Argument Game Proof Theories

6CCS3PRJ Final Year Individual Project Report

Author: Sebastian Oleksa

Supervisor: Dr Sanjay Modgil

Student ID: 20009365

April 5, 2023

Originality Avowal

Abstract

Argumentation plays an important role in many aspects of Computer Science. While representing and resolving conflicting statements is undeniably useful, the theory behind argumentation can be confusing. In this project, I have created an application to build and modify argument frameworks and play argument games. It helps visualise the base concepts behind argumentation, and includes functionalities for creating advanced argumentation frameworks.

Acknowledgements

I would like to thank my supervisor, Dr. Sanjay Modgil, who has provided me support and guidance throughout the whole project. Not only did he introduce me to Argumentation Theory, but also challenged me to go further.

Contents

Chapter 1

Introduction

Argumentation Theory allows us to represent conflicting statements. Abstracting facts and the relations between them can make it easier to determine which are true. This can be used to great effect in many areas of computer science, such as decision-making in AI or communications between separate systems.

Utilising Dung Argumentation Frameworks abstracts the set of statements into arguments and attacks; often represented as a directed graph with nodes being arguments and arrows between them showing attacks.

These frameworks can be analysed in a multitude of ways. Labellings can tell us which statements are always true or false, and which are not clear. Often there are multiple possible labellings, so different sets of rules exist to differentiate between them.

Frameworks can also be used to play Argument Games – an interaction between two players that "debate" the validity of a chosen argument from the framework. Like a real conversation, they bring up new facts to try to counter the other person's arguments. In the end, the initial fact will be declared true if it is defended from all attacks.

Argumentation can be complicated. Our intuitive understanding of debates can actually make understanding the more rigid and abstract argumentation frameworks harder.

In this project, I have developed an application that can help with that. It allows the user to create and modify their own frameworks, visually representing the graphs. The user can explore their functionality, transforming the framework and seeing how small changes affect the results. They can use them to play argument games, either against the computer or controlling both players.

The application also includes tools to create advanced argumentation frameworks – ones that include attacks that can target attacks. Resulting frameworks can be transformed into Dung frameworks via methods included.

While developing the application, special care was put into the usability of it. Through considerations of visual design and customisation, it was the goal to ensure it would be easy to use. A program with the purpose of education should minimise any distractions, such as poor and unintuitive controls. Achieving basic functionality is not enough; through a level of quality and polish, the application can be more effective at introducing the complicated concepts.

Chapter 2

Background

2.1 Dung argumentation framework

In this project I will analyse arguments with the use of Dung argumentation framework, created by Phan Ming Dung in the paper “On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games” [1]. I will introduce this concept, as it is crucial to the entirety of this project. Please note that many of the contents of this chapter come from mentioned papers, I am merely retelling them. The definitions in the Background chapter are excerpts from the cited works – to preserve their original meanings, they have not been altered.

The argumentation framework is an abstraction of arguments. It puts aside the “contents” of the arguments, instead focusing on the relations between them. In this framework, the acceptability of an argument is determined solely from its ability to successfully “defend” itself from any counterarguments.

Definition 1. — Argumentation Framework An argumentation framework is defined as a pair of a set of arguments, and a binary relation representing the attack relationship between arguments. This pair can be written as:

$$AF = \langle AR, attacks \rangle$$

where AR is a set of arguments, and $attacks$ is a binary relation on AR, i.e. $attacks \subseteq AR \times AR$

For two arguments A and B, the meaning of $attacks(A, B)$ is that A represents an attack against B [1]. Such a framework of two arguments and an attack would be written as:

$$AF = \langle \{A, B\}, \{(A, B)\} \rangle$$

These frameworks can be represented visually as a directed graph, where each argument becomes a separate node, and each attack, an edge.

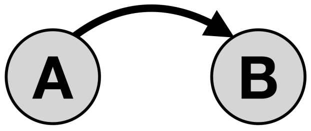


Figure 2.1: An example argument graph, where argument A attacks argument B

These graphs will of course become more complex in tandem with the framework they represent. As another example, here is another graph, and a framework it represents.

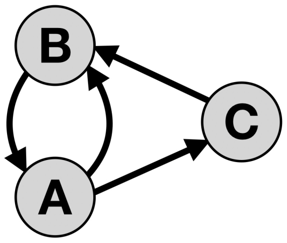


Figure 2.2: A second simple framework graph, representing the following framework:

$$AF = \langle \{A, B, C\}, \{(A, B), (B, A), (A, C), (C, B)\} \rangle$$

2.2 Labelling

This way of representing arguments allows us to focus on their relations, and how they impact each other. To represent the validity of arguments, we will use labellings – this concept comes from the article "Proof Theories and Algorithms for Abstract Argumentation Frameworks" by Sanjay Modgil and Martin Caminada [3]. It provides the baseline for this project.

Definition 2. — **Labelling** For a given argumentation framework AF, a labelling is a function assigning to each argument exactly one label, which can be either IN, OUT or UNDEC.

We define: $in(\mathcal{L}) = \{x|\mathcal{L}(x) = \text{IN}\}$; $out(\mathcal{L}) = \{x|\mathcal{L}(x) = \text{OUT}\}$; $undec(\mathcal{L}) = \{x|\mathcal{L}(x) = \text{UNDEC}\}$; [3].

In short, IN means that the argument is winning, or justified; OUT means that it is losing, or overruled; UNDEC means that it is undecided, as its status can't be fully determined. It is important to note that any labelling can technically be assigned to any argument – the correctness, or legality, of a labelling, is a separate concept.

Definition 3. — **Legal Labelling** Let $\mathcal{L}$ be a labelling for argumentation framework $AF = \langle \mathcal{A}, \mathcal{R} \rangle$ and $x \in \mathcal{A}$

- x is legally IN iff x is labelled IN and every y that attacks x ($y\mathcal{R}x$) is labelled OUT.
- x is legally OUT iff x is labelled OUT and there is at least one y that attacks x ($y\mathcal{R}x$) and y is labelled IN.
- x is legally UNDEC iff x is labelled UNDEC and not every y that attacks x ($y\mathcal{R}x$) is labelled OUT, and there is no y that attacks x such that y is labelled IN. [3]

In other words, for an argument to be legally IN, it must successfully defend itself from all counterarguments. This happens when all arguments that attack it are OUT, i.e. there is no valid counterargument. The OUT labelling is sort of the opposite, where an argument becomes overruled if there is even one justified counterargument, i.e. there exists an attacker that is IN.

The UNDEC label is a bit more tricky to understand. It is usually used when none of the other two labels can be applied. An undecided argument can't be successfully attacked, or it would be legally OUT. Similarly, not every attacker of it can be OUT - else it would be legally IN.

Definition 4. — **Admissible Labelling** An admissible labelling $\mathcal{L}$ is a labelling without arguments that are illegally IN and without arguments that are illegally OUT [3].

Definition 5. — **Complete Labelling** A complete labelling $\mathcal{L}$ is an admissible labelling without arguments that are illegally UNDEC [3].

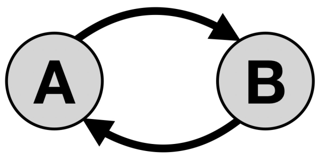


Figure 2.3: Both arguments A and B attack each other

In the above example, two arguments A and B attack one another. Labelling both as UNDEC would be complete – neither one is attacked by an argument that's IN, and both are attacked by at least one argument that isn't OUT.

However, that is not the only complete labelling. We could label A as IN and B as OUT, or the opposite, and these would be equally admissible and complete. As you can see, one argumentation framework can have multiple complete labellings, which creates a need to differentiate them from each other.

Definition 6. — Grounded, Preferred, and Stable Labellings

- $\mathcal{L}$ is a grounded labelling iff there does not exist a complete labelling $\mathcal{L}'$ such that $in(\mathcal{L}') \subset in(\mathcal{L})$
- $\mathcal{L}$ is a preferred labelling iff there does not exist a complete labelling $\mathcal{L}'$ such that $in(\mathcal{L}') \supset in(\mathcal{L})$
- $\mathcal{L}$ is a stable labelling iff $undec(\mathcal{L}) = \emptyset$ [3]

The first two – grounded and preferred – are especially important, as they will return in a slightly different context in the next section. It is important to understand these well. To help do that, please refer to this example:

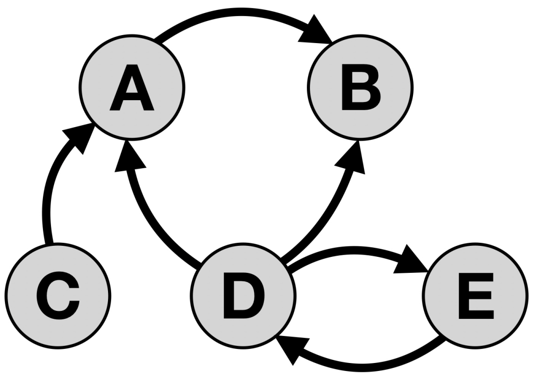


Figure 2.4: An argument graph with 5 nodes

Argument	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$
A	OUT	OUT	OUT
B	UNDEC	OUT	IN
C	IN	IN	IN
D	UNDEC	IN	OUT
E	UNDEC	OUT	IN

Table 2.1: The three complete labellings for Figure 2.4

For the framework above, we could assign the following complete labellings:

Labelling $\mathcal{L}_1$ is grounded, but not stable or preferred.

It has only one argument that is IN - C. For $\mathcal{L}_1$ to not be grounded, there would need to exist a complete labelling $\mathcal{L}_1'$ in which no arguments are labelled IN. As argument C has no attackers, it can only be legally labelled as IN, therefore no such labelling $\mathcal{L}_1'$ exists.

$\mathcal{L}_1$ is not preferred, as there exists a labelling $\mathcal{L}_2$ (shown in the table) for which $in(\mathcal{L}_2) \supset in(\mathcal{L}_1)$

$\mathcal{L}_1$ also is not stable, as it assigns UNDEC to at least one argument

Labellings $\mathcal{L}_2$ and $\mathcal{L}_3$ are both stable and preferred, but they are not grounded.

They are stable, as neither one has any arguments labelled as UNDEC.

They are preferred, as there does not exist a $\mathcal{L}_2'$ such that $in(\mathcal{L}_2') \supset in(\mathcal{L}_2)$, and there does not exist a $\mathcal{L}_3'$ such that $in(\mathcal{L}_3') \supset in(\mathcal{L}_3)$.

