7CCSMPRJ Final Year

An Empirical Analysis of Personality Traits of Artificial Intelligences in Social Card Games

Final Project Report

Author: Alexander J D Hoare

Supervisor: Josh Murphy

Student ID: 20102303

Word Count: 14261

September 3, 2021

Abstract

Artificial Intelligences (AIs) often struggle with being able to correctly understand human emotions, and in particular social elements. In a social card game, players will use persuasion, bluff, deception and manipulation to improve chances of securing victory. Whilst AIs are able to dominate games without social elements (such as chess), they do not perform well in environments where they cannot mathematically quantify the expected utility of a particular move that is based entirely on human psyche and personality.

In this paper, we propose a simple card game based on proposing and guessing numbers, with either rewards or punishments being dealt based on one players intrinsic understanding of one another. The AIs navigate this space by having two main components; player models and the ability to deliberate. The player models store personal attributes (such as trust or deceitfulness), and the argumentation in deliberation allows players to influence others' model of another player.

The paper discovers that, in general, players who greatly trust or distrust others more achieve higher scores. However, the best personality types require synergy between different traits depending on the parameters of the game. In shorter games, for example, it is advantageous to be more aggressive. However when parameters favour longer length games, a player's ability to remain unthreatening to others becomes vital.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 15,000 words.

Alexander J D Hoare

September 3, 2021

Acknowledgements

I would like to thank Josh Murphy for his assistance during this project. I am thankful for Josh's patience, and flexibility to allow me to explore a project that I feel could produce interesting results. His diligence and kindness in making sure I stayed on track was invaluable.

Contents

Chapter 1

Introduction

1.1 Definition

Artificial Intelligence (AI) has a strong ability to mathematically evaluate a situation to influence decision making. However, an area of real-world implementation that an AI typically struggles with has always been human understanding/emulation, and AIs that are able to exhibit social understanding are often ”cheating” or ”mimic[ing]” [1] to give the impression of real social intelligence. When playing competitive, social games (digital or physical), this lack of understanding can often be utilised by human opponents in games that involve social elements to their advantage.

In this paper, we argue that ‘social games’ are defined in this scope as games where one can use cheating, bluffing, deception, teamwork, argumentation or persuasion to improve your chances of victory/a high score. Social elements are difficult for a machine to quantify, let alone understand, resulting in an overall poor performance by AIs especially in social card games such as poker, Werewolf, Hanabi, etc.

1.2 Importance

Artificial Intelligence is a field of Computer Science that is experience huge interest both from academic, but also commercial communities. It is estimated that the artificial intelligence global market is set to reach 169.411bn USD by 2023 [2], showing a massive demand and willingness for investment in this sector. Having an AI more capable of social interactions through human

emulation will open up market investment into sectors which are typically closed-off for AI products.

In 2019, DeepMind's AlphaStar artificial intelligence was able to reach the top league [3] in the popular video-game Starcraft II, a popular electronic sport (e-sport) title released in July 2010. In 2017, the Starcraft II World Championship series, had a prize pool of 700,000 USD [4], showing that learning new strategies from an AI can prove to be lucrative, especially with prize pools of up to 34.3m USD up for grabs in the case of Valve's DoTA2 franchise [5].

Commercially, improving an AI's understanding of a human can also help improve machine-human relations, making it easier for people to accept machines in environments typically associated with a much more emotional element, such as a teacher, doctor or lawyer.

1.3 Difficulties

The problem is difficult firstly because, unlike in the case of Deepmind's Alphastar, it is infeasible to create a machine learning model running on a cutting-edge supercomputer trained in every single possible card game. There exists too many variations on rules, limitations on computational power as well as social card games being balanced around the social element, rather than discovering a dominant strategy and exploiting it.

Key to strong results in social card games is understanding what your fellow players might be doing, What their goals are, how they will react to certain actions and their style of play. These 'soft' metrics can prove to be challenge for a computer to quantify, especially as the accuracy of the model is critical in achieving victory.

1.4 Approach

To create an effective simulation of a social card game, we will create one with our own new ruleset. These rules are designed to be as simple as possible, to allow for a broad overview in a social card game that does not restrict the AI to niche game mechanics.

From there, we will implement an AI that firstly behaves in a way that is comprised of human characteristics. In a social card game, we believe that any player's actions are influenced

by traits (such as Trusting, Calculating, Aggressive, etc) which are used to influence a personal model, which will then be queried to find which action should be taken and against whom.

Additionally, a key component of a social card game is that you do not know what moves an opponent will make, therefore the better you are at guessing what their intentions will be, the greater your chances of survival. Every AI will also hold their own personal copy of the model of each opponent, updating it depending on what moves other AI's make against who.

Finally, a very understated area of social games is the conversation during play. In a real-world situation of friends/adversaries in a social card game, they will attempt to casually either downplay the strength of their situation, attack another player to coordinate a alliance, or play aggressively to create fear in other players to prevent being attacked. To emulate this, we will be using an argumentation framework during a 'deliberation' phase of the game, which will allow each player to take it in turns to make statements about other players, which will further tweak other players' perceived models of other AI's in the game.

Chapter 2

Background

2.1 Technical Background

2.1.1 Rules of the Game

The social card game "The Number Guessing Game" involves an unlimited number of players greater than 1. The game is broken down into two phases; the Gameplay phase and the Deliberation phase.

