exponential of the adjacency matrix. Additional static and dynamic heuristics will rely on that matrix exponential.

Definition 4.9. An AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ can be represented as its **adjacency matrix** $A \in \{0, 1\}^{n \times n}$ with $n = |\mathcal{A}|$. Without any loss of generality, we assume an ordering over all arguments, which can then be denoted as $a_1, a_2, \ldots, a_n$. An entry a_{ij} of the adjacency matrix is 1 if a_i attacks a_j and otherwise 0.

$$a_{ij} = \begin{cases} 1 & \text{if } a_i \rightarrow a_j \\ 0 & \text{otherwise} \end{cases}$$

The adjacency matrix A represents the number of paths with length one between every pair of arguments a_i and a_j in the attack graph as $a_{ij} = 1$ iff a_i attacks a_j. The squared adjacency matrix A^2 contains the number of all paths with length two in the attack graph.

$$(A^2)_{ij} = \sum_{k=1}^{n} a_{ik} \cdot a_{kj} = |\{a_k \in \mathcal{A} \mid a_i \rightarrow a_k \text{ and } a_k \rightarrow a_j\}|$$

Analogously for all $n > 0$, $(A^n)_{ij}$ is equal to the number of paths of length n which start in a_i and end in a_j.

$$(A^n)_{ij} = |\{\pi \in \Pi_n \mid \alpha(\pi) = a_i \text{ and } \omega(\pi) = a_j\}|$$

Every row vector of the matrix potential corresponds to paths beginning in a certain argument and every column vector corresponds to paths ending in an argument. Thus with the help of a matrix potential, the number of paths of a certain length beginning and ending in a node can easily be computed.

$$d_n^+(a_i) = \sum_j (A^n)_{ij}$$

$$d_n^-(a_i) = \sum_j (A^n)_{ji}$$

Again ingoing paths of odd length need to be weighted negatively. To solve this problem, the negative adjacency matrix is introduced.

Definition 4.10. If A is the adjacency matrix of an AAF Γ, $-A$ is called its **negative adjacency matrix**. One of its elements $-a_{ij}$ is equal to -1 if a_i attacks a_j and otherwise 0.

The matrix potential $(-A)^n$ is now equal to $(-1) \cdot (A^n)$ if n is odd and equal to A^n otherwise. Therefore the n-th potential of $-A$ still represents the number of paths between arguments if n is even, but minus the number of paths when n is odd [11].

$$(-A)^n = \begin{cases} A^n & \text{if } n \text{ even} \\ -A^n & \text{if } n \text{ odd} \end{cases}$$

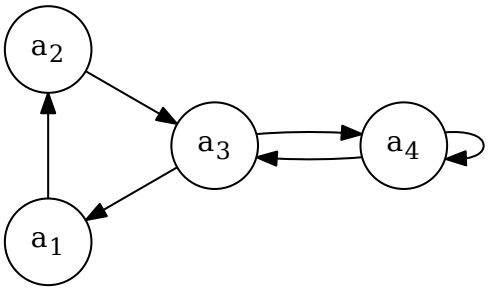
Matrix exponentials have been suggested to be used as heuristics for heuristic search in abstract argumentation [11]. Analogously to the real exponential function $\exp(x) = e^x$, the matrix exponential can be expressed as a power series.

Definition 4.11. The **matrix exponential** $\exp : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$ for any real square matrix $A \in \mathbb{R}^{n \times n}$ is defined as

$$\exp(A) = e^A = \sum_{i=0}^{\infty} \frac{A^i}{i!}$$

As is apparent from its depiction as a power series, the matrix exponential $\exp(A)$ combines potentials A^n of the adjacency matrix of an AAF Γ with $n \in \mathbb{N}$. A single entry of the matrix exponential therefore combines all path inside Γ from one specific argument to another one weighted according to their length.

To distinguish between paths of even and odd length, the matrix exponential can be applied to the negative adjacency matrix. For any AAF $\Gamma = (\mathcal{A}, \mathcal{R})$, the exponential $\exp(-A)$ of the negative adjacency matrix contains all paths in Γ weighted



a	$h^s_{exp}(\emptyset, a)$	$h^d_{exp}(\{a_1\}, a)$
a_1	0.649	0.827
a_2	0.260	−0.957
a_3	−0.054	0.526
a_4	0.198	−0.137

Figure 13: *Left:* The AAF used in Example 4.12. *Right:* Values of heuristics based on the matrix exponential

according to their length, especially paths of even and odd length respectively are weighted with different signs. A path with even length $2n$ is weighted with $\frac{1}{(2n)!}$ and a path with odd length $2n+1$ has a weight of $\frac{-1}{(2n+1)!}$ for any $n \in \mathbb{N}$. Therefore paths with an even length are weighted positively while those with odd length are weighted negatively and longer paths are assigned weights with smaller absolute values, while the weights of shorter paths have greater absolute values. We now define a first heuristic based on matrix exponentials. The static heuristic h^s_{exp} maps an argument a to the sum of all elements of $\exp(-A)$ which correspond to paths ending in a [11].

$$h^s_{exp}(E, a_i) = \sum_{a_j \in \mathcal{A}} (\exp(-A))_{ji}$$

When considering the vulnerability of an argument at a certain point during algorithm execution, paths which begin in an argument which is part of the current partial solution are especially interesting. Arguments which are attacked by the partial solution should not be chosen, while those which are supported by it should be given precedence. Therefore an additional, dynamic heuristic h^d_{exp} is defined. It resembles the static heuristic but only considers paths beginning in arguments which are part of the current partial extension. These paths which start in the partial extension might be most relevant as for example an outgoing path of length 1 from the partial extension to any argument makes the inclusion of that argument into the extension impossible [11].

$$h^d_{exp}(E, a_i) = \sum_{a_j \in E} (\exp(-A))_{ji}$$

Example 4.12. Given the AAF Γ which is visualised by the graph on the left of Figure 13, we compute the adjacency matrix A. The i-th row vector of A represents attacks on the argument a_i and the i-th column vector indicates which arguments

are attacked by a_i. The squared adjacency matrix A^2 represents all paths of length two in Γ. For example $(A^2)_{1,3}$ is 1, because a_1 reaches a_3 in two steps via a_2.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad A^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 2 \end{pmatrix}$$

We now compute the exponential of the negative adjacency matrix:

$$\exp(-A) \approx \begin{pmatrix} 0.827 & -0.957 & 0.526 & -0.137 \\ 0.526 & 0.827 & -1.094 & 0.389 \\ -1.094 & 0.526 & 1.217 & -0.704 \\ 0.389 & -0.137 & -0.704 & 0.649 \end{pmatrix}$$

From this matrix we can extract values of the static heuristic h^s_{exp} by summing the corresponding column value for each argument. This sum represents the number of paths ending in a specific argument, weighted according to each path's length. The results are shown in the table in Figure 13. The argument a_1 is chosen by that heuristic and added to the partial solution. It is considered the argument which is least vulnerable to defeat, as it is attacked only attacked by a_3 which in turn is attacked by two other arguments. After a_1 was added to the partial solution, the dynamic heuristic h^d_{exp} is computed, where every value $h^d_{exp}(\{a_1\}, a_i) = (exp(-A))_{1,i}$ is composed only of one element of the first row vector of the matrix exponential because the partial solution only contains a_1. Among the remaining arguments the heuristic chooses a_3, which seems reasonable as a_1 defeats a_2 which attacks a_3, but at the same time creates a dead end situation as $\{a_1, a_3\}$ is not conflict-free.

4.5 Centrality Measures

Centrality measures could be helpful heuristics as a central argument will attack more arguments and therefore help to defeat arguments outside the partial solution. Two common centrality measures, indegree and outdegree, have already been considered in Section 4.2. Several other centrality measures from network theory have also been considered as static heuristics. One such centrality measure is the betweenness centrality.

Definition 4.13. Let the **betweenness centrality** of an argument $a \in \mathcal{A}$ in a graph $\Gamma = (\mathcal{A}, \mathcal{R})$ be dfined as

$$C_B(a) = \sum_{s \neq a \neq t \in \mathcal{A}} \frac{\sigma_{st}(a)}{\sigma_{st}}$$

where σ_{st} is the number of shortest paths between the arguments s and t and $\sigma_{st}(a)$ is the number of shortest paths between those arguments which contain a.

The betweenness centrality is used as a static heuristic h_B.

$$h_B(E, a) = C_B(a)$$

Another centrality measure is the Eigenvector centrality, which is also used as a heuristic.

Definition 4.14. Let $A \in \mathbb{R}^{n \times n}$ be the adjacency matrix of a graph, e.g. an abstract argumentation framework. For an argument a_i, the **eigenvector centrality** is defined as the i-th element of the eigenvector $x \in \mathbb{R}^n$ which corresponds to the eigenvalue $\lambda \in \mathbb{R}$ with the largest absolute value of A.

$$C_{eig}(a_i) = x_i$$

$$\text{where } Ax = \lambda x \text{ with } |\lambda| \text{ max.}$$

The eigenvector centrality is used as a static heuristic h_{eig}.

$$h_{eig}(E, a) = C_{eig}(a)$$

4.6 Strongly Connected Components

Some methods of reasoning for abstract argumentation rely on the partition of an AAF into strongly connected components and a special property of these components called SCC recursiveness. A hierarchy can be established among strongly connected components such that the labelling of a component depends only on the labelling of components above this components. To exploit this a static heuristic will be defined which sorts strongly connected components in such a way that their order reflects this hierarchy.

Definition 4.15. A **strongly connected component (SCC)** refers to a set of arguments where every element of that set is connected to every other element. In other words, in an AAF $\Gamma = (\mathcal{A}, \mathcal{R})$, $S \subseteq \mathcal{A}$ is an SCC iff for every pairing of arguments $a, b \in S$ with $a \neq b$, it holds that there is a path $a \to \cdots \to b$ from a to b as well as a path $b \to \cdots \to a$ from b to a. All strongly connected components of an argument set are disjoint as an argument cannot be part of two different strongly connected components and every argument is part of an SCC as it could from an SCC of size one on its own. Therefore the set of strongly connected components of a set is a partition of that set. The strongly connected component of an argument $a \in \mathcal{A}$ is denoted as $SCC(a)$.

$$\begin{aligned} SCC(a) = \{b \in \mathcal{A} \mid &\text{there is a } \pi \in \Pi \text{ with } \alpha(\pi) = a \text{ and } \omega(\pi) = b \\ &\text{and there is a } \pi' \in \Pi \text{ with } \alpha(\pi') = b \text{ and } \omega(\pi') = a\} \end{aligned}$$

Additionally the set of all strongly connected components of the AAF is called SCC_Γ.

$$SCC_\Gamma = \{S \subseteq \mathcal{A} \mid S = SCC(a) \text{ for any } a \in \mathcal{A}\}$$

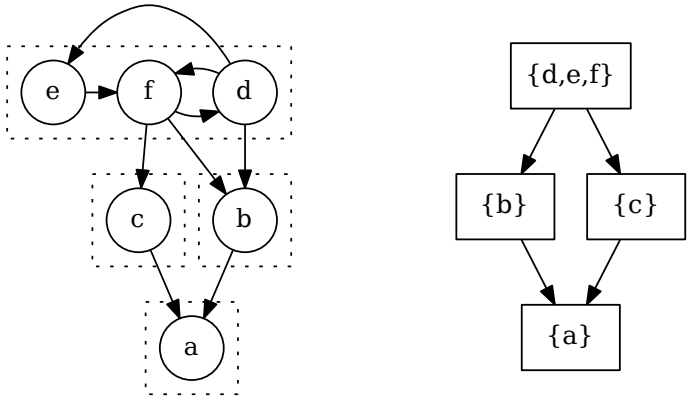


Figure 14: *Left:* The AAF used in Examples 4.16 and 4.23. *Right:* The same AAF decomposed into its strongly connected components, which are composed into a hierarchical graph.

Example 4.16. The AAF given on the left in Figure 14 has four strongly connected components: One is $S_1 = \{a\}$ as a does not attack any other argument. $S_2 = \{b\}$ and $S_3 = \{c\}$ form strongly connected components on their own too. Both attack a, but there is no path leading back from a. The last one is $S_4 = \{d, e, f\}$ as the circular path $\pi = d \to e \to f \to d$ contains a subpath from every element of S_4 to every other element.

Theorem 4.17. All strongly connected components of an AAF form an acyclic directed graph, where each SCC is a node and edges are attacks from one SCC to another one. In this graph there cannot be a cycle as all nodes which were part of a cycle would form a single SCC [19].