They are, however, not grounded, as there exists a labelling $\mathcal{L}_1$ (also shown in the table) for which $in(\mathcal{L}_1 l) \subset in(\mathcal{L}_2)$

2.3 Argument Games

Building complete labellings, and especially proving that an argument is in such labelling, can become difficult. To help showcase these processes, the concept of Argument Games has been introduced. Like the labellings, this comes from the article by S. Modgil and M. Caminada. [3]

Like many games, an argument game is played between two players: a proponent, and an opponent. It starts with the proponent selecting an argument, followed by both players alternating moving arguments onto the game tree, trying to counter the last moved argument. The game ends once no more arguments can be moved. If the proponent has managed to successfully defend the initial argument from all opponent's attacks, the proponent wins. If the opponent manages to form a line of attack that the proponent cannot defend, the opponent wins. The argument games come in two variants, which somewhat change the rules: the grounded and preferred variants, corresponding to grounded and preferred labellings.

The rules of the game

An argument game is always based on an existing argumentation framework AF. That framework needs to be constructed before the game starts and cannot change throughout the course of the game.

At the beginning of the game, one of the players is chosen to go first. They become the "proponent", with the other player being the "opponent". The proponent makes the first move, selecting one of the arguments from the framework and moving it to the game tree. This first argument is called the initial argument.

After selecting the initial argument (let's call it A), the proponent's turn is over. Now, the opponent will make their move. They must choose an argument B such that in AF $attacks(B, A)$ (i.e. in the argumentation framework argument B attacks argument A), and move it to the game tree. Then, the opponent's turn is over, and now the proponent will attempt to move an argument C such that $attacks(C, B)$.

The players will continue performing these turns. If a player cannot make a move, they pass, and the other player goes again. When neither player can make a move, the game ends.

The game tree might split into multiple branches, but these branches can never merge again. From any node in that tree, there is exactly one path to the initial argument, which is called a *dispute*.

Now, there are a few more rules to the game. Firstly, if a player cannot counter the last argument moved by the other player, they may backtrack and counter one of the arguments the other player moved before. When doing so, they must attack it with a different argument than before. It will create a new branch in the game tree. It is possible to move an argument that already exists somewhere else on the game tree. In that case, a new node should be added, even though it still represents the same argument.

Depending on the game variant, an additional rule is added:

- **Grounded game:** The Proponent may not move an argument they have already moved within a dispute.
- **Preferred game:** The Opponent may not move an argument they have already moved within a dispute.

These rules place a restriction on one of the players. They ensure the game cannot last indefinitely. Because of that rule, one of the sides has a finite number of possible moves. As the players make alternating moves, it is certain at some point one side will have no more possible moves left, thus ending the game.

Upon the game end, we can label the game tree as we did with the framework. If the initial argument is IN, it is winning and the proponent has won. Otherwise, if the opponent has attacked it in a sequence that the proponent could not defend, the initial argument is OUT and the opponent has won the game.

Example: We will play a game using the same framework shown in the previous example (Figure 2.4). To show the differences, we will play one grounded and one preferred game, both with the exact same initial argument.

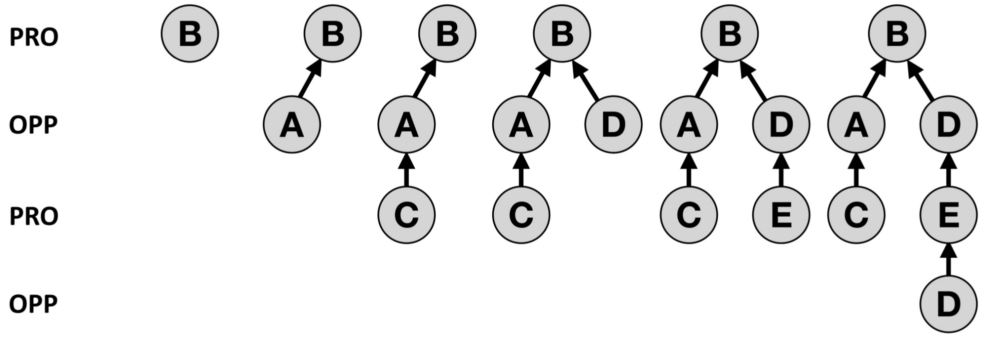


Figure 2.5: An example grounded game

The proponent makes the first move, moving the initial argument B to the game tree. At this moment, no argument attacks it, so B is winning. The opponent then makes their move. In the framework, there are two arguments that attack B – A and D. The opponent could choose any one of them. In this example game, they choose A and move it to the tree. This node will be placed in the layer just below B, with a directed edge towards it, to show that A attacks B.

Now it's the proponent's turn again. Currently, there are no arguments attacking A, so A is winning. As B is attacked by a winning argument, B is currently losing. In the framework, there is only one argument challenging A. So, the proponent chooses C and moves it to the game tree. Just like before, it will appear in a new layer, with an edge towards the argument it attacks.

There are no arguments challenging C in the framework. As the opponent can't counter it, the dispute has been won by PRO, which we call a *line of defence* for *B*[3].

The game, however, is not yet over, as the opponent can backtrack and counter a previous argument in this dispute. They can move D to the game tree and attack B again, creating a new branch – a new dispute. This new node will be placed in the layer just below the node it counters.

The next proponent's move is simple, as there is only one node to counter (just-added D) and only one argument that can counter it (E). They will do just that and move E to the layer just below D.

Now, the opponent has to attack E. The only argument that counters E is D, which has already been moved by the opponent in this dispute. However, we are playing a grounded game, so this is not an issue; this variant allows the opponent to repeat moves in any way they like. So, the opponent will move D to the game tree again. It will be represented by a new node, placed just below the argument it attacks.

It is now the proponents turn again. They need to either counter the last argument moved (D) or backtrack to another argument moved by the opponent – which is either A or another D.
The first is not possible, as the only way to attack D is by moving E – the proponent has already moved E within this dispute, so, under grounded rules, they cannot do that again.
Backtracking is also impossible. The only counter to A is C – which has already been used. If proponent tried to backtrack to the previous D node, they would need to attack it differently than before – they cannot just move E again. However, E is the only argument that attacks D... which means that the proponent is out of options for this dispute.

There is a way to backtrack here for the proponent, shown below.

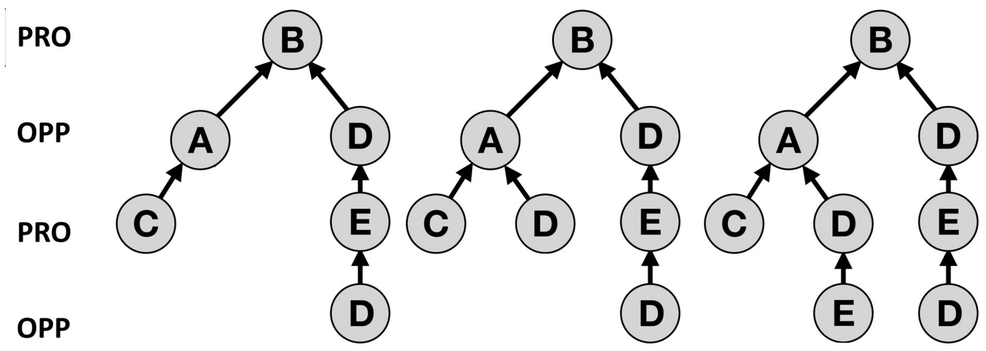


Figure 2.6: An example grounded game: part 2

The proponent can attack A again with D. The resulting dispute is similar to the other one, yet with one key difference – now it is the proponent that moves D. The opponent can counter with E, but the grounded rules prohibit the proponent from answering that.

They cannot make a move, so they pass. The opponent has no more possible moves remaining either, which means the game comes to an end. There were three disputes created. Under the final game tree, the initial argument is losing. The opponent is declared the winner.

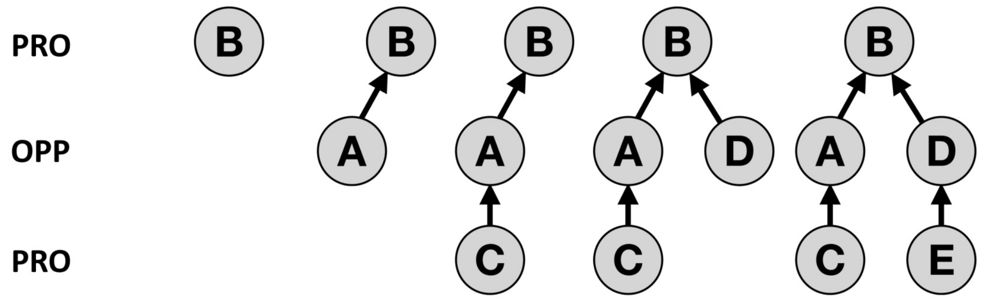


Figure 2.7: An example preferred game

Above is the same game, but played with the preferred variant. At first, the game will proceed the same, until proponent moves E. Under preferred rules, the opponent may not repeat an argument within a dispute. This means they cannot use D to attack E, like they did in the previous example. There are no other arguments that attack E, so the opponent's only option is to backtrack. The only other proponent's node in this dispute is initial B. B only has two arguments that attack it, and both have already been used. Opponent is forced to pass.

Once again, proponent can backtrack.

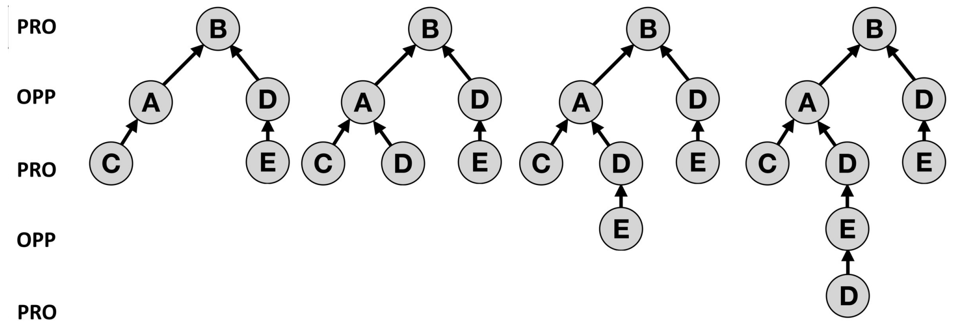


Figure 2.8: An example preferred game: part 2

The proponent can attack A again with D. The resulting dispute is similar to the other one, yet with one key difference – now it is the proponent that moves D. The opponent can counter with E, and the rules allow the proponent to counter that with D again. This dispute is successfully defended as well.

At this point, neither player has any possible moves, so the game ends. The proponent has successfully defended all disputes. Under the final game tree, the initial argument is winning. so they win the game.