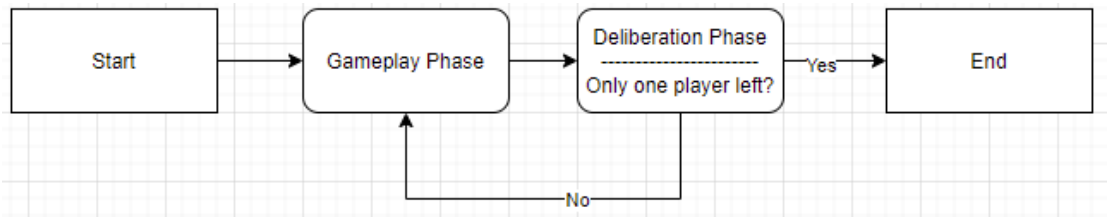

Figure 2.1: UML Diagram of Player

Gameplay Phase

A random player starts at the beginning of a round. The player whose turn it is puts down a numbered card (1-9) face down, and targets it at another player.

The player then makes a statement about what the card is, such as "It is a 5". Note that a player can make any statement about the card, including truths or lies. The player being targeted can then make a guess about which number it is.

If the guess is correct, both players gain one life. If the guess in incorrect, the guessing player loses life equal to the difference between the actual number and the guessed number. For example, if the number is 5 and the player guesses 8, the guess is incorrect and they lose 3 health points. Each player starts with a total of 20 health points. If you have less than 1 health point, you are eliminated from the game.

The game continues in a circular, (anti)clockwise rotation until every player has had a turn. At this point, the game goes into the deliberation phase.

Once the deliberation phase is completed, a new player is selected at random to start a new round. This process continues until only one player remains in the game and is declared the victor.

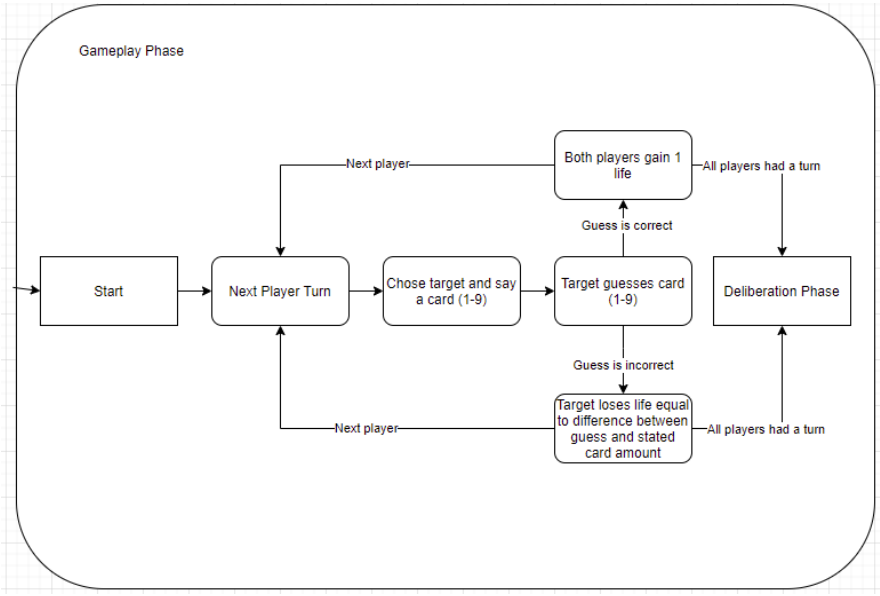

Figure 2.2: UML Diagram of Player

Deliberation Phase

The deliberation phase is a part of the game in-between rounds in which players can make arguments about other players, in an attempt to influence other player's decision making towards others.

