Implementing an Argumentation Game for ADFs under Preferred Semantics

Bachelor's Project Thesis

Marieke Bouma, m.bouma.10@student.rug.nl
Supervisors: A. Keshavarzi Zafarghandi, MSc. & Prof. Dr. L.C. Verbrugge

Abstract: Formal argumentation can be used to model and evaluate reasoning, making it a powerful support for AI. Further, AI methods can help find answers to logical problems in formal argumentation. This project explores Abstract Dialectical Frameworks (ADFs), an argumentation formalism structured as a network of arguments. A propositional formula is attached to each argument, to indicate the conditions under which an argument can be accepted and to show the type of relation between arguments.

One logical problem in ADFs is the credulous decision problem, which describes the question of whether there exists a set of truth values for the arguments under certain semantics in which a specific argument has a given value. Keshavarzi presents a discussion game algorithm between two players, which solves this problem under preferred semantics. In this game, the players take turns trying to find new information given a claim the previous player made about the truth values of arguments. Depth-First Search is applied to find the final set of truth values needed to solve the credulous decision problem. If DFS fails, then the initial claim is not satisfiable. This project consists of the implementation of this discussion game in Python.

Keywords: abstract dialectical frameworks · discussion game · argumentation theory

1 Introduction

Argumentation has recently received increased attention within Artificial Intelligence (AI). Especially argumentation formalisms such as abstract representations of arguments are a popular topic. One such formalism is the concept of Abstract Frameworks (AFs), described first by Dung (1995). An AF has the structure of a graph, in which each node represents an argument. The content of the arguments is not of importance here; rather, the focus lies with the relations between arguments. In AFs, each edge from one argument to another represents an attack relation. To illustrate, consider two contradicting arguments A and B. Then the arguments could be represented as in Figure 1.1.

Figure 1.1: AF consisting of two contradicting arguments.

To generalise this concept, Brewka and Woltran (2010) introduced Abstract Dialectical Frameworks (ADFs). Contrary to AFs, the relations between arguments in ADFs are quite flexible: they can not only be attacking, but also supporting, both attacking and supporting, or neither. The flexibility of relations between arguments in ADFs is realised through the definition of a so-called *acceptance condition* for each argument. This acceptance condition is in fact a propositional formula, which allows us to make inferences about the truth values of arguments. Continuing the example in Figure1.1, the acceptance conditions of a and b are $\neg b$ and $\neg a$, respectively.

In AFs and ADFs, **semantics** are specific criteria used to settle the acceptance of arguments. Though there are several types of semantics, this project focuses on preferred semantics, where we want as much as possible information about the truth values of arguments to be known. These semantics are very powerful tools to answer logical problems encountered in ADFs.

One such problem is the credulous decision problem. For a given ADF, the key question here is: "does there exist a set of truth values for the arguments in the ADF, such that a given argument has a given truth value?".

A number of theoretical solvers for this problem exist, such as K++ (Linsbichler, Maratea, Niskanen, Wallner, and Woltran, 2018) and YADF (Brewka, Diller, Heissenberger, Linsbichler, and Woltran, 2017a). However, the role of discussion in the reasoning used to solve such problems in ADFs, has not been elaborated upon so far. To further investigate this role, Keshavarzi Zafarghandi, Verbrugge, and Verheij (2019) have developed the first existing discussion game algorithm to solve the credulous decision problem of ADFs under preferred semantics.

In this game, two players (a proponent and opponent) take turns trying to find new information based on information the previous player has presented about the truth values of certain arguments. This is called a *claim* when presented by the proponent, or a *challenge* when presented by the opponent. The game starts with an *initial claim* by the proponent. Depth-First Search is applied to find the final set of truth values needed to satisfy the initial claim. If DFS fails, then the initial claim is not satisfiable; otherwise, it is.

This project consists of the implementation of the discussion game in Python. As such, this thesis consists of a theoretical discussion of ADFs, preferred semantics, and the theoretical part of the discussion game, and a practical description of the implementation of the game, and experiments to evaluate the implementation.

Testing is done mainly by testing benchmark inputs for which the correct outputs are already known, and tracking computation time to evaluate different sub-algorithms. All in all, the research question we will try to answer in this thesis is this:

- How can the credulous decision problem for ADFs under preferred semantics be solved automatically by a discussion game implemented in an efficient object-oriented program?

The contents of this thesis will follow the structure of this question: first the concepts of ADFs, preferred semantics, and the credulous decision problem will be introduced. Then, we introduce the theory of the discussion game, and lastly discuss the core of the project, namely the implementation of this game in a program, and the testing and analysis of the program and its efficiency.

2 Background

2.1 Abstract Argumentation Frameworks (AFs)

Abstract argumentation frameworks (AFs), first concretely described by Dung (1995), are directed network graphs, where each node represents an argument and each edge represents an attack relation between two arguments.

As we will see later, other relations between arguments can be described by more complex types of frameworks. At the base, though, AFs are defined as follows:

Definition 1. *An abstract argumentation framework F is a pair $\langle Ar, R\rangle$ where Ar is a set of arguments, and R is a set of attacks between two arguments in Ar, i.e. $R \subseteq Ar \times Ar$. An element of R, e.g. an attack from a to b for $a, b \in Ar$, is denoted as $att(a, b)$ or $(a, b) \in R$.*

Within these frameworks, there is much we can do with the information that we have. Most importantly, we want to find sets of arguments which can be pooled together. The principles used to indicate which sets can be accepted together are called *semantics*.

In AFs, the semantics are defined based on two ways of pooling together arguments, the first of which is called *extensions*. The second way of pooling together arguments is called *labelling*. The notion of labelling will be explained in Section 2.1.1.

One of the notions useful for our current purpose is that of a *conflict-free* extension, in which none of the arguments attack each other. Another is an *admissible* extension, which is not only conflict-free, but in which each argument is also "defended" from its attackers, by some argument in the extension. A defense here means that the attacker of an argument is in turn attacked by some argument. Formally, these definitions are:

Definition 2. *Let $S \subseteq Ar$ and let $a \in S$. Then a is considered defended by, or acceptable w.r.t. S if for any $(b, a) \in R$ there exists $c \in S$ such that $(c, b) \in R$.*

Definition 3. *A set $S \subseteq Ar$ of arguments is called conflict-free if there are no internal conflicts in the set; i.e., there are no two arguments $a, b \in S$ such that a attacks b. Note that a and b can also be the same argument.*

Definition 4. *A set $S \subseteq Ar$ of arguments is called an admissible extension of an abstract argumentation framework F if*

(1) it is conflict-free, and

(2) each argument $a \in S$ is acceptable with respect to S, i.e. a is defended by S.

For a given argumentation framework F, we can find the set of conflict-free and the set of admissible extensions. We denote these sets, respectively, as $cf(F)$ and $adm(F)$. Note that the empty set $\varnothing$ is a conflict-free and admissible extension of any F. Let's take a look at an example:

Example 1.
Let $F := \langle\{a, b, c, d\}, \{(a, b), (b, c), (c, b), (d, d)\}$ be an AF, depicted in Figure 2.1 below. Then $cf(F) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, c\}\}$. Note that $\{d\}$ is not conflict-free because d attacks itself. From these conflict-free sets, we can select the admissible sets: $adm(F) = \{\varnothing, \{a\}, \{c\}, \{a, c\}\}$.