2.4 Winning strategy

To ensure proponent's victory, they can find and follow a winning strategy. If they are able to do that, they will always win the argument game, regardless of the opponent's actions.

We can start by constructing a *dispute tree* - a tree representing all the possible moves in a game. Below, you can see a dispute tree for a grounded and preferred game with initial argument B using the example framework from Figure 2.4.

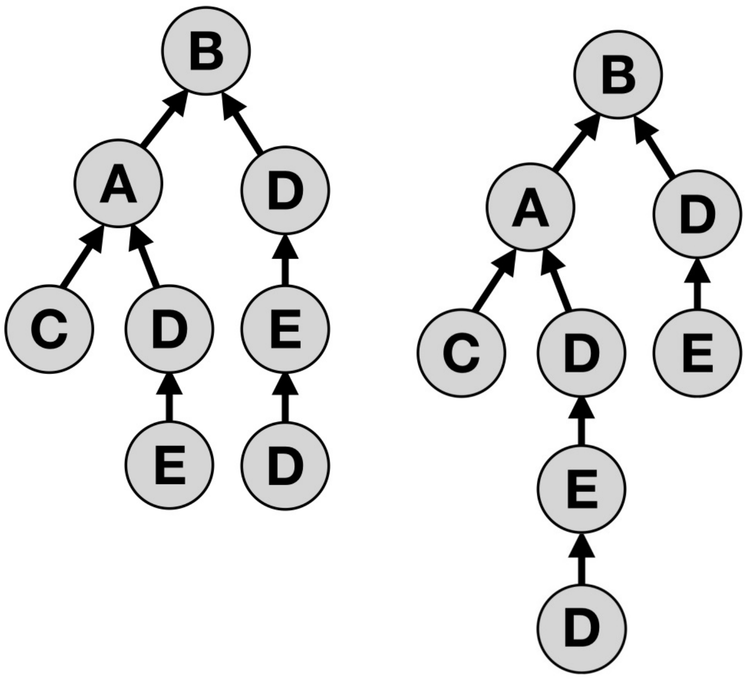


Figure 2.9: A grounded (left) and preferred (right) dispute trees

As you can see, they include the nodes that appeared in our games.

Definition 8. — **Winning strategy** Let $AF = \langle AR, attacks \rangle$, T the dispute tree induced by a in AF, and T' a subtree of T. Then T' is a winning strategy for a iff:

1. The set $D_{T'}$ of disputes in T' is a non-empty finite set such that each dispute $d \in D_{T'}$ is finite and is won by PRO (terminates in an argument moved by PRO)
2. $\forall d \in D_{T'}, \forall d'$ such that d' is some sub-dispute of d and the last move in d' is an argument x played by PRO, then for any y such that $y\mathcal{R}x$, there is a $d'' \in D_{T'}$ such that $d' - y_{OPP}$ is a sub-dispute of d''. [3]

To put it in other terms, the winning strategy is a subtree of the dispute tree, where every dispute is won by the proponent. However, for every move of the proponent, the winning strategy subtree must also include all possible opponent moves that can counter it.

If that rule is kept, and if the proponent only makes moves that are in the winning strategy, any of their opponent's possible moves will stay within the winning strategy. And since all disputes in the winning strategy end in proponent winning, then so will the game itself.

For the previous examples, the grounded game does not have a winning strategy. We can prove this by contradiction.

Suppose a winning strategy exists. As B is the initial node, it has to be included in the winning strategy.We also must add all the opponent's responses to the initial argument B: A and D. Let's focus on D: we cannot end the dispute there, or it will end in the opponent winning, thus violating the definition of the winning strategy. So, we must add E, and following the rule of adding all the opponent's responses, D again. This dispute ends there, with no other move that can be added. However, it ends in an OPP move, violating the definition of the winning strategy. That's a contradiction, so such a subtree cannot exist.

For the preferred game, the whole dispute tree is a winning strategy, as every single dispute ends in PRO argument. We can easily see that this does not violate the definition of the winning strategy, so it is valid.

For one more example, take a look at this different dispute tree.

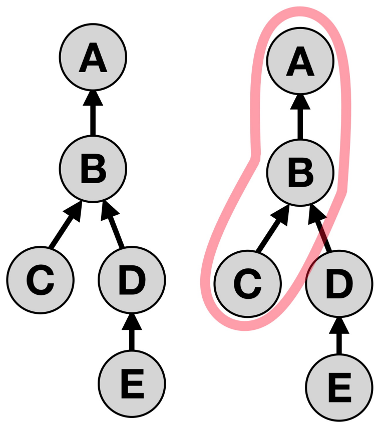


Figure 2.10: A dispute tree (left) and its winning strategy (right)

Here, the subtree marked in red is the winning strategy. It only has one dispute, but it ends in a PRO argument. Additionally, it includes all OPP responses to every PRO move included within the subtree. If the proponent sticks to the moves within the subtree, they will win every dispute and thus the game.

Note: A way of finding these winning strategies is implemented within my project and described within the main section of the report.

2.5 Advanced argumentation frameworks

To explore an even more interesting side of this topic, my project will include a variant of the argumentation framework in which an argument can attack not only other arguments, but also other attacks themselves. I will call these attacks from an argument to another attack, *meta attacks.*

Definition 10. — Advanced Argumentation Framework An advanced argumentation framework is defined as a trio of a set of arguments, a binary relation representing the attack relationship between arguments, and a binary relation from arguments to attacks. This trio can be written as:

$$AAF = \langle AR, attacks, metaattacks \rangle$$

where AR is a set of arguments, $attacks$ is a binary relation on AR, and $metaattacks$ is a binary relation from AR onto $attacks$ i.e. $metaattacks \subseteq AR \times attacks$

Just like Dung frameworks, these can be represented with a graph.

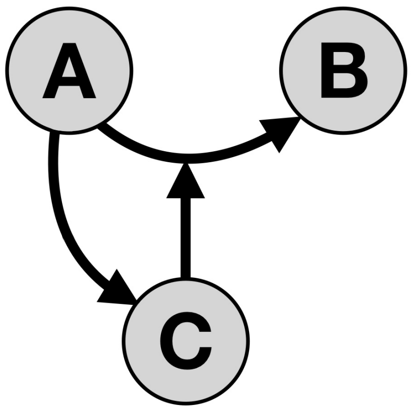


Figure 2.11: An example advanced graph

The above changes are crucial, and require us to redefine some concepts.

First, now labellings must assign a label to both the arguments and the attacks in a framework. Just like arguments, attacks can be IN, OUT, or UNDEC.

Definition 11. — Advanced Legal Labelling Let $\mathcal{L}$ be a labelling for an advanced argumentation framework AAF and x an argument (node) or an attack (edge) within that framework.

- x is legally IN iff x is labelled IN and for every argument y that attacks x, either y or $attacks(y, x)$ is labelled OUT.

- x is legally OUT iff x is labelled OUT and there is at least one y that attacks x such that y is labelled IN and $attacks(y, x)$ is labelled IN.
- x is legally UNDEC iff x is labelled UNDEC and:
    1. There is no y that attacks x such that y is labelled IN and $attacks(y, x)$ is labelled IN.
    2. There is at least one y that attacks x such that neither y nor $attacks(y, x)$ is labelled OUT.

While the definition can appear confusing at first, even more than for regular legal labellings, these are fairly intuitive. If we look at the example figure 2.11, we can quickly find a labelling for it. A, $attacks(A, C)$, and $attacks(C, attacks(A, B))$ have no attackers, so they should be labelled IN.

As both A and $attacks(A, C)$ are IN, C is OUT. If we look at $attacks(A, B)$, it is attacked by C. However, of the pair $attacks(C, attacks(A, B))$ and C, at least one is OUT (C). According to the definition, this makes $attacks(A, B)$ IN.

Lastly, both A and $attacks(A, B)$ are IN, therefore B is OUT.

The above is the only complete labelling; this will often not be the case for other graphs.

2.6 Stratified Advanced Argumentation Frameworks

In this project, I want to focus specifically on the stratified advanced frameworks, also knows as Hierarchical Extended Argumentation Frameworks.

Definition 11. — Stratified Advanced Argumentation Frameworks are frameworks that can be divided into levels such that each level is a Dung framework. Within a level, all attacks are regular Dung attacks between two arguments from that level. There exist no regular attacks between two arguments from two different levels.
Meta attacks in a stratified framework can only exist between two different layers. Additionally, these layers must be ordered: i.e. for any two layers, all meta attacks between that must go in the same direction. [2]

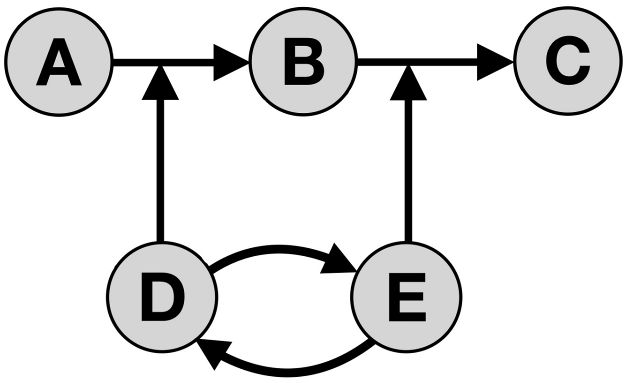


Figure 2.12: A stratified framework

2.7 Converting stratified frameworks to Dung frameworks

Playing argument games on an extended framework is an advanced topic that goes beyond what I can cover in this project. I did, however, want to show one way of handling that for the stratified frameworks.

By following a particular algorithm, we can convert any stratified framework into a Dung framework, on which we can then play the Argument Game.

Let AAF be a stratified advanced argument framework:

$$AAF = \langle AR, attacks, metaattacks \rangle$$

To construct a new Dung framework based on AAF, build a new framework AF such that:

$$AF' = \langle AR', attacks' \rangle$$

1. $\forall a \in AR$ add arguments $j(a)$ and $r(a)$ into AF'. These represent 'justified a' and 'rejected a'.

2. $\forall attacks(a, b) \in attacks$ add an argument $a \rightarrow b$ into AF'. Additionally, add attacks $attacks(r(a), a \rightarrow b)$ and $attacks(a \rightarrow b, j(b))$ to $attacks'$. This represents the attack.

3. $\forall metaattacks(a, attacks(b, c)) \in metaattacks$ add an argument $a \rightarrow (b \rightarrow c)$ into AF'. Also, add attacks $attacks(r(a), a \rightarrow (b \rightarrow c))$ and $attacks(a \rightarrow (b \rightarrow c), b \rightarrow c)$ to $attacks'$ [2]

The result is a Dung framework without any meta attacks, that preserves the original relations found within the advanced argument framework.

