FernUniversität in Hagen
Fakultät für Mathematik und Informatik

AIG
Artificial Intelligence Group

Schlussfolgern in Argumentationsgraphen durch randomisierte Wahl von Verteidigern

Bachelorarbeit

zur Erlangung des Grades einer Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Carina Sophie Benzin

Erstgutachter:	Prof. Dr. Matthias Thimm Artificial Intelligence Group
Betreuer:	Prof. Dr. Matthias Thimm Artificial Intelligence Group

Erklärung

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

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☒	☐
Der Veröffentlichung dieser Arbeit auf der Webseite des Lehrgebiets Künstliche Intelligenz stimme ich zu.	☒	☐
Der Text dieser Arbeit ist unter einer Creative Commons Lizenz (CC BY-SA 4.0) verfügbar.	☒	☐
Der Quellcode ist unter einer GNU General Public License (GPLv3) verfügbar.	☒	☐
Die erhobenen Daten sind unter einer Creative Commons Lizenz (CC BY-SA 4.0) verfügbar.	☒	☐

Crivitz, 04.09.2023 C. Bu

(Ort, Datum) (Unterschrift)

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Konzeption, Implementation und Evaluation eines Argumentationslösers. Das Hauptinteresse liegt auf der Anwendung der stochastischen Lokalsuche zur Bewältigung des Problems der leichtgläubigen Akzeptanz von zulässigen Mengen. Das zugrunde liegende Prinzip ist der sukzessive Aufbau einer zulässigen Menge durch Hinzufügen von randomisiert ausgewählten Verteidigern. Um dieses Konzept umzusetzen, wurde ein Algorithmus in C++ entwickelt[1]. Dieser wird vorgestellt. Die Arbeit endet mit einer Darstellung der durchgeführten Experimente und ihrer Auswertung.

Abstract

This thesis deals with the conception, implementation and evaluation of an argumentation solver. The focus of this work is examining whether the approach of stochastic local search is competitive in solving the credulous acceptance problem regarding admissible sets. To solve this problem, an admissible set is gradually constructed by adding randomly selected defenders. The algorithm developed to realize this concept[2] is introduced. To conclude, the conducted experiments and their evaluation are presented.

[1] Auf Github verfügbar: https://github.com/gigarina/taas-fudge (Im *main-Branch* unter dem *Tag* 'abgabe' auffindbar.)

[2] Available on Github: https://github.com/gigarina/taas-fudge (In the main-branch with the tag 'abgabe'.)

Inhaltsverzeichnis

1 Einleitung

Abstrakte Argumentation behandelt das Zusammenspiel von Aussagen in Diskussionen. Von den Argumenten wird inhaltlich abstrahiert. Es werden allein die Beziehungen zwischen den Argumenten betrachtet. Das Thema wurde erstmals in [Dun95] beschrieben. Der Kerngedanke der Konzentration auf die Beziehungen der Argumente gründet dabei auf dem folgenden Prinzip: Ein Argument A ist glaubwürdig, wenn Gegenargumente mit Argumenten, die A unterstützen, entkräftet werden können [Dun95]. In anderen Worten kann die Glaubwürdigkeit eines Argumentes daran festgemacht werden, ob der Standpunkt, zu dem das Argument gehört, verteidigt werden kann. Somit bietet abstrakte Argumentation einen Ansatz zur Analyse unterschiedlichster Gegebenheiten der realen Welt. Mögliche Anwendungsgebiete reichen von der Analyse von Diskussionen in Online-Foren bis zur Rechtssprechung.

Abstrakte Argumentationsgraphen bieten einen Formalismus, um abstrakte Argumentation anwendbar zu machen. Bei dieser Datenstruktur handelt es sich um einen gerichteten Graphen. Die Knoten entsprechen Argumenten. Eine gerichtete Kante eines Knotens zu einem anderen stellt einen Angriff vom ersten auf den zweiten Knoten dar [Dun95]. Diese Relation führt zu sogenannten Extensionen. Dies sind Mengen von Argumenten, die bestimmte Bedingungen erfüllen und somit einen Standpunkt bilden.

In dieser Arbeit soll ein Algorithmus konzipiert, implementiert und evaluiert werden, der für einen gegebenen Argumentationsgraphen und eines seiner Argumente entscheidet, ob das gegebene Argument in einer zulässigen Extension enthalten ist. Eine zulässige Extension ist ein Standpunkt, der nicht widersprüchlich ist und gegen alle Gegenargumente verteidigt werden kann [Dun95, CDG+15]. Dieses Entscheidungsproblem ist unter dem Namen des leichtgläubigen Akzeptanzproblems bezüglich zulässiger Extensionen (engl. *credulous acceptance problem regarding admissible extensions*) bekannt [LLMR21]. In der Literatur werden zahlreiche vollständige Algorithmen für die Lösung von Problemen dieser Art beschrieben. Im Gegensatz dazu ist der Ansatz der vorliegenden Arbeit einen randomisierten, unvollständigen Algorithmus zu entwickeln und dessen Performanz experimentell festzustellen.

Die Entwicklung von Algorithmen zur abstrakten Argumentation wird wissenschaftlich vor allem durch die *International Competition of Computational Models of Argumentation* (ICCMA) vorangetrieben. Die ICCMA besteht seit 2015. Es ist ein Wettkampf, bei dem Algorithmen zur abstrakten Argumentation gegeneinander antreten. Solche Algorithmen werden auch Argumentationslöser (engl. *argumentation solver*) genannt. Mit den durchgeführten Wettkämpfen erfasst die ICCMA den Stand der Technik von Argumentationsformalismen [JLN23]. Aus diesem Grund werden die Argumentationslöser, die in den durchgeführten Experimenten als Vergleich dienen, aus der letzten Instanz der ICCMA entnommen. Zu diesem Zeitpunkt ist der letzte abgeschlossene Wettkampf die ICCMA21 [Thi19].

2 Abstrakte Argumentation

Bevor die Ergebnisse dieser Arbeit dargestellt werden können, folgen einige formale Definitionen. Die Basis der Bearbeitung sind abstrakte Argumentationsgraphen, welche bereits informell dargestellt wurden. Die folgenden drei Definitionen gründen auf [Dun95].

Definition 1 (Abstrakter Argumentationsgraph)**.** *Ein gerichteter Graph $AF = \langle R, T \rangle$ mit*
R - Menge von Argumenten
T - die Attacken zwischen den Argumenten. Formal eine binäre Relation zwischen den Elementen von R, also $T \subseteq R \times R$.

Abstrakte Argumentationsgraphen werden nach ihren englischen Namensgebern *abstract argumentation frameworks* in der Literatur mit AF abgekürzt [Dun95, CDG+15]. Zum allgemeinen Verständnis wird diese Abkürzung auch in dieser Arbeit übernommen. Da das Wort Framework im Deutschen ein Neutrum ist[3], wird für die Abkürzung AF der Artikel 'das' verwendet, also: das AF.

Beispiel 2.1. Sei $F = \langle R, T \rangle$ ein AF mit R={0, 1, 2} und T={(1, 2), (2, 0)}. F ist in Abb. 1 als Graph dargestellt. Es ist deutlich erkennbar, dass R die Knoten des Graphen beschreibt und T die Kanten.

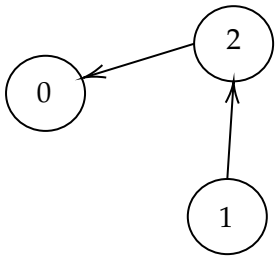

Abbildung 1: AF aus Beispiel 2.1

In AFs können verschiedene Arten von Argument-Mengen gebildet werden. Im Mittelpunkt dieser Arbeit stehen die zulässigen Mengen. Um zulässige Mengen zu definieren, müssen die Begriffe Konfliktfreiheit und Akzeptierbarkeit in abstrakten Argumentationsgraphen betrachtet werden.

Definition 2 (Konfliktfreiheit)**.** *Sei $AF = \langle R, T \rangle$ ein Argumentationsgraph. Eine Menge von Argumenten S $\subseteq$ R ist konfliktfrei, wenn es keine zwei Argumente A,B in S gibt, die sich gegenseitig angreifen.*
Formal: $S \subseteq R : \nexists\, A, B \in S : (A, B) \in T$

Beispiel 2.2. In dem AF aus Abb. 1 sind die Argument-Mengen {0}, {1}, {2} und {0,1} konfliktfrei.

[3] Siehe [oJ]

Definition 3 (Akzeptierbarkeit)**.** *Sei $AF = \langle R, T \rangle$ ein Argumentationsgraph. Ein Argument $A \in R$ ist akzeptierbar im Bezug auf eine Menge $S \subseteq R$ von Argumenten, wenn A von S verteidigt wird. Das heißt, wenn gilt: $A, B \in R : (B, A) \in T : \exists C \in S : (C, B) \in T$.*

Beispiel 2.3. In dem AF aus Abb. 1 ist das Argument 0 akzeptierbar im Bezug auf die Menge {1}, da {1} das Argument 0 vor Argument 2 verteidigt.

Zulässigkeit (engl. *admissibility*) ist eine Semantik von Argumentationsgraphen. Eine Semantik σ weist einem Argumentationsgraphen $AF = \langle R, T \rangle$ eine Menge von Argumenten $\sigma(AF) \subseteq R \times R$ zu. Welche Argumente der Menge zugewiesen werden, ist von der verwendeten Semantik σ abhängig [CDG$^+$15]. Eine weniger formelle Betrachtungsweise ist es Semantiken als Lösung für das Problem, welche Argumente bzw. Argumentmengen akzeptiert werden können, zu sehen. Je nach Formulierung der Semantik können unterschiedliche Argumente bzw. Argumentmengen akzeptiert werden [BCG11]. Die akzeptierten Argumente sind Elemente der durch $\sigma(AF)$ definierten Menge. In dieser Arbeit steht die Semantik der zulässigen Extensionen im Fokus.

Definition 4 (Zulässige Extension)**.** *Eine Menge S von Argumenten in einem Argumentationsgraphen F ist eine zulässige Extension, wenn sie konfliktfrei ist und S akzeptierbar im Bezug auf sich selbst ist.*

Um dieses zentrale Konzept zu verdeutlichen, folgt ein Beispiel.

Beispiel 2.4. Im AF aus Abb. 1 sind die zulässigen Extensionen die folgenden Argument-Mengen: {1}, {0, 1}.

Die Begriffe zulässige Extension und zulässige Menge werden im Weiteren synonym verwendet, beides beschreibt eine Argument-Menge nach Defnition 4.

In dieser Arbeit geht es um die Lösung des leichtgläubigen Akzeptanzproblems bezüglich zulässiger Mengen. Dieses Problem kann formal wie folgt definiert werden.

Definition 5 (Leichtgläubiges Akzeptanzproblem bezüglich zulässiger Extensionen)**.** *Sei $AF = \langle R, T \rangle$ ein Argumentationsgraph mit $a \in R$. Das leichtgläubige Akzeptanzproblem bezüglich zulässiger Mengen ist ein Entscheidungsproblem $Cred_{adm}(a, AF)$, welches erfüllt ist, wenn das Argument a in mindestens einer zulässigen Menge des Argumentationsgraphen enthalten ist. In allen anderen Fällen ist es nicht erfüllt. [CDG$^+$15]*

Beispiel 2.5. Gegeben sei das AF aus Abb. 1. Das leichtgläubige Akzeptanzproblem bezüglich zulässiger Extensionen ist für die Argumente 0 und 1 erfüllt, da beide in mindestens einer zulässigen Extension enthalten sind (vergl. Beisp. 2.4). Argument 2 hingegen kann bezüglich zulässiger Mengen nicht leichtgläubig akzeptiert werden, da es nicht vor dem Angriff von Argument 1 verteidigt wird.

Weitere grundlegende Semantiken im Forschungsgebiet der abstrakten Argumenation sind vollständige Extensionen (engl. *complete Extensions*) und präferierte Extensionen (engl. *preferred Extensions*). Beide Definitionen sind in Anlehnung an [Dun95] und [CDG$^+$15] formuliert.

Definition 6 (vollständige Extension)**.** *Eine zulässige Menge S, bei dem jedes Argument, das im Bezug auf S akzeptierbar ist, in S enthalten ist. Es gilt also: E ist eine vollständige Extension genau dann, wenn E zulässig ist und ∀ Argumente a: a ist akzeptierbar im Bezug auf E: a ∈ E.*

Definition 7 (präferierte Extension)**.** *Eine maximale zulässige Menge. Sei S eine präferierte Extension, dann gilt: ∄ zulässige Menge Z: $S \subset Z$.*

Die Ergebnisse der leichtgläubigen Akzeptanz von zulässigen, vollständigen und präferierten Extensionen sind immer identisch. Das liegt daran, dass vollständige und präferierte Extensionen per Definition zulässig sind. Für vollständige Extensionen gilt: Wenn ein Argument A eines AFs in einer zulässigen Menge S ist, dann kann diese durch hinzufügen von akzeptierbaren Argumenten in Bezug auf S erweitert werden, bis S vollständig ist. Damit ist A auch bezüglich vollständiger Extensionen leichtgläubig akzeptierbar. Für präferierte Exensionen ist in [Dun95] dargestellt, dass alle zulässigen Mengen in einem AF in mindestens einer präferierten Extension enthalten sind. Daraus folgt, dass alle Argumente, die bezüglich zulässiger Extensionen leichtgläubig akzeptiert werden, auch bezüglich präferierter Extensionen leichtgläubig akzeptiert werden.

Da leichtgläubige Akzeptanz bezüglich zulässiger Extensionen in der ICCMA nicht getestet wird, können für den Performanzvergleich die Lösung der leichtgläubigen Akzeptanz bezüglich vollständiger und präferierter Extensionen genutzt werden.

Die in der ICCMA getesteten Argumentationsprobleme werden als Aufgaben bezeichnet. Das Entscheiden der leichtgläubigen Akzeptanz wird mit DC für *Decide Credulous acceptance* also entscheide leichtgläubige Akzeptanz abgekürzt. Darauf folgen ein Bindestrich und eine Abkürzung für die betrachtete Semantik. Die Entscheidung des leichtgläubigen Akzeptanzproblems bezüglich vollständiger Extensionen wird somit als DC-CO das bezüglich präferierter Extensionen mit DC-PR abgekürzt [LLMR21]. Die Definition der Aufgaben DC-CO und DC-PR wird nachfolgend erklärt. (in Anlehnung an [LLMR21]).

Definition 8 (DC-CO)**.** *Input: Ein Argumentationsgraph $AF = \langle R, T \rangle$, ein Argument A ∈ R*
Output: YES, wenn ∃ vollständige Extension E: A ∈ E . NO sonst.

Definition 9 (DC-PR)**.** *Input: Ein Argumentationsgraph $AF = \langle R, T \rangle$, ein Argument A ∈ R*
Output: YES, wenn ∃ präferierte Extension E: A ∈ E . NO sonst.