Example 4.18. On the right side of Figure 14, the strongly connected components of AAF from Example 4.16 are ordered into an acyclic graph according to Theorem 4.17. All other components depend on $S_4 = \{d, e, f\}$ because it directly attacks $S_2 = \{b\}$ and $S_3 = \{c\}$ which in turn both attack $S_1 = \{a\}$. This leads to $S_1 = \{a\}$ additionally depending on both $S_2 = \{b\}$ and $S_3 = \{c\}$.

Definition 4.19. Let $\Gamma = (\mathcal{A}, \mathcal{R})$ be an AAF. For any subset $S \subseteq \mathcal{A}$ we now call $\Gamma_{|S} = (S, \mathcal{R} \cap (S \times S))$ the **restriction** of Γ to S. The restriction $\Gamma_{|S}$ is again an AAF which consists of the node set S and all edges which begin in S and end in S as well [3].

Definition 4.20. A semantics σ is called **SCC-recursive** according to Baroni et al. [3] iff for every AAF $\Gamma = (\mathcal{A}, \mathcal{R})$, it holds that $\mathcal{E}_\sigma(\Gamma) = GF_\sigma(\Gamma, \mathcal{A})$ where $GF_\sigma(\Gamma, C) \subseteq \mathcal{P}(\mathcal{A})$ is recursively defined in the following way

- if $|SCC_\Gamma| = 1$, $GF_\sigma(\Gamma, C) = BF_\sigma(\Gamma, C)$, where the base function BF_σ is specific for the semantics σ. $BF_\sigma(\Gamma, C) = \mathcal{E}_\sigma(\Gamma) \cap \mathcal{P}(C)$ is a set of partial extensions that

is similar to the set of all σ-extensions of Γ but considers arguments which are not part of C to be automatically defeated

- otherwise an argument set $E \subseteq \mathcal{A}$ lies inside $GF_\sigma(\Gamma, C)$ iff for every $S \in SCC_\Gamma$ it holds that $(E \cap S) \in GF_\sigma(\Gamma_{|S'}, U \cap C)$ where $S' = S \setminus (E \setminus S)^+$ is the part of S which is not attacked by $(E \setminus S)$ and $U = \{a \in S' \mid \{a\}^- \setminus S \subseteq E^+\}$ is the part of S which is defended against outside aggressors by E

SCC recursiveness has been proven by Baroni et al. [3] for complete, preferred, stable and grounded semantics. An AAF can be decomposed into a set of subframeworks, where every subframeworks corresponds with one SCC. Among those a partial ordering is established according to the acyclic graph from Theorem 4.17. Because of SCC-recursiveness the extensions of one subframeworks depend only on extensions of subframeworks which are above it in the acyclic graph. For any SCC-recursive semantics extensions of an AAF can now be computed by combination of extensions of its subframeworks. The extensions in subframeworks of strongly connected components which are not attacked by other strongly connected components can be computed independently. Afterwards the extensions of he remaining subframeworks can be computed recursively. The following example is meant to clarify the idea of SCC-recursive semantics.

Example 4.21. All stable extensions from the AAF $\Gamma = (\mathcal{A}, \mathcal{R})$ which is given in Figure 14 shall be enumerated with the help of SCC recursiveness. We therefore construct $\mathcal{E}_{ST}(\Gamma) = GF_{ST}(\Gamma, \mathcal{A})$ recursively. Γ is partitioned into the strongly connected components $S_1 = \{a\}$, $S_2 = \{b\}$, $S_3 = \{c\}$, $S_4 = \{d, e, f\}$ as shown in Example 4.16. Partial extensions have to be determined for each SCC. According to the hierarchy established in Example 4.18 $GF_{ST}(\Gamma_{|S_4}, S_4)$ has to be constructed first as S_4 is the at the top of this hierarchy.

S_4: The restricted AAF contains all of $S_4 = \{d, e, f\}$ as this SCC is unattacked and therefore $S_4 = S_4' = U_4$. The AAF restricted to S_4 has one partial stable extension as d attack all other arguments inside that SCC

$$GF_{ST}(\Gamma_{|S_4'}, U_4) = BF_{ST}(\Gamma_{|S_4}, S_4) = \{\{d\}\}$$

The components S_2 and S_3 are directly below S_4 in the hierarchy and are handled next as they depend only the partial extension $\{d\}$. If there were several partial extensions in S_4, it would be necessary to resume and construct different partial argumentation frameworks for every partial extension and every SCC.

S_2: First S_2', the part of $S_2 = \{b\}$ which is not attacked by the partial extension $\{d\}$, is determined to be empty as b is attacked by d.

$$S_2' = S_2 \setminus \{d\}^+ = \emptyset$$

There also is no part of S_2 which is defended by outside aggressors by the partial extension, therefore $U_2 = \emptyset$. The constructed restriction is empty and an empty AAF has an empty stable extension

$$GF_{ST}(\Gamma_{|S'_2}, U_2) = BF_{ST}(\Gamma_{|\emptyset}, \emptyset) = \{\emptyset\}$$

S_3: First S'_3, the part of $S_3 = \{c\}$ which is not attacked by the partial extension $\{d\}$, is determined.

$$S'_3 = S_3 \setminus \{d\}^+ = S_3$$

The only argument in S_3 is defended by d, therefore $U_3 = S_3$. The constructed restriction consists of one argument and such an AAF has one stable extension which contains that argument.

$$GF_{ST}(\Gamma_{|S'_3}, U_3) = BF_{ST}(\Gamma_{|\{c\}}, \{c\}) = \{\{c\}\}$$

When combining the results achieved for the components S_2, S_3 and S_4, one receives a combined partial solution $\{c, d\}$. As a final step a partial extension for S_1 must be determined.

S_1: First S'_1, the part of $S_1 = \{a\}$ which is not attacked by the partial extension $\{c, d\}$, is determined to empty as a is attacked by c.

$$S'_1 = S_1 \setminus \{c, d\}^+ = \emptyset$$

There also is no part of S_1 which is defended by outside aggressors by the partial extension, therefore $U_1 = \emptyset$. The constructed restriction is empty and an empty AAF has an empty stable extension

$$GF_{ST}(\Gamma_{|S'_1}, U_1) = BF_{ST}(\Gamma_{|\emptyset}, \emptyset) = \{\emptyset\}$$

Because a partial extension has been found for every SCC, these partial extensions can be combined into a stable extension for Γ. There is exactly one stable extension as every SCC has exactly one partial extension. Several stable extension would have been possible if one of the components would have had several partial extensions, while if for any SCC no partial extension could have been found there would be no stable extension. The combination of the partial extensions from all components yields the stable extension $\{c, d\}$.

SCC recursiveness can be utilised for heuristical search. A heuristic h_{SCC} is defined, which sorts arguments according to the SCC they belong to. First the argument set is partitioned into strongly connected components and then all strongly connected components are ordered according to the partial order established by the acyclic graph given in Theorem 4.17 and each assigned a value which reflects this order. Any implementation of the SCC heuristic has to fit the following definition.

Definition 4.22. With regard to any AAF $\Gamma = (\mathcal{A}, \mathcal{R})$, for any two arguments $a, b \in \mathcal{A}$ the **SCC heuristic** h_{SCC} fulfils the following constraints

i) $h_{SCC}(E, a)$ is independent from E and can therefore be written as $h_{SCC}(a)$, this means that h_{SCC} is static

ii) $h_{SCC}(a) = h_{SCC}(b)$ iff $SCC(a) = SCC(b)$

iii) if $SCC(a) \neq SCC(b)$ and there is an argument $c \in SCC(a)$ and an argument $d \in SCC(b)$ such that c attacks d, it has to hold that $h_{SCC}(a) > h_{SCC}(b)$

Let the following example illustrate how an SCC-based heuristic would work when the SCC-hierarchy does not constitute a total order.

Example 4.23. When tasked to compute the scores for the arguments of the AAF given in Frigure 14, an SCC-based heuristic would first divide the argument set into strongly connected components and then establish a hierarchy among those as was done in Example 4.16. The heuristic would then assign a value to each SCC such that no two components have the same value and components that have outgoing attacks into another components are assigned a value greater than the value of the attacked component. Therefore a would be assigned the smallest value and d, e and f would be assigned the greatest value. b and c would be both assigned a different intermediate value. An SCC-based heuristic h_{SCC} for the AAF could for example yield the results which are listed in the following table:

x	a	b	c	d	e	f
$h_{SCC}(x)$	1	3	2	4	4	4

This SCC-based heuristic ensures that arguments which belong to influential strongly connected components are handeled early and therefore limit the number of potential labelling in dependant strongly connected components. It could therefore increase performance by narrowing the search space. The SCC heuristic should be used as a dominant component combined with other static and dynamic heuristics. It should be dominant as strongly connected components should not be mixed and it should not be used on its own as an SCC can be rather large and therefore its arguments should also be distinguished among each other.

Let us finally summarise the different classes of heuristics which have been defined in the course of this section. First static heuristics are considered:

- several degree-based heuristics $\deg^-$, $\deg^+$, h^{Σ}_{deg}, h^{Δ}_{deg}, h^{Π}_{deg}, $h^{\div}_{deg}$ and $h^{\div *}_{deg}$
- several path-based heuristics h^{-}_{path}, h^{+}_{path}, h^{Σ}_{path}, h^{Δ}_{path}, h^{Π}_{path} and $h^{\div}_{path}$
- a static matrix exponential heuristic h^{s}_{exp}
- centrality-based heuristics h_B and h_{eig}

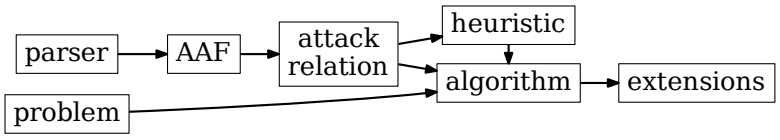


Figure 15: The different objects created by the HEUREKA solver and their relations for a stable, complete or preferred enumeration task. Other tasks will have a different form of output instead of a set of extensions. Also For some task-semantics combinations the heuristic object will not be constructed.

- the SCC heuristic h_{SCC}

The following list contains all defined dynamic heuristics.

- aggressor-count-based heuristics h_{aggr} and $h^{\div}_{aggr}$
- the defensor count heuristic h_{defor}
- a dynamic matrix exponential heuristic h^{d}_{exp}

Section 5 will provide details on the implementation of static as well as dynamic heuristics in the abstract argumentation solver HEUREKA. In Section 6 the performance of different heuristics and combinations thereof will be compared.

5 Implementation

The HEUREKA solver was implemented in C++ and can be compiled under Unix-like systems as well as Windows systems. It fulfils the command line interface specifications of ICCMA'17 [17]. The source code of the project can be accessed on github [1]. This section will first give an overview of the process of solving an abstract argumentation problem by the HEUREKA solver. Then it will detail on the computation of the scores of static and dynamic heuristics and finally describe the notation that allows the combination of different heuristics in HEUREKA and also explain the limitations of such combinatory heuristics.

5.1 Solver Structure

HEUREKA structures the reasoning process by constructing several objects. Each object implements another part of the reasoning process. The relations between those objects in shown in Figure 15. Some objects exists in several alternative versions, for example there are parsers for different file formats and algorithms for different semantics. HEUREKA chooses among these alternatives based on the problem it is

[1] `https://github.com/nilsgeilen/heureka`

tasked to solve. The AAF is parsed from a file, which is specified as a command line parameter, and then transformed into an attack relation object. This object is more favourable for algorithmic purposes as it allows the set of attackers and the set of attackees of an argument to be accessed cheaply. A heuristic object is created out of this attack relation according to a heuristic function. This heuristic function can be specified as a command line parameter. If no heuristic is specified as a command line parameter, HEUREKA employs a semantics-specific default heuristic. Additionally the abstract argumentation problem has to be given as a command line parameter. HEUREKA implements Algorithms 1, 5 and 7. Depending on the problem task and semantics, one of these algorithms is executed on the attack relation with help of the heuristic. When the execution has finished, it will output a result that fits the problem task, in the case of enumeration that is for example a set of extensions. Problems which can be reduced to finding the grounded extension, and therefore are solved with Algorithm 1, do not require a heuristic, the process of reasoning is therefore similar but somewhat simpler in those cases. HEUREKA's command line interface conforms to the requirements of ICCMA'17 [17]. When it is invoked, HEUREKA is therefore passed the following set of command line parameters:

- `-f` specifies the relative or absolute path of the file where the AAF which shall be analysed is stored. This parameter is mandatory.
- `-fo` specifies the file format that the AAF is stored in. This parameter is not required if the file format is apparent from the file ending, otherwise it is mandatory.
- `-p` specifies the abstract argumentation problem that HEUREKA is tasked to solve. The problem has to be given in the concise form `{CO,PR,GR,ST}-{EE,SE,DC,DS}`. This parameter is mandatory.
- `-a` is the argument which is to be justified. This parameter is mandatory for justification problems, but unnecessary for other problems.
- `-H` defines of a custom heuristics, using the notation given in Section 5.3. If no custom heuristics is specified, HEUREKA will select a heuristic according to the abstract argumentation problem that it is tasked to solve.