To better illustrate this concept, below is an example simple stratified framework and the result of converting it to a Dung framework.

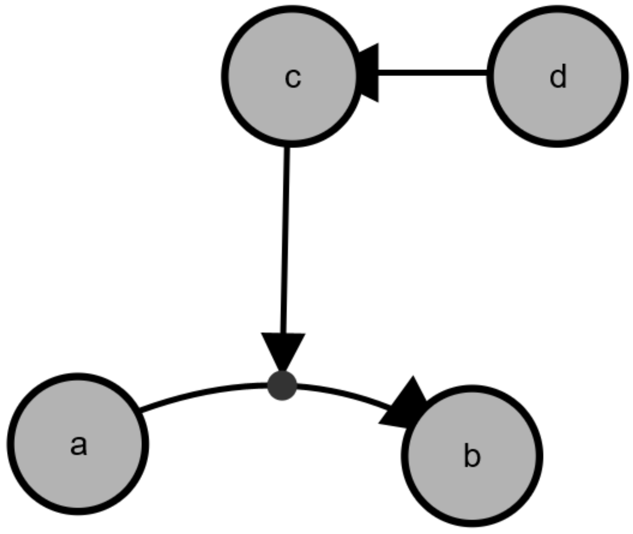


Figure 2.13: A stratified framework to convert

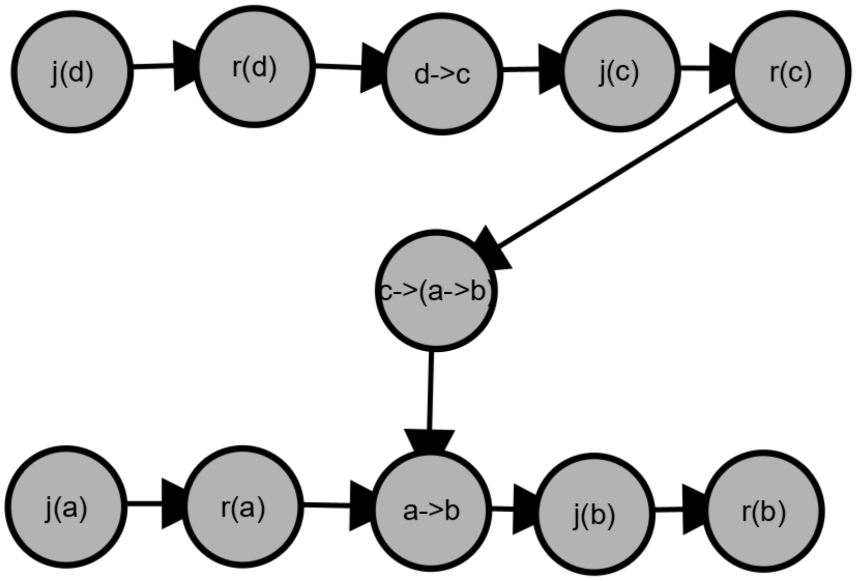


Figure 2.14: Resulting Dung Framework

Chapter 3

Design & Specification

The aim of the project is to create an application that will allow the user to work with argumentation frameworks and play argument games. It should represent these concepts visually to provide the user with clear understanding of the concepts.

3.1 User Requirements

Argumentation Frameworks

1. The user is able to create a new argumentation framework.
2. The user can modify the current framework by adding new arguments, deleting existing ones, creating new attacks, and modifying or deleting existing ones.
3. The user can save the current framework to a file and load a framework from file. The loaded framework must be the same, even if loaded in a different environment.
4. The user can display the current framework.
    - The user can adjust the view of the graph by panning or zooming.
    - The user can edit this displayed graph by moving the nodes around or bending the edges. This can provide visual clarity, but does not modify the framework itself.
    - The user can choose to include this visual configuration in the save file of the framework. The displayed graph of this file when loaded is be the same as when it was saved.

5. The user can make an advanced framework where edges can attack other edges, not just nodes.

6. If the current advanced framework is stratified, the user can convert it to a Dung Framework.

Argument Games

1. The user can start an argument game using the loaded argumentation framework. They can choose whether they want to play the grounded or preferred game.

2. The user can choose whether both players are controlled by the user, or whether one of them should be controlled by the computer.

3. The user can select the initial argument, even if they do not control the proponent.

4. On their turn, a player can select a move to make.

   - During a game, a player may always look at the framework. It is not possible to edit the framework during a game.

5. The application detects when the game ends and displays a message with the winner.

6. When not in a game, the user may display the full game tree for a given framework. This tree will have a winning strategy marked, if one exists.

3.2 System Requirements

1. The application can represent the current framework visually through a directed graph.

   - It can generate the first visual configuration of a framework, if one is not provided in the save file.
   - When creating a new argument, it can find an empty space to put it on the graph to make sure it's visible to the user.

2. The application must be able to construct the game tree and check for a winning strategy. It can then use this strategy in a game or display it to the user, if prompted.

3. During a game:

- The application can determine which moves are allowed and which aren't. On a player's turn, the application will display their possible moves and prevent the player from making invalid moves. If there are no possible moves, the game will terminate.
- If the computer controls one of the players, it will attempt to find the winning strategy. If one exists, the computer will always win.
- At all times throughout a game, the game tree is displayed, showing the current state.

4. Upon the end of the game, the application will show a result screen with the outcome of the game and the final game tree displayed. The user may spend as long as they want on this screen. The user may exit the game with a press of a button, returning to viewing the framework.

Chapter 4

Argument Games Application

The practical product in this project is an application that implements the ideas described before. The primary function is to visually represent the Argumentation Frameworks and Argument Games. It was created with the primary intention of being a teaching tool, so particular care had to put in the application's ease-of-use and, most importantly, correctness.

In this section of the report, I will showcase the design of the application. I will also delve deeper into it's features and their implementation.

4.1 User Interface and Visual Considerations

Due to the intent of this program to be used as a teaching tool, the ease-of-use is important. An intuitive program can allow the user to focus on understanding the topic, not the tool. Because of that, attention must be placed in the design of the user experience.

As an additional goal for this project, I made the decision not to use external libraries for representing graphs in the application. Creating them by hand allows me to fully adjust them to the project's needs, while also presenting an interesting challenge.

The interface is created using JavaFX. I have considered other frameworks, like Swing, or even programming languages, but JavaFX was the best fit for the project. It is made for desktop applications in mind and offers a wide range of features that I took advantage of. While not outdated by any means, it has been around for a number of years, so it has a very high number of libraries and support materials. It is possible there exist libraries that would have been a better, more efficient fit, yet I was able to achieve everything I set out to do using JavaFX.

Below is an early design of how I intended the program to look like.

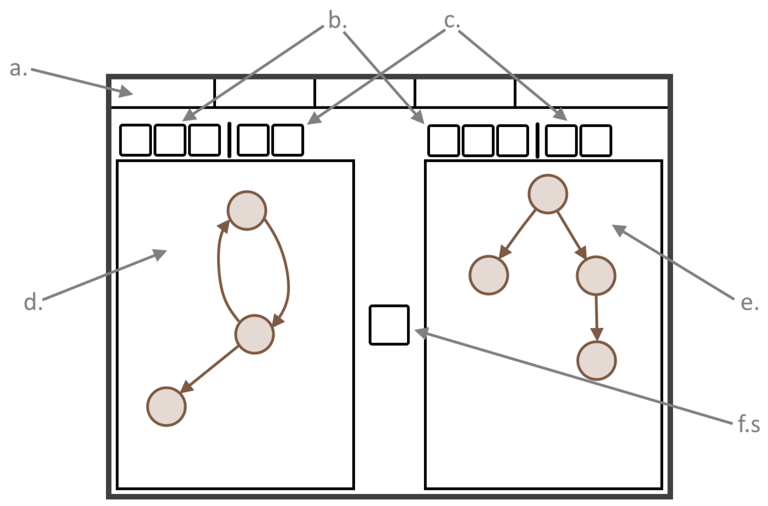


Figure 4.1: A simple design diagram for the application

- a The menu bar. Includes buttons that open submenus, such as File, Edit etc. Used for interactions that affect the entire application, such as loading and saving files. Also used for actions not directly connected to the framework and game tree, such as finding help.
- b Buttons for changing the current mode of the window below. The interaction modes are: Select, Move, and Pan. Clicking one button will unclick the others and swap the interaction mode accurately. The interaction modes are explained in more detail later.
- c Buttons for other actions. These involve mostly adding and deleting arguments and attacks between them.
- d Argument Framework graph panel. This smaller window showcases the currently loaded framework. Arguments are represented by circles, attacks by arrows between them. Arrows can be curved manually by the user to improve readability of the graph.
- e Game Tree panel. A window of the same size as the framework graph panel. It shows a visual representation of the current game tree. During a game, it updates to showcase the moves that have happened. Like on the framework panel, the arguments are represented by circles and attacks by arrows. Unlike the framework panel, the arrows cannot be curved – due to the structure of a tree, this is not necessary.
- f Begin game button. Clicking it prompts the user to select game settings, such as whether they want to play against the computer or not. Upon confirmation, a game begins.

This diagram is a result of many conscious decisions with the aim of making the application intuitive to use. Firstly, the buttons are placed in different sections to reflect which parts of data they affect. Additionally, the buttons that affect the application (interaction modes) are separated from those that affect the data (add/delete elements). The “begin game” button is placed in the middle, between the two panels, as it is effectively a transition from one to the other.

Another important design element is the visual representations of graphs and trees. By keeping the look consistent, it helps to convey to the user that the arguments in the two panels are one and the same.

4.1.1 Interaction Modes

Both panels in the application can be set to one of three interaction modes using the buttons located above. They are independent, i.e. the two panels can be in two different interaction modes at the same time.

Select mode allows the user to click on arguments/attacks. They are then highlighted. The selected item will be the source of some actions; for example, when clicking the “Add Attack” button, the selected argument will be the source of the attack. Likewise, when clicking the “Delete” button, the selected argument or attack will be the one deleted.

Move mode allows the user to rearrange the layout of the attacks and arguments on the screen without changing the underlaying framework. By pressing the left mouse button and holding, arguments can be moved around the panel. The connected attacks will immediately update as well. By clicking on an attack, a small circle will appear. By dragging that circle, the user can change the curvature of the attack, without affecting the origin and end point. The arrow tip will rotate to face the control point.

Pan mode allows the user to move all elements of the pane at once. This is effectively like moving the camera around the graph, as the positions of the elements relative to each other stay the same.