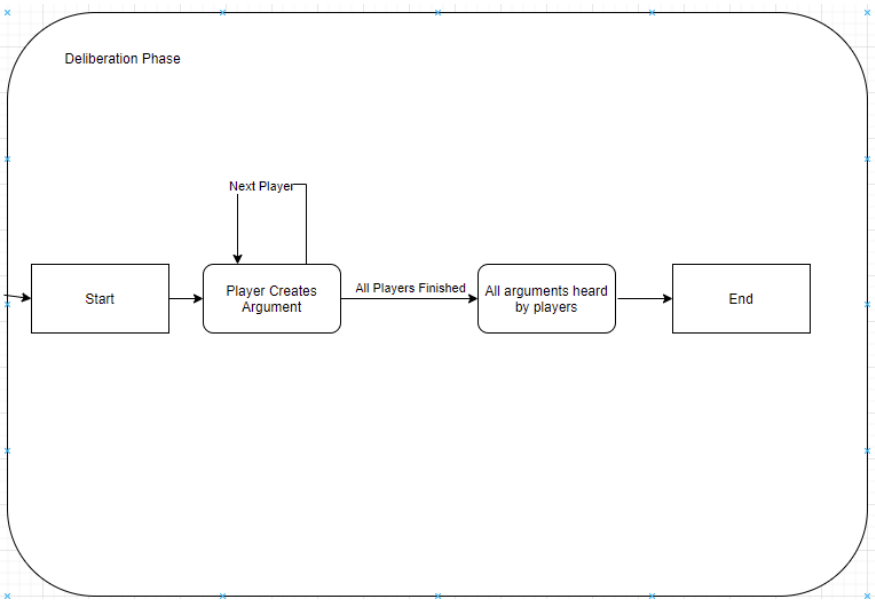

Figure 2.3: UML Diagram of Player

2.1.2 Player-Model

Each player will have two types of models; a personal model, and a model of perceived values for each other player in the game. In the personal model, variables are stored to represent each facet of a personality that are applicable to the Number Guessing Game. These numbers will be created and assigned to a player's personal model at the beginning of the game, and will remain unchanged throughout. Every action a player makes will query these values to decide which is the best choice, and the values may also be used to change one's opinion of the actions of others. For example, a player that is very trustworthy is more likely to believe that an opponent's deceit was merely a one-off, dampening the negative effects of the player's perceived player model of the opponent.

2.1.3 Perceived Player-Model

The perceived player-model is the core of this project, offering insight into which players will antagonise each other, co-operate with one another, or view each other as threatening or trustworthy. The perceived player-model will be stored values of the beliefs of every opponent's personal player-model. The perceived-player model will be updated based on actions taken, arguments heard and strong/weak positioning over the course of a game. A player can, for example, use the perceived player-model to calculate the likelihood that the opponent currently targeting them is acting deceitfully, or in good faith. Using this, a player can locate players

they can trust, work together with, or alternatively try to eliminate or antagonise.

2.1.4 Traits

We propose that the relevant human elements of a social card game is the evaluation of personal values, as highlighted in the player-model, but accompanied with traits and personal strategies, which for human players are learned through previous experiences unrelated to the game being played, but for a machine can be emulated by hardcoding strategies and/or traits.

In this scenario, traits are used as permanent modifiers to the player-model. For example, the trait "Trusting" will increase the player's personal model's trust variable by 20, whereas the "Untrusting" trait will decrease it by the same amount. This is done firstly to quantify human-assigned traits, making the results easier to digest and for more comparative personality trait conclusions to be drawn.

2.1.5 Argumentation Framework

An Argumentation Framework (AF) is a directed graph formulated as $\langle A, R \rangle$, where A is the set of possible arguments that can be made, and R is the set of attack relations, where (a, b) $\in$ R means that node (player) a attacks node b with an argument $\in$ A.

For example, Player 1 "Bob" attacks Player 2 "Alice" with the argument 'Deceitful', meaning; "This player is deceitful". This would create the argument:

$$\langle "Deceitful", (Bob, Alice) \rangle$$

Each argument can make one of six different arguments, a positive integer effect on each of the variables in each player's perceived player model, and a negatively effecting one. The usage of argumentation framework in the 'deliberation' phase will be determined by an examination of the player's own player-model, the selection of an appropriate target influenced by its stored player-model on another players, which is then affected by the player's own traits to eventually come up with an argument against a player.

2.1.6 Custom Game Justification

The creation of a new, general social card game was undertaken to improve the overall flexibility of the results of the analysis. If an analysis is performed on a niche, bespoke game such as Werewolf or Hanabi, then the results may only be applicable to that particular domain.

In popular social card games, design is often centred around maximising the enjoyment of particular players. Outside of gambling social card games (such as poker), the cost of a defeat is often a loss of personal pride amongst friends. As a result, players are often not motivated to always seek dominant strategies and exploit their opponents to maximise their chances of winning. The 'Number Guessing Game', whilst lacking depth and being unlikely to be particularly enjoyable for a human player to play, is designed to be as broad as possible. This breadth allows for the results to be applied to most social card games, albeit unlikely to show with certainty which personality traits perform best in all social card games. The structure can be broken up into three constituent parts: cards, gaining/losing health and making simple arguments/statements.

Cards

The first step of achieving this increased simplicity was first to transform all playable 'cards' into simple numeric values. A card is a integer value, and therefore holds no secret side effects, intrinsic value or additional effects on the game. A benefit of this simpler implementation is that it becomes much easier for an AI to quantify how much threat a particular player poses when playing a card, or how much they have been deceived by when they face a loss of health. Compared to a game like Werewolf, where there are three different card roles, each with three different abilities during different points in time during the course of the game, an AI may find it difficult to exactly pinpoint where the greatest expected utility may lie in a social context.

Gaining/Losing Health

Another step in generalising and making quantifying easier is the way that progress throughout the game is tracked. The higher your health, the longer you can survive and therefore the greater your chances of victory. When another player takes an action that deliberately causes you to lose health, an AI will be able to directly attribute that to a social deception. Similarly, when an AI takes an action that causes a player to gain life, the AI knows that a socially beneficial action has taken place. This lack of ambiguity allows AIs to directly assign in-game

actions to social constructs, such as having been deceived or not.

Simple Statements

An Argumentation Framework is an comprehensive framework for reaching a conclusion on a contentious issue. The contentious issue in question in the context of this game is to convince other players that your perception of that player is the most accurate. Whilst normal, in-person deliberation has an infinite number of points to be made, conclusions to draw and structure of arguments, this game seeks to simplify it into simple, one-line statements about a single one of another player's perceived-player model variables. This simplification allows us to effectively simulate deliberation, without having to delve deeply into the psychological aspects of the construction of an argument by limiting the arguments to be made by the AIs.