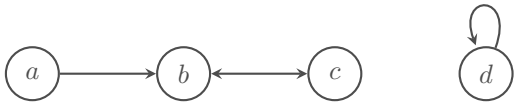

Figure 2.1: AF of Example 1

Although quite a number of other semantics have been defined for AFs, in this work the focus lies with the notion of preferred semantics.

Definition 5. *A preferred extension of an abstract argumentation framework F is an admissible set S that is maximal with respect to set inclusion; that is, all admissible sets that are pure subsets of S, are not preferred extensions. A preferred extension of F is denoted $prf(F)$.*

In Example 1, we get $prf(F) = \{\{a, c\}\}$. To illustrate the criterion of maximal set inclusion, here is another example:

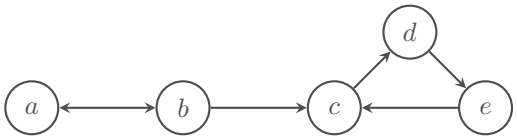

Figure 2.2: AF of Example 2

Example 2.
Let $F := \langle\{a, b, c, d, e\}, \{(a, b), (b, a), (b, c), (c, d), (d, e), (e, c)\}$ be an AF, depicted in Figure 2.2. Then $cf(F) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, c\}, \{a, d\}, \{a, e\}, \{b, d\}, \{b, e\}\}$, and $adm(F) = \{\varnothing, \{a\}, \{b\}, \{b, d\}\}$. We get $prf(F) = \{\{a\}, \{b, d\}\}$.

Note that since the empty set $\varnothing$ is a subset of all non-empty sets, we have that $\varnothing$ is a preferred extension if and only if there are no non-empty admissible sets of F. In this case, $\varnothing$ is of course also the **only** preferred extension of F.

2.1.1 Labelling

A labelling is a mapping of each argument in an ADF to either *in*, *out*, or *undec*, each of which represents a truth value, i.e. a function $lab : Ar \longmapsto \{in, out, undec\}$. Here, *in* is true, *out* is false, and *undec* stands for undecided. The first two mappinsg can also be seen as an argument being "in" or "out" of the set of accepted arguments. In the context of AFs, a labelling is a specific representation of the set of all arguments in an AF, such that each argument has a label. Labelling can be used as a tool to find arguments in AFs that can be accepted together.

The transition from extensions to labellings happens via a function called *Ext2Lab*. This function was defined by Baroni, Caminada, and Giacomin (2011), and works as follows: for an extension ε we check for each argument in our framework, its relation to ε. Based on this, we label the argument *in*, *out*, or *undec*. Formally, the definition is as follows:

Definition 6. *Let ε be an extension of an AF F. Then for an argument $a \in F$,*

$$Ext2Lab(\varepsilon)(a) = \begin{cases} in, & \text{iff } a \in \varepsilon \\ out, & \text{iff } \exists\ b \in \varepsilon \text{ s.t. } (b, a) \in R \\ undec, & \text{otherwise.} \end{cases}$$

Thus, a labelling for an AF F is a set containing all arguments of F, with their labels. To illustrate, let us look at the AF in Example 2 again.

Example 2 (continued)**.**
We have $\{a\} \in prf(F)$*, so* $Ext2Lab(prf(F))(a) = in$*, i.e. our labelling must contain* $a \to in$*. Since* a *attacks* b*, we have* $b \to out$*. We do not know the label of* e *yet, so* $c \to undec$*, and in turn* $d \to undec$ *and* $e \to undec$*, since the only attackers of those arguments are labelled undec as well. Thus,* $Ext2Lab(\{a\}) = \{a \mapsto in, b \mapsto out, c \mapsto undec, d \mapsto undec, e \mapsto undec\}$*.*

For our second preferred extension, $\{b, d\}$*, we get* $\{b \to in, d \to in\}$*. Since* d *is an attacker of* e*, we must get* $e \to out$*. Since* b *attacks* a *and* c*, we know that* $\{a \mapsto out, c \mapsto out\}$*. Thus,* $Ext2Lab(\{b, c\}) = \{a \mapsto out, b \mapsto in, c \mapsto out, d \mapsto in, e \mapsto out\}$*.*

2.2 Abstract Dialectical Frameworks (ADFs)

A powerful generalisation of the concept of AFs is that of abstract dialectical frameworks (ADFs). The main difference between AFs and ADFs is that the relations between arguments in ADFs are much more flexible. The flexibility of relations between arguments in ADFs is realised through defining so-called *acceptance conditions* for each argument.

ADFs were defined first by Brewka and Woltran (2010), who revised their definition twice later on (Brewka, Ellmauthaler, Strass, Wallner, and Woltran, 2013, 2017b). ADFs are defined formally below, in Definition 7.

Definition 7. *An abstract dialectical framework* F *is a tuple* $\langle A, L, C\rangle$ *where* A *is a set of arguments (or statements),* L *is a set of relations (or links) between two arguments in* A *(i.e.* $L \subseteq A \times A$*), and* C *is a set of acceptance conditions.*

The acceptance condition for an argument $a \in A$, denoted φ_a, is a propositional formula. This way, we can have many different links between two arguments. The four types of links between arguments and some examples are denoted in Table 2.1.

While the attacking and supporting links are rather straightforward, the links of "both" and "neither" may require some explanation. The handbook of ADFs (Brewka et al., 2017b) is a useful source to clarify this.

Link	Example φ_a
Attacking	$\varphi_a : \neg b$
Supporting	$\varphi_a : c$
Both	$\varphi_a : d \vee \neg d$
Neither	$\varphi_a : e \iff f$

Table 2.1: Types of links between arguments b, c, d, e, f **and** a**, and some examples of corresponding acceptance conditions of** a**.**

To paraphrase and applied to the examples in Table 2.1: a link such as (d, a) is both supporting and attacking, and called *redundant*, because switching the truth value of d does not change the evaluation of φ_a. By contrast, the links (e, a) and (f, a) are neither supporting nor attacking, and also called *dependent*, because the influence of the truth value of e on the evaluation of φ_a depends on the truth value of f, and vice versa.

The *parents* of an argument $a \in A$ in an ADF F are all the arguments with a link to a, i.e. $\{b \in A \mid (b, a) \in L\}$. If an argument does not have any parents, it is called an *initial argument* and its acceptance condition is either $\top$ or $\bot$.

2.2.1 Semantics of ADFs

ADFs use *interpretations* to pool together arguments in a useful way. An interpretation, denoted v, is a function which maps a statement a to a truth value $\mathbf{x}$, where $\mathbf{x} \in \{\mathbf{t},\mathbf{f},\mathbf{u}\}$ (standing for true, false, undecided). Thus, for example, an interpretation can be $v = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{u}, c \mapsto \mathbf{f}\}$. To make for better readability, an interpretation can also be rewritten to the sequence of the truth values of the arguments in lexicographic order. The example interpretation above then becomes $v = \mathbf{tuf}$.