4.1.2 Implementation

In the final product, I was able to achieve the design goals I set for the project.

The end product aligns very closely with the design, with minor differences. These came mostly from elements I did not foresee would be included at the design stage, such as the choice box for the preferred vs grounded game. For these new elements, I tried to follow the previous reasoning to find the best place for them.

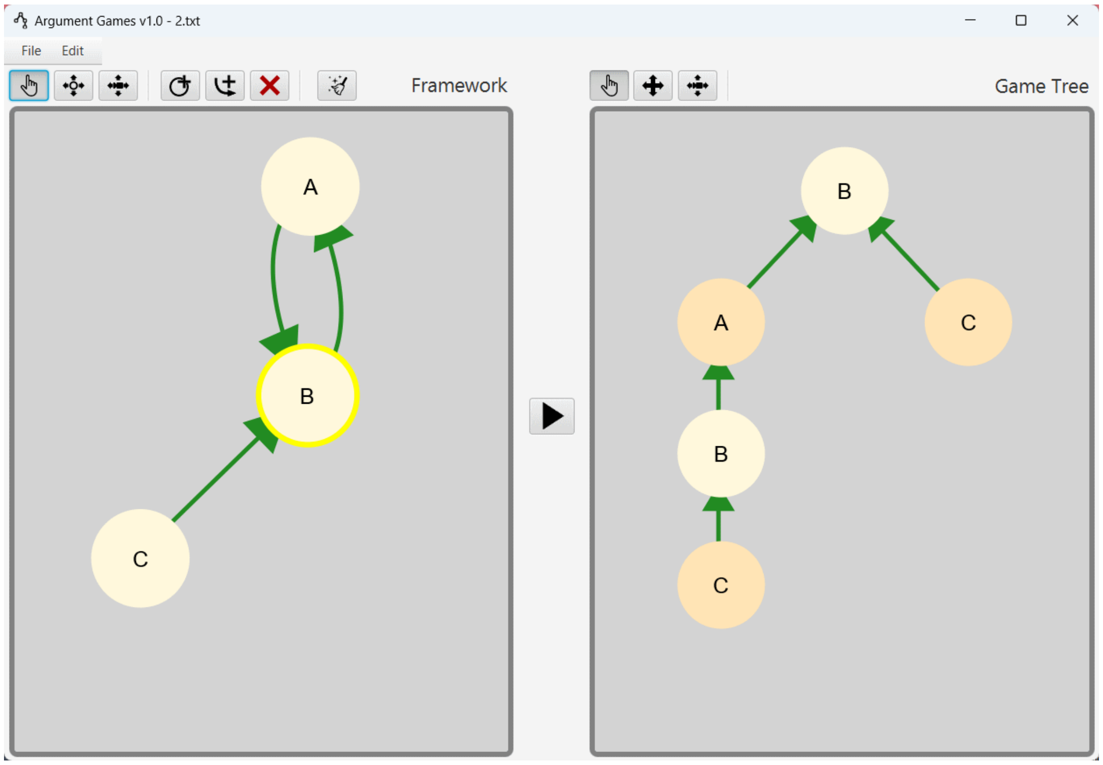


Figure 4.2: The final look of the application

One significant addition to the initial design is the status bar placed at the top of the Game Tree pane. It is used during the argument game to provide short messages to the user, informing them of the state of the game. It makes the process much less confusing and removes ambiguity about whose turn it is or what the user is prompted to do. Messages are kept short and concise, never more than a single line.

4.2 Argument Frameworks

The first section of the application's functionalities is related to the Argument Frameworks. The product must be able to represent these frameworks in multiple ways: in memory, in saved files, and visually – on the user's screen.

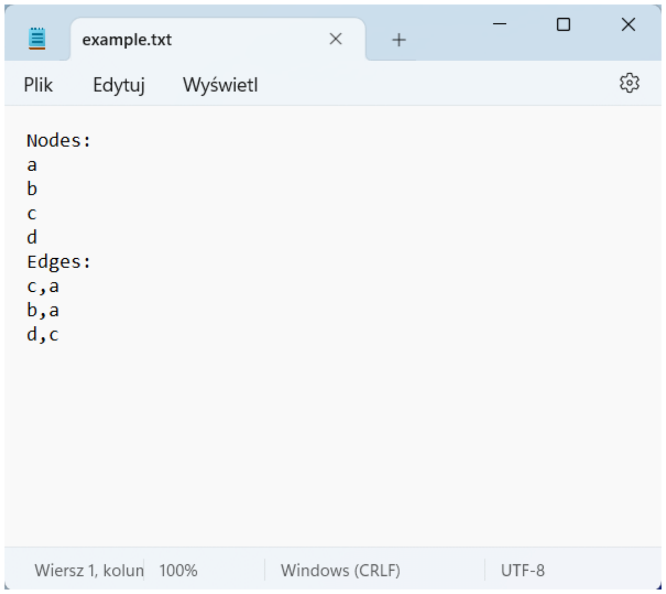
The program is written in Java, a language with many object-oriented capabilities. To take advantage of these features, data such as frameworks, arguments, attacks etc. are stored as objects of custom classes. While not the most data efficient approach, it does provide better code readability and extendability. It would result in worse performance if dealing with a very large framework. However, the application is intended to be a teaching tool, and is not expected to process frameworks nearly large enough for that performance difference to become noticeable.

4.2.1 Data and visual representation

The visual representation (circle in the application) and the underlying data (framework argument) are two different objects in the application. There most functions interact with only one of the two. Following best coding practices, separating them provides cleaner code. It gives the data a layer of protection, making it harder to accidentally affect it.

4.2.2 Saving frameworks to files

The application supports saving the current framework to a file and loading a framework from a file. The files have .txt extension, so they can be easily modified by hand in a text editor. The data saved is intentionally made easy to understand to further encourage that.



```
Nodes:
a
b
c
d
Edges:
c,a
b,a
d,c
```

Figure 4.3: Contents of the saved file

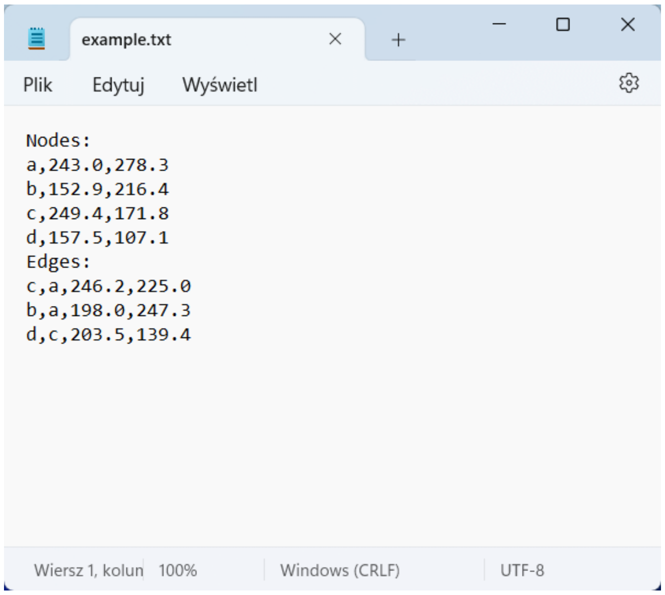
The contents are very similar to the definition of an argument framework:

$$AF = \langle AR, attacks \rangle$$

The file shown above would represent the framework:

$$AF = \langle \{A, B, C, D\}, \{(C, A), (B, A), (D, C)\} \rangle$$

The user can turn on the option to also save the positions of the arguments – this will ensure the current visual configuration will be retained when the file is loaded. It is, however, optional.



```
Nodes:
a,243.0,278.3
b,152.9,216.4
c,249.4,171.8
d,157.5,107.1
Edges:
c,a,246.2,225.0
b,a,198.0,247.3
d,c,203.5,139.4
```

Figure 4.4: Save file with the positions saved

Loading a file first clears the currently open framework, then parses the file. For each line, a circle, or an arrow is created – depending on the section. If the optional position arguments have not been provided, the new circles will be placed in the middle of the pane. When the process finishes, a clean-up function is run. It separates the circles from each other to ensure they do not overlap. Additionally, it makes all attacks straight lines. The function is discussed in more detail further in the report.

If, at any point, the parser encounters a line it cannot interpret, the loading function is cancelled and a message is displayed to the user.

Upon loading the previously shown file (without saved positions) into the application, the following is presented on screen:

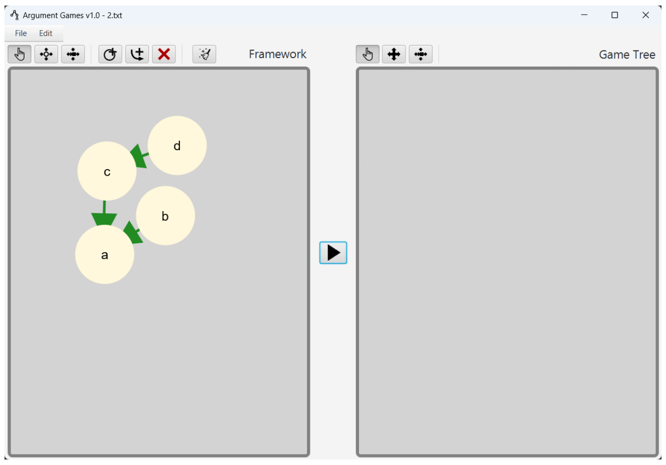


Figure 4.5: Visual graph generated by loading the previous file

The user is of course free to further adjust the visible graph to their needs by moving the nodes, but the load from file function provides a good, readable starting point.

4.2.3 Clean-up function

Whether by adding a new node, loading a file, or simply moving elements around, it is possible to obscure parts of the graph and make it hard to read. The clean-up function attempts to move the elements of the graph and improve visibility. It is automatically invoked after loading a file without specified positions, or can be performed manually by pressing the corresponding button.

The function will run for each circle in the graph until either no improvements are made or until a maximum number of iterations is reached. This ensures the function never lasts indefinitely.

For a circle, it will look at its position and find the circle that is the closest to it. If the distance to it is lower than the defined minimum, the circle will move a small distance directly away. A small degree of random noise is added to prevent any loops from forming.

The defined minimum and step distance are two parameters that can be changed to adjust how the function performs. For my application

$minimum\_distance = 2 * circle\_radius + step\_distance$

$step\_distance = 10$

A special case exists when two nodes are directly on top of each other. The node will then move slightly in a random direction – the next iterations will further separate them.

If the function runs for all circles without triggering any movement, it ends, as no further improvements are possible. This solution is not perfect, for example it does not take into account the arrows between the circles. A more sophisticated version would be possible, yet it would require a considerable amount of work, and the current function is sufficient for the majority of use cases for the program.