2.2 Literature Review

2.2.1 Dung 1995

P.M Dung's 1995 [6] paper on the acceptability of arguments in n-person games is incredibly important when exploring co-operative game theory. The paper highlights the reasons why humans use argumentation in a social game, citing that it is "essential for humans to understand new problems", and to "defend their opinions".

However, the paper explores the uses of argumentation in co-operative game theory to work towards conflict resolution. This, whilst using many similar notations, ends up not being the purpose of argumentation in this paper. Here, we utilise argumentation to primarily create conflicts, sabotage reputations whilst also having the ability to persuade another that you are more trustworthy with a finite number of arguments to be made, whereas Dung explores all arguments possible in theory. The paper also goes into detail of the utilisation of many different subsets of arguments, including conflict-free, stable-extensions, grounded, etc, which play a heavy role in influencing the examination of AI's using argumentation frameworks, that in this paper are not relevant.

The paper, however does highlight the importance of the "fundamental role our theory of argumentation can play in investigating the logical structure of many social and economic problems.". Using this, we can use Dung's theory of the acceptability of arguments to effectively

weight a particular argument based on the person who said it and to whom, allowing us to quantify an arguments strength, and therefore its effect on all players' perceived player model.

2.2.2 Axelrod's Tournament

Axelrod's Tournament [7] was a multiplayer game based on the Prisoner's Dilemma held in 1980. In this game, multiple AIs competed against each other in an experiment on game-theory using the famous dilemma. In this tournament, AIs were created that adopted different strategies, such as "Always Defect", or "Tit-For-Tat".

The paper demonstrates the impact that different strategies have on the overall success of a game requiring co-operation, or absolute selfishness. With scores listed to quantify success, we are able to see benchmarks of whether more trustworthy or co-operative AIs by Axelrod's breakdown of "properties of successful rules"/strategies. In this section, we see that many of the sucessful strategies, outside of Tit-For-Tat, often focused on their "niceness" and "forgiveness" as key attributes. In our solution, we implement these features in our player model, in the form of personal trust (niceness) and the perceived trust of another player (forgiveness), with additional modifiers affecting these variables based on an individuals personal traits.

However, Axelrod's Tournament is limited when making a direct comparison mainly due the vastly different format of the game at hand. In the prisoner's dilemma, each player deals exactly with one player at a time, and has no influence over the decision making process of others. Additionally, the results of Axelrod's Tournament are in fact limited as it gives the impression that the best-performing strategies are in fact the best. After the tournament had taken place, it was found that there are many rules that could defeat Tit-For-That (the winner) in the environment. The proof that there exists additional strategies that can outperform the current winners results shows no actual indication of whether more forgiving/nice AIs perform potentially in co-operative/non-co-operative games, as there may be rules that are more ruthless in strategy, that could be the better. To improve this, the Number Guessing game will bring multi-player, open games that will allow for strategies to naturally form, rather than being hard-coded. This change to natural formation will allow for context-sensitive actions to be undertaken by the player, as in a realistic game a player will not make a move that is guaranteed to have a negative effect, despite it being against their strategy.

2.2.3 Modelling social action for AI agents

'Modelling social action for AI agents' [8] outlines the keys to success in any AI in a social environment. It explains the importance of behaving in a way to encourage coalitions, versus being selfish. It also brings about the idea of breaking down a social AI into three basic ontological categories: "action, structure and mind".

The paper gives insight into the application of agent sociality, especially in regards to agents interfering or depending on each other. The paper argues that "mind is not enough", highlighting the importance of introducing additional modifiers and variables to improve the emulation of a human personality in a social card game. The three mentioned ontological categories can form a structure from which to build the AI's personalities around, to observe which personality types, emergent coalition structures, and strategies perform best in the game simulation.

However the paper bases the model of social action for AI agents mostly on a goal-based approach, in which the social actions are primarily motivated by the persuit of a particular end-goal. In this paper, whilst it is obviously ideal for an AI agent to win the game, the primary driver in its selection of its actions is to make an action against the most appropriate player, be it a positive one (gaining both parties a life) or a negative one (attacking a player). This key, but subtle difference in primary agent motivation results in many approaches mentioned in this paper to be not directly applicable, such as goal-delegation of cooperating agents.

2.2.4 Personalities for Synthetic Agents

In 1996, Daniel Rousseau and Barbara Hayes-Roth [9] explore the effects of differing personalities on their ability to improvise and behave differently depending on the situation they're in. In this example, they give each agent different ratings of "friendliness, confidence and activity" and ask an agent to carry out a conversation in a CyberCafe.

The paper brings insight into the constitution of a human, and therefore AI agent emulating a human's personality. It raises concerns that "there are many theories proposing various personality traits and none of them is universally accepted", including that "Psychologists disagree about the number and the identity of basic traits". This lack of cohesive universal understanding of the building blocks of a personality both hinder but also help the scope of the project. On one hand, the lack of consensus means that there is a lot more freedom in what traits/person-

alities may be implemented. With no framework to adhere to, a more bespoke solution can be crafted that lends itself to the problem domain more, without corrupting essential structures set in stone within AI Psychology. However, this lack of framework does also leave a lot of ambiguity as to what does constitute a personality. Without a consensus, there is very little way of knowing if a constructed personality for the purposes of this paper actually plays social card games well, or if it just plays the Number Guessing Game well.