We can perform an *informational ordering* on two interpretations, i.e. order them based on how much information they give us about the truth values of the arguments. Naturally, if an argument is labelled u, then it does not give us as much information as it would if that argument would be labelled true or false. More formally, for the truth values $\{\mathbf{t},\mathbf{f},\mathbf{u}\}$, we have that

- $\mathbf{u} <_i \mathbf{t}$ and $\mathbf{u} <_i \mathbf{f}$.
- $\mathbf{t} =_i \mathbf{t}$, $\mathbf{f} =_i \mathbf{f}$, and $\mathbf{u} =_i \mathbf{u}$.
- $\leq_i$ is the transitive and reflexive closure of $<_i$.

The semantics of ADFs are based on interpretations being created via the *characteristic operator*. Given an interpretation v, we want to "update" it; in other words, we want to use the information in v to form a new interpretation v'. The characteristic operator Γ_F does this by evaluating the acceptance condition of each argument.

Definition 8. *The characteristic operator for an ADF F, denoted Γ_F, is a function on interpretations v. It takes each argument $a \in A$ and performs*

$$\Gamma_F(v)(a) = \begin{cases} \boldsymbol{t}, & \text{iff } \varphi_a^v = \top \text{ (irrefutable)} \\ \boldsymbol{f}, & \text{iff } \varphi_a^v = \bot \text{ (unsatisfiable)} \\ \boldsymbol{u}, & \text{otherwise} \end{cases}$$

In this equation, for p a parent of argument a,

$$\varphi_a^v : [p/\top : v(p) = \boldsymbol{t}][p/\bot : v(p) = \boldsymbol{f}].$$

Just as before, we can get information about interpretations by looking at admissibility and preferredness. For an interpretation v to be admissible, its "updated" interpretation $v' = \Gamma_F(v)$ needs to contain more or equally as much information. This also means that all arguments a that are mapped to **t** or **f** in v must remain mapped to that truth value in $\Gamma_F(v)$. The formal definition by Brewka et al. (2013) is as follows:

Definition 9. *A three-valued interpretation v for an ADF F is admissible iff $v \leq_i \Gamma_F(v)$.*

Naturally, a preferred interpretation must be admissible. Next to that, it is an interpretation that contains as much information as we can possibly have gotten from the acceptance condition of each argument. Formally, this is:

Definition 10. *A three-valued interpretation v for an ADF F is preferred iff it is $\leq_i$-maximal admissible.*

To illustrate, let us look at another example ADF and the logical steps taken to find its admissible and preferred interpretations. Example 3 shows the game play for a simple ADF, whose admissible interpretations are displayed by informational ordering in the lattice on the right side of Figure 2.3.

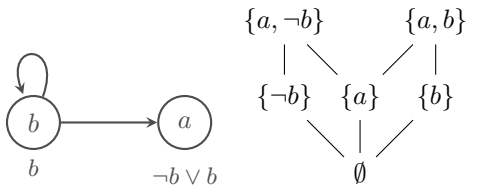

Figure 2.3: ADF of Example 3 and informational ordering of its admissible interpretations.

Example 3. *Let F an ADF as in Figure 2.3.*

To find $adm(F)$ and $prf(F)$, we continuously evaluate the acceptance conditions of a and b under a given interpretation v, and check if $v \leq_i \Gamma_F(v)$. For some interpretations, we can reason about admissibility without explicitly testing the acceptance conditions.

Since we have $\varphi_a = \neg b \vee b$ a tautology, $\varphi_a = \top$ no matter the truth value of b. We start with an initial $v_u = \boldsymbol{uu}$:

$v_u = \boldsymbol{uu}$
$\varphi_a^{v_u} = \top, \varphi_b^{v_u} = b$
$\Gamma_F(v_u) = \boldsymbol{tu}$
so v_u is admissible.

Since the acceptance condition of a is irrefutable, $\boldsymbol{tu}$ is also admissible, but $\boldsymbol{fu}$ is not.

$v_1 = \boldsymbol{ut}$
$\varphi_a^{v_1} = \top, \varphi_b^{v_1} = \top$
$\Gamma_F(v_1) = \boldsymbol{tt}$
so v_1 is admissible.

$v_2 = \boldsymbol{uf}$
$\varphi_a^{v_2} = \top, \varphi_b^{v_2} = \bot$
$\Gamma_F(v_2) = \boldsymbol{tf}$
so v_2 is admissible.

Again, since a has no influence on any argument, $\boldsymbol{tt}$ and $\boldsymbol{tf}$ are admissible as well. However, since $\varphi_a^v = \top$ for all v, we have that $\boldsymbol{ft}$ and $\boldsymbol{ff}$ are both not admissible. Thus:

$adm(F) = \{\emptyset, \boldsymbol{tu}, \boldsymbol{ut}, \boldsymbol{uf}, \boldsymbol{tt}, \boldsymbol{tf}\}$
$prf(F) = \{\boldsymbol{tt}, \boldsymbol{tf}\}$. □

2.3 Preferred Discussion Game

A central problem in the study of ADFs is the *credulous decision problem*. It describes the evaluation of the truth value of an argument in a given ADF. The *credulous* acceptance (or denial) of an argument under a given semantics is based on the mere existence of any interpretation in that semantics under which the argument is acceptable (deniable).

Definition 11. *Given an ADF $F = \{A, L, C\}$, an argument $a \in A$ under a given semantics σ is called* credulously acceptable *if there exists an interpretation $v \in \sigma(F)$ such that φ_a is irrefutable under v. Similarly, an argument is credulously deniable if there exists an interpretation $v \in \sigma(F)$ such that φ_a^v is unsatisfiable.*

A number of algorithms have been developed to deal with this problem, among which the K++ solver (Linsbichler et al., 2018) and YADF (Brewka et al., 2017a), but also the current work of my supervisor, Atefeh Keshavarzi Zafarghandi. Her work focuses on the credulous decision under preferred semantics, which is realised through a discussion game between two players, first described in 2019 (Zafarghandi et al., 2019).

2.3.1 Overview

The discussion game entails a dialogue between a proponent P and an opponent O. P starts the game and first dialogue by presenting an assignment $a \mapsto \mathbf{x}$, where $\mathbf{x} \in \{\mathbf{t}, \mathbf{f}\}$. This assignment is called the *initial claim*. Then the dialogue continues, until one of two conclusions is drawn:

- An agreement is found, which means that there exists indeed an admissible interpretation which assigns $a \mapsto \mathbf{x}$.* P wins the game.
- A contradiction is reached in the current dialogue. One interpretation at a time, P must search for an opportunity to start an alternative dialogue. If all possible dialogues have been found and none reach an agreement, then P is convinced that the initial claim was wrong, and P loses the game.

*While the algorithm aims to answer the credulous decision problem under preferred semantics, it is enough to find an admissible interpretation, because if that exists, then there must also exist some preferred interpretation that contains the assignment described by the initial claim.