4.3 Argument Game Trees

Similarly to the frameworks, game trees are also represented through circles and arrows. The application also uses multiple classes – one for the visual representation and another for the underlying data.

4.3.1 Generating a tree

While generating the data behind the tree is relatively standard, what's more interesting is placing the elements in the panel. The function provided can display any tree, regardless of the number of children per node. It does so while providing visual clarity, making sure that neither the nodes nor arrows overlap.

To do this efficiently, the program calculates a "width" for every node. For a leaf, this width is always 1. For all other nodes, it first sums the widths of its non-leaf children, and the total number of its children. That node's width will be the higher one of these two sums.

After generating the widths, it places the circles corresponding to the arguments inside the panel, starting from root level, going level by level. At each one, it will reserve a box for each non-leaf node based on its width. That node will then be placed in its middle. It will also save the free positions within the box, if it is wide enough. The method will then call itself recursively on that node, building the subtree rooted in it, and restricting it to the generated box.

After every non-leaf node is placed at the level, the program will slot the leaf nodes into the saved spaces, if possible. If there aren't enough of these empty spaces, it will place them in boxes of width 1, just like the non-leaf nodes.

This algorithm makes the generated trees more compact, as some branches can slot together and occupy the same "column" – minimizing the horizontal size of the tree, making it easier to display and more visually appealing.

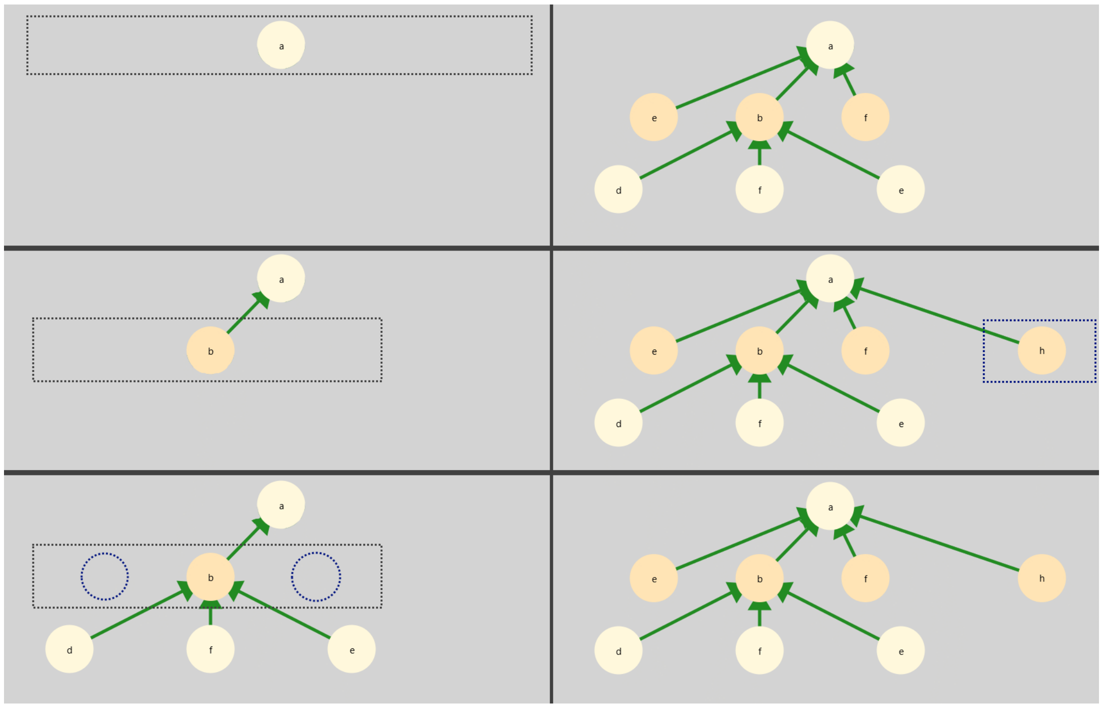


Figure 4.6: Generating a game tree

4.4 Argument Game

The key part of the program is the Argument game being played between the proponent and the opponent. Its structure follows the definitions presented in the Background section.

The game starts with the proponent selecting an argument from the framework to become the "root" of the game. Then, the players perform their turns, alternating between the opponent and the proponent.

4.4.1 Turn Structure

On their turn, a player chooses an argument from the framework and moves it to the game tree. The moved argument must counter one of the arguments moved by the other player. Additionally, the argument being countered must currently be in – this prevents a player from stalling by introducing irrelevant arguments and makes the end condition more clear. The reasoning behind this rule is provided in the background section of the report.

At the beginning of the player's turn, all arguments, in the game tree, moved by the other player, that are currently in, will become highlighted. Clicking on one will select it as the one the player wants to counter. This selection can be changed.

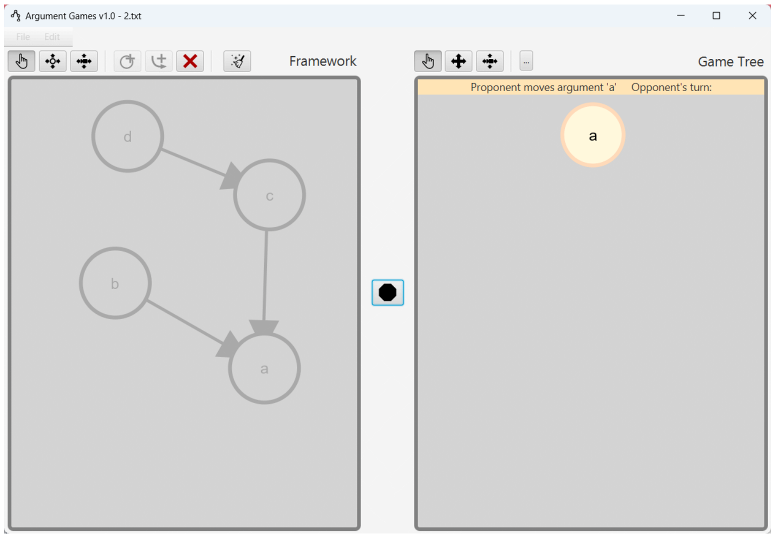


Figure 4.7: Beginning of player's turn

Upon choosing an argument to counter, the framework graph (left panel) will highlight the corresponding argument in the framework. Additionally, all arguments that counter it will be highlighted in a different shade – these are the arguments that can be moved by the player.

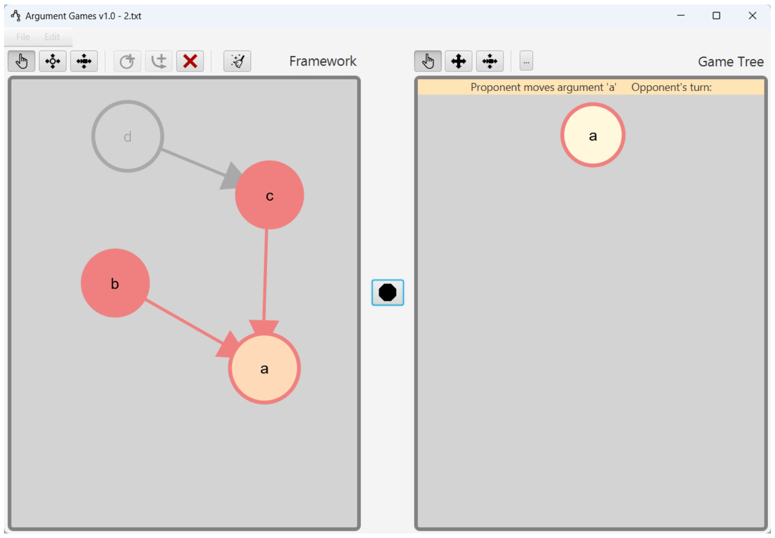


Figure 4.8: After selection of argument to counter

Clicking on one moves it to the game tree and ends the player's turn. The other player will now repeat the same process.

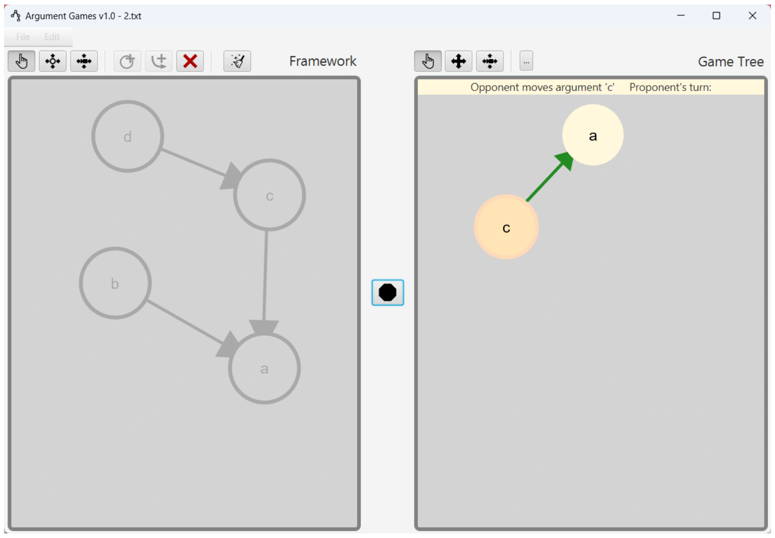

Argument Games v1.0 - 2.txt
File Edit
Framework
Game Tree
Opponent moves argument 'c' Proponent's turn:
d
c
b
a
a
c

Figure 4.9: Beginning of next player's turn

4.4.2 Ending the game

There are two ways in which the game can end: either the user stops it manually, or one of the players wins.

The former happens upon clicking the "End Game" button, located in the same place as the "Start Game" button previously. To differentiate, they have different icons clearly showing the start and stop symbols. Manually stopping the game will not produce any pop-up messages; it will just restore the application to the state it was in before the game started.

The win condition is reached when one of the players has no possible moves to perform. That player will lose the game, and their opponent is the winner. A pop-up message with the result is shown to the user. Upon closing it, no more game actions can be made, but the user is free to look at the resulting game tree as long as they like. Clicking the "End Game" button will fully finish the game and restore the application to the state it was in before the game began.

4.5 Playing Argument Games with the computer

The program allows the user to play an Argument Game with the computer. While the user selects the root argument of the game, the computer takes over the role of the proponent and will make decisions separate from the user. After the game is initialised and the user chooses to play against the computer, the program immediately searches for a winning strategy. If one exists, it will follow it until the game ends, with the computer winning. If one does not exist, the computer will make random moves at each stage. The game will then inevitably end with the opponent winning.