Wir definieren für den in dieser Arbeit betrachteten Anwendungsfall eine Aufgabe zur Lösung des leichtgläubigen Akzeptanzproblems bezüglich zulässiger Extensionen.

Definition 10 (DC-ADM)**.** *Input: Ein Argumentationsgraph $AF = \langle R, T \rangle$, ein Argument $A \in R$*
Output: YES, wenn ∃ zulässige Extension E: A ∈ E . NO sonst.

Das DC-ADM Problem ist NP-Vollständig [DBC02, DM04]. Dieses Hindernis muss in dieser Ausarbeitung überwunden werden.

Es gibt verschiedene Möglichkeiten, das Problem der leichtgläubigen Akzeptanz bezüglich zulässiger Mengen zu lösen. Wie in [Rod18, CGTW17] beschrieben, können Argumentationslöser in zwei Kategorien unterteilt werden. Auf der einen Seite finden sich Algorithmen, die auf Reduktion basieren, auf der anderen Seite Algorithmen, die direkt im Argumentationsgraphen agieren.

Es gibt für andere Formalismen wie z. B. SAT gut erforschte Algorithmen, die effizient arbeiten. Diese Tatsache nutzen die Argumentationslöser aus, die auf Reduktion basieren. Sie reduzieren das Problem des Argumentationsgraphen auf ein Problem in einem anderen Formalismus, für den es bereits effiziente Algorithmen gibt und nutzen das Ergebnis dieser Algorithmen als ihr eigenes.

In dieser Arbeit nehmen wir den direkten Ansatz. Dieser unterscheidet sich von dem Ansatz der Reduktion in der Hinsicht, dass das Problem nicht umgewandelt wird, sondern im Formalismus von abstrakten Argumentationsgraphen bestehen bleibt [Rod18, CGTW17]. Bei der Umformung der Argumentationsgraphen im Reduktionsansatz kann es zu einem Informationsverlust kommen.

Der direkte Ansatz bietet den Vorteil, dass die Eigenschaften von Argumentationsgraphen in der Problem-Lösung direkt genutzt werden können. Das kann zur Beschleunigung der Lösungsfindung führen, da ein Informationsvorsprung besteht und die formalen Eigenschaften von Argumentationsgraphen ausgenutzt werden können [CDG+15].

2.1 Kennzeichnungen in abstrakter Argumentation

Viele direkte Ansätze nutzen Kennzeichnungen (engl. *labelings*). Dabei werden die einzelnen Knoten des Argumentationsgraphen Schritt für Schritt mit den Kennzeichnungen `in`, `out` und `undec` versehen [Cam06], die sich in jeder Iteration des Algorithmus ändern. Normalerweise terminieren solche Algorithmen, nachdem keine weiteren Änderungen auftreten [CDG+15]. Die Kennzeichnung eines Arguments beschreibt, ob das Argument in der derzeitigen Iteration akzeptiert wird (`in`) oder nicht (`out`) oder ob der Status der Akzeptanz noch unentschieden ist (`undec`).

Kennzeichnungen haben die Charakteristik, dass die Kennzeichnung von einem Argument die Kennzeichnung seiner Nachbarn beeinflusst. Der Grund hierfür ist wie folgt. Zulässige, präferierte und vollständige Extensionen haben ganz grundlegend gemeinsam, dass sie konfliktfreie Mengen von Argumenten darstellen. Formal bedeutet das: Sämtliche Semantiken von Interesse basieren in abstrakter Argumentation alle auf Konfliktfreiheit. Daraus lässt sich folgern, dass, wenn ein Argument a als `in` gekennzeichnet wird, alle seine Nachbarn als `out` gekennzeichnet werden müssen, um Konfliktfreiheit in der Extension sicherzustellen. Da durch Kennzeichnungen diese Eigenschaft von Argumentationsgraphen zur schnelleren Lösungsfindung ausgenutzt werden können, hebt [CDG+15] den Kennzeichnungsansatz als einen Vorteil des direkten Ansatzes zur Lösung von Argumentations-Problemen

hervor.

Nach [CG09] ist eine Kennzeichnung wie folgt definiert.

Definition 11 (Kennzeichnung)**.** *Angenommen $F = \langle R, T \rangle$ ist ein AF. In diesem Kontext ist eine Kennzeichnung eine vollständige Funktion: $\mathcal{L} : R \to \{\texttt{in}, \texttt{out}, \texttt{undec}\}$*

Im folgenden verwenden wir nach [Cam06] $\texttt{in}(\mathcal{L})$ für $\{A|\mathcal{L}(A) = \texttt{in}\}, \texttt{out}(\mathcal{L})$ für $\{A|\mathcal{L}(A) = \texttt{out}\}$ und $\texttt{undec}(\mathcal{L})$ für $\{A|\mathcal{L}(A) = \texttt{undec}\}$. Eine Kennzeichnung ist zulässig, wenn folgendes gilt:

Definition 12 (Zulässige Kennzeichnung)**.** *Sei $AF = \langle R, T \rangle$ ein AF und L seine Kennzeichnung. L ist eine zulässige Kennzeichnung, wenn die folgenden Voraussetzungen gelten:*

1. *Wenn $\mathcal{L}(A) = \texttt{in}$, dann gilt für alle Angreifer von A, dass sie als $\texttt{out}$ gekennzeichnet sind. Formal: $\forall B \in R : B \to A : \mathcal{L}(B) = \texttt{out}$*
2. *Wenn $\mathcal{L}(A) = \texttt{out}$, dann gibt es mindestens einen Angreifer von A, der als $\texttt{in}$ gekennzeichnet ist. Formal: $\exists B \in R : B \to A : \mathcal{L}(B) = \texttt{in}$*

Zulässige Mengen und zulässige Kennzeichnungen sind äquivalent. Man kann eine zulässige Menge S als Kennzeichnung darstellen, indem alle Argumente $\in$ S als $\texttt{in}$ gekennzeichnet sind. Alle Angreifer der Argumente aus $\texttt{in}(\mathcal{L})$ werden als $\texttt{out}$ gekennzeichnet und alle übrigen Elemente bleiben $\texttt{undec}$ [CDG$^+$15, CGTW17]. Da zulässige Mengen und Kennzeichnungen äquivalent sind, ist der Kennzeichnungsansatz auch für diese Ausarbeitung geeignet.

Aufgrund des zuvor beschriebenen Vorteils, dass Kennzeichnungen ($\texttt{in}$, $\texttt{out}$, $\texttt{undec}$) im Vergleich zu Argument Mengen (enthalten oder nicht) eine zusätzliche Dimension der Information bieten, werden für die Ausarbeitung des Algorithmus Kennzeichnungen verwendet.

2.2 Analyse der modernen Argumentationslöser

Da die Forschung im Bereich abstrakter Argumentation vor allem durch die ICCMA vorangetrieben wird, soll analysiert werden, welche Ansätze die Argumentationslöser nutzen. Die Analyse beschränkt sich auf die Argumentationslöser der ICCMA21, da diese zur Zeit der Bearbeitung dieser Arbeit den Stand der Technik darstellen. In der ICCMA21 haben die Argumentationslöser PYGLAF, μ-TOKSIA in den Bereichen DC-CO und DC-PR sehr gut abgeschnitten. Beide Argumentationslöser basieren auf dem Ansatz der Reduktion. Direkte Argumentationslöser in der ICCMA21 waren AFGCN [Lar21], HARPER++ [Thi21] und MATRIXX [Hei21]. AFGCN und HARPER++ wurden nur auf dem *Approximate Track*[4] eingesetzt. AFGCN hatte sowohl in DC-CO als auch in DC-PR in etwa doppelt so viele Punkte wie HARPER++ [LLMR21]. MATRIXX schnitt in DC-CO von allen Argumentationslösern am schlechtesten ab. Da MATRIXX keine präferierten Extensionen unterstützt, nahm dieser Löser nicht an DC-PR teil. Da die Ergebnisse von MATRIXX

[4] Mehr Informationen dazu in [LLMR21]

unterdurchschnittlich waren, wird dieser Argumentationslöser im Folgenden nicht weiter analysiert.

Es wird im Folgenden ausschließlich auf die Lösung von leichtgläubigen Akzeptanzproblemen (DC - aus dem Englischen "*Decide Credulous acceptance*") eingegangen, da sie für die durchgeführte Forschung relevant sind.

HARPER++ nutzt spezielle Eigenschaften fundierter Extensionen (engl. *grounded Extensions*) aus, um Argumentations-Probleme zu lösen. Eine fundierte Extension ist eine minimale vollständige Extension. Durch ihre speziellen Eigenschaften kann die Lösung von DC Problemen davon abgeleitet werden, ob das analysierte Argument von einem Argument in der fundierten Extension des entsprechenden AFs angegriffen wird [Thi21].

Der AFGCN Argumentationslöser nutzt zur Lösung von DC-Problemen ein Annäherungsverfahren basierend auf einem *Graph Convolutional Network Model* [Mal22] einer künstlichen Intelligenz, die mit Daten aus früheren ICCMA Instanzen trainiert wurde [Lar21].

Die Analyse der modernen Argumentationslöser hat gezeigt, dass eine breite Auswahl von Ansätzen genutzt werden kann, um Argumentations Probleme zu lösen. Keiner der analysierten Ansätze verfolgt den Ansatz, der in dieser Arbeit erforscht werden soll.

2.3 Stochastische Lokalsuche

Eine verwandte Arbeit ist [Thi18]. Sie stellt eine Familie von Algorithmen zum Finden einer stabilen Extension (engl. *stable Extension*) in einem gegebenen AF vor und erforscht dessen Performanz. Dabei wird der Ansatz der stochastischen Lokalsuche genutzt. Algorithmen, die stochastische Lokalsuche nutzen sind unvollständige Algorithmen, die ihr Ergebnis sukzessive durch zufällige Veränderungen verbessern. Weitere Informationen zu stochastischer Lokalsuche sind z. B. in [Bie09] zu finden. Die Algorithmus-Familie aus [Thi18] baut auf den bekannten stochastische Lokalsuche Algorithmen GSAT [SLM92] und WalkSAT [SKC99] auf. GSAT und WalkSAT sind Algorithmen zur Lösung des Booleschen Entscheidbarkeitsproblems (engl. *boolean SATisfiability problem*) - auch bekannt als SAT-Problem. In der Arbeit von [Thi18] wird das Konzept von GSAT/WalkSAT direkt auf AFs angewendet. Der grundlegende Algorithmus namens WalkAAF arbeitet mit Kennzeichnungen und ändert in jeder Iteration die Kennzeichnung von einem Argument, um eine stabile Kennzeichnung aufzubauen. Da der in dieser Arbeit zu erforschende Algorithmus einen ähnlichen Aufbau hat wie WalkAAF, wird hier auf den Aufbau des WalkAAF-Algorithmus eingegangen. Der Algorithmus ist in [Thi18] genauer formalisiert.

Algorithmus 1 zeigt den allgemeinen Ablauf von WalkAAF. Er startet mit der Eingabe eines AFs und zwei ganzen Zahlen M, N, welche sicherstellen, dass der unvollständige Algorithmus in jedem Fall terminiert. Für die festgelegte Anzahl von Versuchen N (Zeile 1) wird Folgendes getan: Für jedes Argument wird zufällig `in` oder `out` in der Kennzeichnung L gesetzt (Zeile 2). Danach wird für die vorgege-

Algorithmus 1 WalkAAF

Require: $AF = \langle R, T \rangle$, ganze Zahlen N, M (=Anzahl Versuche)
Ensure: Eine stabile Kennzeichnung L oder FAIL bei fehlgeschlagener Suche

```
1:  for (1,...,N) do
2:      L ← setze in und out Kennzeichnungen zufällig für alle B ∈ R
3:      for (1,...,M) do
4:          if (L ist stabil) then return L
5:          else
6:              A = zufälliges falsch gekennzeichnetes Argument
7:              L(A) = ¬L(A)
8:          end if
9:      end for
10: end for
11: return FAIL;
```

bene Anzahl von Versuchen M (Zeile 3) überprüft, ob L stabil ist. Ist L stabil, wird L zurückgegeben (Zeile 4). Anderenfalls wird zufällig ein falsch gekennzeichnetes Argument A ausgewählt (Zeile 6) und seine Kennzeichnung wird umgekehrt (Zeile 7), das bedeutet: War L(A) = `in` wird es zu L(A)= `out` vice versa. Sind alle Versuche M, N verbraucht, bevor eine stabile Kennzeichnung zurückgegeben wurde, wird FAIL zurückgegeben.

Ein Argument A ist dabei nach [Thi18] falsch gekennzeichnet, wenn A:

1. als `undec` gekennzeichnet ist.
2. als `out` gekennzeichnet ist, aber keiner seiner Angreifer ist als `in` gekennzeichnet ist.
3. als `in` gekennzeichnet ist, aber
   a) nicht alle Angreifer von A als `out` gekennzeichnet sind.
   b) mindestens ein Argument, das A angreift, nicht als `out` gekennzeichnet ist.

Wenn es in einer Kennzeichnung kein falsch gekennzeichnetes Argument gibt, ist es eine stabile Kennzeichnung [Thi18]. Es wird bei dieser Definition auf die Ähnlichkeit zu zulässigen Kennzeichnungen hingewiesen. Die Voraussetzungen 2 und 3.b von stabilen Kennzeichnungen gelten auch für zulässige Kennzeichnungen. Dies ist darauf zurückzuführen, dass stabile Extensionen auch zulässige Extensionen sind [CDG$^+$15].

Die Auswertung der Ergebnisse aus [Thi18] ergab, dass WalkAAF mehr falschnegative Ergebnisse liefert als vergleichs-*Argumentationslöser*. Dennoch ist WalkAAF aufgrund seiner Geschwindigkeit konkurrenzfähig.

Aufbauend auf diesen Ergebnissen soll in dieser Arbeit ein Ansatz nach stochastischer Lokalsuche erforscht werden, der das DC-ADM Problem löst.

In diesem Kapitel wurden die für das Thema dieser Arbeit relevanten Definitionen vorgestellt und mit Beispielen veranschaulicht. Es wurde auf verwandte Arbeiten eingegangen.

3 Konzeption

Das Ziel des zu implementierenden Algorithmus ist es, das leichtgläubige Akzeptanzproblem bezüglich zulässiger Mengen zu lösen. Die Idee ist es angefangen mit dem Argument, dessen leichtgläubige Akzeptanz nachgewiesen werden soll, sukzessive eine zulässige Kennzeichnung aufzubauen, indem Verteidiger für `in`($\mathcal{L}$) zu `in`($\mathcal{L}$) hinzugefügt werden. Die Auswahl von Argumenten, die als `in` gekennzeichnet werden, sollen nach dem Ansatz der stochastischen Lokalsuche randomisiert ausgewählt werden.