So the following command for example invokes HEUREKA which would then enumerate all stable extensions of the AAF represented by the file "graph.apx" using a default heuristic. The answer to this problem HEUREKA would output a list of lists of arguments, where every list of arguments would represent an extension.

```
$ heureka -p EE-ST -f graph.apx
```

The next command tasks HEUREKA to find out whether the argument a_{50} is sceptically justified under preferred semantics in the AAF represented by the file "graph.apx". A custom heuristic $h^*(E, a) = -\deg^-(a)$ which maps an argument to

<**aaf**>	::=	(<**argument**> "\n")* "#\n" (<**attack**> "\n")*
<**argument**>	::=	<**identifier**>
<**attack**>	::=	<**identifier**> "␣" <**identifier**>
<**identifier**>	::=	("A" | ... | "Z" | "a" | ... | "z" | "0" | ... | "9")+

Figure 16: EBNF specification for an AAF in tgf file format

its negative indegree is constructed an used for this task. After finishing the necessary computations, HEUREKA would output either "YES" or "NO" to answer this question.

```
$ heureka -p DS-PR -a a50 -f graph.apx -H "- indegree"
```

When the solver is called, it first reads in an AAF from a file either in Trivial Graph Format (*.tgf) or Aspartix Format (*.apx) [17]. While tgf is more concise, it is also whitespace-sensitive, and while apx is more verbose, it is independent of whitespace. For EBNF specifications of both formats, see Figures 16 and 17. The path of the file where the AAF is stored is specified by a command line parameter. The file contains a list of named arguments and a list of argument tuples representing attacks. It can therefore easily be parsed into a similar representation of the AAF. For computational reasons, all arguments are associated with identifying 32-bit integers and represented by those throughout the program code. Afterwards the AAF is converted into a different representation which allows algorithms to extract informations faster. For every argument it holds a vector containing all of its attackers as well as a vector containing all of its attackees. The set of attackers and the set of attackees can therefore both be accessed cheaply.

The problem description is given to HEUREKA as a command line parameter in the form of a tuple which consists of task and semantics. After task-dependant preparations, an algorithm is invoked according to the semantics. The stable algorithm is used for all stable problems, while the complete algorithm is used for all preferred problems as well as complete enumeration and complete credulous justification. For finding a single complete extension and complete sceptical justification as well as all grounded problems, the grounded algorithm is used. The different algorithms are described in detail in Section 3. There the backtracking algorithms are given in a recursive manner, but for performance reasons, these algorithms are actually implemented in an iterative manner. An algorithm stores the labelling and several other informations about the arguments in vectors and additionally holds a stack which keeps track of all changes to the labelling. For each labelling change it is additionally remembered whether this update was caused by a decision directly or caused by a consequence. When an algorithm has to backtrack, it pops labelling changes from the stack and reverts those until a decision is popped. The respective argument is excluded from the extension and the algorithm will then resume normally.

<**aaf**>	::=	<**argument**>* <**attack**>*
<**argument**>	::=	"arg(" <**identifier**> ")."
<**attack**>	::=	"att(" <**identifier**> "," <**identifier**> ")."
<**identifier**>	::=	("A" | ... | "Z" | "a" | ... | "z" | "0" | ... | "9")+

Figure 17: EBNF specification for an AAF in apx file format

Example 5.1. A backtracking algorithm, either Algorithm 5 or 7, is tasked with finding an extension in the AAF specified in Figure 18. First an initial labelling $\mathcal{L}$ is computed and the stack S is empty.

$$\mathcal{L} = \{(a, \text{BLANK}), (b, \text{BLANK}), (c, \text{BLANK}), (d, \text{UNDEC})\}$$

$$S = ()$$

The algorithm then calls a heuristic to determine the argument which it tries to include first. The heuristic returns c, which is therefore set to IN. Argument c is added to the stack and marked as a decision.

$$\mathcal{L}' = \{(a, \text{BLANK}), (b, \text{BLANK}), (c, \text{IN}), (d, \text{BLANK})\}$$

$$S' = (c^d)$$

As a consequence of this first decision, the algorithm then starts to exclude the arguments which are attacked by c or which attack c from the extension. It starts with setting a to OUT as a is defeated by c. It does the same to d and set b to UNDEC as it attacks c. An updated labelling of the following form is constructed.

$$\mathcal{L}'' = \{(a, \text{OUT}), (b, \text{UNDEC}), (c, \text{IN}), (d, \text{OUT})\}$$

$$S'' = (c^d, a, b, d)$$

All argument label updates are pushed to the stack. While constructing $\mathcal{L}''$, the algorithm recognizes that b is an aggressor for the extension but cannot be defeated by it. It aborts the process and backtracks to the last decision. It pops all updates from the stack until the addition of c and reset all corresponding arguments to BLANK. It sets c itself to UNDEC and adds it to the stack, but this time c is not marked as a decision.

$$\mathcal{L}''' = \{(a, \text{BLANK}), (b, \text{BLANK}), (c, \text{UNDEC}), (d, \text{BLANK})\}$$

$$S''' = (c)$$

The algorithm then resumes by requesting a new candidate for inclusion from the heuristic.

5.2 Computation of Heuristic Scores

The two backtracking algorithms which enumerate stable or complete extensions respectively require a heuristic function to help them take decisions. For the respective problems, an object representing a heuristic function is created. The algorithm will call this function every time it takes a decision and asks it for guidance. The heuristic function object implements a heuristic function of the form $h : \mathcal{P}(\mathcal{A}) \times \mathcal{A} \to \mathbb{R}$ where $\mathcal{A}$ is the argument set of an AAF. This function maps a partial solution and an argument to a heuristic score. A custom heuristic consisting of different heuristic components can be parsed from a command line parameter as described in Section 5.3. The heuristic function can be split into a static part h^s and a dynamic part h^d. When tasked to select the next candidate to be potentially included into the extension, the function is supplied with an integer number representing the number of previous decisions and some information about the current partial extension, for example which arguments are already defeated by the extension. The extension-related information is only used by dynamic heuristics as they rely on the current extension E. The heuristic function object holds an array of tuples, where every tuple corresponds to an argument. A tuple $(a, h^s(\emptyset, a))$ consists of an argument a and the static heuristic value $h^s(\emptyset, a) \in \mathbb{R}$. The value $h^s(E, a) = h^s(\emptyset, a)$ can be statically computed as it does not rely on the current partial extension E. During the intialisation of the heuristic function object, this array is sorted according to the static heuristical value in descending order. The algorithm maintains a counter which keeps track of the number of decisions already taken, but excluding reverted changes. This counter is even incremented when the heuristic suggests an argument which the algorithm has already assigned a label. When such an argument is suggested, the algorithm will just increase the decision counter and request a new argument. When the algorithm backtracks and reverts all changes until the last decision, the counter is set back to the value it had at the time of this decision, and is increased by 1 before the next argument is requested. A heuristic which is made up entirely of static components will always return the i-th element of the array as a response to the i-th decision.

Example 5.2. The AAF given in Figure 18 is read in from a file. As a heuristic a static outdegree heuristic $(E, a) \mapsto \deg^+(a)$ is specified. Therefore the following array is constructed:

$$((a, 1), (b, 1), (c, 2), (d, 2))$$

This array will then be sorted:

$$((c, 2), (d, 2), (a, 1), (b, 1))$$

When no dynamic heuristic is specified the arguments will therefore be returned in the following order: c, d, a, b. The early choice of the self-attacking argument d is not harmful as the algorithm will immediately backtrack and thus exclude d from the start. It is apparent that a second static component should be added to distinguish c from d and a from b.

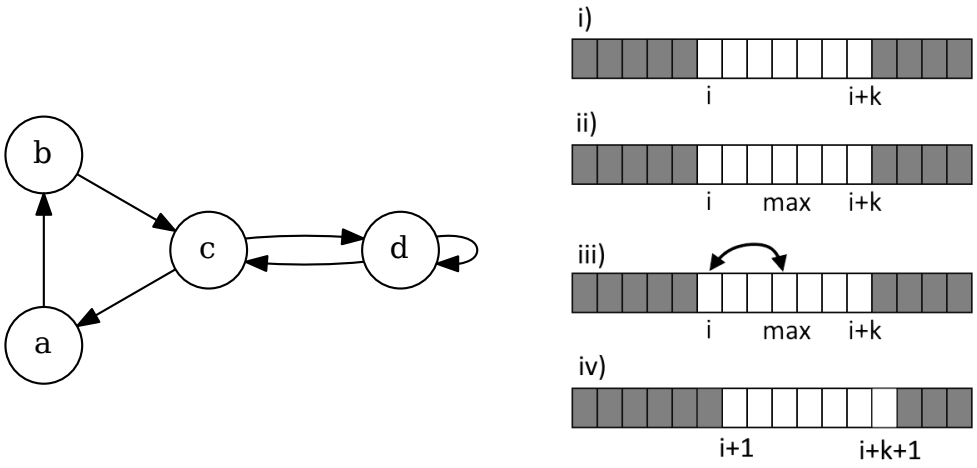


Figure 18: *Left:* The AAF used in Examples 5.1 and 5.2. *Right:* visualisation of the selection process of a dynamic heuristic.

When additionally a dynamic heuristic was specified, the array has to be re-ordered on every decision. For an example of the way dynamic heuristics work, see Example 5.3.

A dynamic heuristic in contrast will compare the k most promising arguments, where k is a specified window size. Assume that all arguments are pre-sorted by a static heuristic. Let now the argument with the highest static heuristic value be called a_1, the argument second highest value a_2 and so on, which results in an ordering $a_1 > a_2 > \cdots > a_n$ with $n = |\mathcal{A}|$. When tasked to take the i-th decision, it will compute $h^s(\emptyset, a_j) + h^d(E, a_j)$ for every argument a_j inside the window with $j = i, \ldots, i + k$ where a_j is the j-th element of the array. To do this it utilizes the information about the partial extension E that have been supplied by the algorithm. The argument inside the window with the maximal combined heuristical value will be given priority and its position in the array will be swapped with the i-th element, so it will not be considered for the next decision, which will consider a window from $i + 1$ to $i + k + 1$. The argument on position i of the array is only considered again, when the algorithm backtracks to a decision which happened before the i-th decision.

Example 5.3. An array representation for a dynamic heuristic in a simple AAF is visualised on the right in Figure 18. The heuristic is tasked with deciding which argument should be tried to be included into the extension next. All arguments are stored in an array, pre-sorted according to a static heuristic. As this is the i-th decision, all arguments sorted at a lower position than i have already been considered. Therefore all arguments which are still blank are ordered at a position greater than i and arguments on lower positions can savely be ignored. The window size is specified as k, therefore all arguments ordered on a position greater then $(i + k)$ are not

Algorithm 9 Power Iteration

Input: $A \in \mathbb{R}^{n \times n}$ square matrix

Output: $x \in \mathbb{R}^n$ dominant eigenvector

1: $x_i \leftarrow 1/n$
2: **while** x changed **do**
3: $\quad x \leftarrow \frac{Ax}{\|Ax\|_2}$
4: **return** x

considered due to their low static heuristic score. In Figure 18 the window of arguments which are considered for inclusion from i to $(i + k)$ is coloured white (i). The dynamic heuristic computes a score for all arguments in this window and selects the maximum (ii). It then swaps the positions of the i-th element and the maximal element (iii). The maximal element which is selected by the algorithm. The selected element will then no longer be considered when the heuristic is called next time, as the window has moved one step "to the right" (iv).