Chapter 3

Specification

3.1 Requirements

Below is a table listing the requirements of the project:

- Understand what personality types/traits perform best in social card games.
- Understand the role of deliberation and argumentation for AIs in social card games.
- Implement the Number Guessing game in C#.
- Implement an argumentation framework for players to utilise.
- Implement user models to hold personal attributes and personality values.
- Implement traits, and how they interact with a user model.
- Implement a user model that holds beliefs about the personal attributes of others.
- Implement how arguments/deliberations influence this perceived player-model.
- Implement how actions undertaken can influence player's perceived player-model.
- Maintain the flow of the game to maximise realistic emulation.
- Implement mechanics to maintain flow such as decay and logical behaviour to avoid infinite player loops.
- Add easy-to-access parameters for additional testing upon final release of project.
- Add different evaluation metrics to assess performance of traits.

3.2 Requirements Testing

The requirements will be tested by playing 100 back-to-back games, each game starting with new players with new traits. The decision to have 100 games is to ensure that a wide variety of different players, personalities, traits and trait synergies are observed and to avoid any unusual distribution that could occur due to the random assignment of traits.

The detailed quantifier used for an AI's success will be victory rate. In this scenario, two approaches can be taken. Firstly, a points-based system based on the number of players in the game. For example, out of 20 players, the victor will claim 1 points, runner up 2 points and the last player to be eliminated will game 20 points, with lower points being better. Alternatively, a system that only counts a victory as the last-player standing will be able to find a dominant personality in a clearer manner. Finally, a system that will measure the survivability of a particular player; that is, how many rounds they were able to stay in the game for.

Ultimately, the project will determine which combination of traits result in a higher win rate and what personality type performs best in a social card game, allowing for a human player to adopt and emulate these strategies in a game of their own, perhaps with friends, to result in a greater chance of winning.

3.3 Benchmarks

One particular benchmark which may be interesting to observe is to see which personality categories (traits, strategies or base player model) have the greatest effect on the victory rate of an AI. In Axelrod's Tournament[7], the Tit-For-Tat strategy was the most effective in securing victory, so it could be expected that behaving like that may enforce other AI's with their strategies that are more forgiving to perform better when against it. However, it could also result in a widespread hatred (low trust) of that AI's strategy, so it might not perform well. As a result, it is challenging to compare the results of Axelrod's Tournament to the potential results of this Number Guessing Game, due to the differences of the game structure, strategy implementations and additional personality category implementations such as traits and a player-model.

Chapter 4

Design

4.1 Models

The decision to opt for the use of user models for each player and opponent was made to offer a differing solution to the majority of research into this field, which primarily focuses on creating a tree-graph of all possible next game states using Monto-Carlo tree searches [10] or machine-learning techniques [11]. Whilst these solutions offer a method to create an AI that will *succeed* in a social card-game environment, these methods do not touch on the 'believability' and accurate emulation of a human player. As a result, user-models are proposed that hold values about players which are used to make decisions about moves.

It should be noted that for both player-models and perceived player-models, the value ranges 0 to 100 have been used. This decision was made in order to keep things relatable to probabilistic analysis (by dividing by 100 to get 0-1 values). The decision to use 0-100 instead of 0-1 values was made for design simplicity. Integer incrementation, implementation and overall analysis of systems within the project were made simpler for development, thus faster progression was able to be made with less resource constraints.

4.1.1 UML Diagram

Below is a UML diagram outlining the Player, PlayerModel, PerceivedPlayerModel and Arguments along with the relationships between them.