The search for this agreement is essentially Depth-First Search: if a dead end is reached in the search tree, another branch is investigated, until either an agreement is found or all options yield a dead end. Thus, following the description of the discussion game as a search tree, one dialogue is one branch. A dialogue is a series of interpretations, which are represented as nodes in the search tree.

2.3.2 Game play algorithms

The first dialogue in the game starts by P making an *initial claim*, which can be written as an interpretation v_0 in which the relevant argument is assigned to the claimed truth value, and all other arguments are assigned to $\mathbf{u}$.

After that, the players take turns in which they apply either the *forward* or *backward* move, until the game stops. To determine which move is applied, each player first evaluates the informational ordering between the most recently presented interpretation v and the previous interpretation w. This is called the *checking step*. The goal is that each newly presented interpretation v in the dialogue contains at least as much information as the previous interpretation w, that is, $w \leq_i v$.

If $w <_i v$, the dialogue is continued through the forward move. If $w =_i v$, then v is indeed an admissible interpretation that satisfies the initial claim, and thus an agreement has been found and P wins the game. However, if $w \not\leq_i v$, there is a contradiction. If P found the contradiction, P applies the backward move, but if O found the contradiction, O simply does not apply any move, and P applies the backward move. The backward move is represented as backtracking in Depth-First Search.

2.3.3 The forward move

The first of the two moves in the game is the forward move. From the checking step, the player knows that there is some set of arguments which were not defined yet in w, but are in v, since $w <_i v$. This set of arguments is called the set of *recently presented* arguments in v with respect to w, and is defined formally in Definition 12.

Definition 12. *The set A' of recently presented argument(s) in a new interpretation v w.r.t the previous interpretation w (i.e. $w \leq_i v$), are the arguments for which $w(a) = \boldsymbol{u}$, and $v(a) \neq \boldsymbol{u}$.*

After checking the informational ordering, each player has two tasks, in the given order:

(1) Checking the consequences of the given interpretation v on the truth value of the recently presented argument(s). This is done through evaluating the acceptance conditions of those arguments under v. For a recently presented argument $a \in A'$, the acceptance condition under v is denoted φ_a^v.

(2) Finding a new interpretation. From the acceptance conditions φ_a^v of each $a \in A'$, the player finds the *minimal satisfiable interpretations* of each a, formally presented in Definition 13. The player then combines that information with v through the *forward move* (see also Definition 14), and presents the new interpretation to the other player. In the case of P, this is called a new *claim*; in the case of O, a new *challenge*.

Definition 13. *Let $a \in A'$ be recently presented in v. A minimal satisfiable interpretation of a is $a <_i$-minimal interpretation over a or parents of a that are in φ_a^v, such that $v(a)$ is satisfied. It is found through the relation $mSAT_F(\varphi_a^v)$ (abbreviated to $mSAT_a$), which is defined using the cases in Equation 1 on page 8.*

To make some more sense of all this, consider the following example.

Example 4. *Example dialogue for a simple ADF $D = \{\{a, b\}, \{(a, b), (b, a)\}, \{\varphi_a = \neg b, \varphi_b = \neg a\}\}$.*

P: initial claim: $\exists v \in prf(D)$ such that $v(a) = \boldsymbol{t}$, i.e. $v_0 = \boldsymbol{tu}$.

O:
- *Indeed $\boldsymbol{uu} <_i v_0$, so the dialogue can continue and $A' = \{a\}$.*
  - *(1) Evaluates $\varphi_a^{v_0} = \neg b$. In this case, there are no changes.*
  - *(2) Finds that $a \mapsto \boldsymbol{t}$ only holds if $b \mapsto \boldsymbol{f}$, i.e. $mSAT_a = \boldsymbol{uf}$. Challenges: "if I agree on $a \mapsto t$, then prove that $b \mapsto f$", i.e. $v_1 = \boldsymbol{tf}$.*

P:
- *Indeed $v_0 <_i v_1$, so the dialogue can continue and $A' = \{b\}$.*
  - *(1) Evaluates $\varphi_b^{v_1} = \neg\top = \bot$.*
  - *(2) Finds that v_1 is enough to answer O's challenge ($mSAT_b = \{a \mapsto \boldsymbol{t}\}$), so $v_2 = v_1$.*

O:
- *We have that $v_1 =_i v_2$, so an agreement is found and the dialogue stops. The game is won by P.*

□

The second step in each turn is the forward move. This move, defined formally in Definition 14, essentially consists of combining the information found in the minimal satisfiable interpretation(s) with the information given in the previous interpretation, yielding a new interpretation to be added to the dialogue.

Definition 14. *For A' the set of arguments $\{a_1, ..., a_n\}$ recently presented in v w.r.t w, and $mSAT_{A'} = \{mSAT_{a_1}, ..., mSAT_{a_n}\}$ the set of minimal satisfiable interpretations $mSAT_{a_i}$ for a_i w.r.t v, for $1 \leq i \leq n$, the forward move is a binary function $\delta(v, mSAT_{A'})$, which is defined using the cases in Equation 2 on page 8.*

Thus, in the forward move, the player looks at the truth value of each recently presented argument a, and checks for any conflicts with the minimal satisfiable interpretations of the other recently presented arguments.

For example, if argument a is defined in v, but there is a conflict in the parents of a in the minimal satisfiable interpretation of any other argument, the truth value of a in the new interpretation is $\mathbf{u}$. On the other hand, if we already had $v(a) = \mathbf{u}$, and a is a parent of any other recently presented arguments a_i, the value of a is $mSAT_{a_i}(a)$ if and only if there are no conflicts between that $mSAT_{a_i}(a)$ and any $mSAT_{a_j}(a)$, for other arguments a_j of which a is a parent and for which $mSAT_{a_j}(a) = \mathbf{t}/\mathbf{f}$.

2.3.4 Dialogues and the backward move

As we have seen, each forward move yields an interpretation. These interpretations are stored together in a dialogue, defined formally as follows:

Definition 15. *A dialogue is a sequence of interpretations $[v_0, ..., v_j]$ for $j \geq 1$, in which all the following conditions hold:*

$$mSAT_a = \begin{cases} w & \text{if } \varphi_a^v \not\equiv \top/\bot \ \wedge \exists w \mid \Gamma_F(w)(a) = v(a) \ \wedge \neg\exists w' \mid (w' <_i w \ \wedge \Gamma_F(w')(a) = v(a)), \\ \{a \mapsto \Gamma_F(v)(a)\} & \text{if } \varphi_a^v \equiv \top/\bot \ \vee \neg\exists w \mid \Gamma_F(w)(a) = v(a) \end{cases}$$

Equation 1: The function $mSAT_a$ to find a minimal satisfiable interpretation for an argument $a \in A'$.

$$\delta(v, mSAT_{A'})(a) = \begin{cases} v(a) & v(a) = \mathbf{t}/\mathbf{f} \wedge a \notin A', \\ v(a) & v(a) = \mathbf{t}/\mathbf{f} \wedge \varphi_a^v = \top/\bot, \\ v(a) & v(a) = \mathbf{t}/\mathbf{f} \ \wedge mSAT_a \neq \{a \mapsto \Gamma_F(v)(a)\} \ \wedge \\ & \neg\exists a_i, c \ s.t. \ (c \in par(a) \wedge mSAT_a(c) \neq mSAT_{a_i}(c)), \\ mSAT_{a_i}(a) & v(a) = \mathbf{u} \ \wedge \exists a_i \in A' \ s.t. \ (mSAT_{a_i}(a) = \mathbf{t}/\mathbf{f} \wedge \\ & \neg\exists a_j \in A' \ s.t. \ mSAT_{a_i}(a) \neq mSAT_{a_j}(a)), \\ \mathbf{u} & otherwise. \end{cases}$$

Equation 2: The forward move $\delta(v, mSAT_{A'})(a)$ is used to form a new interpretation.