3.1 Formale Definition des Algorithmus

In dieser Arbeit wird der direkte Ansatz zur Lösung von AF-Problemen angewendet. Wie in Kapitel 2.1 erwähnt führten die Vorzüge von Kennzeichnungen zu der Design-Entscheidung, diese bei der Ausarbeitung des Algorithmus anstelle von Argument-Mengen zu nutzen. Der ausgearbeitete Algorithmus ist der folgende:

Algorithmus 2 `solve_dc-adm()`

Require: $AF = \langle AR, att \rangle, A \in AR, N \in \mathbb{N}$
Ensure: $\exists$ Kennzeichnung $\mathcal{L}$ von AR: $A \in$ `in`($\mathcal{L}$) $\Rightarrow WAHR$, Sonst $\Rightarrow FALSCH$

```
 1: for (1, ..., N) do
 2:     Verteidigungsstatus VS = Speichert für jedes Argument ∈ AR, ob in(L) es angreift.
 3:     Angriffsstatus AS = Speichert für jedes Argument ∈ AR, ob es in(L) angreift.
 4:     labelIn(A, VS, AS)                          ▷ Siehe Algorithmus 3
 5:     while (!isAdmissible(L, VS )) do            ▷ Siehe Algorithmus 4
 6:         B =findB(VS, AS)                        ▷ Siehe Algorithmus 5
 7:         C = findC(L, B)                         ▷ Siehe Algorithmus 6
 8:         if (C == Null) then
 9:             break
10:         end if
11:         labelIn(C, VS, AS)
12:     end while
13:     if isAdmissible(L, VS ) then
14:         return WAHR
15:     end if
16: end for
17: return FALSCH
```

Zur weiteren Erläuterung folgt eine schrittweise Erklärung. Der Algorithmus agiert auf einem Argumentationsgraphen AF und einem Element a dieses Argumentationsgraphen. Zudem ist eine maximale Anzahl N von Iterationen festgelegt, die

der Algorithmus nicht überschreitet (Zeile 1). Diese Anzahl von Iterationen stellt die Anzahl von Versuchen dar, die unternommen werden, um eine zulässige Kennzeichnung in dem Argumentationsgraphen zu finden, in der $\mathcal{L}$(A) = `in` ist. Wir gehen davon aus, dass zu Anfang alle Argumente mit `undec` gekennzeichnet sind. Um Zeitkomplexität zu sparen, speichern wir für jedes Argument den Verteidigungsstatus (Zeile 2) und den Angriffsstatus (Zeile 3). Der Verteidigungsstatus für ein Argument E ist wahr, wenn `in`($\mathcal{L}$) E angreift. Der Angriffsstatus für ein Argument E ist wahr, wenn E `in`($\mathcal{L}$) angreift. Für den Teil der Konzeption können der Verteidigungs- und der Angriffsstatus als *Arrays* boolescher Werte interpretiert werden, bei denen jeder Index für ein bestimmtes Argument im AF steht. A wird mit der `labelIn()`-Funktion als `in` gekennzeichnet (Zeile 4). Die `labelIn()`-Funktion erfüllt noch weitere Funktionen, die im nächsten Absatz erklärt werden. Solange $\mathcal{L}$ nicht zulässig ist (Zeile 5) wird zufällig ein Angreifer von `in`($\mathcal{L}$) mit der `findB()` Funktion gewählt. Mithilfe dieses Angreifers wird dann ein Verteidiger für $\mathcal{L}$ mithilfe der `findC()`-Funktion ausgewählt (Zeile 6 und 7) und wie oben erwähnt als `in` gekennzeichnet (Zeile 11). Auf die Hilfsfunktionen `findB()` und `findC()` wird später genauer eingegangen. Wenn $\mathcal{L}$ zulässig ist, terminiert der Algorithmus mit der Ausgabe WAHR (Zeile 14). Gibt es (in Zeile 8) keinen Verteidiger für $\mathcal{L}$, wird die derzeitige Kennzeichnung $\mathcal{L}$ verworfen (implizit durch break in Zeile 9) und es wird ein neuer Versuch gestartet, eine zulässige Kennzeichnung zu finden (Zeile 1) insofern nicht alle Versuche verbraucht sind. Sind alle Versuche verbraucht und wurde keine zulässige Kennzeichnung gefunden, terminiert der Algorithmus mit der Ausgabe FALSCH (Zeile 17).

Im Folgenden wird der Algorithmus 2 mit `solve_dc-adm` referenziert. Es ist gut sichtbar, dass `solve_dc-adm` den direkten Ansatz verfolgt, da er direkt auf dem Argumentationsgraphen arbeitet und somit die Informationen über die Angriffs-Beziehungen zwischen den Argumenten zur Lösungsfindung nutzen kann.

Die `labelIn()` Funktion (Algorithmus 3) übernimmt neben dem Kennzeichnen der Argumente die wichtige Aufgabe, den Angriffs- und Verteidigungsstatus im Bezug auf `in`($\mathcal{L}$) für jedes Argument des AFs aktuell zu halten.

Der Ablauf der Funktion ist wie folgt: Wenn das übergebene Argument A sich nicht selbst angreift (Zeile 1) wird es als `in` gekennzeichnet (Zeile 2). Alle seine Angreifer und Kind-Argumente werden als out gekennzeichnet (Zeilen 4 und 8). Dies hat den Grund, dass die Angreifer und Kinder eines Arguments mit diesem in Konflikt stehen. Alle Angreifer von A bekommen einen Angriffsstatus von WAHR (Zeile 5), da sie das als `in` gekennzeichnete Argument A und damit `in`($\mathcal{L}$) angreifen. Alle Kind-Argumente von A bekommen einen Verteidigungsstatus von WAHR (Zeile 9), da A und damit `in`($\mathcal{L}$) sich gegen mögliche eingehende Angriffe von ihnen verteidigt. Diese Vorbereitungen ermöglichen eine effiziente Prüfung der Zulässigkeit von `in`($\mathcal{L}$) sowie die effiziente Identifikation von B- und C-Kandidaten. Damit `solve_dc-adm` erfolgreich eine zulässige Kennzeichnung aufbauen kann, muss sichergestellt werden, dass mit `labelIn()` nur Argumente als `in` gekennzeichnet werden, die nicht mit `in`($\mathcal{L}$) in Konflikt stehen.

Algorithmus 3 `labelIn()`

Require: $AF = \langle AR, att \rangle$, Kennzeichnung $\mathcal{L}$, Verteidigungsstatus VS, Angriffsstatus AS, A $\in$ AR

```
 1: if (A greift sich nicht selbst an) then
 2:     L(A) = in
 3:     for (∀ Angreifer E von A) do
 4:         L(E) = out
 5:         AS[E] = WAHR
 6:     end for
 7:     for (∀ Kinder K von A) do
 8:         L(K) = out
 9:         VS[K] = WAHR
10:     end for
11: end if
```

Die Prüfung der Zulässigkeit einer Kennzeichnung spielt in `solve_dc-adm` eine zentrale Rolle. Damit eine Kennzeichnung zulässig ist, müssen alle Angreifer von `in`($\mathcal{L}$) als `out` gekennzeichnet sein und alle Argumente $\in$ `out`($\mathcal{L}$) müssen mindestens einen Angreifer haben, der als `in` gekennzeichnet ist. Die `labelIn()`-Funktion wird nur mit Argumenten aufgerufen, die konfliktfrei mit `in`($\mathcal{L}$) sind und `labelIn()` stellt sicher, dass alle Angreifer von `in`($\mathcal{L}$) als `out` gekennzeichnet sind. Damit ist die erste Bedingung durch die `labelIn()`-Funktion immer erfüllt. Aus diesem Grund wird in Algorithmus 4 nur die zweite Bedingung überprüft. Dafür wird durch alle Argumente `out`($\mathcal{L}$) iteriert (Zeile 1) und überprüft, ob `in`($\mathcal{L}$) sich gegen eines dieser Argumente nicht verteidigt (Zeile 2). Wenn ja, wird FALSCH ausgegeben (Zeile 3). Wenn alle o $\in$ `out`($\mathcal{L}$) verteidigt werden, also die Bedingung erfüllt ist, wird WAHR ausgegeben (Zeile 6). Diese Konzeption wirkt sich durch ihre starke Abhängigkeit von der `labelIn()`-Funktion negativ auf die Wiederverwendbarkeit aus. Da durch die `labelIn()`-Funktion jedoch die korrekte Funktion der Zulässigkeitsprüfung aus Algorithmus 4 sichergestellt wird, wurde sich durch die potenzielle Zeitersparnis für diese Version der Prüfung entschieden.

Algorithmus 4 `isAdmissible()`

Require: *Labeling* $\mathcal{L}$, Verteidigungsstatus VS
Ensure: WAHR, wenn alle o $\in$ `out`($\mathcal{L}$) verteidigt sind. FALSCH, sonst.

```
1: for (o ∈ out(L) ) do
2:     if (!VS[o] ) then
3:         return FALSCH
4:     end if
5: end for
6: return WAHR
```

Um mit Algorithmus 5 einen unverteidigten Angreifer von `in`($\mathcal{L}$) auszuwählen, gehen wir wie folgt vor: bKandidaten ist eine Liste (Zeile 1). Wir iterieren durch alle Argumente i aus AR (Zeile 2) und fügen die Argumente der bKandidaten-

Liste hinzu, die einen Angriffsstatus von WAHR und einen Verteidigungsstatus von FALSCH haben (Zeilen 3-5). Dies sind alle unverteidigten Angreifer von `in`($\mathcal{L}$). Ist bKandidaten danach nicht leer (Zeile 7), wird ein zufälliges Element aus dieser Liste zurückgegeben (Zeile 8). Sollte bKandidaten leer sein, wird -1 als Fehlerwert zurückgegeben.

Algorithmus 5 `findB()`

Require: $AF = \langle AR, att \rangle$, Verteidigungsstatus VS, Angriffsstatus AS
Ensure: $B \in AR$: B ist unverteidigter Angreifer von `in`($\mathcal{L}$). -1, wenn es kein solches B gibt.

```
1:  bKandidaten = Liste zum Speichern von Kandidaten für B
2:  for (∀ i ∈ AR) do
3:      if (AS[i]==WAHR ∧ VS[i]== FALSCH) then
4:          bKandidaten.add(i)
5:      end if
6:  end for
7:  if (!bKandidaten.empty()) then
8:      return bKandidaten.get(zufälliger_Index)
9:  end if
10: return -1
```

Einen passenden Verteidiger für `in`($\mathcal{L}$) wählen wir mit Algorithmus 6 aus. Die entsprechenden Kandidaten werden in der Liste cKandidaten gespeichert (Zeile 1). Nun wird durch alle Angreifer des übergeben Arguments B iteriert (Zeile 2). Alle Angreifer, die die Kennzeichnung `undec` haben, werden zu den cKandidaten hinzugefügt (Zeile 3-5). Das hat den Hintergrund, dass mit `labelIn()` (siehe Algorithmus 3) sichergestellt wird, dass alle Argumente, die mit `in`($\mathcal{L}$) in Konflikt stehen, als `out` gekennzeichnet werden. Somit sind alle Argumente, die als `undec` gekennzeichnet sind konfliktfrei mit `in`($\mathcal{L}$). Analog zu Algorithmus 5 wird ein zufälliges Argument aus cKandidaten zurückgegeben (Zeile 8), falls cKandidaten nicht leer ist (Zeile 7) und ansonsten wird -1 zurückgegeben (Zeile 10). Im folgenden wird ein

Algorithmus 6 `findC()`

Require: $AF = \langle AR, att \rangle$, Kennzeichnung $\mathcal{L}, B \in AR$
Ensure: $C \in AR : (C, B) \in att \wedge$ C ist konfliktfrei mit `in`($\mathcal{L}$). -1, wenn es kein solches C gibt.

```
1:  cKandidaten = Liste zum Speichern von Kandidaten für C
2:  for (∀ cKandidat - Angreifer von B) do
3:      if (getLabel(cKandidat) == undec) then
4:          cKandidaten.add(cKandidat)
5:      end if
6:  end for
7:  if (!cKandidaten.empty()) then
8:      return cKandidaten.get(zufälliger_Index)
9:  end if
10: return -1
```

erfolgreicher Durchlauf von Algorithmus 2 anhand des Graphen aus Abb. 2 gezeigt.

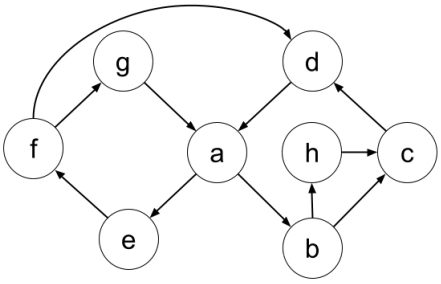

Abbildung 2: Graph für Beispiel 3.1

Beispiel 3.1. Das Beispiel ist grafisch in den Abbildungen 3-6 dargestellt.

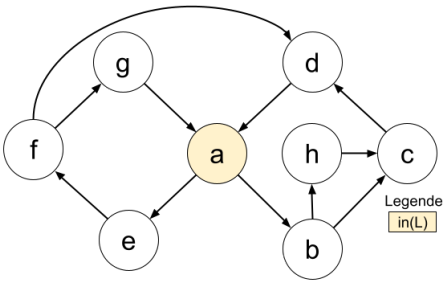

Abbildung 3: **Schritt 1** - Zu Beginn wird a als `in` gekennzeichnet (in gelb dargestellt). `in`($\mathcal{L}$)={a} ist konfliktfrei, aber nicht akzeptierbar und somit nicht zulässig, die Schleife wird betreten.

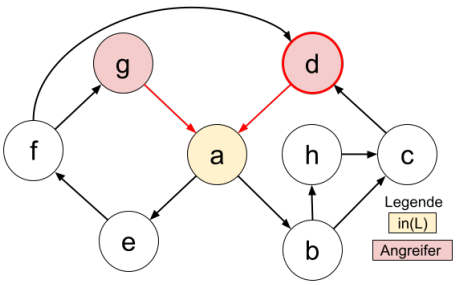

Abbildung 4: **Schritt 2** - Wir betrachten nun alle Angreifer von `in`($\mathcal{L}$) (rot dargestellt).Argument d wird zufällig ausgewählt (rote Umrandung).

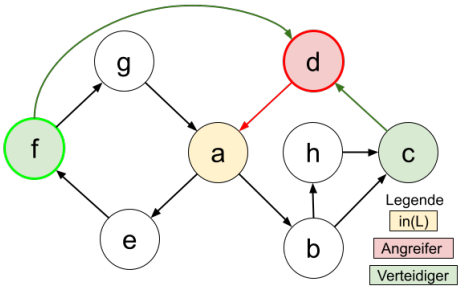

Abbildung 5: **Schritt 3** - Es gibt zwei Argumente, die $\texttt{in}(\mathcal{L})$ gegen d verteidigen. Die Argumente c und f. Es wird zufällig f ausgewählt (grüne Umrandung).