To compute the scores of the arguments of an AAF for some of the implemented heuristics, HEUREKA needs to perform arithmetic operation on vectors and matrices. For this purpose the linear algebra library *Eigen3* [2] is employed which is published under an open source licence. If for example eigenvector centrality is used as a heuristic h_{eig}, the dominant eigenvector of the adjacency matrix of the AAF needs to be computed. HEUREKA employs Algorithm 9 which relies on *power iteration* to compute the dominant eigenvector [12]. The operations on the adjacency matrix of the AAF necessary for the exponential heuristics are also done using *Eigen3*. Additionally several heuristics use algorithms which do not rely on arithmetic operations to compute their scores. The betweenness centrality of the arguments of the AAF, which can also be used as a heuristic h_B, is computed by a version of *Brandes' Algorithm* implemented in HEUREKA [4]. For the SCC-based heuristics, the AAF is first partitioned into strongly connected components with a version of *Tarjan's Algorithm* [25, 23]. Afterwards the strongly connected components are ordered according to the partial hierarchy between them.

5.3 Combination of Heuristics

So far in Section 4 a number of heuristics has been established, which for example includes the following: h^{Σ}_{path}, $h^{\div}_{deg}$, $h^{\div *}_{deg}$, $h^{\div}_{path}$, h_B, h_{eig} and h_{SCC}, which are all static. Additionally dynamic heuristics have been established, which are h_{aggr}, h_{defor}, $h^{\div}_{aggr}$ as well as h_{exp}. All of those have been implemented as part of the HEUREKA solver. HEUREKA parses information about which heuristics it will use from a command line parameter. Do accomplish this it uses a format which will be defined in this section. It knows the following simple static heuristics:

[2] http://eigen.tuxfamily.org/

- `deg beta gamma` is a degree-based heuristic where γ is a floating-point number representing the weight given to the indegree and β is a floating-point number representing the weight given to the outdegree, resulting in a combinatorial value $\beta \deg^- + \gamma \deg^+$
- `indeg` is the pure indegree while `outdeg` is the pure outdegree
- `path k1 beta k2 gamma` where k_1 and k_2 are integer numbers and γ and β are floating-point numbers, is a path-based heuristic which is equal to $h^{\Sigma}_{path} = h^{-}_{path} + h^{+}_{path}$ where h^{-}_{path} has the maximal path depth k_1 and the path weight β while h^{+}_{path} has the maximal path depth k_2 and the path weight γ
- `inpath` is an ingoing-path-based heuristic h^{-}_{path} with fixed default parameters $k = 3$ and $\beta = -0.1$
- `outpath` is an outgoing-path-based heuristic h^{+}_{path} with fixed default parameters $k = 5$ and $\gamma = 0.25$
- `exp k` is equal to the static exponential heuristic h^{s}_{exp} where k is an integer number which stands for the maximal exponent and therefore the maximal path depth
- `extdegrat` is equal to $h^{\dot{-}*}_{deg}$
- `cb` which is equal to h_B and `ceig` which is equal to h_{eig}
- `scc` represents h_{SCC}

These static heuristics can be combined through arithmetic operations. These are addition ($+$), subtraction ($-$), multiplication ($*$), division ($/$) and exponentiation ($\hat{}$). The formulae built with these operations can also contain constants and have to be given in Polish notation, i.e. $a \circ b$ is written as `∘ a b` for $\circ \in \{+, -, *, /, \hat{}\}$. All the remaining static heuristics defined in Section 4 can be defined by combining the heuristics listed above and constants through arithmetic operations. The degree ratio heuristic $h^{\dot{-}}_{deg}$ could for example be expressed as

```
/ + outdeg 1 + indeg 1
```

$$h^{\dot{-}}_{deg}(E, a) = \frac{\deg^+(a) + 1}{\deg^-(a) + 1}$$

Additionally a combined heuristic can contain at most one dynamic component. Dynamic components perform an operation on a precomputed static component during algorithm execution and can be defined with the following syntax.

- `dynindeg k beta` represents the aggressor count heuristic h_{aggr} where k is an integer number which denotes the window size and β is a floating-point number which is used to weigh the aggressor component, this dynamic component is added to a static score $h^s(a)$ resulting in a heuristical score $h^s(a) + \beta h_{aggr}(E, a)$ for an argument a and an extension E

- `dynindegrat k` also corresponds to the aggressor count heuristic h_{aggr} with a window size given as the integer number k, but here the static score $h^s(a)$ is divided by the aggressor count heuristic resulting in a heuristical score $h^s(a)/(h_{aggr}(E,a)+1)$ for an argument a and an extension E
- `dynoutdeg k gamma` is equal to the defensor count heuristic h_{defor} where k is an integer number which denotes the window size and γ is a floating-point number which is used to weigh the defensor component, this dynamic component is added to the static score $h^s(a)$ resulting in a heuristical score $h^s(a)+\gamma h_{defor}(E,a)$ for an argument a and an extension E
- `dynexp k` is equal to the dynamic exponential heuristic h^d_{exp} where k is an integer number which represents the maximal exponent and therefore the maximal path depth, the dynamic component is here again added to the static component

Because dynamic heuristics are recomputed at every step of algorithm execution, they have to work as efficient as possible. Therefore a decision was taken that HEUREKA only allows a single dynamic component to be part of a custom heuristic. Custom arithmetic operations or combinations of different heuristics are only permitted to be computed statically before the algorithm execution starts. An example for a combined heuristic with a dynamic component which sorts arguments according to h_{SCC} first, then according to h_{aggr} and lastly according to outdegree is the following

```
+ * scc 1000000 + dynindeg 100 1000 outdeg
```

which corresponds to $10^6 \cdot h_{SCC} + 10^3 \cdot h_{aggr} + h^+_{path}$. While the weight of static components can be computed with arithmetic operations in Polish notation, the weight of the dynamic aggressor count heuristic has to be specified via a parameter as no custom arithmetic operation can be carried out during algorithm runtime.

Example 5.4. A static component based on outgoing paths and a dynamic aggressor count component could be combined in the following two, fundamentally different ways: By addition a combined heuristic $h^+_{path} - 0.1 \cdot h_{aggr}$ could for example be constructed in the following way:

$$h^+_{path} - 0.1 \cdot h_{aggr} : E, a \mapsto \sum_{i=1}^{5} 0.25^i d^+_i(a) - 0.1 \cdot \left|\{a\}^- \setminus E^+\right|$$

```
+ outpath dynindeg 25 -0.1
```

An instance of the outpath-to-aggressor-count-ratio heuristic $h^{\div}_{aggr}$ on the other hand would be constructed in the following way:

$$h^{\div}_{aggr}(E,a) = \frac{\sum_{i=1}^{5} 0.25^i d^+_i(a) + 0.25}{|\{a\}^- \setminus E^+| + 1}$$

```
/ + outpath 0.25 dynindegrat 25
```

Many combinations of heuristics have been tried out during the evaluation of the HEUREKA solver which is described in section 6.2.

6 Evaluation

The different heuristics implemented in the HEUREKA solver are compared among each other to test the overall effectiveness of heuristics and determine good heuristics for specific problems. Some heuristics have parameters which need to be adjusted before this step. To demonstrate the general effectiveness of heuristic backtracking, HEUREKA is additionally compared to two different approaches to reasoning in abstract argumentation which were submitted to ICCMA'17. All experiments which were conducted as part of this evaluation have been run on an eight-core 2 GHz *Intel Xeon E312xx* CPU with 16 GB RAM attached.

For the evaluation of the heuristics implemented in HEUREKA, several testsets have been generated. Each testset consists of several randomly generated graphs. Random graphs are generated with the help of two different methods: Erdős-Rényi random graphs [16] tend to contain many nodes with an average degree and only few nodes with an usually low or an usually high degree, as an edge appears between every pair of two nodes with the same probability. While Erdős-Rényi random graphs are historically as well as recently common and well-studied, it has been observed that graphs which model real-world networks tend to have a scale-free degree distribution. Those graphs contain many nodes with a low degree and a few hubs which have a very high degree. Therefore the Barabási–Albert method for graph generation [1] was also employed for the evaluation, as it was designed to produce graphs with a scale-free degree distribution. Nofal et al. [22] have observed that argumentation frameworks which are hard to backtracking algorithms tend to have a ratio between arguments and attacks in the range of $2\,|\mathcal{A}| < |\mathcal{R}| < 20\,|\mathcal{A}|$. Therefore the generated graphs should have a node-to-edge-ratio inside that range, as the solver should be optimized for hard problems.

6.1 Optimisation of Parameters for Heuristics

Several parameters for different heuristics need to be tuned. Therefore trainingsets are generated and HEUREKA is invoked on them with a number of heuristic variations that only differ in a single parameter. First the parameters that are used to weight paths of different length for the path-based heuristics as well as the maximal path depth are adjusted. Later a feasible window size for dynamic heuristics is determined.

Experiment 6.1. A trainigset of ten Erdős-Rényi random graphs is generated with 250 nodes and 1500 edges each. For the positive path-based heuristic $h^+_{path}(E, a) = \sum_{i=1}^{k} \gamma^i d^+_i(a)$ both the maximal path depth k as well as the weighting parameter γ

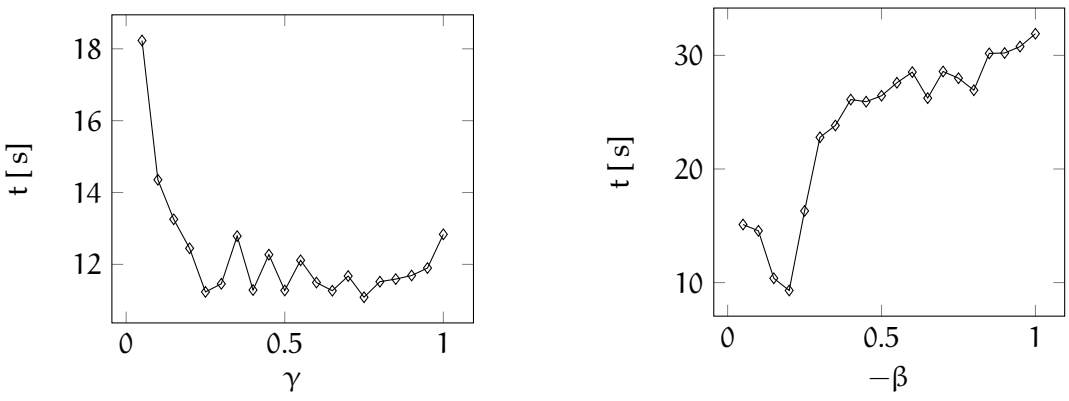


Figure 19: Performance of positive (*left*) and negative (*right*) path-based heuristics with different parameters γ and $-\beta$ as described in Experiments 6.1 and 6.2 respectively

need to be adjusted. With different values of γ in between 0.05 and 1 and a depth of $k = 5$, HEUREKA has been invoked on the trainingset and tasked to enumerate all stable extensions. For each graph the execution time of the solver has been measured with different values γ. With a timeout of ten minutes, all invocations succeeded. The average computation duration for each value γ can be seen in the graph on the left in Figure 19. This graph has three promising local minima at $\gamma = 0.25$, $\gamma = 0.5$ and $\gamma = 0.75$. The experiment was then repeated with these three parameters and different maximal path depths $k = 1, 2, \ldots, 11$. The best results were achieved with the parameter pair $\gamma = 0.25$ and $k = 5$.

Experiment 6.2. For the negative path-based heuristic $h^-_{path}(E, a) = \sum_{i=1}^{k} \beta^i d_i^-(a)$ both the depth k as well as the weighting parameter β need to be adjusted. The same trainingset from Example 6.1 has been used again. With different values of β in between -0.05 and -1, HEUREKA was invoked on the graphs of the trainingset and again tasked to enumerate all stable extensions to measure its performance. After ten minutes, the reasoning process was aborted and counted as a timeout. Out of the ten graphs tested, the computation timed out two times for values $\beta \geq -0.1$ and three times for values $\beta < -0.1$. The graph on the right in Figure 19 visualises the duration of the reasoning process for the seven argumentation frameworks which could be solved with all values of β. A promising value would be $\beta = -0.2$ as it was the fastest to solve the seven easier argumentation frameworks as well as $\beta = -0.1$ as it was the fastest of those who solved eight argumentation frameworks. The ingoing-path-based heuristic was tested with each combination of the parameters $\beta \in \{-0.1, -0.2\}$ and $k = 1, 2, \ldots, 11$. The best results were achieved with the parameter pair $\beta = -0.1$ and $k = 3$.

The values observed in Experiments 6.1 and 6.2 suggest parameters of $\gamma = 0.25$ and $\beta = -0.1$ combined with a maximal path depth of $k = 5$ and $k = 3$ respectively. Based on these results the parameters for path-based heuristics h^+_{Path} and h^-_{Path} are fixed to those values.

$$h^+_{path}(E, a) = \sum_{i=1}^{5} 0.25^i d^+_i(a)$$

$$h^-_{path}(E, a) = \sum_{i=1}^{3} (-0.1)^i d^-_i(a)$$

These results for both maximal path depth as well as weighting parameters suggest that for the heuristics based on ingoing paths only short paths are important. Among outgoing paths on the other hand, somewhat longer paths seem to be relatively influential as well.