The main challenge here is to find the winning strategy or, if one does not exist, communicate it to the game controller. The following is the approach taken within the application.

4.5.1 Finding a winning strategy

A winning strategy is a subtree of the game tree such that each branch ends in a proponent's argument. Additionally, for each proponent's argument in the subtree, all of its children must also be included in the subtree.

Effectively, this means that the strategy includes every single move the opponent can make in reaction to our moves, and every possible sequence of moves within the strategy will result in the proponent's win.

In my implementation, I analyse the previously generated game tree and mark each argument in it as belonging to the winning strategy or not with the following process:

The first step is to recognize, that if an argument is not in the winning strategy, neither is any of its descendants. Otherwise, the winning strategy subtree would be disconnected, which is not possible. Therefore, when we determine that an argument is not in that subtree, we can set it and all its descendant's marks to indicate they do not belong to the winning strategy. This is final, and these nodes will not be reevaluated, so it is performed only when certain.

The process itself starts at the root of the game tree and runs a function that evaluates based on whether the argument is pro or opp, and based on the argument's children.

The first case is when the argument has no children. If it is a pro argument, we mark it as belonging to the winning strategy for now. If it is an opp argument, we mark it as not in the strategy – its inclusion would violate the rule that states all the disputes in the winning strategy must end in a pro argument.

If the argument does have children, we first run this function on them, and then check their states.

- For a pro argument, if any of its children are not in the winning strategy, then neither is that argument. In short, in means that the opponent can answer with a move that will end in the opponent winning the dispute – therefore it cannot be the winning strategy. Otherwise, mark it as part of the winning strategy for now.
- For an opp argument, it is almost the inverse – it is a part of the winning strategy only if

at least one of its children is part of the winning strategy. In other words, that opponent's move can be answered by the proponent in a way that will lead to their victory.

The above function will correctly update the mark of each argument in the game tree and determine the winning strategy. If the root of the tree gains the mark of not being in the winning strategy, this means one does not exist.

This functionality can be used in the application also outside the argument games. By clicking the "Generate game tree" button, the right panel will display the full game tree, with the winning strategy marked in a different colour.

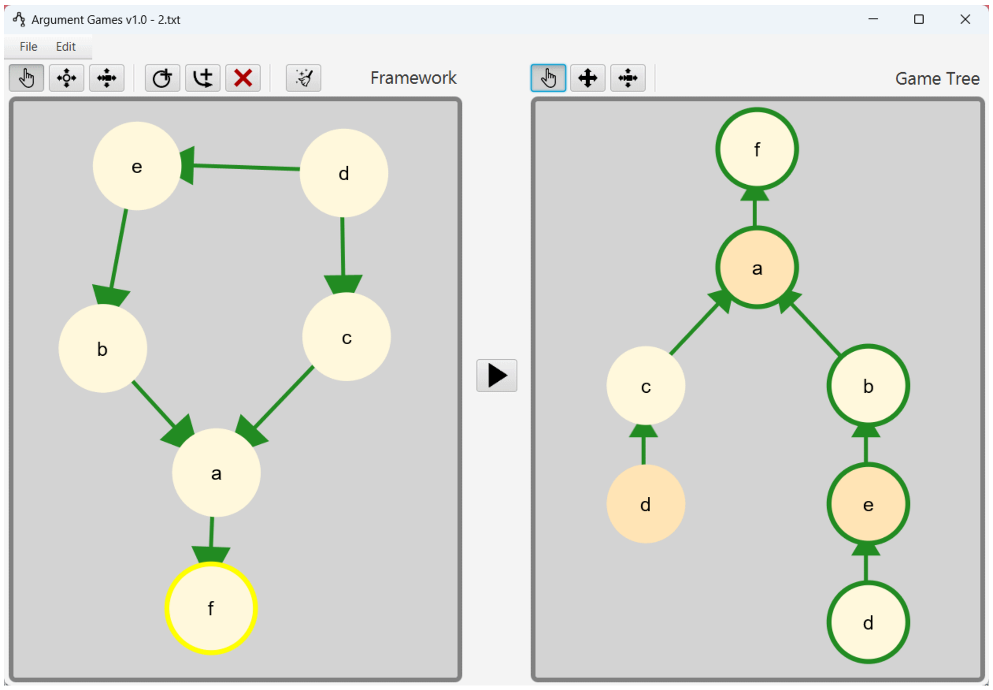


Figure 4.10: The winning strategy marked on the generated game tree

4.6 Advanced Argument Frameworks

The application includes functionality for creating stratified advanced argument frameworks. This needs to first be enabled in the settings, by checking the box "Allow stratified meta argument frameworks".

Once done so, when creating a new edge, the user can click on an existing attack. That will create a meta attack, between the origin argument and the clicked attack.

The program includes a check for whether the resulting advanced framework will be stratified or not. This works by creating an adjacency set for the origin argument. That set includes all arguments that can be reached for it using just regular attacks; this creates a complete set of all arguments in that stratified level.

If the target attack would be a part of that set, the result would not be stratified – so that new meta attack will not be allowed. The second check looks at all meta attacks between the same two levels as the new meta attack. It makes sure all go in the same direction (to satisfy the requirements of a stratified framework). If that rule would be broken by the new meta attack, it will not be allowed.

The result, because of the check, will be a stratified advanced argument framework.

The meta attacks are shown similarly to regular attacks. An arrow is drawn from the origin argument to the midpoint of the target attack. That midpoint will move based on the positions of arguments and the control point, allowing for visual adjustments by the user.

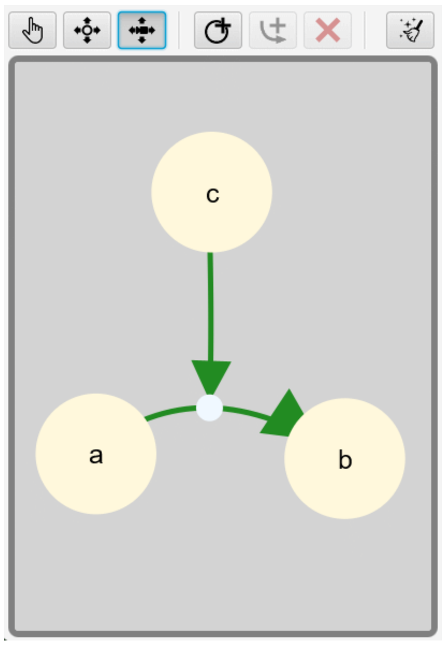


Figure 4.11: A meta attack in an advanced framework

These advanced frameworks cannot be used to play games directly. However, using an option “Transform meta framework to a Dung framework” the user can transform it into a Dung Framework. This utilises the method described in the background section (Section 2.7, page 17). The result can be used to play argument games.

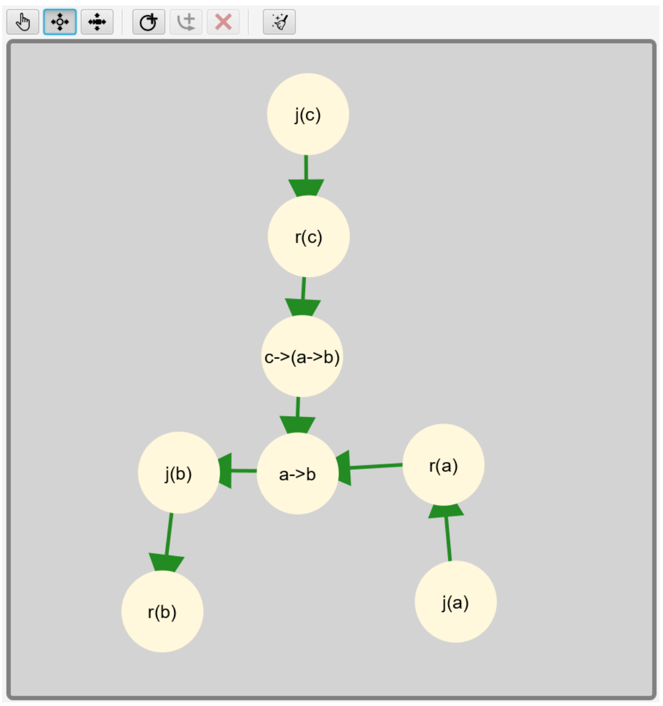


Figure 4.12: The same framework after being transformed and manually rearranged for visual clarity

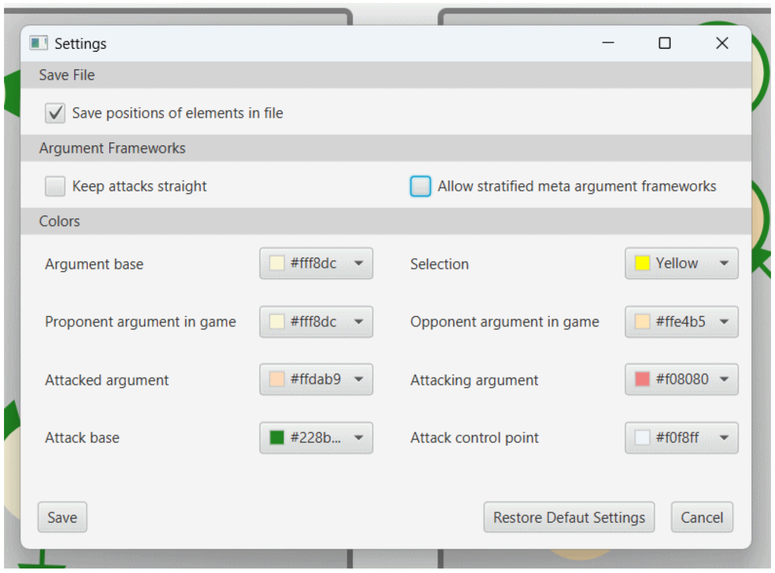
Just like regular frameworks, the advanced frameworks can be normally saved and loaded.

The functionality for advanced argument frameworks in the current version of the application is limited. While it can introduce stratified frameworks and show how they can be transformed into Dung frameworks, it currently has no way of exploring non-stratified argument frameworks. These are outside the theoretical and practical scope of this project.

4.7 Application Settings

In order to further improve the usability and accessibility of the application, a number of settings were included.

By clicking the corresponding button in the menu, the user can open the settings in a new window. Hovering the mouse over these options displays a tooltip – a more in-depth explanation of how that option affects the application. The user can make changes to the settings. Upon saving them, the application will write them to a file and prompt the main controller to reload these values.