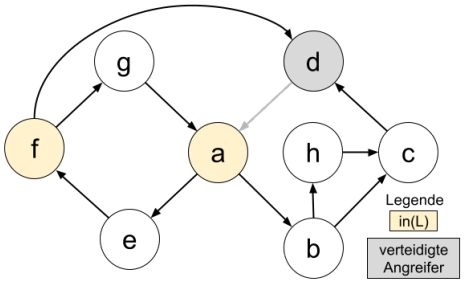

Abbildung 6: **Schritt 4** - f wird als $\texttt{in}$ gekennzeichnet. Ein Durchlauf der while Schleife ist beendet. $\texttt{in}(\mathcal{L})$ ist zulässig und der Algorithmus gibt WAHR aus.

Es ist zu beachten, dass der Algorithmus 2 unvollständig ist. Das bedeutet, dass er falsch-negative Ergebnisse liefern kann. Um diese Eigenschaft klarer zu erläutern, folgt ein Beispiel.

Beispiel 3.2. Die ersten zwei Schritte aus Beispiel 3.1 werden auch hier durchgeführt. Damit liegt folgende Situation vor:

$$\texttt{in}(\mathcal{L}) = \{a\} \text{ B = d, C-Kandidaten = \{c,f\}}$$

Es wird zufällig c als Verteidiger ausgewählt und als $\texttt{in}$ gekennzeichnet. Die Menge $\texttt{in}(\mathcal{L})$ ist nicht zulässig, dadurch wird eine weitere Iteration der while-Schleife gestartet. Es gilt nun:

$$\texttt{in}(\mathcal{L}) = \{a, c\} \text{ B-Kandidaten = \{b, g, h\}}$$

Es wird zufällig h als B ausgewählt. Nun stellen wir fest, dass kein Argument, das mit $\texttt{in}(\mathcal{L})$ konfliktfrei ist, die Menge vor dem Angriff von h verteidigt. Damit wird die While-Schleife verlassen. $\texttt{in}(\mathcal{L})$ ist nicht zulässig und die For-Schleife wird verlassen (da N=1). Es wird FALSCH ausgegeben. Wie in Beispiel 3.1 gezeigt, gibt es jedoch eine zulässige Kennzeichnung, in welchem Argument a $\in \texttt{in}(\mathcal{L})$. Somit handelt es sich bei dem Ergebnis dieses Beispiel-Durchlaufs um ein falsch-negatives Ergebnis.

3.1.1 Formale Eigenschaften des Algorithmus

Im Folgenden werden die formalen Eigenschaften des Algorithmus beschrieben und bewiesen. Dabei wird auf allgemeine Eigenschaften, auf die Korrektheit sowie die Zeit- und Speicherplatzkomplexität eingegangen. Da der Algorithmus dem Ansatz der stochastischen Lokalsuche folgt und somit randomisiert arbeitet, ist er nicht deterministisch. Des Weiteren ist `solve_dc-adm` wie zuvor erwähnt unvollständig. Nun wird die formale Korrektheit des Algorithmus gezeigt.

Proposition 1. *Wenn `solve_dc-adm` bei der Eingabe AF, a WAHR ausgibt, gibt es in AF mindestens eine zulässige Kennzeichnung $\mathcal{L}$ mit $\mathcal{L}(a) =$ `in`.*

Beweis. WAHR wird allein in Zeile 14 zurückgegeben. In Zeile 13 wird sichergestellt, dass dies nur passiert, wenn die Kennzeichnung $\mathcal{L}$ zulässig ist. □

Proposition 2. *Wenn AF keine zulässige Kennzeichnung $\mathcal{L}$ mit $\mathcal{L}(a) =$ `in` beinhaltet, gibt `solve_dc-adm` FALSCH aus.*

Beweis. Wenn $\mathcal{L}$ nicht zulässig ist, verhindert Zeile 13, dass Zeile 14 im Code erreicht wird. Somit wird nich WAHR ausgegeben. Da wir mit endlichen AFs arbeiten, gibt es nur endlich viele Knoten B (Zeile 6) und C (Zeile 7), womit die *break*-Bedingung in Zeile 9 letztlich immer erreicht wird. Da auch N nach Definition eine endliche Zahl ist, wird auch die For-Schleife immer verlassen und somit FALSCH ausgegeben. □

Mit diesem Beweis ist zusätzlich gezeigt, dass `solve_dc-adm` immer terminiert. Der Beweis unterstreicht zusätzlich die Tatsache, dass Algorithmus 2 keine negativen Instanzen bestimmen kann. Er durchläuft in dem Fall, dass ein Argument nicht leichtgläubig akzeptiert werden kann, die For-Schleife so lange, bis keine Versuche mehr zur Verfügung stehen. Zur Analyse der Zeit- und Speicherplatzkomplexität wird als erstes auf alle Unterfunktionen eingegangen.

Proposition 3. *Für eine gegebene Eingabe $AF = \langle AR, att \rangle$, Kennzeichnung $\mathcal{L}$, Verteidigungsstatus VS, Angriffsstatus AS, $A \in AR$ braucht Algorithmus 3 `labelIn()` im Schlimmstfall $O(|AR|)$ Zeit und O(1) Speicherplatz.*

Beweis. Alle Operationen in `labelIn()` haben eine Laufzeitkomplexität von O(1) mit Ausnahme der For-Schleifen in Zeilen 3 und 7. Beide Schleifen können im schlimmsten Fall |AR|-1 mal durchlaufen. Dieser Fall ist gegeben, wenn alle anderen Argumente im Graphen A angreifen und A alle anderen Argumente im Graphen zurück angreift. Durch dich Bedingung in Zeile 1 wird sichergestellt, dass A sich nicht selbst angreift und somit auch nicht zu den Angreifern oder Kindern von A gehört. Damit ist die Laufzeitkomplexität bei $O(|AR| - 1) \cdot O(1) = O(|AR|)$. Die Speicherkomplexität von `labelIn()` liegt bei O(1), da keine neuen Datenstrukturen oder Variablen angelegt werden. □

Proposition 4. *Für eine gegebene Eingabe Kennzeichnung $\mathcal{L}$, Verteidigungsstatus VS braucht Algorithmus 4 im Schlimmstfall $O(|AR|)$ Zeit und O(1) Speicherplatz.*

Beweis. Alle Operationen in `labelIn()` haben eine Laufzeitkomplexität von O(1) mit Ausnahme der For-Schleife in Zeilen 1. Die Schleife kann im schlimmsten Fall |AR|-1 mal durchlaufen. Dies ist der Fall, wenn alle bis auf das zu Anfang gegebene Argument als `out` gekennzeichnet sind. Damit ist die Laufzeitkomplexität bei $O(|AR|-1) \cdot O(1) = O(|AR|)$. Die Speicherkomplexität von Algorithmus 4 liegt bei O(1), da keine neuen Datenstrukturen oder Variablen angelegt werden. □

Proposition 5. *Für einen gegebenen Input $AF = \langle AR, att \rangle$,Verteidigungsstatus VS, Angriffsstatus AS braucht `findB()` (siehe Algorithmus 5) im Schlimmstfall $O(|AR|)$ Zeit und O(|AR|) Speicherplatz.*

Beweis. Alle Operationen in `findB()` haben eine Laufzeitkomplexität von O(1) mit Ausnahme der For-Schleife in Zeile 2. Da die For-Schleife |AR| mal durchläuft, ergibt sich eine Laufzeitkomplexität von $O(|AR|) \cdot O(1) = O(|AR|)$. In `findB()` wird eine Liste bKandidaten erstellt und gefüllt (Zeile 1 und 4). Dabei kann die Liste im Schlimmstfall |AR|-1 Argumente beinhalten, wenn z. B. ein Argument $\in$ `in`($\mathcal{L}$) ist und alle anderen Argumente dieses angreifen. Des Weiteren wird zusätzlicher Speicherplatz für den zufälligen Index aus Zeile 8 benötigt. Daraus ergibt sich eine Speicherkomplexität von $O(|AR|-1) + O(1) = O(|AR|)$. □

Proposition 6. *Für einen gegebenen Input $AF = \langle AR, att \rangle$, Kennzeichnung $\mathcal{L}, B \in AR$ braucht `findC()` (siehe Algorithmus 6) im Schlimmstfall $O(|AR|)$ Zeit und $O(|AR|)$ Speicherplatz.*

Beweis. Alle Operationen in `findC()` haben eine Laufzeitkomplexität von O(1) mit Ausnahme der For-Schleife in Zeile 2. Die For-Schleife läuft im Schlimmstfall |AR| mal durch, wenn alle Argumente im übergebenen AF B angreifen und B sich selbst angreift. Daraus ergibt sich eine Laufzeitkomplexität von $O(|AR|) \cdot O(1) = O(|AR|)$. Die Speicherkomplexität liegt analog zu `findB()` bei $O(|AR|)$ siehe dazu den Beweis von Proposition 5. □

Da die Zeit- und Raumkomplexität von allen Unterfunktionen gezeigt wurde, kann nun auf die Hauptfunktion `solve_dc-adm` eingegangen werden.

Proposition 7. *Für eine gegebene Eingabe $AF = \langle AR, att \rangle, A \in AR, N \in \mathbb{N}$ braucht `solve_dc-adm` im Schlimmstfall $O(N|AR|^2)$ Zeit und $O(|AR|)$ Speicherplatz.*

Beweis. Alle Hilfsfunktionen (dargestellt in Algorithmen 3, 4, 5 und 6) haben eine Laufzeitkomplexität von O(|AR|) (siehe Propositionen 3, 4, 5 und 6).

In `solve_dc-adm` werden die Hilfsfunktionen `labelIn()`, `findB()` und `findC()` nacheinander in einer while-Schleife aufgerufen. Die while-Schleife wird im Schlimmstfall |AR|/2 mal durchlaufen. Dies ist zum Beispiel der Fall, wenn jedes gewählte B von genau einem Argument C verteidigt wird, das wiederum von einem neuen unverteidigten Argument angegriffen wird. Die For-Schleife in Zeile 1 wird genau N mal durchlaufen. Dadurch ergibt sich eine Laufzeitkomplexität von $O(N) \cdot (O(|AR|) + O(|AR|/2) \cdot O(|AR|)) = O(N) \cdot (O(|AR|) + O(|AR|) \cdot O(|AR|)) = O(N) \cdot$

$(O(|AR|) \cdot O(|AR|)) = O(N|AR|^2)$. Die Speicherkomplexität von `solve_dc-adm` lässt sich wie folgt bestimmen. Es werden zwei Datenstrukturen Verteidigungs- und Angriffsstatus erstellt (Zeilen 2 und 3). Diese können, wie zuvor erwähnt, als Arrays der Länge `|AR|` interpretiert werden. Die Speicherkomplexität beträgt dabei $O(|AR|) + O(|AR|) = O(|AR|)$. Die Speicherkomplexität für `labelIn()` beträgt laut Proposition 3 O(1) und kann dadurch vernachlässigt werden. In der While-Schleife (Zeile 5) werden nacheinander `findB()` und `findC()` aufgerufen, welche eine Speicherkomplexität von je O(`|AR|`) haben (siehe Propositionen 5 und 6) wir gehen davon aus, dass der von `findB()` und `findC()` benötigte Speicherplatz nach Ende der Funktionen wieder freigegeben wird. Damit berechnet sich die Speicherkomplexität von `solve_dc-adm` mit $O(|AR|)$ für den Speicherplatz des Verteidigungs- und Angriffsstatus und $O(|AR|)$ für den benötigten Platz von `findB()` und `findC()`. Das Gesamtergebnis für die Speicherkomplexität ist damit $O(|AR|) + O(|AR|) = O(|AR|)$. □

3.2 Limitationen und Vorteile des Algorithmus

Es muss beachtet werden, dass der Algorithmus nicht vollständig ist und es zu falsch negativen Ausgaben kommen kann. Dies ist darauf zurückzuführen, dass Kandidaten für zulässige Mengen zufällig ausgewählt werden. Es ist also möglich, dass eine existierende zulässige Menge zufällig nicht ausgewählt, somit nicht gefunden wird und der Algorithmus 2 FALSCH ausgibt.

Die zufällige Auswahl der möglichen zulässigen Mengen kann zusätzlich als Nachteil gesehen werden, da nicht bevorzugt Mengen gewählt werden, die eine hohe Wahrscheinlichkeit haben, zulässig zu sein. Die Auswahl der Kandidaten findet also nicht systematisch statt.

Auf der anderen Seite ist die zufällige Auswahl der Elemente insofern ein Vorteil, als dass der Algorithmus keine Vorannahmen über das AF macht. Dadurch kann der Algorithmus problemlos bei jeglichen AFs eingesetzt werden. Ein weiterer Vorteil des Algorithmus ist, dass er immer terminiert und somit die NP-Vollständigkeit des zugrunde liegenden leichtgläubigen Akzeptanzproblems erfolgreich bewältigt wurde. Da die maximale Laufzeit des Algorithmus begrenzt ist und es nicht zu Endlosschleifen kommen kann.

Des Weiteren ist der Algorithmus durch die zufällige Auswahl nicht deterministisch. Dadurch ist die Reproduzierbarkeit von Programmabläufen nicht sichergestellt. Das erschwert die Fehlersuche.

Dieses Kapitel stellt die Konzeption des erforschten Algorithmus dar. Er wird formal definiert und Design-Entscheidungen werden erklärt. Des Weiteren werden die formalen Eigenschaften des Algorithmus gezeigt. Abschließend wird in einer kritischen Betrachtung auf die Limitationen und Vorteile des Algorithmus eingegangen.

4 Implementation

In diesem Kapitel werden die Implementation und die damit einhergehenden Entscheidungen bei der Umsetzung des Algorithmus dargestellt. Es werden die verwendeten Datenstrukturen vorgestellt und auf den Programm-Aufbau sowie -Ablauf eingegangen.

Aus Performanz-Gründen ist `solve_dc-adm` mit C++ implementiert. Da der Algorithmus auf Datenstrukturen wie AFs und Kennzeichnungen angewiesen ist, musste eine Entscheidung getroffen werden, diese Datenstrukturen neu zu entwickeln oder bereits bestehende Datenstrukturen zu nutzen.

Es wurde sich für die Nutzung bereits existierender Datenstrukturen entschieden, um den Fokus auf die Entwicklung von `solve_dc-adm` zu legen. Eine Schwierigkeit bei dieser Herangehensweise war die Einarbeitung in den fremden Code und die Sicherstellung des Verständnisses der definierten Datenstrukturen.