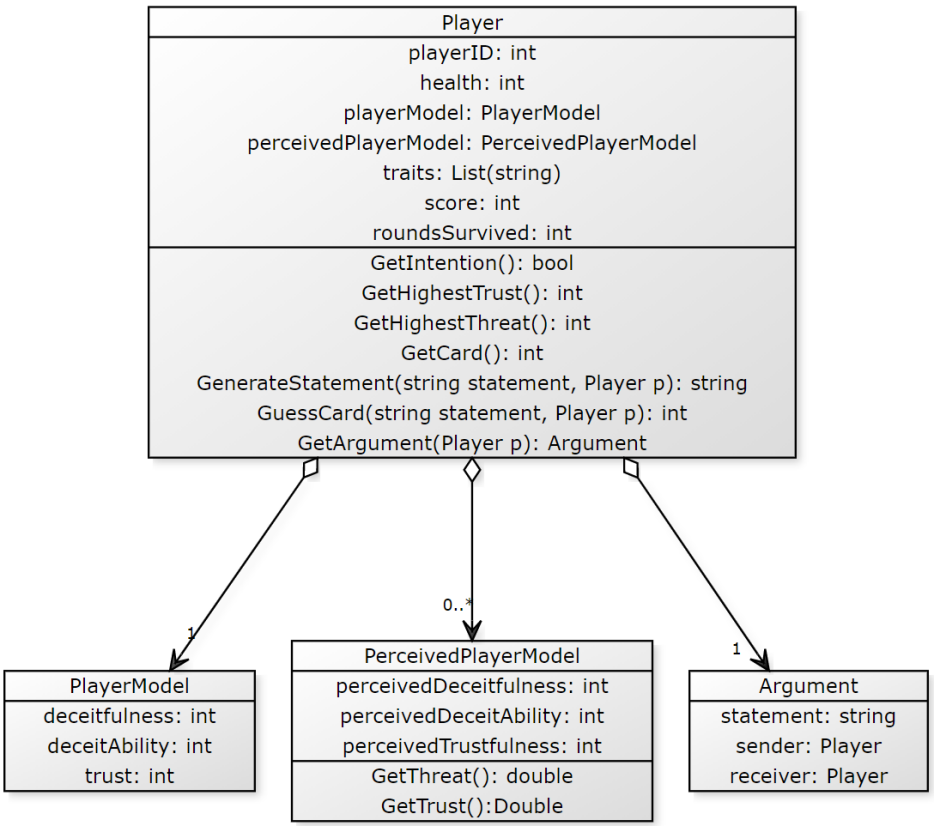

Figure 4.1: UML Diagram of Player

4.1.2 Player Model

The personal Player-Model is used by a player to store attributes about themselves. These values are queried for all actions undertaken by that player, be it for selecting a statement, choosing a target or guessing a card. The following variables are stored about an individual player:

Variable Name	Range	Description
Deceitfulness	0 to 100	How often this player will try to deceive another.
Deceit Ability	0 to 100	How large this player's deception will be.
Trustfulness	0 to 100	How likely a player is to trust/like another

Figure 4.2: Player Model Attributes

These values were selected as they completely encompass the social aspects of what is required in the scope of the Number Guessing game. During the Gameplay phase, the only decisions a player has to make is whether or not they are going to deceive, and if so, how much by. These are encapsulated by 'deceitfulness' and 'deceit ability'. From there, the variable of trustfulness

is used to determine how affected a player is by the actions of others.

Additionally, these same three variables can also be brought across into the Deliberation phase of the game. The same variables are used to determine whether they are going to make a positively effecting argument, or a negatively effecting one. Trustfulness is then used in the argument resolution to determine how much a player believes the argument of an opponent.

4.1.3 Perceived Player Models

Additionally, a perceived model that is update at the end of a round based on a player's action is updated, stored with the following values:

Variable Name	Range	Description
PerceivedTrustfulness	0 to 100	How does this player trust others.
PerceivedDeceitfulness	0 to 100	How likely this player is to deceive me
PerceivedDeceitAbility	0 to 100	How much this player is likely to deceive me by

Figure 4.3: Perceived Player-Model Attributes

The values selected for the perceived player-model (PPM) is identical to the values stored in the personal player-model. The PPM is used to hold the *beliefs* of a player about an opponent. Using these models, a player will be able to better calculate an appropriate target by gauging which opponent is the greatest/least threatening to their position so that they can act accordingly.

4.2 Traits and Arguments

4.2.1 Traits

Below are the list of traits available to a player. The traits are assigned at the beginning of a game, and each player is given a random amount between one and three inclusively. Each trait belongs to a different 'trait group/related player-model' variable, with the groups being; Trust, Deceitfulness and DeceitAbility. Only one trait from each trait group can be selected. This is to avoid trait conflicts such as being both 'trusting' and 'untrusting' at the same time.

Trait Name	Related PM Variable	Effect
Trusting	Trust	Large increase
Untrusting	Trust	Large decrease
Unsuspicious	Trust	Small increase
Suspicious	Trust	Small decrease
Deceitful	Deceitfulness	Large increase
Honest	Deceitfulness	Large decrease
Calculating	Deceitfulness	Small increase
Fair	Deceitfulness	Small decrease
Aggressive	DeceitAbility	Large increase
Passive	DeceitAbility	Large decrease
Audacious	DeceitAbility	Small increase
Timid	DeceitAbility	Small decrease

Figure 4.4: Table of traits and effects

Each trait is assigned either a large/small increase/decrease to its related player-model variable. When a new player is created, each variable is set to a base of 50. A random amount is then added to each variable to create more uniqueness amongst individual players. This is made to avoid two players with only the 'trusting' trait being functionally identical at game start. The random amount added can vary, but for testing purposes has been a value between -20 and 20.

From there, the traits are then translated into the numerical values and the effects it would have on a player's personal-model. During development, the large modifier will be 30, and the small modifier will be 15. Once the traits have been translated and added to the corresponding fields along with the random changes, the player-model is completed.

Example of a player model creation:

Variable Name	Value
Trust	50
Deceitfulness	50
DeceitAbility	50

Figure 4.5: Initial values at creation

Variable Name	Value
Trust	55
Deceitfulness	37
DeceitAbility	59

Figure 4.6: Values after random changes +5 Trust, -13 Deceitfulness, +9 DeceitAbility applied.

Variable Name	Value
Trust	85
Deceitfulness	37
DeceitAbility	44

Figure 4.7: Values after traits Trusting (Large increase to Trust), Timid (Small decrease to DeceitAbility) are applied.