Dynamic heuristics require a window size to be specified, which determines how many of the most promising arguments will be considered for every decision. Therefore an experiment is conducted to measure and compare the performance of a dynamic heuristic with different window sizes. This experiment is repeated with testsets containing graphs of various sizes, as the viability of window sizes might strongly depend on the number of arguments per graph.

Experiment 6.3. Three new trainingsets are generated consisting of ten randomly generated graphs each. While all generated graphs are Erdős-Rényi random graphs with approximately 1000 attacks each, the argumentation frameworks of the first trainingsset contain 100 arguments each, those from the second one 200 and those from the third one 400. The number of arguments differ to explore whether the window size scales with the graph size. The window sizes considered lie in between 5 and 75. HEUREKA was invoked on each graph of each trainingset with the aggressor count heuristic $h_{aggr}(E, a)$ pre-sorted according to outdegree resulting in a combined heuristic $\deg^+(a) - h_{aggr}(E, a)$ with different window sizes $k = 5, 10, \ldots, 70, 75$. For the combination of each graph and window size, the execution time of the solver was measured with a timeout of ten minutes. For argument counts 100 and 400 all tests ran within the time limit, while with 200 arguments and a window size $k \geq 35$ HEUREKA timed out once and twice for values $k \geq 55$. The performance of this heuristics with different window sizes is shown in the graphs of Figure 20 for each testset. The optimum is the smallest considered value $k = 5$ for the graphs with 200 or 400 arguments. For graphs with 100 arguments $k = 5$ and $k = 10$ behave approximately equally well. This is surprising as earlier it was assumed that larger argument sets could require larger windows. But this could be explained by tests on the smallest traingset behaving unstable compared to tests on the other trainingsets.

The results from Experiment 6.3 suggest that the window size should be chosen small. Similar results have been observed for bigger graphs. The aggressor count

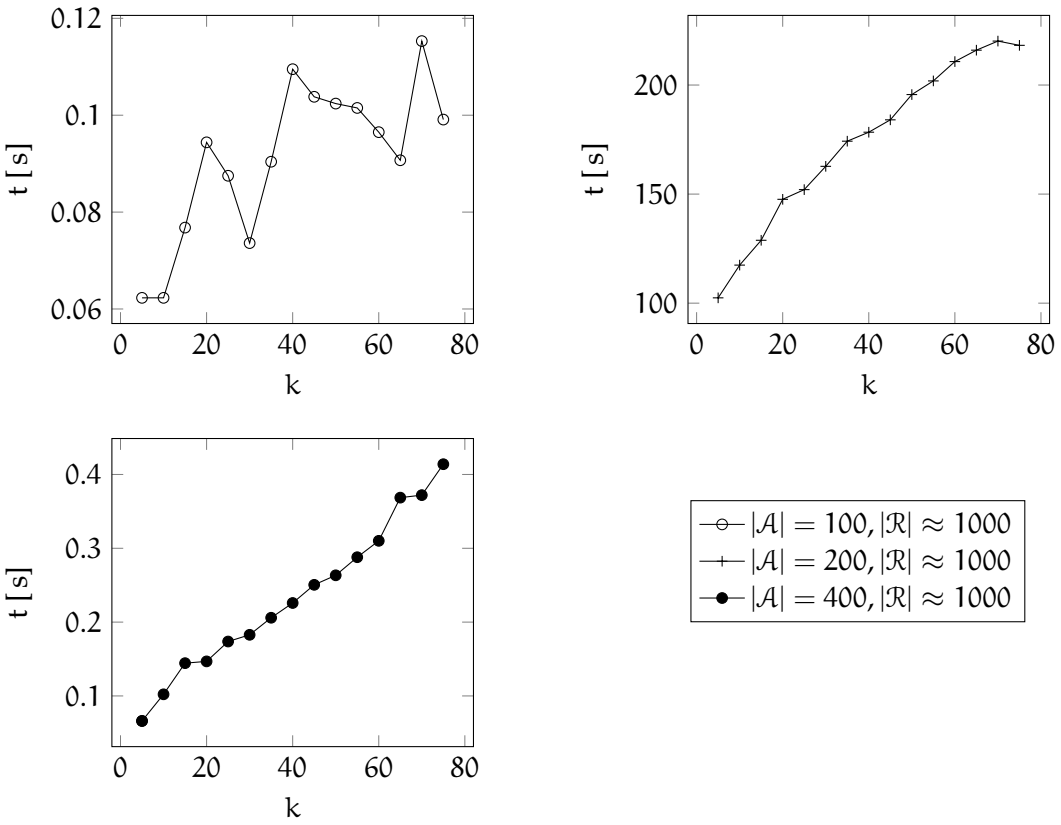


Figure 20: Performance of dynamic aggressor count heuristic with different window size k on graphs with $|\mathcal{A}|$ arguments and $|\mathcal{R}| \approx 1000$ attacks as described in Example 6.3. Each of the three graphs represents a testset, each with a different number of arguments.

heuristic is already the cheapest dynamic heuristic as all algorithms automatically keep track of the number of aggressors of each argument for algorithmic purposes. It can therefore be assumed that other heuristics, which are far more costly to compute, will not work better on large windows. The window size is thus set to 5 for the following experiments, which are conducted to first compare the different heuristics and then to compare HEUREKA to other abstract argumentation solvers.

6.2 Comparison of Heuristics

As a next step, experiments are conducted which compare different heuristics which have been defined in Section 4 and combinations thereof. Thereby we expect to find insights into which heuristics are generally useful and which are not. Additionally for every class of abstract argumentation problems, a heuristic shall be identified

which is suggested to be used with these problems. The HEUREKA solver will then use these heuristics by default to solve the respective problems.

This section concentrates on the tasks of extension set enumeration and finding single extensions as these are corner cases of justification problems. In the best case scenario to solve a justification problem it is sufficient to find a single extension and in the worst case scenario it is necessary to find all extensions. Therefore finding a single extensions is similar to the best case in both sceptical and credulous justification and the same holds for enumerating all extensions and the worst case of any justification problem. Thus if a heuristics works best for both finding and enumerating extensions under certain semantics, it will be suggested to be utilised for justification problems under this semantics also. Problems under grounded semantics and finding a single complete extension are also disregarded as those can be solved using Algorithm 1 alone which does not require a heuristic. This leaves the following relevant problems: enumeration of all extensions under complete, preferred and stable semantics and finding a single extension under preferred and stable semantics.

Several simple and compound heuristics which were regarded promising have been chosen for evaluation. Compound heuristics are characterised by a listing of their components ordered by the expected weight of those components in such a way that the most influential component is listed first. Overall 19 heuristics have been tested, 13 of which are purely static heuristics. First indegree and outdegree have been combined in different ways to define degree-based heuristic, some of which also contain SCC-based components. When the SCC heuristic is utilised, it is the dominant component, as it would not make sense to mix arguments from different strongly connected components. The simple degree ratio heuristics $h^{\div}_{deg}$ has been augmented with a path-based component because it tends to produce large groups of arguments with equal values which need to be distinguished further.

- h^{Σ}_{deg}
- h^{Π}_{deg}
- $(h^{\div}_{deg}, h^{\Sigma}_{path}) = h^{\div}_{deg} + 0.001 \cdot h^{\Sigma}_{path}$
- $h^{\div *}_{deg}$
- $(h_{SCC}, h^{\div}_{deg}, h^{\Sigma}_{path}) = 1000 \cdot h_{SCC} + h^{\div}_{deg} + 0.001 \cdot h^{\Sigma}_{path}$
- $(h_{SCC}, h^{\div *}_{deg}) = 1000 \cdot h_{SCC} + h^{\div *}_{deg}$

Additionally path-based heuristics with and without SCC-based components are also included in the evaluation.

- h^{+}_{path}
- h^{Σ}_{path}
- h^{Π}_{path}
- h^{-}_{path}
- h^{Δ}_{path}
- $h^{\div}_{path}$
- $(h_{SCC}, h^{+}_{path}) = 1000 \cdot h_{SCC} + h^{+}_{path}$

Six of the used heuristics contain dynamic components like the aggressor count heuristic which was used with and without an SCC-based component and the similar outpath-to-aggressor-count-rate heuristic:

- $(h^+_{path}, h_{aggr}) = h^+_{path} - 0.1 \cdot h_{aggr}$
- $h^{\div}_{aggr}$
- $(h_{SCC}, h^+_{path}, h_{aggr}) = 1000 \cdot h_{SCC} + h^+_{path} - 0.1 \cdot h_{aggr}$

A dynamic outgoing-path heuristic emphasizes paths which defend the current solution:

- $(h_{defor}, h^+_{path}) = 10 \cdot h_{defor} + h^+_{path}$
- $(h_{SCC}, h_{defor}, h^+_{path}) = 1000 \cdot h_{SCC} + 10 \cdot h_{defor} + h^+_{path}$

The dynamic matrix exponential heuristic is used with a outgoing-path-based static component as it only considers ingoing paths:

- $(h^d_{exp}, h^+_{path}) = h^d_{exp} + h^+_{path}$

HEUREKA has been evoked with all of the listed heuristics on testsets of different shape to compare the measured runtime using different heuristics. Centrality-based heuristics have been excluded from this evaluation as initial tests have deemed them barely useful. The exponential heuristic has only been evaluated in its dynamic form as the static exponential heuristic h^s_{exp} is very similar to the ingoing-path-based heuristic h^-_{path}.

The algorithm runtime depends heavily on both argument and attack count, but the required effort does not scale monotonically with the graph size. Increasing the number of arguments or attacks of an AAF can either make problems in that AAF harder or easier solvable, as can for example be observed in Figure 20 when comparing the performance on graphs with different argument counts. Therefore to compare the different heuristics abstract argumentation frameworks with differing amounts of arguments and a varying argument-attack-ratio were utilized. The number of attacks was chosen in such a way that it would be hard to solve for backtracking algorithms.

The performance of heuristics for a specific problem will be measured in the following way: HEUREKA will be invoked in combination with a heuristic and tasked to solve that problem for every graph in the testset. If this process succeeds within ten minutes, the runtime is measured and it is counted as a success, otherwise it will count as a timeout. When this experiment has been conducted for all relevant heuristics, the results achieved using each heuristic can be compared. A heuristic which has produced more successful results for a specific problem than another heuristic is considered better fit for that problem than the other heuristic. If two heuristics have returned the same number of successful results, the one that has completed these tasks with a lower average runtime is considered better. The average runtime is the arithmetic mean of the runtime of all successful reasoning processes.

testset	complete/preferred		stable	
	arguments	attacks	arguments	attacks
small	200	~ 1000	200	~ 2000
medium	400	~ 1250	400	~ 2000
large	800	~ 2250	800	~ 3000
scale-free	200	~ 1600	600	~ 4800

Figure 21: Size of the small, medium and large testsets generated for Experiment 6.4 and the scale-free testsets that has been generated for Experiment 6.5.

Experiment 6.4. To compare the different heuristics, testsets of Erdős-Rényi random graphs were generated with 200, 400 and 800 arguments per graph and ten graphs each. For each testset the number of attacks per graph was adjusted for each semantics such that the resulting argumentation frameworks would be hard to solve. Different values were chosen for complete and preferred semantics on one hand and stable semantics on the other hand as problems which are easy under stable semantics might be hard under other semantics. The size of the argument sets and the attack sets of the generated graphs are given in Figure 21. HEUREKA has been invoked on all graphs of all testsets and tasked to enumerate all complete extensions (EE-CO), enumerate all preferred extensions (EE-PR), find a single preferred extension (SE-PR), enumerate all stable extensions (EE-ST) and find a single stable extension (SE-ST). The runtime of the execution has been measured and the process was aborted after ten minutes.

Experiment 6.5. The heuristics have also been tested on scale-free graphs, additionally to the testsets from Experiment 6.4. Again two different testsets have been generated, one for complete and preferred semantics with 200 nodes and approximately 1600 edges and another one for stable semantics with 600 nodes and approximately 4800 edges, as the algorithm which enumerates stable extension works far more efficient than the algorithm which enumerates complete extensions. HEUREKA was again invoked on this testset to solve the relevant tasks EE-CO, EE-PR, SE-PR, EE-ST, SE-ST and the runtime was again limited to ten minutes.