In der Ausarbeitung von `solve_dc-adm` werden die Datenstrukturen des modernen Argumentationslösers taas-fudge, der ebenfalls in C++ implementiert ist, genutzt[5].Im folgenden werden die Namen taas-fudge und FUDGE synonym für diesen Argumentationslöser verwendet. Die Implementation von `solve_dc-adm` und alle dafür nötigen Hilfsfunktionen sowie Datenstrukturen wurden hinzugefügt. Da leichtgläubige Akzeptanz bezüglich zulässiger Extensionen von Argumentationslösern im Allgemeinen nicht unterstützt wird, wird `solve_dc-adm` bei der Lösung von den DC-CO und DC-PR Aufgaben aufgerufen. Dieser Schritt wird unternommen, um Experimente mit anderen Argumentationslösern zu ermöglichen. Der entwickelte Argumentationslöser wird im folgenden als adm-fudge referenziert. Sein Quellcode ist online[6] verfügbar.

Im folgenden wird detaillierter auf die Implementation eingegangen.

4.1 Datenstrukturen

In diesem Unterkapitel werden die in der Implementation verwendeten Datenstrukturen vorgestellt. Die Abhängigkeiten der verwendeten Datenstrukturen werden im UML-Diagramm aus Abb. 7 dargestellt. Der «struct» Stereotyp bedeutet, dass es sich um eine C++-Struktur handelt. In Abb. 7 ist die Implementation von Algorithmus 2 dargestellt. Hierbei handelt es sich um eine C++-Funktion. Funktionen werden normalerweise nicht in UML-Diagrammen dargestellt, jedoch wurde sich hierbei dafür entschieden, um die Nutzung von Datenstrukturen in `solve_dc-adm` darzustellen. Die Boxen um die verschiedenen Strukturen stellen die Zusammengehörigkeit der Datenstrukturen zu ihrer jeweiligen Quelle dar. Die mit «use» beschrifteten Pfeile von `solve_dc-adm` zu der taas-fudge und der GLib 2.0 Box bedeuten, dass `solve_dc-adm` alle Strukturen in dieser Box verwendet. Die in Abb. 7 erwähnten Datenstrukturen werden nachfolgend vorgestellt.

[5] Verfügbar unter [TCV23]

[6] Auf Github: https://github.com/gigarina/taas-fudge (Im *main-Branch* unter dem *Tag* 'abgabe'.)

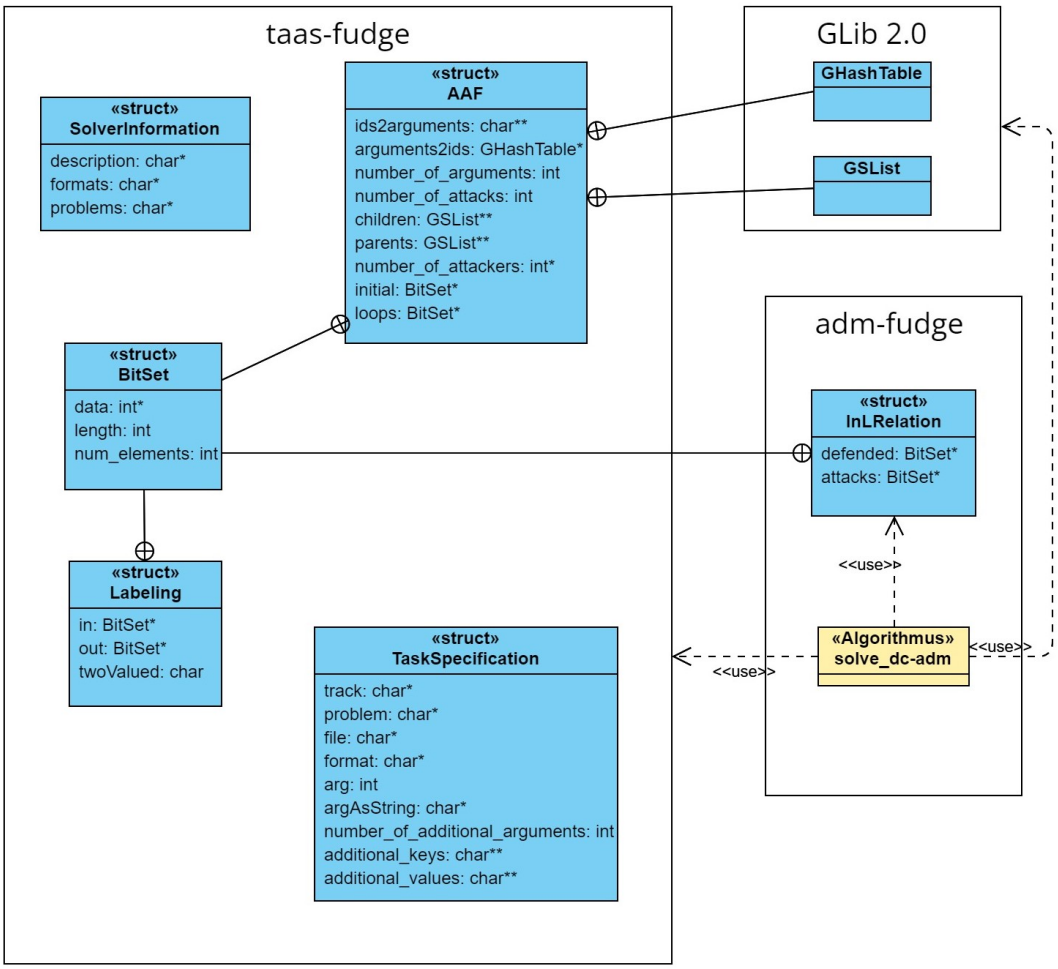

Abbildung 7: UML-Diagramm der in der Implementation von Algorithmus 2 verwendeten Datenstrukturen

4.1.1 Datenstrukturen aus taas-fudge

Die in `solve_dc-adm` verwendeten Datenstrukturen von taas-fudge werden nun vorgestellt. Es handelt sich bei allen Datenstrukturen um C++-Strukturen (engl. *structs*). Für eine Übersicht über alle Mitglieder der jeweiligen Strukturen wird auf Abb. 7 verwiesen. Es werden die für den implementierten Argumentationslöser relevanten Mitglieder kurz erläutert.

- **TaskSpecification** ist eine Datenstruktur zur Speicherung der relevanten Informationen über die vom Argumentationslöser zu lösende Aufgabe. Darunter sind das zu lösende Problem, die AF-Datei und das Format, in dem die Datei vorliegt. In `solve_dc-adm` wird sie insbesondere dafür genutzt, das Argument zu extrahieren, für welches leichtgläubige Akzeptanz nachgewiesen werden soll.
- **`BitSet`** eine Implementation eines *Bitsets*. Es ist ähnlich wie ein *Array* mit Werten von 0 und 1. Wird im Argumentationslöser genutzt, um binäre Infomationen zu speichern. Dabei steht jedes Bit für ein Argument, in dem entsprechenden AF aufsteigend sortiert nach den Argument-IDs. Stellt unter anderem die Basis für Kennzeichnungen dar.
- **`Labeling`** stellt eine Datenstruktur zu den vorgestellten Kennzeichnungen vor. Dabei hat ein `Labeling` je ein `BitSet` für `in`($\mathcal{L}$) und `out`($\mathcal{L}$) sowie ein Mitglied *twoValued*, welches festlegt, ob in der `Labeling` Struktur Werte nur mit `in` und `out` gekennzeichnet werden können oder mit `in, out` und `undec`.
- **AAF** die Datenstruktur für AFs. Speichert alle relevanten Informationen über das zu bearbeitende AF. Dabei bekommt jedes Argument eine fortlaufende Identifikationsnummer angefangen bei 0, um die Verarbeitung im Programm zu vereinfachen. Die Zuordnung von IDs und Argumentnamen ist in den Mitgliedern ids2arguments und arguments2ids gespeichert. Auch die Anzahl aller Argumente und Angriffsrelationen im AF werden gespeichert. Von besonderer Bedeutung für die Implementation von Algorithmus 2 sind die Mitglieder children und parents, welche für jedes Argument A mit der ID i mit children[i] eine Liste von Kindern und mit parents[i] eine Liste von Eltern Argumenten für Argument A zurückgeben.

4.1.2 Hinzugefügte Datenstrukturen

Es wurde eine Datenstruktur hinzugefügt, die auf dem Prinzip der `Labeling` Datenstruktur basiert. Die Implementation der Basisfunktionen wie Initialisierung, Lese- und Schreibzugriff sowie Freigeben des Speichers funktionieren dabei analog zu den Basisfunktionen der `Labeling`s aus taas-fudge. Die hinzugefügte Datenstruktur ist die folgende:

- **`InLRelation`** Diese Datenstruktur implementiert den Verteidigungs- und Angriffsstatus aus Algorithmus 2. Sie speichert das Verhältnis jedes Arguments des entsprechenden AFs zu den Argumenten in `in`($\mathcal{L}$). Dafür gibt es zwei `BitSet`s als Mitglieder (engl. *member*) *attacks* und *defended*. Das Mitglied *attacks* entspricht dem Angriffsstatus aus Algorithmus 2 und speichert analog zu der `Labeling` Struktur aus taas-fudge für jedes Argument A eines AFs, ob A `in`($\mathcal{L}$) angreift. Das Mitglied *defended* entspricht dem Verteidigungsstatus aus Algorithmus 2 und speichert für jedes Argument A des entsprechenden AFs, ob `in`($\mathcal{L}$) sich gegen A verteidigt. Die Menge `in`($\mathcal{L}$) verteidigt sich gegen A, wenn ein Argument B $\in$ `in`($\mathcal{L}$) A angreift. Diese Struktur ermöglicht einen einfachen Test dafür, ob *in*($\mathcal{L}$) sich gegen alle Angreifer verteidigt.

4.2 Projekt-Aufbau

In diesem Abschnitt wird die Ordner-Struktur des adm-fudge Argumentationslösers erläutert. Im Hauptordner befindet sich das Skript `build-adm-fudge.sh`. Durch dessen Ausführung im Terminal wird adm-fudge kompiliert und die *binary*-Datei `adm-fudge` erzeugt. Die Datei `adm-fudge.cpp` enthält den grundlegenden Ablauf des Programms angefangen bei der `main()`-Funktion. Diese Datei ist dafür zuständig die relevanten Datenstrukturen aufzusetzen und `solve_dc-adm` aufzurufen, wenn nötig. Der Hauptordner hat die relevanten Unterordner `adm-fudge_ressources, lib, taas` und `util`. Die Ordner `lib, taas` und `util` wurden aus taas-fudge übernommen. Dabei enthält `lib` die benötigten *Libraries* für adm-fudge. Da adm-fudge keinen externen SAT-Löser nutzt, wurde die CaDiCaL [Lin] *Library* aus diesem Ordner entfernt. Die pstreams *Library* wird weiterhin genutzt. Im `taas` Ordner befinden sich alle relevanten Datenstrukturen und Funktionen, die aus taas-fudge übernommen wurden. Darunter sind die Strukturen für AFs, Kennzeichnungen und alle nötigen Funktionen zum Aufbau dieser Strukturen aus der Eingabe. An keiner dieser Dateien wurden Änderungen vorgenommen. Der `util` Ordner enthält Hilfsfunktionen. Dieser Ordner und sein Inhalt wurden nicht verändert.

Der `adm-fudge_ressources` Ordner enthält den neu hinzugefügten Code. Im Unterordner `adm_util` befinden sich Hilfsfunktionen sowie zur Umsetzung nötige Datenstrukturen. Die Datei `adm_basic_util.cpp` enthält alle benötigten Hilfsfunktionen für `solve_dc-adm`. Die `adm_InLRelation.cpp` Datei enthält die Struktur `InLRelation` sowie alle relevanten Funktionen. Die `adm_setup_util.cpp` Datei enthält alle Hilfsfunktionen zur Initialisierung der Hilfsstrukturen `Labeling` und `InLRelation`. Im `tasks` Ordner befindet sich die Implementation der `solve_dc-adm` Funktion sowie alle unmittelbar dafür nötigen Funktionen.

4.3 Programmablauf

Der allgemeine Programmablauf des entwickelten Argumentationslösers ist von dem Ablauf in taas-fudge inspiriert und wurde an die bearbeitete Problemstellung

angepasst. Der Ablauf beginnt mit dem Aufruf der `main()` Funktion mit einer Eingabe der entsprechenden Parameter im ICCMA-Format. Für DC-CO und DC-PR erfolgt die Eingabe nach dem folgenden Schema:

```
./solver -p <Problem> -fo <Dateiformat> -f <Eingabedatei> -a
<Argument>
```

Dabei ist solver eine kompilierte, ausführbare Datei des Argumentationslösers. Als Problem können für adm-fudge DC-CO oder DC-PR eingegeben werden. Das Dateiformat beschreibt das Dateiformat der Eingabedatei. Der adm-fudge Argumentationslöser unterstützt nur tgf als Dateiformat. Dies muss bei der Angabe des Dateiformats und der Eingabedatei beachtet werden. Für das übergebene Argument wird DC-ADM und damit gleichzeitig DC-CO und DC-PR entschieden. Eine gültige Eingabe für adm-fudge ist z. B. die folgende:

```
./adm-fudge -p DC-CO -fo tgf -f graph.tgf -a 1
```

Bei der Eingabe wird vorausgesetzt, dass graph.tgf ein AF im .tgf Format ist und ein Argument 1 beinhaltet.

Die `main()`-Funktion ruft die `adm__solve()`-Funktion auf. Diese verarbeitet die Eingabe aus dem Programmaufruf. Dafür werden Funktionen aus taas-fudge genutzt, um eine TaskSpecification und eine AAF Struktur aufzubauen, die in `solve_dc-adm` genutzt werden können. Nach dem Aufruf von `solve_dc-adm` werden die erstellten Strukturen zerstört und der Speicherplatz wieder freigegeben. Es wird mit der Rückgabe 0 zu `main()` zurückgekehrt und `main()` gibt das Ergebnis von `adm__solve` zurück. In dem folgenden Unterabschnitt wird auf die Implementation von `solve_dc-adm` eingegangen.

4.4 Umsetzung des Algorithmus

Dieser Abschnitt soll die Implementation von `solve_dc-adm` erläutern. Die vollständige Implementation inklusive aller relevanten Funktionsaufrufe ist im Flussdiagramm aus Abb. 8 dargestellt. Der Startwert für den Zufallszahlengenerator wird auf die derzeitige Systemzeit gesetzt. Dies stellt sicher, dass bei den gleichen Voraussetzungen unterschiedliche Zufallszahl-Sequenzen generiert werden. Danach wird die For-Schleife gestartet. Auf die Wahl der maximalen Iterationen für die Experiment-Durchführung wird in Kapitel 5.1.1 eingegangen.