4.2.2 Arguments

Below are a list of arguments that each player can make. For simplicity, each player can choose one argument that either positively or negatively effects each of their opponent's perceived player-models attributes. This basic approach to argumentation and which arguments a player can make has been made in order to easily quantify the effect of an argument, taking elements of social nuance out of the arguments that a human player may exhibit. Whilst this may reduce the believability of the AIs playing the game, accurately representing dialogue is still "radically different in nature to those of importance when considering the concept of argumentation as it is familiar from everyday contexts" [12]. Therefore, the reduction in complexity does not detract too much from overall emulation, as argumentation framework as a framework for emulation is already not realistic enough.

Argument Name	Related PPM Variable	Effect
Trustful	Trust	Increase
Untrustful	Trust	Decrease
Deceitful	Deceitfulness	Increase
NotDeceitful	Deceitfulness	Decrease
Aggressive	DeceitAbility	Increase
NotAggressive	DeceitAbility	Decrease

Figure 4.8: Table of Arguments and effects

Each player must make a single argument. A player can choose to either make a positive argument (Trustful, NotDeceitful, NotAggressive) or a negative one (Untrustful, Deceitful, Aggressive) based on their calculated intention. Once each player has selected a target and an argument to make towards that target, all arguments are resolved at the end, with each argument having an effect of ± 1 on every player's perceived player-model of that target.

Argumentation Example

Below are three players, along with a demonstration of which arguments will be made and why.

Variable	Value
Trust	60
Deceitfulness	82
DeceitAbility	37

Figure 4.9: Personal Player-Model of Player BOB

Variable	Value
PerceivedTrustfulness	60
PerceivedDeceitfulness	82
PerceivedDeceitAbility	37
Health	15

Figure 4.10: Perceived Player-Model of Player ALICE

Variable	Value
PerceivedTrustfulness	22
PerceivedDeceitfulness	76
PerceivedDeceitAbility	95
Health	27

Figure 4.11: Perceived Player-Model of Player ELLEN

When Player BOB needs to make an argument during the first Deliberation phase of the game, he will query his own personal player-model first to determine his intention. His intention is calculated using the formula in Figure 3.12. Using this, a random value will be produced between 0 and $60 + 82 + 0 = 142$. In this case, the value outputted is 105, which is higher than BOB's trust. Bob is therefore going to act deceitfully.

Next, BOB must calculate a target. As he is acting deceitfully, he must calculate the biggest threat between players ALICE and ELLEN using the threat calculation formula in Figure 3.14. For ALICE, her threat is $82 + 37 - 15 + (0 - 0) = 104$, and ELLEN's threat is $76 + 95 - 27 + (0 - 0) = 144$. As ELLEN has the highest threat, BOB will target her. Now that BOB has a target and an intention, he will select a random negative argument from the list of possible negative arguments. BOB randomly selects the *Deceitful* argument. Therefore, BOB makes the following argument:

$$< "Deceitful", (BOB, ELLEN) >$$

4.3 Key Algorithms

4.3.1 Target Selection and Intention

During the start of a player's turn in the Gameplay phase, the player must first make the decision of whether or not they are going to be deceitful or truthful. This stage is called getting the players **intention**. This intention is calculate as a random number between 0 and:

$$trust + deceitfulness + consecNoChanges$$

Figure 4.12: Intention calculation

Variable	Description
trust	The current player's trust attribute.
deceitfulness	The current player's deceitfulness attribute.
consecNoChanges	How many rounds since a player was last eliminated.

Figure 4.13: Intention calculation variables

From there, if the number is less than or equal to trust, then the player is going to act honestly, otherwise they will act deceitfully. The introduction of the consecNoChanges variable will be done to prevent a roster of players that all like each other, resulting in a never-ending game where each player will behave honestly towards the other, boosting each other's health. This change also brings about an element of realism in the behaviour of the AIs, where if many rounds pass with no eliminations, it helps the player's realise that they will have to turn on one of their friends in order to achieve victory.

4.3.2 Calculating Threat

When a player makes the decision to behave in a deceptive manner, a target must be found to be deceitful towards. In order to do so, each opponent's perceived player-model is used to calculate a threat score, calculated as below:

$$threat = D + A + H + (B - G)$$

Figure 4.14: Threat calculation.

Symbol	Variable	Description
D	perceivedDeceitfulness	The belief of how likely this player is to be deceptive.
A	perceivedDeceitAbility	The belief of how much this player is likely to deceive by.
H	playerHealth	How much remaining health this player has.
G	goodActions	How many times this player has been truthful towards me.
B	badActions	How many times this player has been deceitful towards me.

Figure 4.15: Threat calculation variables.

perceivedDeceitfulness and perceivedDeceitAbility are used to get an overall sense of their deceptiveness. playerHealth is also taken into account to see which player poses a greater chance of winning the game. To add additional personality to the game, and to avoid the problem of all player's ganging up against one other player before simulatenously moving on to their next target, goodActions and badActions are used to hold how many times the player has behaved in a trustful (goodActions) or deceitful (badActions) manner in the past. This also will bring the additional benefit of allowing for personal alliances/coalitions to form between players, as well as being able to form a 'nemesis' over time, as two player's battle to defeat the other, worsening each other's threat scores of each other.

4.3.3 Calculating Trust

When a player makes the decision to behave in an honest manner, a target must also be chosen for them to be truthful towards. For this, each opponent's perceived player-model is queried to calculate a trust score:

$$trust = T - H + (G - B)$$

Figure 4.16: Trust calculation.