- *v_0 is an initial claim;*
- *for each $i > 0$, $v_i = \delta(v_{i-1}, mSAT_{A'})$ such that A' is the set of arguments recently presented in v_{i-1} and $mSAT_{A'}$ is a minimal satisfiable interpretation of A';*
- *for each $0 \leq i < j - 1, v_i <_i v_{i+1}$.*

As discussed, the forward move is applied until a contradiction is found because $w \not\leq_i v$. In this case, the backward move is applied. This move consists of P either searching for another claim, or asking O for another challenge, thus for another dialogue D'. The backward move is formally defined in Definition 16.

Definition 16. *For an initial claim v_0, a dialogue $D = [v_0, ..., v_n]$ blocked by a contradiction and S_D the corresponding set of minimal satisfiable interpretations of D, the binary function β takes D and S_D and returns $D' = [v_0, ..., v_{j'}]$ and $S_{D'}$, which satisfy the following conditions:*

- *$[v_0, ..., v'_j]$ is a dialogue with $1 \leq j \leq i$ and $v'_j \neq v_j$;*
- *the part $[v_0, ..., v_{j-1}]$ of D' is equal to the first part $[v_0, ..., v_{j-1}]$ of D;*
- *$D' = [v_0, ..., v'_j]$ is the maximal alternative dialogue, in the sense that there is no dialogue $[v_0, ..., v'_k]$ that is equal to a part $[v_0, ..., v_{k-1}]$ of D such that $j < k$ and $v'_k \neq v_k$;*
- *$S_{D'}$ is the set of minimal satisfiable interpretations for D'.*

If no dialogue $[v_0, ..., v'_j]$ with $j \geq 1$ that satisfies these conditions exists, then β returns $D' = [v_0]$ and S_D.

This is where Depth-First Search comes in: in a search tree, the backward move makes the search go to a different branch. O wins if the entire search tree has been traversed and no agreement is found.

In theory, P is responsible for finding another claim or challenge, since P is the player with the goal of finding an agreement. If n is even, then P "asks" O to present another challenge, and if n is odd, then P tries to present another claim. In practice, this search, nor any of the other moves and processes discussed, differ between players.

Another difference to note between the theoretical and practical steps of the backward move is that a player applying the backward move must know the evaluated acceptance condition φ_a^v of an argument a for which an unused $mSAT_a$ is to be found. In theory, φ_a is evaluated under v again, but in practice, this step does not yield any new results since v and φ_a have not changed. As such, in the examples, the evaluating step of the backward move is not shown.

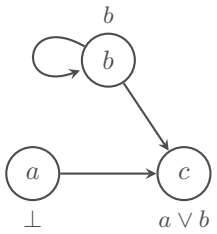

Figure 2.4: ADF $F = \{\{a, b, c\}, \{(a, c), (b, b), (b, c)\}, \{\varphi_a = \bot, \varphi_b = b, \varphi_c = a \vee b\}\}$**.**

Example 5 showcases the checking step and backward move. The example ADF is shown in Figure 2.4 above.

Example 5. *Example dialogue for an ADF* $F = \{\{a, b, c\}, \{(a, c), (b, b), (b, c)\}, \{\varphi_a = \bot, \varphi_b = b, \varphi_c = a \vee b\}\}$.

P: initial claim: $\exists v \in prf(D)$ *such that* $v(c) = \boldsymbol{t}$*, i.e.* $v_0 = \boldsymbol{uut}$*.*

O: – $v_0 = \boldsymbol{uut}$ *contains strictly more information than* $\boldsymbol{uuu}$*, so the dialogue continues.* $A' = \{c\}$*.*

(1) Evaluates $\varphi_c^{v_0} = a \vee b$*.*

(2) Finds the set of minimal satisfiable interpretations for $\varphi_c^{v_0}$ *and presents* $mSAT_c = \boldsymbol{tuu}$*. This yields* $\delta(v_0, mSAT_{A'}) = \boldsymbol{tut} = v_1$*.*

P: – $v_0 <_i v_1$*, so the game continues.* $A' = \{a\}$*.*

(1) Evaluates $\varphi_a^{v_1} = \bot$*.*

(2) Finds $mSAT_a^{v_1} = \boldsymbol{fuu}$*. Since there is a contradiction between* $mSAT_a^{v_1}(a)$ *and* $v_1(a)$*,* $\delta(v_1, mSAT_{A'}^{v_1})(a) = \boldsymbol{u}$*, so* $\delta(v_1, mSAT_{A'}^{v_1}) = \boldsymbol{uut} = v_2$*.*

P: – $v_1 \not\leq_i v_2$*, so the dialogue* $[v_0, v_1, v_2]$ *is blocked.*

(2) Searches for another result for $mSAT_a^{v_1}$*, but none exists. Asks O to present another challenge, i.e. another* $\delta(v_0, mSAT_c)$*.*

O: – *Just as before,* $\boldsymbol{uuu} <_i v_0$*, so there is no contradiction.* $A' = \{c\}$*.*

(2) Finds another result for $mSAT_c$ *with* $\varphi_c^{v_0}$*, namely* $mSAT_c' = \boldsymbol{utu}$*. Applies the forward move,* $\delta(v_0, mSAT_c') = \boldsymbol{utt} = v_1'$*.*

P: – $v_0 <_i v_1'$*, so the dialogue* $D' = [v_0, v_1']$ *continues.* $A' = \{b\}$*.*

(1) Evaluates $\varphi_b^{v_1'} = \top$*.*

(2) Finds $mSAT_b = \boldsymbol{utu}$*, so* $\delta(v_1', mSAT_b) = \boldsymbol{utt} = v_2'$*.*

O: – $v_1 =_i v_2$*, so* $A' = \{\}$*, and an agreement is found. P is the winner.* □

3 Implementation

The main goal of this project is the implementation of the preferred discussion game. As such, this section describes the practical matter of implementing relevant algorithms. The original code can be found at `https://github.com/piekb/adfs`.