Wie in Kapitel 4.1.2 erwähnt, implementiert die Datenstruktur `InLRelation` den Angriffs- und Verteidigungsstatus aus der Konzeption. Algorithmus 3 ist unter dem Funktionsnamen `labelIn()` ausprogrammiert. Die Implementierung des Algorithmus 4 hat den Namen `adm__isAdmissible`.

Algorithmus 5 ist unter dem Funktionsnamen `findBFirst()` implementiert. Die Liste bKandidaten wurde durch einen Vektor umgesetzt. Diese Wahl hat den folgenden Grund. Die zufällige Auswahl eines Kandidaten für B geschieht durch die zufällige Wahl eines Index des Vektors mithilfe der C++ `rand()`-Funktion. Daher wird indizierter Zugriff benötigt, welcher bei Vektoren in einer Zeitkomplexität von O(1) möglich ist.

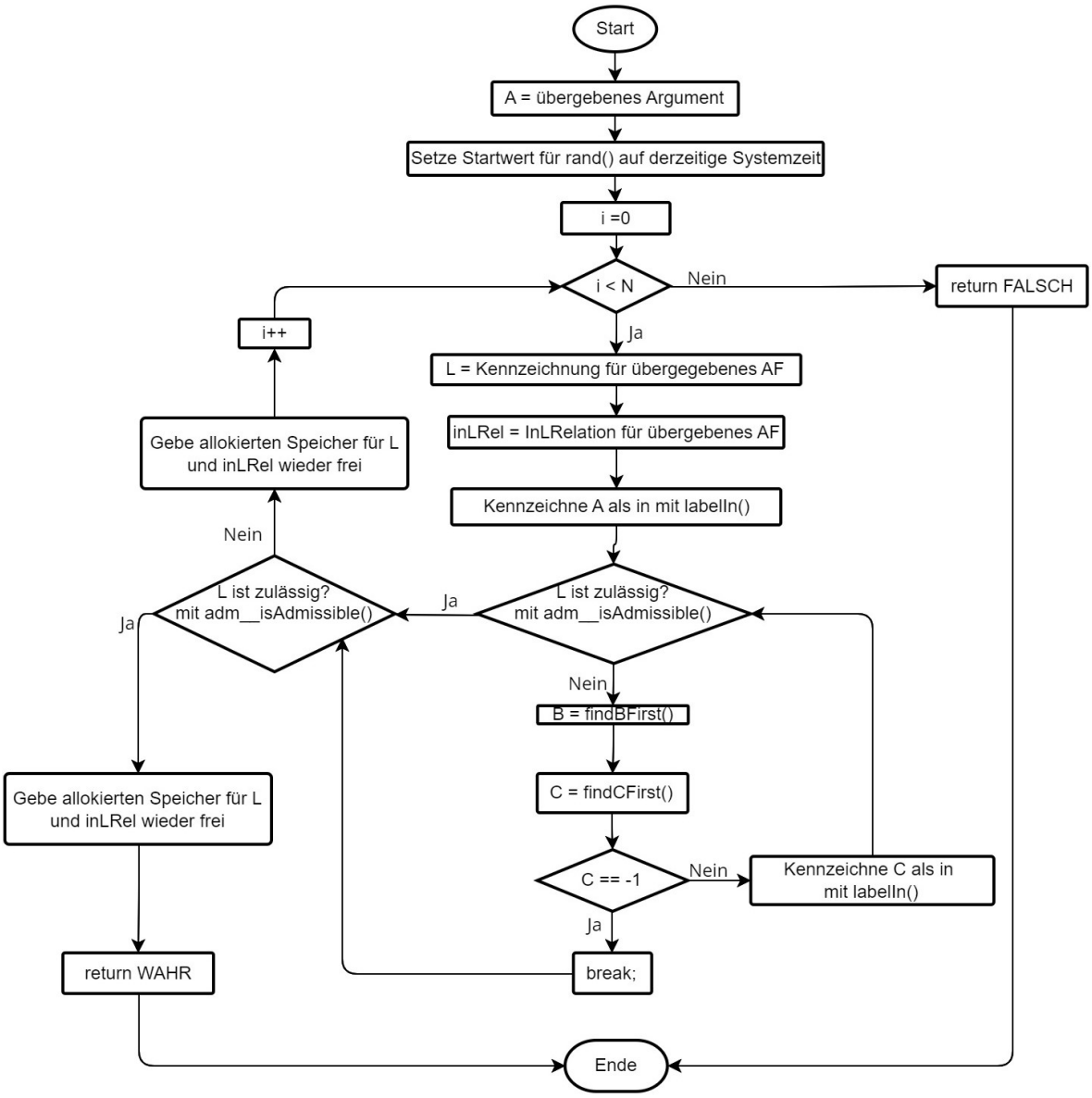

Abbildung 8: Flussdiagramm zum Ablauf der Implementation von Algorithmus 2

Der Algorithmus 6 ist in adm-fudge als `findCFirst()` Funktion ausprogrammiert. Analog zur `findBFirst()`-Funktion wird die Kandidaten-Liste durch einen Vektor umgesetzt.

In diesem Kapitel wurden alle relevanten Informationen zur Implantation des Algorithmus dargestellt. Zu diesen Informationen gehören die verwendeten Datenstrukturen, der Aufbau des Projekts, der Ablauf des Programms sowie Umsetzung des Algorithmus `solve_dc-adm`.

5 Experimente

Im Folgenden werden der Aufbau und die wichtigsten Erkenntnisse der durchgeführten Experimente vorgestellt. Das Ziel der Experimente war es, die Performanz des entwickelten Argumentationslösers im Vergleich zu Argumentationslösern des Stands der Technik zu evaluieren. Alle Experimente und ihre Auswertungen wurden mit probo2 auf einem 64-Bit (x86_64) Ubuntuserver durchgeführt. Die Betriebssystem-Version des Servers ist Ubuntu 20.04.6 LTS (Long Time Support). Probo2 ist ein Programm unter der MIT-Lizenz zum Vergleich von Argumentationslösern. Es bietet alle benötigten Funktionen, um Leistungen der Argumentationslöser zu messen, zu vergleichen und grafisch darzustellen. Die neueste Version von probo2 steht online[7] zur Verfügung.

5.1 Experiment Aufbau

Nachfolgend wird der Experiment-Aufbau zur Rekonstruierbarkeit dargestellt. Dazu wird auf die Vergleichs-Argumentationslöser sowie die verwendeten Benchmarks eingegangen.

5.1.1 Wahl der maximalen Anzahl der Iterationen

Es musste eine Entscheidung über die Anzahl der maximalen Iterationen N getroffen werden. Dafür wurde der Algorithmus mit verschiedenen Werten für N getestet. Da die Anzahl von Argumenten und Angriffen in den AFs der Benchmark-Graphen stark variieren, wurde N in Abhängigkeit des entsprechenden AFs gewählt. Es wurde ein einfacher Laufzeitvergleich mit nur einem einzigen Durchlauf pro Argumentationslöser-Version in probo2 durchgeführt. Die Benchmark-Menge war die in Kapitel 5.1.3 beschriebene Auswahl der ICCMA17-Benchmark Graphen. Der Grund für den einmaligen Durchlauf war, mehr Werte für N vergleichen zu können. Verglichen wurden für jedes $AF = \langle R, T \rangle$ folgende Werte für N:

$|R| \cdot |T|$, $1000 \cdot |R|$, $1000 \cdot |T|$, $1000 \cdot |R| \cdot |T|$, $10000 \cdot |R|$, $10000 \cdot |T|$

Die Ergebnisse des Performanz-Vergleichs der verschiedenen Werte für N ist in Tabellen 1 und 2 dargestellt.

[7] Siehe [Kle23]

N	$\lvert R\rvert \cdot \lvert T\rvert$	$1000 \cdot \lvert R\rvert$	$1000 \cdot \lvert T\rvert$
Timeouts	5	1	11
Richtig gelöste Instanzen	81	80	81

Tabelle 1: Die Anzahl von *Timeouts* (bei 600 Sekunden) und die Anzahl von richtig gelösten Instanzen für einige Werte für N.

N	$1000 \cdot \lvert R\rvert \cdot \lvert T\rvert$	$10000 \cdot \lvert R\rvert$	$10000 \cdot \lvert T\rvert$
Timeouts	16	5	28
Richtig gelöste Instanzen	80	83	85

Tabelle 2: Die Anzahl von *Timeouts* (bei 600 Sekunden) und die Anzahl von richtig gelösten Instanzen für einige Werte für N.

Wie zu erwarten steigt die Anzahl von *Timeouts* bei steigenden Werten von N. Auch die Anzahl von richtig gelösten Instanzen steigt an. Da $N = 1000 \cdot |R|$ mit Abstand zu den wenigsten *Timeouts* geführt hat und dennoch nur 6% weniger Instanzen richtig lösen konnte als $N = 10000 \cdot |T|$, wurde $N = 1000 \cdot |R|$ gewählt. Damit liegt die Laufzeitkomplexität der Implementation von `solve_dc-adm` bei $O(|AR|^3)$ (vergl. Proposition 7).

5.1.2 Auswahl der modernen Argumentationslöser

Da der entwickelte Argumentationslöser auf den Datenstrukturen des taas-fudge Argumentationslösers aufbaut, wird auch dieser als Vergleichs-Argumentationslöser dienen. In der ICCMA21 haben die Argumentationslöser PYGLAF und μ-TOKSIA in den Bereichen DC-CO und DC-PR am besten abgeschnitten. Frühere Versionen dieser Argumentationslöser haben schon in der ICCMA19 gute Ergebnisse in DC-CO und DC-PR erzielt. Da sich diese Argumentationslöser in den zwei für diese Arbeit relevanten Aufgaben etabliert haben, werden sie als moderne Argumentationslöser in den hier durchgeführten Experimenten dienen. Bei allen Argumentationslösern handelt es sich dabei um die Version aus der ICCMA21. Da diese Versionen auf den Versionen der ICCMA19 aufbauen, wurden nur die neuesten Versionen als Vergleichs-Argumentationslöser gewählt. Es folgt eine Vorstellung der Argumentationslöser.

- **FUDGE** nutzt Reduktion auf SAT, um Argumentationsprobleme zu lösen. Zur Lösung von Problemen bezüglich idealer Extensionen sowie der skeptischen Akzeptanz von präferierten Extensionen werden neue Ansätze der Codierung genutzt. FUDGE nutzt die C++ API des SAT-Argumentationslösers CaDiCaL[8], um die SAT-Probleme zu lösen [TCV21a]. Für die Definitionen skeptischer Akzeptanz und idealen Extensionen sowie weiteren Informationen zu genutzten Codierungen verweisen wir auf [TCV21a] und [TCV21b].

[8] Siehe [Lin]

- **PYGLAF** nutzt *circumscription* zur Lösung von Argumentations-Aufgaben. *Circumscription* ist eine nicht-monotone Logik erstmals erwähnt in [McC80]. Dafür nutzt PYGLAF den *circumscription* Argumentationslöser CIRCUMSCRIPTINO, welcher auf dem SAT-Argumentationslöser GLUCOSE [AS09] basiert. Der in Python implementierte Argumentationslöser codiert die Argumentations-Probleme für CIRCUMSCRIPTINO, welcher die Aufgaben löst. In [Alv21, Alv17] ist PYGLAF in mehr Detail beschrieben.

- **μ-TOKSIA** ist ein in C++ implementierter Argumentationslöser, der gegebene Argumentationsprobleme durch Reduktion auf SAT löst. Dabei werden die in SAT codierten Probleme durch den SAT-Argumentationslöser CryptoMiniSat [SNC09] gelöst. Verschiedene Aufgaben verfolgen hier unterschiedliche Ansätze. Für DC-CO reicht der einmalige Aufruf des zugrunde liegenden SAT-Argumentationslösers aus. Für die Lösung von DC-PR Aufgaben werden CEGAR Prozeduren (wie beschrieben in [DJWW14]) genutzt. Der Argumentationslöser nutzt noch weitere Ansätze, die für diese Arbeit jedoch nicht relevant sind und auf welche deshalb nicht eingegangen wird. Für weitere Informationen zu μ-TOKSIA verweisen wir auf [NJ21] und [NJ20].

5.1.3 Benchmarks

Für die Experimente wurden eine Auswahl der Benchmarks der ICCMA17 und ICCMA19 Benchmarks genutzt. Da adm-fudge unvollständig ist und im Falle eines nicht leichtgläubig akzeptierbaren Arguments alle Iterationen durchläuft, wurden bei der Auswahl der Benchmark Graphen nur diejenigen übernommen, die in DC-CO das Ergebnis YES erzielen. Dies sorgt dafür, dass die Experimente in einer angemessenen Zeit durchlaufen.

Um diese Graphen zu identifizieren, wurden für die verschiedenen Benchmark-Mengen unterschiedliche Ansätze gewählt. Für die ICCMA17 liegen die Ergebnisse der DC-CO Aufgabe online[9] vor. Diese Ergebnisse wurden mithilfe von Tabellen-Kalkulationsprogrammen analysiert und die Graphen, die mindestens einer der teilnehmenden Argumentationslöser mit dem Ergebnis YES lösen konnte, wurden in die Auswahl aufgenommen. Für die Benchmark-Graphen der ICCMA19 löste zunächst der μ-TOKSIA Argumentationslöser in probo2 die DC-CO und DC-PR Aufgaben für alle Benchmark-Graphen aus der ICCMA19. Durch die von probo2 generierten Ergebnis-Dateien wurden die für diese Arbeit relevanten Graphen identifiziert und in die Auswahl der verwendeten Benchmark-Graphen übernommen.