The results which have been obtained from Experiments 6.4 and 6.5 for the different heuristics are listed in Figures 22 and 23. Figure 22 contains the results under complete and preferred semantics while Figure 23 contains the results under stable semantics. When comparing these results one notices that the scores of different heuristics are similar for all problems under complete and preferred semantics on one hand and for all problems under stable semantics on the other hand. At the same time the results for the first group (EE-ST, EE-PR and SE-PR) of problems differs significantly from the results of problems from the second group (EE-ST and SE-ST). From this we can draw the conclusion that different heuristics could perform drastically different in combination with different enumeration algorithm

heuristic	EE-CO		EE-PR		SE-PR	
	ans.	**average**	**ans.**	**average**	**ans.**	**average**
h^{Δ}_{deg}	32	80.8 s	32	78.6 s	36	45.9 s
h^{+}_{path}	35	51.2 s	35	49.9 s	36	24.2 s
h_{SCC}, h^{+}_{path}	35	53.1 s	35	52.0 s	36	25.9 s
h^{-}_{path}	7	74.0 s	7	67.4 s	9	22.7 s
h^{Σ}_{path}	35	60.7 s	35	59.4 s	36	24.2 s
h^{Δ}_{path}	35	44.4 s	35	43.5 s	36	22.1 s
h^{Π}_{deg}	33	54.2 s	33	52.9 s	37	54.4 s
h^{Π}_{path}	**36**	**27.0 s**	**36**	**26.7 s**	**37**	**14.9 s**
$h^{\div}_{deg}, h^{\Sigma}_{path}$	16	157.1 s	16	153.7 s	28	75.2 s
$h_{SCC}, h^{\div}_{deg}, h^{\Sigma}_{path}$	16	153.7 s	16	150.8 s	27	75.5 s
$h^{\div *}_{deg}$	10	79.5 s	10	76.2 s	26	71.4 s
$h_{SCC}, h^{\div *}_{deg}$	10	79.2 s	10	75.7 s	26	70.4 s
$h^{\div}_{path}$	15	118.2 s	16	144.1 s	26	70.3 s
h^{+}_{path}, h_{aggr}	35	65.3 s	35	66.4 s	36	29.0 s
$h_{SCC}, h^{+}_{path}, h_{aggr}$	35	68.4 s	35	68.9 s	36	31.8 s
$h^{\div}_{aggr}$	35	76.8 s	35	76.6 s	36	35.5 s
h_{defor}, h^{+}_{path}	35	88.5 s	35	87.3 s	36	39.8 s
$h_{SCC}, h_{defor}, h^{+}_{path}$	35	74.6 s	35	68.9 s	36	37.7 s
h^{d}_{exp}	10	30.1 s	10	30.3 s	19	45.7 s

Figure 22: Results for complete and preferred semantics from Experiments 6.4 and 6.5 combined. Every column refers to a problem and every row refers to a heuristic. Every entry consists of the number of answers and the average runtime. The number of answers is the number of problems which could be solved before a timeout of ten minutes out of 40. The average runtime is the arithmetic mean of the time that it took to successfully solve these problems.

heuristic	EE-ST		SE-ST		overall	
	ans.	**average**	**ans.**	**average**	**ans.**	**average**
h^{Δ}_{deg}	38	64.0 s	38	58.3 s	176	64.8 s
h^{+}_{path}	39	82.9 s	39	58.4 s	**184**	**53.9 s**
h_{SCC}, h^{+}_{path}	39	92.3 s	39	67.3 s	184	58.9 s
h^{-}_{path}	25	85.2 s	28	78.6 s	76	72.7 s
h^{Σ}_{path}	39	75.5 s	39	51.0 s	184	54.4 s
h^{Δ}_{path}	38	80.8 s	38	55.2 s	182	49.7 s
$h^{\prod}_{deg}$	25	89.9 s	25	68.8 s	153	62.2 s
$h^{\prod}_{path}$	25	93.1 s	26	73.0 s	160	42.0 s
$h^{\div}_{deg}$	**40**	**37.4 s**	**40**	**24.5 s**	140	68.3 s
$h_{SCC}, h^{\div}_{deg}$	40	37.7 s	40	24.7 s	139	67.7 s
$h^{\div *}_{deg}$	40	62.1 s	40	38.9 s	126	59.2 s
$h_{SCC}, h^{\div *}_{deg}$	40	64.8 s	40	40.1 s	126	60.1 s
$h^{\div}_{path}$	37	37.4 s	38	22.8 s	132	61.8 s
h^{+}_{path}, h_{aggr}	38	72.9 s	38	46.8 s	182	56.1 s
$h_{SCC}, h^{+}_{path}, h_{aggr}$	38	80.6 s	38	54.6 s	182	60.9 s
$h^{\div}_{aggr}$	38	77.1 s	38	48.8 s	182	62.8 s
h_{defor}, h^{+}_{path}	38	87.4 s	38	56.8 s	182	71.8 s
$h_{SCC}, h_{defor}, h^{+}_{path}$	37	77.1 s	38	60.7 s	181	64.5 s
h^{d}_{exp}	25	111.9 s	28	56.8 s	92	75.4 s

Figure 23: Results for stable semantics from Experiments 6.4 and 6.5 combined. The column which is titles "overall" combines the results from all five problems, which are listed in this figure and in Figure 22. Every entry consists of the number of answers and the average runtime. The number of answers for the stable problems is the number of problems which could be solved before a timeout of ten minutes out of 40. For the overall score for all five problems the total number of tests is 200 instead. The average runtime is the arithmetic mean of the time that it took to successfully solve these problems.

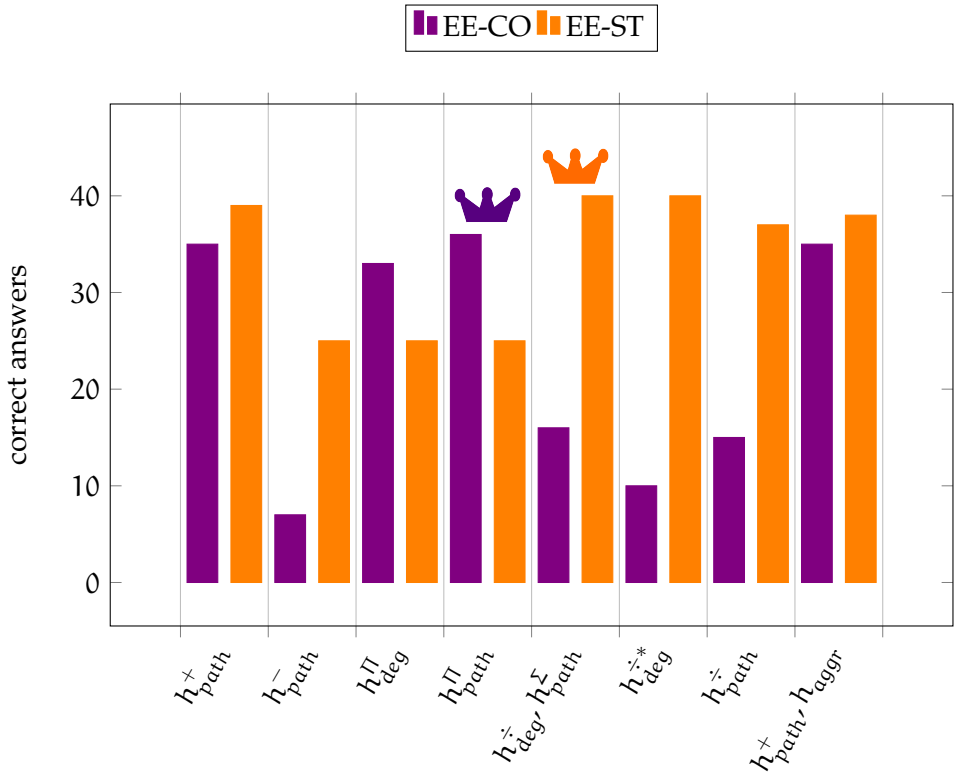


Figure 24: Comparison of the performance of a selection of heuristics under complete and stable semantics. The height of each bar represents the number of answers which were received inside ten minutes during Experiments 6.4 and 6.5 when using the corresponding heuristic. The two heuristics which have proven most fit for a specific tasks have been are marked with a crown in the corresponding colour. Please note that the results under complete and stable semantics respectively cannot be directly compared as they have been obtained using different testsets.

because problems under complete and preferred semantics are handled by Algorithm 5 while problems under stable semantics are handled by Algorithm 7. The different performance of a number of heuristics for both algorithms is visualised by the bar chart in Figure 24, which compares the runtime of complete and stable extension enumeration, which are two exemplaric tasks that are each performed by another one of the two backtracking algorithms.

From the results of Experiments 6.4 and 6.5, conclusions about the fitness of the tested heuristics for abstract argumentation problems shall be drawn. This is first done separately for problems under different semantics as heuristics can perform drastically different under stable semantics than under complete and preferred semantics. A partial order over all heuristics for problems under complete or preferred semantics can be derived from the results which are listed in Figure 22. A heuristic

is considered to have performed better then another heuristic if the algorithm has returned more results before the timeout of ten minutes for all three relevant tasks when using that heuristic. To improve readability, compound heuristics are here denoted by their characteristic component. Student's t-test [18] can be utilised to test whether two series of measurements differ significantly. Compound heuristics with a dominant SCC-based component have been left out because their performance does not differ significantly from the performance of the respective heuristic without that component. The following partial order reflects the results for problems under complete and preferred semantics.

$$h^{\Pi}_{path} > \begin{matrix} h^{+}_{path} \\ h^{\Sigma}_{path} \\ h^{\Delta}_{path} \\ h^{\Pi}_{deg} \\ h_{aggr} \\ h^{\div}_{aggr} \\ h_{defor} \end{matrix} > h^{\Delta}_{deg} \gg h^{\div}_{deg} > h^{\div}_{path} > h^{\div *}_{deg} > h_{exp} > h^{-}_{path}$$

A similar partial order is derived from the results of stable problems given in Figure 23.

$$\begin{matrix} h^{\div}_{deg} \\ h^{\div *}_{deg} \end{matrix} > \begin{matrix} h^{+}_{path} \\ h^{\Sigma}_{path} \end{matrix} > \begin{matrix} h^{\Delta}_{deg} \\ h^{\Delta}_{path} \\ h_{aggr} \\ h^{\div}_{aggr} \\ h_{defor} \end{matrix} > h^{\div}_{path} \gg \begin{matrix} h^{-}_{path} \\ h_{exp} \end{matrix} > h^{\Pi}_{path} > h^{\Pi}_{deg}$$

The two partial orders above can be combined into a third partial order which is given below. Here a heuristic is considered better than another heuristic if it fared better on all tested problems under the relevant semantics (EE-CO, EE-PR, SE-PR, EE-ST and SE-ST).

$$\begin{matrix} h^{+}_{path} \\ h^{\Sigma}_{path} \end{matrix} > \begin{matrix} h^{\Delta}_{path} \\ h_{aggr} \\ h^{\div}_{aggr} \\ h_{defor} \end{matrix} > h^{\Delta}_{deg} > h^{\div}_{path} > h_{exp} > h^{-}_{path}$$

$$\wedge \qquad \wedge$$

$$h^{\div}_{deg} > h^{\div *}_{deg}$$

$$h^{\Pi}_{path} > h^{\Pi}_{deg}$$

There are therefore four heuristics, h^{+}_{path}, h^{Σ}_{path}, h^{Π}_{path} and $h^{\div}_{deg}$ which are not dominated by another heuristic for all problems. At the same time h^{-}_{path} has strictly been least effective out of the tested heuristics.

To choose default heuristics, the best-performing heuristic has been determined for every abstract argumentation problem which has been tested during Experiments 6.4 and 6.5. The heuristic which lead to the fewest timeouts is considered best,

task	CO	PR	ST
EE	h^{Π}_{path}	h^{Π}_{path}	$(h^{\div}_{deg}, h^{\Sigma}_{path})$
SE	-	h^{Π}_{path}	$(h^{\div}_{deg}, h^{\Sigma}_{path})$

Figure 25: The heuristics which yielded the best results for the enumeration of extensions (EE) and single extension (SE) problems for different semantics in Experiments 6.4 and 6.5

when tied the one with the lowest average process runtime is preferred. The best-performing heuristics per problem are depicted in the table in Figure 25. The heuristic $(h^{\div}_{deg}, h^{\Sigma}_{path})$, which sorts arguments according to their indegree-to-outdegree-ratio, has worked best for both enumerating all stable extensions as well as finding a single stable extension. For complete enumeration problems the heuristic h^{Π}_{path}, which combines ingoing and outgoing paths by multiplication, has worked best. This also holds true for enumerating all extensions under preferred semantics and finding a single preferred extension. The choice of the default heuristic therefore depends more on the used algorithm than on the task which is to be solved. All problems which use Algorithm 7, that enumerates stable extensions, performed best with the heuristic $(h^{\div}_{deg}, h^{\Sigma}_{path})$, while all problem which use Algorithm 5, that enumerates complete extensions, performed best with the heuristic h^{Π}_{path}.