3.1 General

The program takes as input an ADF. In the input file, the ADF is represented as follows: each line either describes the existence of an argument a in the ADF or the acceptance condition of an argument a. For example, when $\varphi_a = \neg b \vee (a \wedge b)$ and $\varphi_b = \top$, the input file looks as follows:

```
s(a).
s(b).
ac(a,or(neg(b),(and(a,b)))).
ac(b,c(v)).
```

Upon execution, the program gets the name of the input file from the command line, and then prints an overview of the ADF. Then, the program gets the argument and truth value of the initial claim from the command line. Throughout the execution of the program, interpretations at nodes of the search tree are printed, and the steps taken (forward, backward, or conclusion of the game) are printed. This way, the user can reconstruct the dialogue that led to the outcome of the game.

To deal with logical expressions in the implementation, being able to import external libraries and packages is almost unavoidable. Moreover, to represent certain data types, object oriented programming proves useful. To this end, the chosen programming language was Python.

3.2 Data structures

At the base level of an ADF, there are arguments and interpretations that assign truth values to those arguments. In the program, an interpretation is represented as a string of truth values, one for each argument alphabetically.

We need some way of representing an argument. Making a class Argument provides a solution here. The Argument type contains a string `Name` and a propositional formula `ac` to represent its acceptance condition. Since the propositional formula is made using the Python library Logic, an Argument also has a Logic symbol made from its name to deal with certain restrictions by the library. Lastly, an argument has an int `dex` for index, which is used to find the assignment of the argument to a truth value in an interpretation.

3.2.1 Search tree

As described before, the search for an agreement in the game happens via Depth-First Search (DFS). A straightforward tree structure is used, by simply constructing one `Root` from the initial claim, and for each node a new `Node`.

A `Root` contains `data` (the interpretation string) and `children` (a list of `Node`s). A `Node` contains these elements as well as one `parent` node, to make navigating through the tree a bit simpler. Both of these structures contain a function `add_child` that initializes a child `Node` and adds it to the `children` of the current node.

When a number of minimal satisfiable interpretations is found for a node, the set of these interpretations becomes `n.msats`. This is explained further in Section 3.4.1.

Whenever a new interpretation is found through the forward move, that interpretation is given as the data of a new child node, and the new node is accessed. For the backward move, the tree applies standard DFS backtracking. This is elaborated on in Section 3.5.

3.3 Informational ordering

The first key task in the algorithm is performing the checking step, in which the informational ordering of two interpretations is found. Using an old interpretation w and new interpretation v, we must have $w \leq_i v$, or we get a contradiction. In practice, an advantage of this step is that one function can combine detecting a contradiction, detecting an agreement ($w =_i v$), and finding the set of recently presented arguments A'.

To do this, the program must loop through v using the counter variable `i`, and check for each assignment whether it coincides with the assignment of that argument in w. This happens in the function `check_info`, which takes two interpretations and performs Algorithm 3.1. The algorithm performs three checks:

- if w`[i]` == **u** and v`[i]`! = **u**, add the argument at place `i` to `a_prime`;
- if w`[i]`! = **u** and v`[i]`! = w`[i]`, indicate that there is a contradiction;
- if after looping through v, `a_prime` is empty, indicate that there is an agreement.

Algorithm 3.1 Pseudo-algorithm for `check_info`.

```
a_prime ← []
contra ← False
found ← False

for i in enumerate(v) do
    if w[i]='u' then
        if v[i]!='u' then
            append v[i] to a_prime
        end if
    else if v[i]!=oldv[i] then
        contra ← True
    end if
end for

if a_prime is empty then
    found ← True
end if
```

3.4 Forward move

To apply the forward move, the program first finds a set of minimal satisfiable interpretations, i.e. $mSAT_{A'}$. This set is used by the function `forward`, which loops through the set of arguments A' and finds $\delta(v, mSAT_{A'})$ for each argument.

The implementation follows the five cases from the function for δ as outlined in the theory. The first, second, and fifth case from δ are distinguished in regular `if`-statements, while there are separate functions `third` and `fourth` to take the steps described on the right-hand side of the third and fourth cases in Equation 2 on page 8.

Both of the latter functions make use of another function `no_conflict`. For a given argument a and the mSAT for some other argument a_i, named $mSAT_{a_i}$, the function checks whether there is any other $a_j \in A'$ for which $mSAT_{a_j}(a) \neq mSAT_{a_i}(a)$. This function is called as `no_conflict(msat_ai,a_prime,a)`; in the third case of δ, it is called once for every parent c of a, i.e. `no_conflict(msat_a,a_prime,c)`. In the fourth case of δ, it is called just once, for a itself.

3.4.1 Minimal satisfiable interpretations

To search for a minimal satisfiable interpretation that has not been used before, one straightforward strategy would be to simply generate all satisfiable interpretations for φ_a for $a \in A'$, and then find the set of those interpretations with minimal length[†].

However, there is one important problem with this strategy: the computational complexity. To find the full satisfiable interpretations for φ_a, we need to first generate all interpretations of length $n-1$, where n is the number of arguments in the ADF. Then we must loop through all of them, pad a **u** on the place of a, and check whether the resulting interpretation satisfies φ_a.

Using the options **t,f,u**, we get a truth table of $n-1$ propositional atoms for three truth values, which results in an exponential complexity: this table has 3^{n-1} rows. Moreover, padding the **u** at the place of a requires another loop of n iterations.

Random generation of interpretations

Another approach to finding interpretations could be to try out random truth value assignments over the parents of the argument for which an interpretation must be found. This is more computationally efficient than computing the full set of satisfiable interpretations.

To improve this further, we can:

- check whether this random interpretation actually satisfies the acceptance condition of the argument, before using the interpretation in δ;
- keep track of a "blacklist" of already used sets of interpretations (one per argument in A') by a node, so that these sets are not used twice.

One thing to note about this method is that the satisfiable interpretations found are not always minimal. Another, more pressing issue is the following. If we use a completely random distribution of truth value assignments, but the only satisfiable interpretation is already in the blacklist, the search will go on forever. Without the blacklist, this problem occurs as well, only in this form the one satisfiable interpretation would just be used in a new branch of the tree and cause a contradiction again and again. For this reason, an optimization of the first approach seems the better option here.

Algorithm 3.2 Pseudo-algorithm for smart computation of the set of mSATs for an argument $a \in A'$ with `k` parents.

```
inters ← combinations of t,f,u of length k
cnt ← k-1
msats ← [ ]
for inter in inters do
    if inter.count('u') < cnt then
        if msats not empty then
            break
        else
            cnt-=1
        end if
    end if
    m ← '', j ← 0
    for i in range(n) do          ▷ Pad with u's
        if argument at i is in parents(a) then
            m ← inter[j], j+=1
        else
            m ← m +' u'
        end if
    end for
    if m satisfies phi(a) then
        msats.append(m)
    end if
end for
return msats
```

[†]Note that this section only describes the implementation of the first case of the formal definition of $mSAT_a$; if $\varphi_a^v \equiv \top/\bot$, the program returns $\{a \mapsto \Gamma_F(v)(a)\}$ as $mSAT_a$.

Improving computational search

One first optimization of the non-random search for mSATs is to check satisfiability only for interpretations over the parents of an argument $a \in A'$. This more closely follows the theory than checking satisfiability for interpretations over all arguments.