Die in dieser Arbeit verwendeten Benchmark-Graphen stehen online[10] zur Verfügung. Es handelt sich hierbei um insgesamt 292 Graphen. Davon sind 115 aus der ICCMA17 und 177 aus der ICCMA19. Im folgenden gehen wir auf die Unterschiede der ausgewählten Benchmark-Mengen ein, da sie sich stark in ihrer Komplexität sowie der Verteilung von Argumenten pro Graph unterscheiden. Aus den Säulen-

[9] Siehe [Thi17]

[10] https://drive.google.com/drive/folders/1PjkLHspbdiXRA9fCMzmqZCL6rXBsXjGI?usp=sharing

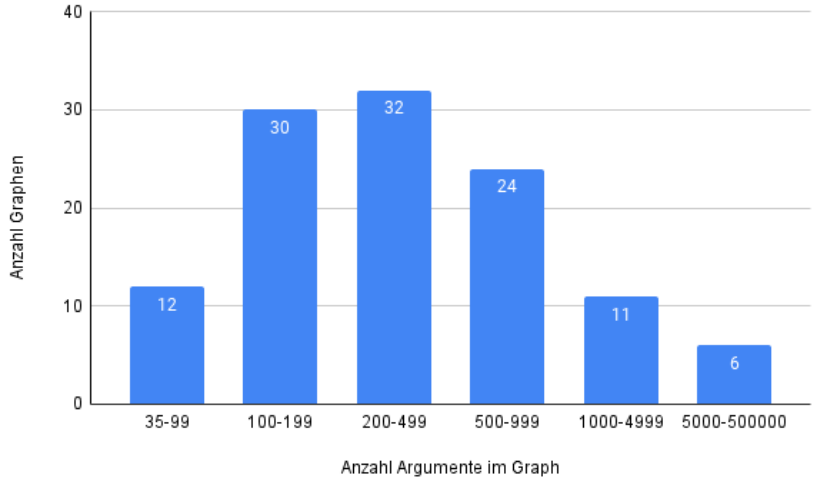

Abbildung 9: Anzahl von Argumenten pro Graph für die Auswahl der ICCMA17 Benchmarks

diagrammen in Abb. 9 und 10 ist ersichtlich, dass sich die zwei Benchmark-Mengen insbesondere durch die Anzahl der Argumente pro Graph unterscheiden. In der ICCMA17 Auswahl liegen die meisten der Graphen bei einer Größe von 100 bis 999 Argumenten. Bei der ICCMA19 Auswahl haben die meisten Graphen unter 100 Argumente. Auch die Minimal und Maximalanzahl von Argumenten pro Graph sind bei den beiden Mengen sehr unterschiedlich. Der kleinste Graph in der ICCMA17 Auswahl hat 35 Argumente, wohingegen der kleinste Graph aus der ICCMA19 fünf Argumente hat. Der größte Graph aus der ICCMA17 hat 500 000 Argumente. Der größte Graph aus der ICCMA19 hat 10 000. Diese Unterschiede in Graph-Größen wirken sich stark auf die Verarbeitungszeit der Argumentationslöser aus. Die Graphen aus der ICCMA17 Auswahl sind größer und damit komplexer als diejenigen aus der ICCMA19 Auswahl.

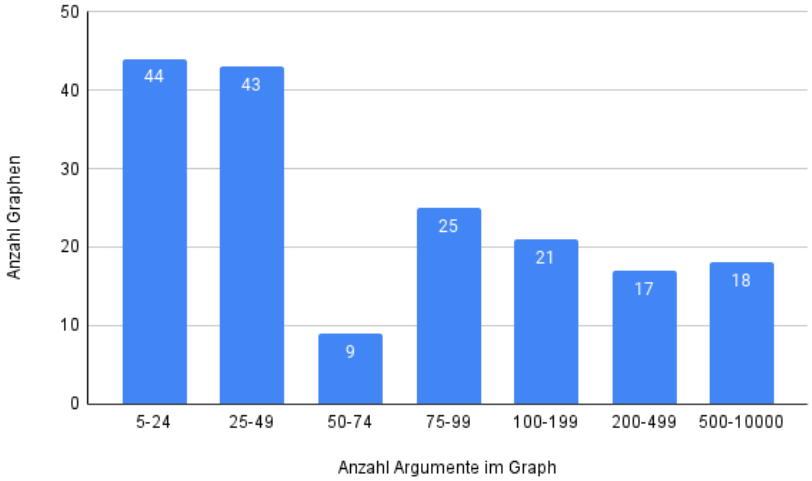

Abbildung 10: Anzahl von Argumenten pro Graph für die Auswahl der ICCMA19 Benchmarks

5.2 Ergebnisse

Nachfolgend werden die Ergebnisse der Argumentationslöser adm-fudge, FUDGE, μ-TOKSIA und PYGLAF für die Lösung des DC-CO Problems dargestellt und interpretiert. Dabei löste adm-fudge das DC-ADM Problem, um DC-CO zu lösen. In den Grafiken ist adm-fudge als adm-final_BA23 angegeben.

	ICCMA17		ICCMA19		Total	
Löser	Positive Ergebnisse	*Timeouts*	Positive Ergebnisse	*Timeouts*	Positive Ergebnisse	*Timeouts*
adm-fudge	87	1	144	1	231	2
FUDGE	95	19	146	0	241	19
μ-TOKSIA	96	18	146	0	242	18
PYGLAF	91	23	146	0	237	23

Tabelle 3: Anzahl der richtig positiv entschiedenen Instanzen und Anzahl der *Timeouts* pro Argumentationslöser

In Tabelle 3 sind die richtig als positiv entschiedenen Ergebnisse jedes betrachteten Argumentationslösers sowie dessen Anzahl von *Timeouts* bezüglich der verschiedenen Benchmark-Mengen sowie insgesamt dargestellt. Es ist gut sichtbar, dass adm-fudge in beiden Benchmark-Mengen von allen Argumentationslösern die wenigsten Graphen richtig positiv entscheiden konnte. Wird μ-TOKSIA als Referenz genommen, konnte adm-fudge in der ICCMA17-Benchmark Auswahl 90,6% der Graphen richtig lösen und in der ICCMA19-Benchmark Auswahl 98,6%. Es fällt

auf, dass adm-fudge der einzige Löser ist, der nicht alle ICCMA19 Graphen richtig lösen konnte. Bei den komplexeren Benchmark-Graphen aus der ICCMA17 führte die Wahl der maximalen Iterationen von adm-fudge mit 1000 mal die Anzahl der Graph-Argumente zu nur einem *Timeout*. Bezüglich der Korrektheit der Ergebnisse ist der entwickelte Argumentationslöser also nicht wettbewerbsfähig mit modernen Argumentationslösern. Betrachten wir Tabelle 4 hat adm-fudge für den ICCMA19 Benchmark mit 5,41 Sekunden die schlechteste durchschnittliche Laufzeit. Für Die Auswahl der ICCMA17 Benchmark Graphen liegt die Laufzeit von adm-fudge weit unter der Durchschnittslaufzeit der anderen Löser. Der nächst schnellste Löser ist μ-TOKSIA, welcher eine 726% höhere Durchschnittslaufzeit aufweist als adm-fudge. Dies ist auch in dem *Cactus Plot* aus Abb. 11 zu beobachten. Der adm-fudge Ar-

	ICCMA17	ICCMA19	Total
Löser	Durchschnittliche Laufzeit	Durchschnittliche Laufzeit	Durchschnittliche Laufzeit
adm-fudge	13,16	5,41	8,46
FUDGE	102,37	0,12	40,39
μ-TOKSIA	95,6	0,23	37,79
PYGLAF	122,22	0,59	48,49

Tabelle 4: Durchschnittliche Laufzeit der Argumentationslöser in Sekunden

gumentationslöser hat eine ähnliche Performanz wie PYGLAF bis der Performanz Graph von PYGLAF kurz vor ca. 90 Instanzen stark ansteigt. Der Graph von adm-fudge hat zwischen ca. 90 und 95 Instanzen eine höhere Steigung als die von μ-TOKSIA und FUDGE. Ab 95 Instanzen bis ca. 112 Instanzen bleibt der Anstieg der CPU-Zeit in Sekunden einheitlich. Danach steigt auch für adm-fudge die Zeit stark an. Diese scheinbar bessere Performanz bei hoher Anzahl von Instanzen kann möglicherweise darauf zurückgeführt werden, dass *Timeouts* in *Cactus Plots* nicht beachtet werden. In adm-fudge ist eine künstliche obere Grenze der Iterationen gesetzt. Bei AFs, für die andere Argumentationslöser in *Timeouts* laufen, entscheidet adm-fudge die Instanz falsch-negativ. Diese falsch-negativen Ergebnisse werden in den *Cactus Plot* aufgenommen, die *Timeouts* der anderen Löser nicht. Im weiteren Verlauf werden die Daten bereinigt, um zu identifizieren, ob der Performanz-Vorsprung von adm-fudge auf die maximale Anzahl der Iterationen zurückzuführen ist. Betrachten wir den *Cactus Plot* aus Abb. 12, sehen wir, dass adm-fudge für die einfacheren Graphen aus der ICCMA19-Auswahl am schlechtesten abschnitt. Dieser Unterschied zwischen den Ergebnissen der beiden Benchmark-Mengen ist signifikant. Der *Cactus Plot* aus Abb. 13 zeigt die Performanz aller Argumentationslöser für alle Graphen aus der ICCMA17 Auswahl, die adm-fudge korrekt lösen konnte. Im Vergleich zu Abb. 11 ist erkennbar, dass der Performanz-Vorsprung zu μ-TOKSIA und FUDGE durch die maximale Anzahl der Iterationen von adm-fudge bedingt war. Dennoch liegt ein Vorsprung zu PYGLAF vor und der Verlauf des Graphen für adm-fudge ist fast vollständig mit dem von μ-TOKSIA überschnei-

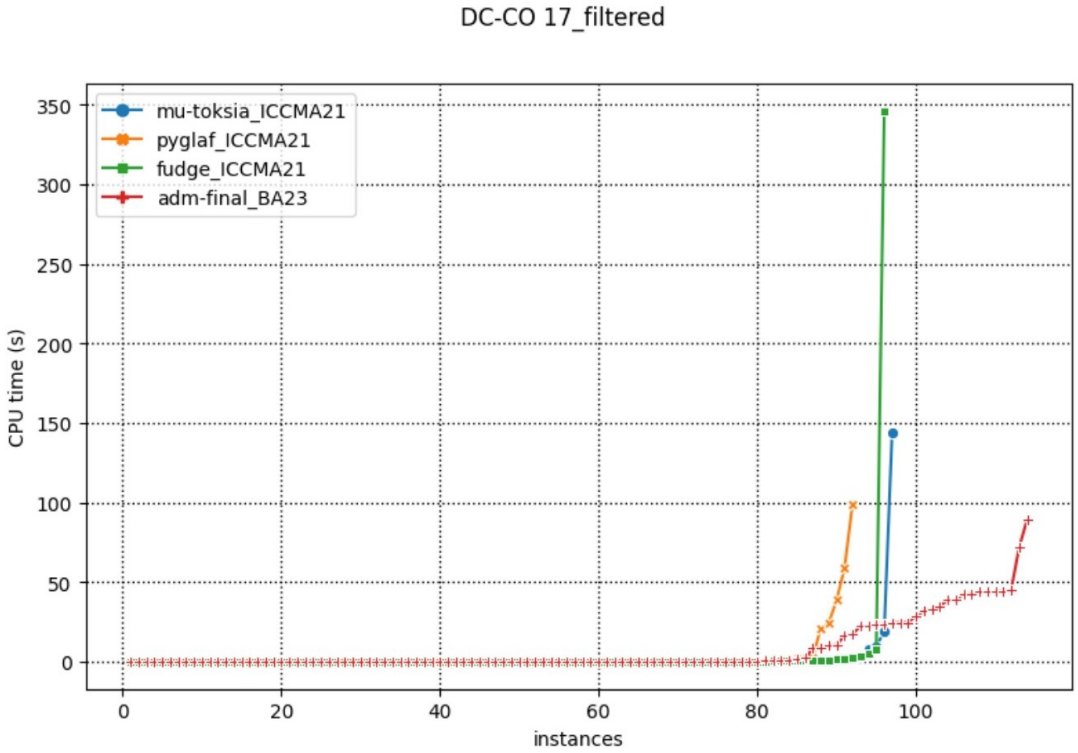

Abbildung 11: Performanzvergleich der Argumentationslöser auf der Auswahl der ICCMA17-Benchmark Graphen

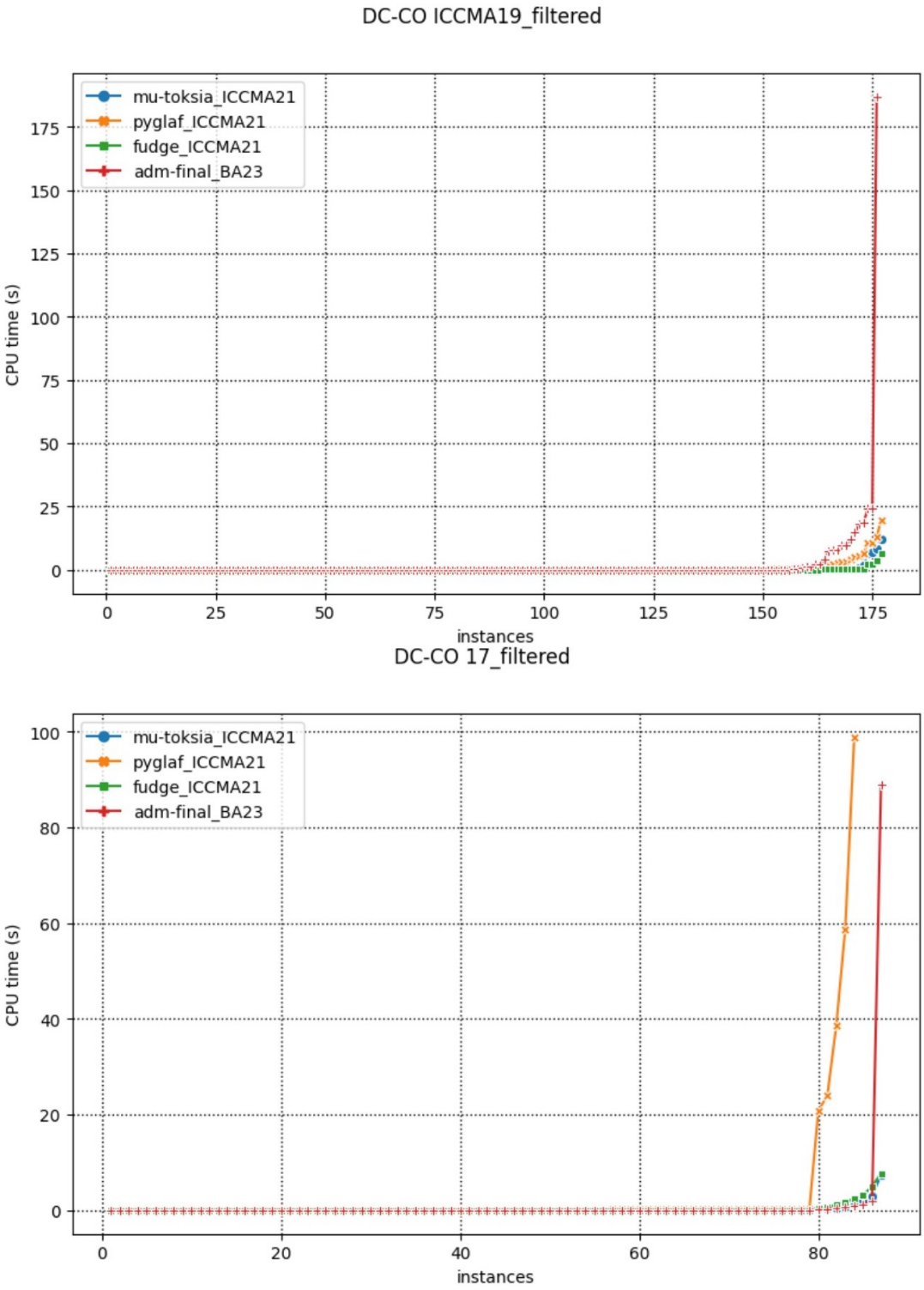

Abbildung 13: Performanzvergleich auf den bereinigten Ergebnissen der Argumentationslöser auf den ICCMA17-Benchmark Graphen. Es wurden nur die Datenpunkte von den Argumentationsgraphen übernommen, die adm-fudge im Experiment lösen konnte.