Interestingly the ratio-based heuristics, which have scored best on stable semantics, have scored extremely bad when using them with complete or preferred semantics, as can be observed in Figure 24. Another way to explain the extreme differences between problems under different semantics is the fact that different testsets have been used to measure the performance for these problems. Therefore it is possible that the choice of the default heuristic has been optimised towards a specific graph structure. That is the structure of the argumentation frameworks inside the testset used for the evaluation of stable semantics. To clarify this inconsistency the tests regarding stable semantics from Experiment 6.4 have been rerun. But this time these problems have been solved on the smaller graphs from the testsets which have been used for complete and preferred semantics. The size of those graphs are described in the respective column of Figure 21. The results of these tests were again surprisingly ambiguous. While all ratio-based heuristics scored extremely bad when trying to enumerate all stable extensions of the smaller graphs, the same ratio based heuristics were among the best heuristics for searching a single stable extension. Therefore the differences in performance cannot be explained with a biased selection of graphs.

The results of Experiments 6.4 and 6.5, which are given in Figures 22 and 23, additionally allow it to draw general conclusions about the usefulness of several of the tested heuristics and their fitness to be employed together with different semantics. Heuristics with SCC-based components have produced results which are very similar to the same heuristics without these components. While these components

had no significant impact on the performance on Erdős-Rényi random graphs, they brought slight advantages on the scale-free Barabási-Albert random graphs.

When comparing the results of the different purely path-based heuristics that are listed in Figures 22 and 23, it becomes apparent that outgoing paths are a lot more influential than ingoing paths. Especially the outgoing-path-based heuristic h^+_{path} scored a lot better than the ingoing-path-based heuristic h^-_{path}. This holds true for all tested problems and testsets with graphs of different shapes and sizes. When combining h^+_{path} and h^-_{path} additively, one can either choose to weigh h^-_{path} positively resulting in the path sum heuristic $h^\Sigma_{path} = h^+_{path} + h^-_{path}$ or negatively resulting in the path difference heuristic $h^\Delta_{path} = h^+_{path} - h^-_{path}$. Both heuristics favour arguments with many outgoing paths, but while the former tries to avoid vulnerable arguments the latter encourages them to be picked early. Both heuristics work well for all problems, but h^Σ_{path} outperforms h^Δ_{path} under stable semantics while h^Δ_{path} works better under the remaining semantics. This combined with the similar behaviour of ratio-based as well as product-based heuristics suggest that the early inclusion of vulnerable arguments is desirable when using Algorithm 5. At the same time these arguments should be avoided when using Algorithm 7.

Different dynamic heuristics have scored very well when the results of all testsets and abstract argumentation problems are combined. These overall results are listed in the rightmost column in Figure 23. The dynamic heuristics which performed best are in this order: the aggressor count heuristic h_{aggr}, the outpath-to-aggressor-count-ratio heuristic $h^\div_{aggr}$ and the defensor count heuristic h_{defor}. The same ordering could be achieved by ordering the dynamic heuristics by the complexity of the required computations during the algorithm execution. This combined with the results of Experiment 6.3, which suggest small window sizes, gives the impression that dynamic heuristics should avoid expensive computations during algorithm runtime.

Considering the results of the experiments conducted during this section, default heuristics $default_\sigma : \mathcal{P}(\mathcal{A}) \times \mathcal{A} \to \mathbb{R}$ which HEUREKA will utilise to solve problems under semantics σ when no custom heuristic is specified are now defined. For each semantics the heuristic was chosen which performed best under this semantics. This heuristic is additionally augmented with an SCC-based component as these have proven beneficial in a few cases. All chosen default heuristics are purely static.

$$default_{CO}(E, a) = default_{PR}(E, a) = 1000 \cdot h_{SCC}(E, a) - h^-_{path}(E, a) \cdot h^+_{path}(E, a)$$

$$default_{ST}(E, a) = 1000 \cdot h_{SCC}(E, a) + \frac{deg^+(a) + 1}{deg^-(a) + 1} + \frac{h^-_{path}(E, a) + h^+_{path}(E, a)}{1000}$$

$$\text{where } h^-_{path}(E, a) = \sum_{i=1}^{3} (-0.1)^i \cdot d^-_i(a) \text{ and } h^+_{path}(E, a) = \sum_{i=1}^{5} 0.25^i \cdot d^+_i(a)$$

The SCC-based component h_{SCC} is static, it is therefore independent of the partial extension E and its values can be written as $h_{SCC}(a) = h_{SCC}(E, a)$. Additionally h_{SCC} must be implemented in such a way that it fulfils the following constraints:

i) $h_{SCC}(a) = h_{SCC}(b)$ iff $SCC(a) = SCC(b)$, i.e.
the both arguments a and b lie inside the same SCC

ii) if $SCC(a) \neq SCC(b)$ and there is an argument $c \in SCC(a)$ and an argument $d \in SCC(b)$ such that c attacks d, it has to hold that $h_{SCC}(a) > h_{SCC}(b)$

To summarise the observed results, outgoing-path-based heuristics have worked very well for solving problems under all semantics, while ratio-based heuristics have have proven useful for problems under stable semantics, but not useful when using other semantics. Product-based heuristics have on the other hand worked well under all semantics except stable semantics. Several dynamic heuristics have also produced good results for all problems, but have been been surpassed by some static heuristic for all specific purposes. For the following evaluations and as default heuristics for HEUREKA, the degree-ratio-based heuristic $(1000 \cdot h_{SCC} + h^{\div}_{deg} + 0.001 \cdot h^{\Sigma}_{path})$ has been chosen to solve problems under static semantics while the path-product-based heuristic $(1000 \cdot h_{SCC} + h^{\Pi}_{path})$ has been chosen to solve problems under complete and preferred semantics.

6.3 Comparison of Solvers

HEUREKA in combination with the default heuristics which were selected in Section 6.2 is now compared to two other abstract argumentation solvers. Thereby insight is expected to be found into the effectiveness of the implemented algorithms and heuristics for different problem classes and graphs of different sizes and forms. HEUREKA should compare against a backtracking approach which scored well in ICCMA'15 and additionally to a good approach which does not rely on backtracking. Therefore ARGTOOLS [22] and ARGSEMSAT [8] have been chosen to be tested against HEUREKA. Both take part ICCMA'17 and both scored well in the previous competition ICCMA'15. In both cases the version that was submitted to ICCMA'17 is used for this evaluation. ARGTOOLS is a backtracking solver with minimal support for heuristics that came in sixth place in ICCMA'15, which makes it the most successful backtracking solver. HEUREKA builds on several ideas which were introduced in ARGTOOLS. ARGSEMSAT is an approach based on reduction to SAT which came in second place in ICCMA'15. ARGSEMSAT has been chosen because COQUIAAS, the solver which landed on the first place, did not compile on the test system. First the correctness of HEUREKA is tested against the other two solvers:

Experiment 6.6. To test the correctness of the results returned by the HEUREKA solver, a testset of ten easy to solve random graphs is generated, where each graph contains 100 arguments and 300 attacks. The seven different justification problems

were used, which are credulous and sceptical justification under complete, preferred and stable semantics and justification under grounded semantics. HEUREKA, ARGSEMSAT and ARGTOOLS have all been tasked to solve the seven different justification problems for ten arguments in each of the ten graphs. Afterwards the results of the different problems for each argument produced by the different solvers are compared. The results are the same for all solvers on all problems except sceptical justification under stable semantics where the results of ARGTOOLS differ. This could be explained by the fact that ARGTOOLS returned wrong results for sceptical justification under stable semantics in a number of tests in ICCMA'15 [3].

Afterwards an experiment is conducted to measure the performance of HEUREKA and compare it to the other solvers:

Experiment 6.7. The three solvers HEUREKA, ARGTOOLS and ARGSEMSAT have been compared on the testset which was used in ICCMA'15 which can be downloaded from the competition's website [3]. This testset consists of eight subtestsets that fall into three groups which are described there in the following way:

A The first group of testsets consists of graphs with a very large grounded extension and many nodes in general

- A1 features small graphs with a large grounded extension
- A2 features medium-sized graphs with a large grounded extension
- A3 features large graphs with a large grounded extension

B The second group of testsets consists of graphs which feature many complete, preferred and stable extensions

- B1 features small graphs with many extensions
- B2 features medium-sized graphs with many extensions

C The third group of testsets consists of graphs which feature a rich structure of strongly connected components

- C1 features small graphs with many strongly connected components
- C2 features medium-sized graphs with many strongly connected components
- C3 features large graphs with many strongly connected components

Each of these testsets consists of 24 graphs, therefore there are 192 graphs overall. All three solvers have been invoked on every graph once for every problem SE-GR, DC-GR, EE-CO, SE-CO, DS-CO, DC-CO, EE-PR, SE-PR, DS-PR, DC-PR, EE-ST, SE-ST, DS-ST and DC-ST. For every justification problem one argument has been randomly chosen out of every graphs argument set. All three solvers have then been

[3] http://argumentationcompetition.org/2015/results.html

problem	HEUREKA		ARGTOOLS		ARGSEMSAT	
	ans.	average	ans.	average	ans.	average
SE-GR	192	0.1 s	192	4.2 s	191	3.2 s
DC-GR	192	0.1 s	192	4.2 s	191	3.3 s
EE-CO	139	11.1 s	130	11.9 s	184	24.0 s
SE-CO	192	0.1 s	192	4.2 s	192	2.7 s
DS-CO	192	0.1 s	192	4.2 s	192	2.7 s
DC-CO	150	5.3 s	163	27.4 s	192	1.6 s
EE-PR	139	11.0 s	130	10.9 s	191	19.3 s
SE-PR	139	5.5 s	134	10.1 s	192	6.3 s
DS-PR	139	5.7 s	152	13.0 s	192	3.1 s
DC-PR	150	6.9 s	167	30.9 s	192	1.4 s
EE-ST	156	10.0 s	146	26.6 s	188	9.3 s
SE-ST	173	15.9 s	158	71.2 s	192	3.8 s
DS-ST	173	13.5 s	105	62.5 s	191	4.6 s
DC-ST	178	14.6 s	172	41.8 s	191	1.3 s

Figure 26: Results of Experiment 6.7 for each individual task. The column titled "ans." lists the number of correct answers given before a timeout of ten minutes out of a total of 192 invocations per task and the column titles "average" contains the average runtime which was spend to compute these answers.

tasked to justify this argument. Again the runtime of the solvers has been measured and runtimes of more than ten minutes have been counted as timeouts. The results of this experiment are listed in Figure 26 sorted by problem class, and in Figure 27 sorted by the structure of the analysed graphs. ARGTOOLS uses a misleading output format for single extensions which does not conform to the ICCMA'15 or ICCMA'17 standard, it for example reports an empty extensions when it has not been able to find a stable extension. These results have been counted as correct answers as this bug should be able to be easily corrected. ARGTOOLS has returned several wrong answers when performing sceptical justification under stable semantics (SE-ST), these have not been counted as correct answers.