Though in many cases this strategy requires less satisfiability checking, the program still loops through 3^k interpretations, where k is the number of parents of a. This means that in the worst case, i.e. when all arguments of the ADF (except a) are parents of a, we must still check the satisfiability of 3^{n-1} interpretations.

A way to further improve the complexity of the search is to sort the set of interpretations and keep track of the number of arguments not assigned to **u** (i.e. the "level" of informational ordering). This way, we only need to check satisfiability for one level at a time. If any satisfiable interpretations have been found at one level, then the next level need not be checked; the loop simply breaks. This is the algorithm used in the final version of the program, and is given in Algorithm 3.2 on page 11.

In the worst case, i.e. when the minimal satisfiable interpretation is also maximal w.r.t informational ordering, we still need to check the satisfiability of 3^k interpretations. Moreover, the padding with **u** slows down the program quite a bit for increasing size of the ADF. In better cases, though, the overall complexity of this version of the algorithm is much lower than the previously presented algorithms.

The implementation used for finding mSATs is a modification of the function given in the theory, since the full set of mSATs is returned and attached to the node in the search tree. To make the program a bit more dynamic by maintaining some kind of randomization, a random set of mSATs (one for each $a \in A'$) is chosen to continue the game with, and that set of mSATs is removed from the list of options for the node, called `n.msats`. This way, no options are used twice, but the options are not used in a fixed order.

Example 6 shows how mSATs for multiple recently presented arguments are combined, such that each combination can be safely removed from `n.msats` when accessed.

Example 6. *Consider an ADF of arguments* $\{a, b, c, d\}$ *where, at some point in solving an initial claim, we have* $A' = \{a, d\}$*. Say we have* $\varphi_a = b \vee c$*, and* $\varphi_d = \bot$*. Then*

- *for* φ_a*, we have two options for* $mSAT(a)$*: either* ***utuu****, or* ***uutu****.*
- *for* φ_d*, we have only* $mSAT(d) =$ ***uuuf****.*

A list of two sets is returned to `n.msats`*, each of which has the form* $\{a : mSAT(a),\ d : mSAT(d)\}$*. In this example, we get:*

`n.msats =` [{`a:utuu,d:uuuf`},{`a:uutu,d:uuuf`}]

One of these two options is chosen randomly, used for the forward move, and removed from `n.msats`*. In case of a contradiction, the backtracking step checks if there are any sets of the form* $\{a : mSAT(a),\ d : mSAT(d)\}$ *left.*

3.5 Backward move

When a contradiction is found in one node of the search tree, the program must backtrack. For clarity, let v_n be a node that caused a contradiction.

First, it is checked whether v_n is the root node, i.e. represents the initial claim, in which case P automatically loses and the program finishes. Otherwise, from v_n, the program enters a loop in which it is checked whether another set of $mSATs$ exists under the parent node, v_{n-1}. If no other set exists, the program backtracks further.

The loop is exited either when the program has backtracked all the way to the root node without finding another branch, or when an unused set of $mSATs$ is found under some node. Again, if the current node is the root, P loses, but otherwise, the forward move is applied on v_{n-i} using the newly found set of $mSATs$, and the resulting interpretation is stored as the data of a new child node which is then accessed.

The algorithm for backtracking is Algorithm 3.3 on page 13. For brevity, when the `data` of a node `n` is passed to a function, the algorithm states `n` instead of `n.data`.

As mentioned earlier in this report, a dialogue in the game is represented as a branch of the search tree. Since the backtracking algorithm for the tree structure solves the implementation of the backward move, we opted out of implementing dialogues and the function β from the theory explicitly.

Algorithm 3.3 Pseudo-algorithm for backtracking

```
    if n is Root then break          ▷ P loses game.
    end if

    n ← n.parent
 5:
    while n is not Root & found_msat=False do
        n ← n.parent
        a_prime ← check_info(n, n.parent)[0]
        if n.msats is not empty then
10:         m ← random.choice(n.msats)
            remove m from n.msats
            found_msat ← True
        end if
    end while
15:
    if n is Root then break          ▷ P loses game.
    else
        update ← forward(n, a_prime, msat)
        n.add_child(update)
20:     n ← n.children[i+1]
    end if
```

Algorithm 3.4 Pseudo-algorithm for the game.

```
    n.msats ← find_msat(n, a_prime)
    m ← random.choice(n.msats)
    remove m from n.msats
    update ← forward(v_0, a_prime, m)
 5: n.add_child(update)
    n ← n.children[0]

    while True do
        check ← check_info(n, n.parent)
10:     a_prime, contra, found ← check
        if contra then backtrack
        else if found then break     ▷ P wins game.
        else
            n.msats ← find_msat(n, a_prime)
15:         m ← random.choice(n.msats)
            remove m from n.msats
            update ← forward(n, a_prime, m)
            n.add_child(update)
            n ← n.children[0]
20:     end if
    end while
```

3.6 Full algorithm

The algorithm that controls the moves of the game from the main program is one that initializes the tree structure, followed by continuous applications of the forward move until either

- a contradiction is found, at which point we backtrack, or
- an agreement is found, at which point the game ends with P as the winner.

As explained in Subsection 3.3, contradictions and agreements are found by the checking step, which is performed at the beginning of the loop. As explained in Subsection 3.5, if at any stage of backtracking we have reached the root node, the game ends with P losing.

The full algorithm of the game is given more schematically in Algorithm 3.4. Again, for brevity, when the `data` of a node `n` is passed to a function, the algorithm states `n` instead of `n.data`. At the conclusion of the game, the full search tree is printed. If P has won the game, the "winning" interpretation is printed, along with "YES". If P has lost the game, the program outputs "NO".

4 Testing

Due to the practical nature of the project, experimenting consists first and foremost of testing whether the implementation works. To do so, a number of different inputs have been tested. These inputs range from rather basic, such as the ADF of Example 4 on page 7, to complex ADFs with notable acceptance conditions, so that specific elements of the program can be tested.

These special elements include:

- correctly evaluating different acceptance conditions: $\neg, \vee, \wedge, \top, \bot, \iff, \implies$, but also tautologies such as $\varphi_a = d \vee \neg d$, and contradictions such as $\varphi_d = \neg d$;
- searching for alternative children for nodes at any depth of the search tree, and continuing the game with a new branch if needed;
- finding alternative mSATs in a smart way. A key requirement here is that the computational complexity is not too high; see also Section 3.4.1.

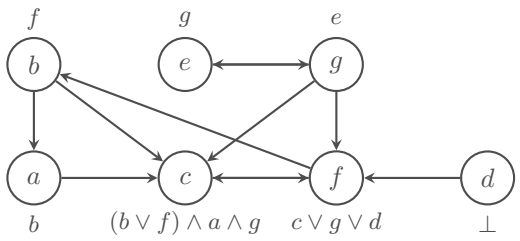

Figure 4.1: Complex ADF used as input to test computation times.

4.1 Efficiency of finding mSATs

As described earlier in Section 3.4.1, multiple algorithms for finding appropriate mSATs were explored in the project. As the various versions of the algorithm which finds random interpretations cause other bugs in the code that make the program incorrect, it does not make sense to include this algorithm when analyzing the complexity and computation time of the final algorithm.