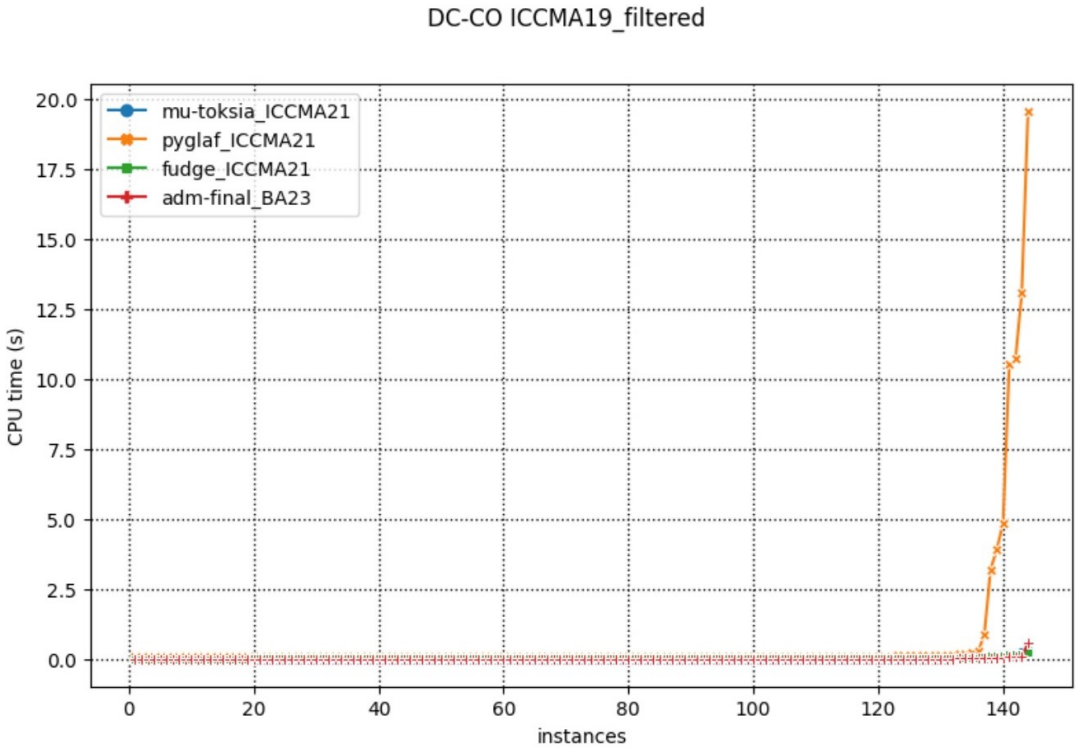

Abbildung 14: Performanzvergleich auf den bereinigten Ergebnissen der Argumentationslöser auf den ICCMA19-Benchmark Graphen. Es wurden nur die Datenpunkte von den Argumentationsgraphen übernommen, die adm-fudge im Experiment lösen konnte.

dend. Dieselbe Beobachtung entsteht bei der Abb. 14. Die Analyse der Daten zeigt, dass adm-fudge den Großteil der vorgelegten Aufgaben korrekt lösen kann (in den Experiment-Ergebnissen 90,6% - 98,6%). Der im Vergleich zu anderen Argumentationslösern hohe Anteil von falsch negativen Ergebnissen war zu erwarten, da nach dem Ansatz der stochastischen Lokalsuche randomisiert vorgegangen wird. Der signifikante Unterschied in Performanz bei den verschiedenen Benchmark-Mengen ist ebenfalls auf diese falsch-negativen Ergebnisse zurückzuführen. Ein falsch negatives Ergebnis bedeutet, dass adm-fudge die For-Schleife bis zum Erreichen von der maximalen Anzahl von Iterationen durchlaufen hat. Bei den komplexen Graphen aus der ICCMA17 Auswahl benötigt die strategische Verarbeitung, die die anderen Löser verfolgen mehr Prozessorzeit als das Durchlaufen aller Iterationen der For-Schleife für adm-fudge. Bei den simplen Graphen aus der ICCMA19 Auswahl ist das vollständige Durchlaufen der For-Schleife laufzeitintensiver als das strategische Vorgehen der anderen Löser. Damit kommen wir zu dem Schluss, dass adm-fudge wettbewerbsfähig ist, wenn die maximale Anzahl von Iterationen auf die Komplexität und Größe der Benchmark-Graphen angepasst ist.

Dieses Kapitel zeigt den Aufbau und die Ergebnisse der durchgeführten Experimente. Der entwickelte Algorithmus zeigt bei einer angemessenen Wahl von maximalen Iterationen überdurchschnittlich geringen Laufzeiten. Dabei ist der Abstrich der Genauigkeit der Ergebnisse verhältnismäßig gering.

6 Fazit

Diese Arbeit beschreibt die Konzeption, Implementation und Evaluation eines stochastischen Lokalsuche Algorithmus zur Lösung des leichtgläubigen Akzeptanzproblems bezüglich zulässiger Extensionen. Dafür wurde der Algorithmus-Aufbau sowie der Aufbau dessen direkter Unterfunktionen dargestellt und Designentscheidungen erklärt. Die Umsetzung des Algorithmus inklusive des Aufbaus des Programms sowie der genutzten Datenstrukturen wurde aufgezeigt. In dem nachfolgenden Experiment wurde nachgewiesen, dass der entwickelte Argumentationslöser bei einer angemessen gewählten maximalen Anzahl von Iterationen einen Großteil der Aufgaben zur leichtgläubigen Akzeptanz bezüglich zulässiger Extensionen in einem Bruchteil der Zeit lösen kann, die hochmoderne Argumentationslöser benötigen. Dabei muss die maximale Anzahl von Iterationen auf die Größe der Test-AFs bezüglich der Anzahl der Argumente angepasst werden. Der entwickelte Argumentationslöser eignet sich zur schnellen Identifikation von AFs, für welche das leichtgläubige Akzeptanzproblem bezüglich vollständiger, präferierter oder zulässiger Extensionen erfüllt ist. Die Instanzen, für die der entwickelte Argumentationslöser entschieden hat, dass sie nicht leichtgläubig akzeptiert werden können, sollten jedoch von einem Löser überprüft werden, der systematisch vorgeht. Diese zweite Überprüfung kann falsch negative Entscheidungen des entwickelten Argumentationslösers identifizieren. Der schwerwiegendste Nachteil des entwickelten Argumentationslösers ist der hohe Anteil von falsch negativen Ergebnissen, welcher durch den Ansatz der stochastischen Lokalsuche bedingt ist. Diese Eigenschaft des Lösers beeinträchtigt seine Wettbewerbsfähigkeit von allen am meisten. Die in [Thi18, SKC99] dargestellten *greedy moves* könnten die Performanz in diesem Bereich verbessern. Es handelt sich um gelegentlich durchgeführte systematische Aktionen, die zu der Lösung des gestellten Problems führen sollen. In dem Falle des bearbeiteten leichtgläubigen Akzeptanzproblems bezüglich zulässiger Mengen könnte zum Beispiel der Verteidiger in einem *greedy move* zur Extension hinzugefügt werden, welcher diese vor den meisten Angreifern verteidigt. Ob die vorgeschlagene Änderung zu einer Reduktion von falsch negativen Ergebnissen führt, muss in weiterführender Forschung beurteilt werden.

Zusammenfassend wird in dieser Arbeit ein vielversprechender Ansatz zur Lösung des leichtgläubigen Akzeptanzproblems bezüglich zulässiger Mengen erforscht. Der entwickelte Argumentationslöser zeichnet sich bei korrekt gesetzten Parametern durch eine überdurchschnittlich geringe Laufzeit aus, obwohl er eine leicht höhere Quote falsch-negativer Ergebnisse aufweist. In weiterführender Forschung könnten Schwachpunkte des Algorithmus identifiziert und verbessert werden, um

seine Wettbewerbsfähigkeit weiter zu erhöhen.

Literatur

[Alv17] Mario Alviano. Ingredients of the Argumentation Reasoner pyglaf: Python, Circumscription, and Glucose to Taste. 2017.

[Alv21] Mario Alviano. THE PYGLAF ARGUMENTATION REASONER (ICCMA2021). `http://argumentationcompetition.org/2021/downloads/pyglaf.pdf`, 2021. (Online. Zugriffsdatum: 03.08.2023).

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, July 2009. Morgan Kaufmann Publishers Inc.

[BCG11] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *Knowledge Eng. Review*, 26:365–410, December 2011.

[Bie09] Armin Biere, editor. *Handbook of satisfiability*. Number v. 185 in Frontiers in artificial intelligence and applications. IOS Press, Amsterdam, The Netherlands ; Washington, DC, 2009. OCLC: ocn290492523.

[Cam06] Martin Caminada. On the Issue of Reinstatement in Argumentation. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *Logics in Artificial Intelligence*, Lecture Notes in Computer Science, pages 111–123, Berlin, Heidelberg, 2006. Springer.

[CDG+15] 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.

[CG09] Martin W. A. Caminada and Dov M. Gabbay. A Logical Account of Formal Argumentation. *Studia Logica*, 93(2):109, November 2009.

[CGTW17] Federico Cerutti, Sarah A Gaggl, Matthias Thimm, and Johannes P Wallner. Foundations of Implementations for Formal Argumentation. *IFCoLog Journal of Logics and Their Applications*, 4(8), 2017.

[DBC02] Paul E. Dunne and T. J. M. Bench-Capon. Coherence in finite argument systems. *Artificial Intelligence*, 141(1):187–203, October 2002.

[DJWW14] Wolfgang Dvořák, Matti Järvisalo, Johannes Peter Wallner, and Stefan Woltran. Complexity-sensitive decision procedures for abstract argumentation. *Artificial Intelligence*, 206:53–78, 2014.

[DM04] Sylvie Doutre and Jérôme Mengin. On Sceptical Versus Credulous Acceptance for Abstract Argument Systems. In Jóse Júlio Alferes and João

Leite, editors, *Logics in Artificial Intelligence*, Lecture Notes in Computer Science, pages 462–473, Berlin, Heidelberg, 2004. Springer.

[Dun95] 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.

[Hei21] Maximilian Heinrich. The MatrixX Solver For Argumentation Frameworks, September 2021.

[JLN23] Matti Järvisalo, Tuomo Lehtonen, and Andreas Niskanen. ICCMA 2023: 5th International Competition on Computational Models of Argumentation. `https://iccma2023.github.io/`, 2023. (Online. Zugriffsdatum: 01.09.2023).

[Kle23] Jonas Klein. probo2. `https://github.com/aig-hagen/probo2`, 3 2023. (Github. Online. Zugriffsdatum: 01.09.2023).

[Lar21] Malmqvist Lars. AFGCN: AN APPROXIMATE ABSTRACT ARGUMENTATION SOLVER, 2021.

[Lin] Johannes Kepler Universität Linz. Cadical simplified satisfiability solver. `https://fmv.jku.at/cadical/`. (Online. Zugriffsdatum: 01.09.2023).

[LLMR21] Jean-Marie Lagniez, Emmanuel Lonca, Jean-Guy Mailly, and Julien Rossit. Design and Results of ICCMA 2021, October 2021. arXiv:2109.08884 [cs].

[Mal22] Lars Malmqvist. *Approximate Solutions to Abstract Argumentation Problems Using Graph Neural Networks*. phd, University of York, July 2022.

[McC80] John McCarthy. Circumscription—A form of non-monotonic reasoning. *Artificial Intelligence*, 13(1):27–39, 1980.

[NJ20] Andreas Niskanen and Matti Järvisalo. µ-toksia: An Efficient Abstract Argumentation Reasoner. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 800–804, 9 2020.

[NJ21] Andreas Niskanen and Matti Järvisalo. µ–TOKSIA AT ICCMA'21. `http://argumentationcompetition.org/2021/downloads/mu-toksia.pdf`, 2021. (Online. Zugriffsdatum: 03.08.2023).

[oJ] Dudenredaktion (o. J.). „Framework" auf Duden online. `https://www.duden.de/node/291047/revision/1307857`. (Online. Zugriffsdatum: 01.09.2023).

[Rod18] Odinaldo Rodrigues. An investigation into reduction and direct approaches to the computation of argumentation semantics. *Festschrift in Honour of Tarcisio Pequeno, Tributes. College Publications, To appear*, 2018.

[SKC99] Bart Selman, Henry Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 26, September 1999.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. July 1992.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584, pages 244–257. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[TCV21a] Matthias Thimm, Federico Cerutti, and Mauro Vallati. FUDGE: A LIGHT-WEIGHT SOLVER FOR ABSTRACT ARGUMENTATION BASED ON SAT REDUCTIONS. `http://argumentationcompetition.org/2021/downloads/fudge.pdf`, 2021. (Online. Zugriffsdatum: 03.08.2023).

[TCV21b] Matthias Thimm, Federico Cerutti, and Mauro Vallati. Skeptical Reasoning with Preferred Semantics in Abstract Argumentation without Computing Preferred Extensions. May 2021.

[TCV23] Matthias Thimm, Federico Cerutti, and Mauro Vallati. taas-fudge. `https://github.com/aig-hagen/taas-fudge`, 3 2023. (Github. Online. Zugriffsdatum: 01.08.2023).

[Thi17] Matthias Thimm. ICCMA17 - Competition dc-co results. `http://argumentationcompetition.org/2017/DC-CO.results`, 2017. (Online. Zugriffsdatum: 01.08.2023).

[Thi18] Matthias Thimm. Stochastic Local Search Algorithms for Abstract Argumentation under Stable Semantics. In Sanjay Modgil, Katarzyna Budzynska, and John Lawrence, editors, *Proceedings of the Seventh International Conference on Computational Models of Argumentation (COMMA'18)*, volume 305 of *Frontiers in Artificial Intelligence and Applications*, pages 169–180, Warsaw, Poland, September 2018.

[Thi19] Matthias Thimm. ICCMA - Competition 2021. `http://argumentationcompetition.org/2021/index.html`, 2019. (Online. Zugriffsdatum: 01.09.2023).

[Thi21] Matthias Thimm. Harper++: Using Grounded Semantics for Approximate Reasoning in Abstract Argumentation. In *The Fourth International Competition on Computational Models of Argumentation (ICCMA'21)*, May 2021.