As can be observed from the results listed in Figure 26, which are also illustrated by Figure 6.3, the reduction-based approach ARGSEMSAT could solve nearly all tasks easily, while both backtracking-based approaches HEUREKA and ARGTOOLS struggled with some problems. All three solvers handled problems under grounded semantics well. HEUREKA solved these problems faster than ARGTOOLS which in turn solved them faster than ARGSEMSAT. The same holds true for SE-CO and DS-CO as these problems are similar to SE-GR and DC-GR respectively, except that ARGSEMSAT here performed slightly better than ARGTOOLS. The remain-

testset	HEUREKA		ARGTOOLS		ARGSEMSAT	
	ans.	average	ans.	average	ans.	average
A1	336	0.1 s	334	3.8 s	336	0.6 s
A2	336	0.1 s	335	7.0 s	333	1.8 s
A3	336	0.3 s	327	34.2 s	330	9.6 s
B1	219	37.8 s	242	54.6 s	333	14.5 s
B2	138	15.8 s	139	89.5 s	336	17.9 s
C1	336	0.01 s	314	0.3 s	335	0.3 s
C2	301	3.6 s	273	13.6 s	333	0.3 s
C3	302	13.5 s	263	15.5 s	335	2.5 s

Figure 27: Results of Experiment 6.7 for each individual testset. The column titled "ans." lists the number of correct answers given before a timeout of ten minutes out of a total of 336 invocations per testset and the column titles "average" contains the average runtime which was spend to compute these answers.

ing two problems under complete semantics were solved fastest by ARGSEMSAT. Among the backtracking-based solvers, HEUREKA outperformed ARGTOOLS for EE-CO which in turn outperformed HEUREKA in DC-CO. A similar picture shows for preferred semantics: ARGSEMSAT dominates, while the other two solvers produce mixed results. Again HEUREKA solved some problems better (EE-PR and SE-PR) while it is outperformed by ARGTOOLS regarding the justification problems (DS-PR and DC-PR). Problems under stable semantics were solved best by ARGSEM-SAT again, which performed better than HEUREKA which in turn performed better than ARGTOOLS for all problems under stable semantics. This trend was intensified by the fact that ARGTOOLS returned several wrong answers under stable semantics. Problems under grounded and stable semantics can therefore be identified as relative strengths of HEUREKA, while justification tasks under preferred semantics and credulous justification under complete semantics would be its particular weaknesses.

All three solvers have also performed differently on the different testsets which contain graphs of various sizes and with different structural properties. Results for the different testsets are listed in Figure 27 and are also illustrated by Figure 6.3. The testsets A1, A2 and A3 contain graphs which are relatively easy to solve but are tendentially larger than the graphs from other testsets. HEUREKA performed best on these graphs. It solved all problems within the time limit while the other solvers struggled especially with large graphs. The two testsets B1 and B2 contain especially hard graphs which contain many extension and caused problems for the backtracking-based solvers. Here out of all three tested solvers HEUREKA worked worst. Testsets C1, C2 and C3 contain graphs with many strongly connected com-

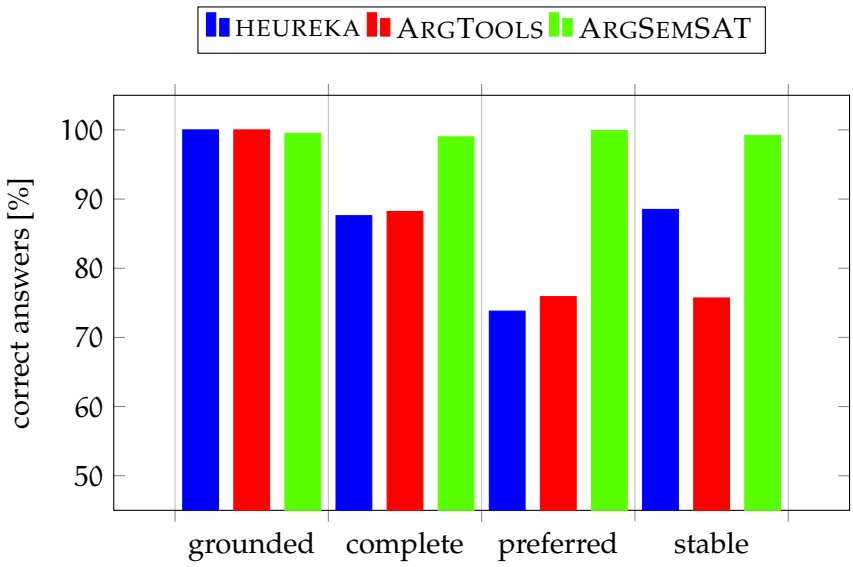


Figure 28: Share of correct answers within ten minutes among all invocations of different solvers when tasked to solve problems under specific semantics given in percentage. The underlying data has been measured during Experiment 6.7, the results broken down by individual tasks can be found in the table in Figure 26.

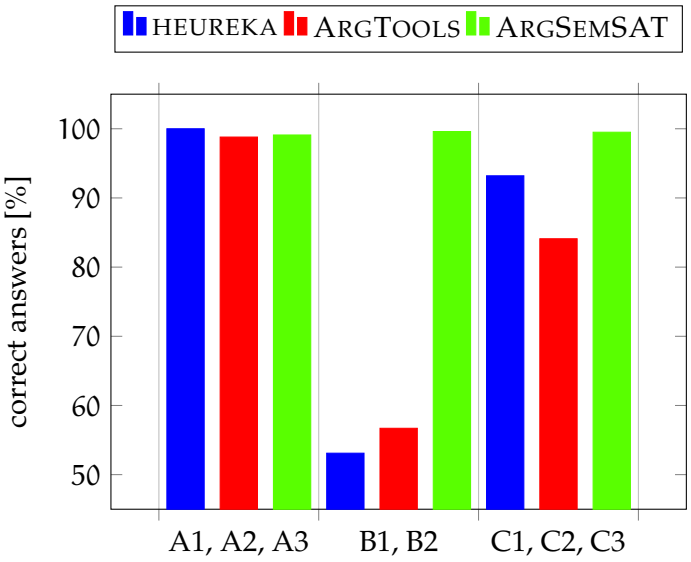


Figure 29: Share of correct answers within ten minutes among all invocations of different solvers on different classes of graphs given in percentage. The underlying data has been measured during Experiment 6.7, the results broken down by individual testsets can be found in the table in Figure 27.

ponents. On these graphs HEUREKA worked better than ARGTOOLS but worse then ARGSEMSAT.

When combining the all results obtained in Experiment 6.7, the following overall numbers of correct answers out of 2688 problems are observed:

ARGSEMSAT	HEUREKA	ARGTOOLS
2671	2304	2225

The augmentation of backtracking algorithms with heuristic guidance could therefore not yield results which surpass the performance of reduction-based solvers, but it has proven to have the possibility to substantially increase the performance of backtracking algorithms. It could thus help to make the direct approach to abstract argumentation more viable.

7 Conclusion

Various heuristics have been defined in the course of this thesis and implemented as part of the HEUREKA solver. Promising approaches include heuristics based on node degree as well as a node's in- or outgoing paths. Additionally heuristics which order arguments according to their strongly connected components have worked well on some argumentation frameworks. The performance of several different heuristics has been measured under complete, preferred and stable semantics. Heuristics have performed similarly under complete and preferred semantics, but drastically different under stable semantics. This is not surprising as a separate algorithm is utilised to solve problems under stable semantics. The algorithm which searches stable extensions has proven to be generally more effective than the algorithm which is used to solve problems under preferred semantics and most problems under complete semantics. Both algorithms have different requirements for heuristics as several heuristics have performed extremely different in combination with these algorithms.

For each specific abstract argumentation problem, some static heuristics performed better than all dynamic heuristics. Among those path-based heuristics with a dominant outgoing-path-based component have constantly shown a solid performance under all semantics. But under stable semantics these have been outperformed by degree ratio heuristics while path product heuristics have worked better for complete and preferred semantics. For each relevant problem recommended heuristics have been identified. These are $default_{ST}$ for all problems under stable semantics and $default_{CO} = default_{PR}$ for all relevant problems under complete

and preferred semantics. These recommended heuristics are used as default heuristics by the HEUREKA solver.

$$\begin{aligned} default_{CO} &= 1000 \cdot h_{SCC} + h^{\Pi}_{path} \\ default_{PR} &= 1000 \cdot h_{SCC} + h^{\Pi}_{path} \\ default_{ST} &= 1000 \cdot h_{SCC} + h^{\div}_{deg} + 0.001 \cdot h^{\Sigma}_{path} \end{aligned}$$

The effectiveness of an argument is its potential to defeat or defend other arguments. Outgoing paths increase the effectiveness of an argument. Effectiveness has proven a favourable property for all purposes and thus heuristics should try to prioritise arguments with this property. Ingoing attacks on the other hand make it harder to defend an argument and thereby increase its vulnerability. The early inclusion of arguments which are vulnerable against attacks however is only beneficial under complete and preferred semantics. The stable algorithm works better when these arguments are avoided.

HEUREKA has performed relatively strong for problems under grounded and stable semantics, while it has performed relatively weak for justification tasks under preferred semantics and credulous justification under complete semantics. A way to overcome these weaknesses would be further customisation of the backtracking search algorithm for those problems, which could require a separate algorithm for preferred semantics. Another strength of HEUREKA is its solid performance on large argumentation frameworks with several thousand arguments.

Great possibilities to further improve backtracking algorithms for abstract argumentation might lie in further development and refinement of dynamic heuristics. All dynamic heuristic that have been introduced in the course of this thesis relied solely on the distinction of arguments which are inside or outside of the partial solution, i.e. arguments which are labelled IN on one hand and the remaining arguments on the other hand. Further research might include the definition and evaluation of dynamic heuristics which additionally distinguish between arguments which are labelled OUT and UNDEC. Another way to improve the performance of a heuristic-backtracking-based abstract argumentation solver would be to let the solver first analyse the structure of the AAF and then choose a heuristic based on the abstract argumentation problem as well as structural properties of the AAF. Such properties could for example be the degree distribution of the AAF as the conducted experiments show that some heuristics have performed quite differently on graphs with short and long tailed degree distribution. Additionally to the four semantics originally defined by dung several alternative semantics have been defined, further research could also focus on the influence of different heuristics on the performance of backtracking algorithms under these semantics.

Overall heuristics have proven useful to guide backtracking algorithms for abstract argumentation and there are several possible improvements which could help to further increase their impact on algorithm performance.

References

[1] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[2] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *The Knowledge Engineering Review*, 26(04):365–410, 2011.

[3] Pietro Baroni, Massimiliano Giacomin, and Giovanni Guida. SCC-recursiveness: a general schema for argumentation semantics. *Artificial Intelligence*, 168(1-2):162–210, 2005.

[4] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[5] Martin Caminada. Comparing two unique extension semantics for formal argumentation: ideal and eager. In *Proceedings of the 19th Belgian-Dutch conference on artificial intelligence (BNAIC 2007)*, pages 81–87, 2007.

[6] Martin W. A. Caminada, Walter A. Carnielli, and Paul E. Dunne. Semi-stable semantics. *Journal of Logic and Computation*, page exr033, 2011.

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

[8] Federico Cerutti, Massimiliano Giacomin, and Mauro Vallati. Argsemsat: Solving argumentation problems using sat. *COMMA*, 14:455–456, 2014.

[9] Federico Cerutti, Mauro Vallati, and Massimiliano Giacomin. Where are we now? state of the art and future trends of solvers for hard argumentation problems. *Computational Models of Argument: Proceedings of COMMA 2016*, 287:207, 2016.

[10] Günther Charwat, Wolfgang Dvořák, Sarah A. Gaggl, Johannes P. Wallner, and Stefan Woltran. Methods for solving reasoning problems in abstract argumentation–a survey. *Artificial intelligence*, 220:28–63, 2015.

[11] Carl Corea and Matthias Thimm. Using matrix exponentials for abstract argumentation. In *Proceedings of the First Workshop on Systems and Applications of Formal Argumentation (SAFA'16)*, pages 10–21, September 2016.

[12] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer, Berlin, Heidelberg, 2008.

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

[14] Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. Assumption-based argumentation., 2009.

[15] Wolfgang Dvorák. Computational aspects of abstract argumentation. 2012.

[16] Paul Erdős and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[17] Sarah A. Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran. The second international competition on computational models of argumentation (ICCMA'17). 2016.

[18] Udo Kuckartz, Stefan Rädiker, Thomas Ebert, and Julia Schehl. *Statistik: eine verständliche Einführung*. Springer-Verlag, 2013.

[19] Beishui Liao, Liyun Lei, and Jianhua Dai. Computing preferred labellings by exploiting sccs and most sceptically rejected arguments. In *Theory and Applications of Formal Argumentation: Second International Workshop, TAFA 2013, Beijing, China, August 3-5, 2013, Revised Selected Papers*, volume 8306, page 194. Springer, 2014.

[20] Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In *Argumentation in artificial intelligence*, pages 105–129. Springer, 2009.

[21] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Algorithms for argumentation semantics: labeling attacks as a generalization of labeling arguments. *Journal of Artificial Intelligence Research*, 49:635–668, 2014.

[22] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Looking-ahead in backtracking algorithms for abstract argumentation. *International Journal of Approximate Reasoning*, 78:265–282, 2016.

[23] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[24] Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1-2):81–132, 1980.

[25] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[26] Bart Verheij. Two approaches to dialectical argumentation: admissible sets and argumentation stages. *Proc. NAIC*, 96:357–368, 1996.