- [✓] Save positions of elements in file

- [ ] Keep attacks straight
- [ ] Allow stratified meta argument frameworks

Argument base	#fff8dc	Selection	Yellow
Proponent argument in game	#fff8dc	Opponent argument in game	#ffe4b5
Attacked argument	#ffdab9	Attacking argument	#f08080
Attack base	#228b...	Attack control point	#f0f8ff

Figure 4.13: Settings window

Upon starting the application, it will attempt to read the local settings file for the values. If the file does not exist or is corrupted, a set of default values is set.

The settings include changes to the behaviour – save file settings and choosing whether to play with the computer – and changes to the visuals. The latter is colour choices, which can fully customize how the application displays frameworks, trees, and games. This also means that if a user has difficulty differentiating between the default colours used by the application, they can change them according to their needs.

Chapter 5

Evaluation

The intended use case for the application created in this project is as a teaching tool to help understand the concepts of Argumentation Frameworks and Argument Games. As such, I wanted to put extra emphasis on the usability and polish of the program. The way to do this was to conduct small-scale usability testing near the end of the project. The resulting insights could then be addressed with a number of small changes to improve the user experience.

5.1 User Experience Testing

This evaluation took place once the program was in theoretically complete state. All features have been completed and the functionally correct. The goal of this evaluation was to judge the application and find areas where it could be improved.

Care was taken to ensure this process strictly classified as service evaluation, rather than research, to comply with the ethical guidelines. As such, all tasks involved only evaluating the application. No personal data was collected.

Additionally, all collected data was merged together without differentiating by participant. This means it is not possible to determine with certainty that any two insights were provided by the same participant, let alone that participant's identity. It is also not noted how many users raised a particular issue. This does not impact the usability of the results, as the testing was qualitative rather than quantitative.

The testing consisted of a 20-minute individual interview with three different participants, who were asked to perform a series of tasks in the program while being watched by the interviewer. The participants were asked to try to speak their thoughts out loud to get insights into their expectations of how the program will work. All participants have had no previous experience with the application nor Argumentation Theory.

All interviews followed the script provided below.

5.2 UX Interview Script

Preparation

Load default settings for the application. Ensure the file game_framework.txt is correct and located on desktop.

Introduction

Introduce yourself. Inform the user the interview should take about 30 minutes, then describe what Usability Testing is. Ask them to try to speak their thoughts out loud as they complete the tasks. Assure them there are no wrong answers – this is a test of the application, not of them.

Describe background

Describe the general idea behind argument frameworks. Introduce an example:

Tomato is a fruit ← Tomatoes have a savoury flavour ← Some tomato varieties are sweet

Argument Frameworks

Open the program and ask the participant to perform a series of tasks.

1. Represent the previous tomato example in the app.
2. Save the resulting framework to a file. Then, open a new file.
3. Create a framework of arguments A and B that attack each other. Ask user to adjust the visuals, so it's clearly visible.

Argument Games

Ask the user to load file game_framework from desktop. That framework is shown at the end of the script.

1. Play a game without the computer on the preferred framework.

Meta Argument Games

1. Reopen the tomato framework. Ask the user to add an argument representing: Fruits are not classified by sweet flavour to the framework.

2. Add an attack from the new argument onto the attack from “Tomatoes have a savoury flavour” to “Tomato is a fruit”. NOTE: Under current build, meta argument frameworks must be enabled in the settings. This is not true under default settings – the user will need to turn these on manually.

Settings

Ask the user to open settings and describe what they believe each option means (after reading in-app descriptions)

Ask them to change the colour of arrows to any colour they want.

Conclusion

Ask the user whether there is anything else they would like to tell us about. Thank them for their participation and reiterate the value it provides.

game_framework.txt contents:

```
Nodes:
a,300.0,506.0
b,118.0,369.0
c,468.0,328.0
d,433.0,159.0
e,194.0,152.0
Edges:
c,a,391.0,330.0
c,d,451.0,244.0
d,b,281.0,263.0
a,c,394.0,496.0
e,c,331.0,240.0
d,e,313.0,155.0
b,a,214.0,436.0
```

5.3 UX Insights

While the interviews took a very short time to carry out, it produced valuable feedback. The users were able to carry out most tasks without any help needed. The icons made for the application were intuitive, except for the “Clean graph” button, which caused confusion with some participants. Based on that, tooltips were added to the buttons and labels were added above the panes with the framework and the game tree.

Another source of confusion was the method of adjusting the curve of the arrow. The users did not expect the arrow to be selectable. To address that, when the user creates a circle or arrow, they are immediately selected. This instantly shows to the user the possibility, and improves the flow of work – as it was found the users very often followed a “Add Argument” action immediately with a “Add Attack” action, originating from that newly created argument.

Another issue that was brought up was upon selecting a New File. That action, at the time of the interview, immediately prompted the user to pick a place to save the new file. Some participants were confused by that and thought they clicked the wrong button. Based on that, the button now clears the application, but no longer prompts the save.

However, not all the raised opinions have been addressed. During the interviews, participants pressed keyboard buttons, expecting them to perform certain actions. This includes ’delete’ to delete the currently selected item, ’ctrl-s’ to save, and ’ctrl-shift-s’ to save a new file. These keyboard shortcuts are not used by the application, even though the related actions are implemented – just performed using on-screen buttons. Adding these keyboard shortcuts in the future could further improve the usability of the program.

When asked for features they would like to see in the application, participants mentioned:

a keyboard shortcuts

b zoom function

c editing names of existing arguments

d option to highlight connected arrows when selecting an argument

These are good considerations for future expansion of the application and are discussed further in the Conclusion chapter.

5.4 Program Evaluation

Besides the insights gained from the Usability Testing, I have more to say about the evaluation of the application. While I am happy with the current state of it, especially given the improvements made from the interviews, given more time, it could be improved further.

The possibly largest piece of criticism could be the internal format of the data. The frameworks are created as a series of many objects of different classes. This approach is not the fastest nor the most memory efficient. However, this was a conscious choice, as I wanted to focus on the readability and expandability of the code. Dividing the many methods between relevant classes achieves that better. This does not change the fact that the program will be slower when processing very, very large frameworks, even if data of such size is unlikely to be used in a teaching program.

The topic of advanced argumentation frameworks is one I wish I had been able to explore more in this project. Unfortunately, using them to play argument games directly is complicated and would require serious theoretical and practical work. I have attempted to tackle the theoretical side, specifically by considering a labelling for the attacks in addition to arguments, but the result was not satisfactory.

Though not the most important, usability was a side-focus of the project. It was important to me to produce a piece of good quality software that can be easily used by any user. I believe I have achieved the standard I was aiming for, both by careful considerations during production and by conducting appropriate evaluation. In doing so, I followed both the good practices in the field and the ethical guidelines. While not all ideas could be realised, the improvements were still significant.

I discuss ideas for further work, including features I did not realise in the current version of the project, in the Conclusion chapter.

Chapter 6

Legal, Social, Ethical, and Professional Issues

For the entire duration of the project, care was taken to consider possible ethical issues. The project is individual, and does not involve other people. That, in addition to the topic, might give the idea that such concerns are unnecessary.

That is not true. Disregarding the discussion of ethical concerns might make it possible for biases to slip by unnoticed. That is why I wanted to approach this topic seriously.

The biggest possible concern comes from the topic of the project – Argumentation theory. While dialogue and arguments are part of daily life, they can delve into socially and ethically unacceptable areas. In my project, I have to present examples of arguments. I made sure that the examples given are not harmful.

Another concern relates to the application's intended use. It is meant to be a teaching tool, to help explain a complicated concept. If the application behaves incorrectly, it can disrupt the learning process of the user. An error in code can give them incorrect knowledge. As such, the correctness of the application is crucial.

The last concern regarded the accessibility of the application. Every student should be able to use it. To accommodate for that, most written communication from the app comes in the form of pop-up windows. The user can take any amount of time reading it, and the window will disappear only once closed by the user. Another feature that improves accessibility is the option menu, where the user can freely adjust the colours the app uses to mark its elements. Users with, for example, colour blindness, can adjust these options to make sure the colours used are easy for them to differentiate.

Chapter 7

Conclusion and Future Work

The aim of this project was to produce a piece of software that can be used to create argumentation frameworks and run argument games, in order to help understand these complex topics. This aim has been achieved, with an application that is not only functional, but also polished to provide a good user experience.

Achieving it produced many varied challenges, both on the front-end and the back-end. It required me to come up with many uncommon methods, such as clearing up the graph and calculating the positions of nodes in a displayed tree. In such cases, my approach was to start on paper, considering the many cases and desired outcomes. I did not start writing code until I had a first draft in mind. The project validated this approach, as it resulted in a success in each case.

It has also shown the importance of considering priorities in the objectives. Not all work items I came up with could have been produced in the time-span of the project. However, by focusing on the base functionalities first, with minor considerations to UI design early on, I have been able to ensure achieving them before focusing on improving the application's usability.

Throughout the project, I have gained a lot of knowledge of Argumentation Frameworks. I realised their importance, which only fuelled my motivation for creating a program to help understand them. I am proud of the final result.

7.1 Future Work

Possibly the most interesting idea to expand upon are the advanced argumentation frameworks. The addition of attacks on attacks change more than expected with how the labellings, games, and winning strategies are implemented. The functionality included in the current application is just a small part of that.

Another technically interesting item of future work would be the improvement of clean-up function. Currently, it only takes note of circles, moving them away from each other if overlapping. It does not care about the arrows between them. A more sophisticated version could consider these, or possibly move the control points so that as few arrows cross as possible. I have made an attempt at that, but the issue turned out to be much more complicated than expected. While not related to the argumentation frameworks directly, it is still an interesting computational and mathematical challenge.

Then, there are many possible quality-of-life improvements. One that was considered during the development was the ability to zoom in by using the scroll wheel. While the work on it was started, it was declared low priority. It was eventually not included in the final release.

The editing of names and usage of keyboard shortcuts, that came up during the usability testing, are more possible improvements. While not expanding the range of final results possible to create in the application, they can make the process faster for many users.

Lastly, are the options to improve the visual clarity. As suggested by the testing participants, having an option to highlight the arrows connected to a circle when selected can make it easier to process, especially with more complex or cluttered frameworks. However, care would need to be taken to ensure these highlight do not mix up with the highlights when selecting an arrow.

Bibliography

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

[2] Modgil, S. (2009). Reasoning about preferences in argumentation frameworks. *Artificial Intelligence (AIJ) Volume 173, Issues 9-10*, pages 901–1040.

[3] Modgil, S. and Caminada, M. (2009). Proof theories and algorithms for abstract argumentation frameworks. *Argumentation in AI*, pages 105–132.