As such, this section describes experiments on two algorithms on two input ADFs, testing the influence of the "smart" computation as described in Algorithm 3.2 on page 11. Since the different parts of the program work as they should, the main thing to test here is computation time as a function of the size of the set of arguments in the input ADF. The computation time is measured over the entire program, using one of the following algorithms for finding mSATs:

(1) Smart computation over parents

(2) Computation over parents

The two example ADFs and initial claims used to test these algorithms, are

- the simple ADF with three arguments from Example 5, as in Figure 2.4 on page 9, with initial claim $\{c \mapsto \mathbf{t}\}$;
- an ADF with seven arguments as in Figure 4.1, with initial claim $\{a \mapsto \mathbf{t}\}$. Especially φ_c makes this ADF more complex than the first.

To account for the randomization in the program, the average computation time is taken over 50 runs. The results are presented in Table 4.1.

Algorithm	Simple ADF	Complex ADF
1	0.311283659	1.902142177
2	0.318055177	2.654839830

Table 4.1: Average computation times (in seconds) of the program, using different algorithms for finding mSATs, for a smaller and more complex input ADF.

These computation times clearly reflect that the "smart" algorithm makes the program more efficient, since with increasing number of arguments, the computation time for the second algorithm increases much faster.

An interesting phenomenon showing that the program works as it should, is that the computation times for both algorithms show spikes on certain (rough) lines. This distribution reflects the fact that choosing the correct minimal satisfiable interpretation right away leads to a solution faster than having to backtrack because the program chose an option leading to a contradiction. Figure 4.2 displays this phenomenon for the simple input.

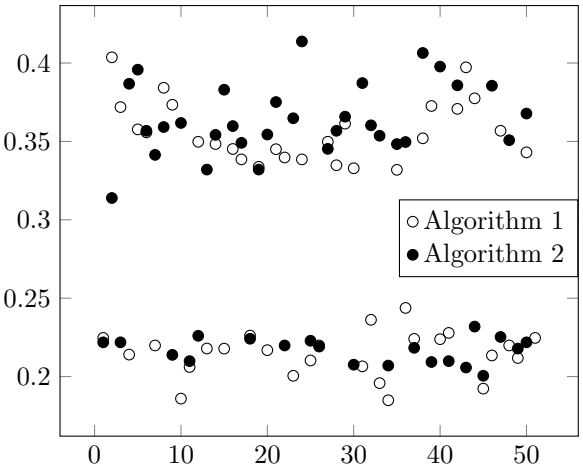

Figure 4.2: Computation times of the two algorithms on the simple input over 50 runs.

In the case of the simple input ADF, when $mSAT_c = \mathbf{utu}$ is chosen in the first turn by the opponent, the game reaches an agreement faster than when $mSAT_c = \mathbf{tuu}$ is chosen in that turn, because the backward move need not be applied.

5 Conclusion

This project explored the preferred discussion game for abstract dialectical frameworks and how to implement said game in an efficient Python program.

Overall, the implementation of the game is as it should be. The tree structure provides an elegant game play, with underlying functions in neat, direct implementations of the theoretical formulae. The program is user-friendly, and has the option of printing the different intermediate steps of the game and the search tree, showcasing the processes unique to the algorithm provided by the theory. This way, the program provides a useful aid for future research on the game, and a base for the implementation of similar games.

As expected, the use of Python libraries simplifies the evaluation of logical expressions greatly, though it should be noted that the Python SymPy library used to read input in the current implementation does not read input files correctly if the names of arguments in the file are integers.

A number of different algorithms were compared for the search for a useful minimal satisfiable interpretation. The two most efficient algorithms were compared by tracking the computation time. Though for complex inputs, the algorithms are not as efficient as expected, the computation time is not very high for simpler problems.

In particular, the random generation of these interpretations is something worth exploring further. In the scope of this project, problems concerning infinite search loops restricted the use of random generation in the game. In the future, one solution might be provided by exhausting the search set. This is actually used in the current implementation, where randomly visited options are removed from a set so that they are not visited twice.

6 Discussion

As the discussion game introduced by Keshavarzi and implemented in this thesis is the first of its kind, many options for more general further research exist.

For starters, one direct continuation of this project would be to use the implementation of the current game to evaluate the game algorithm against other existing solvers such as K++ and YADF. As the implementations of these solvers give answers mere microseconds, even for extremely large inputs, it did not make sense to evaluate their computation times against those of the current implementation of the discussion game algorithm.

Moreover, while the game discussed in this work solves the credulous decision problem under preferred semantics, similar games could be developed for the skeptical decision problem, which revolves around the question whether not just some, but *all* interpretations of a given type of semantics assign a certain argument to a certain truth value.

Lastly, while the game algorithm was developed with preferred semantics in mind initially, the underlying concepts are robust enough to be applied to other semantics as well, such as grounded and complete semantics. In fact, a similar version for grounded semantics has been recently developed by Keshavarzi Zafarghandi, Verbrugge, and Verheij (2020, in press), and has been accepted to the 8th International Conference on Computational Models of Argument.

References

Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *The Knowledge Engineering Review*, 26 (4):365–410, 2011.

Gerhard Brewka and Stefan Woltran. Abstract dialectical frameworks. In *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 102–111, 2010.

Gerhard Brewka, Stefan Ellmauthaler, Hannes Strass, Johannes Peter Wallner, and Stefan Woltran. Abstract dialectical frameworks revisited. In *Twenty-Third International Joint Conference on Artificial Intelligence*, pages 803–809, 2013.

Gerhard Brewka, Martin Diller, Georg Heissenberger, Thomas Linsbichler, and Stefan Woltran. Solving advanced argumentation problems with answer-set programming. In *Thirty-First AAAI Conference on Artificial Intelligence*, pages 1077–1083. AAAI press, 2017a.

Gerhard Brewka, Stefan Ellmauthaler, Hannes Strass, Johannes P Wallner, and Stefan Woltran. Abstract dialectical frameworks: An overview. *The IfCoLog Journal of Logics and their Applications*, 4(8):2263–2317, 2017b.

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

Thomas Linsbichler, Marco Maratea, Andreas Niskanen, Johannes Peter Wallner, and Stefan Woltran. Novel algorithms for abstract dialectical frameworks based on complexity analysis of subclasses and sat solving. In *IJCAI*, pages 1905–1911, 2018.

Atefeh Keshavarzi Zafarghandi, Rineke Verbrugge, and Bart Verheij. Discussion games for preferred semantics of abstract dialectical frameworks. In *European Conference on Symbolic and Quantitative Approaches with Uncertainty*, pages 62–73. Springer, 2019.

Atefeh Keshavarzi Zafarghandi, Rineke Verbrugge, and Bart Verheij. A discussion game for the grounded semantics of abstract dialectical frameworks. In *Proceedings of COMMA 2020: Computational Models of Argument*, 2020, in press. URL `https://comma2020.dmi.unipg.it/index.html`.