Symbol	Variable	Description
T	perceivedTrustfulness	The belief of how trustful this player is.
H	playerHealth	How much remaining health this player has.
G	goodActions	Number of times player has been truthful towards me.
B	badActions	Number of times player has been deceitful towards me.

Figure 4.17: Trust calculation variables.

Similar to the threat calculation, a trust score takes into account the overall perceivedTrustfulness of the opponent, as well as how little health they have remaining, as well as seeing if this

opponent has overall acted in a truthful manner towards the player by taking into account the overall good actions and bad actions made.

4.3.4 Argument Resolution

At the end of each gameplay phase, the game will enter into the Deliberation phase. In this phase, each player makes an argument, and they are all resolved at the end of the phase. During resolution, each argument will adjust the relevant perceived-player model attribute by ± 1 from the argument table shown in Figure 3.8. For example, the argument 'Trustful' will adjust each player's perceivedTrustfulness attribute about the target player by 1. Each argument is resolved, before the turn ends and returns to the gameplay phase.

4.3.5 Algorithm Justification

Creating an algorithm for target selection, intention and calculations of trust/threat present a challenge of simulatenously creating an algorithm which is representative of realistic human behaviour, whilst also making sense within the confines and mechanics of the Number Guessing game. The choice of variables used in Figure 3.12 to calculate intention was made as it, in very simple terms, fully encapsulates the decision making process of whether or not a particular player is going to behave deceitfully. As the personal player-model simplifies a player's deceitfulness into a singular probabilistic attribute, it can easily be queried (offset by personal trust and consecNoChanges) to get an indication of intention.

Similarly, the algorithms used to calculate trust and threat encompass all variables that could be relevant in evaluating an opponent's trust or threat. The challenge, therefore, is determining how these variables influence the output. As the variables can be assigned into clearly marked beneficial or non-beneficial categories (e.g perceivedDeceitfulness is *always* more threatening), the exact structure of the formula for calculation changed to a problem of basic arithmetic. Additionally, the choice to include previous actions undertaken by opponents in the goodActions and badActions variable is based on research that evaluations of trust in particular "differ according to the previous knowledge that players have of their partners" [13], increasing the overall quality of using trust and deceitfulness measurements.

Alternative algorithms could also have been used. Firstly, an approach that targets the weakest players instead of the most threatening players could have been brought in. However,

this would have resulted in the weakest players quickly being overwhelmed and eliminated from the game, causing a less natural, less distributed style of game where the first player attacked at game start would be perceived as the weakest, effectively resulting in players being eliminated one-by-one whilst others remained largely ignored. Introducing a biggest threat calculation that takes into account player health introduces player threat being reduced as they are being attacked, which brings a more balanced, healthy game.

4.4 Realistic Emulation

4.4.1 Decay

When humans play games against each other, people's beliefs about their opponents change over time based on that actions made against them, or against others. However, it is unlikely that a player over the course of a game, will still hold a grudge against another based on actions made at the start of the game, potentially hundreds of turns ago. As a result, to emulate 'forgiveness' in the players, a 'decay' mechanic has been implemented.

The decay mechanic sets a base of 50 for each of the perceived player-models attributes, and tends towards that base over time. The rate of decay is set to 0.1, with a decay occuring for all players' perceived player-models at the end of every turn (playing + deliberation phases). As a result, this should prevent perceived player-models attributes being stuck at a boundary (0-100), despite no action being undertaken by that player for a long time.

Chapter 5

Implementation

5.1 Language

The project was written in Microsoft's C# language in Visual Studio 2019. This choice was made primarily due to overall familiarity with the language. The advantages of C#, such as object-orientation allowed for easy abstraction and implementation of objects such as the Argument, PlayerModel and PerceivedPlayerModel classes. During the design phase of the development of this project, it was clear that the computational resources required to play the game, update all models for every player and resolve all arguments made in the deliberation phase would increase exponentially as new players were added. As a result, a language that featured strong speeds through code compilation rather than an interpreter was a necessity.

Whilst C# may not show pure runtime speed gains over languages such as C++/C, it makes up for it in Microsoft's excellent documentation, updates and external library support, such as LINQ queries, which were used extensively for optimisation. Overall, all these factors contributed to C# being the easiest language to implement the project in, without making large compromises to speed.

5.2 Classes

5.2.1 Argument.cs

The Argument.cs file contains a simple description of what comprises an argument in our project's context.

```
string statement;
Player sender;
Player receiver;
```

The argument objects contains the player that has made the argument, the player that is on the receiving end of the argument, and the statement made about that player.

5.2.2 Game.cs

Game.cs can be considered the main function of the program. This is where the implementation begins, and it controls the flow of the game and the game rules itself. Game.cs ensures the correct flowing of the Gameplay phase and Deliberation phase, removing dead players to keep the Number Guessing Game intact.

Main()

Main() focuses on the creation of the players within the game, and is the first entry point of the program upon running. It dictates how many players will be in the game, and how many back-to-back games will be played to get results from. It is responsible for setting all initial variables before beginning the GameLoop, as well as displaying results at the end of a game.

GameLoop()

GameLoop() is the main loop for the Number Guessing Game. It flows the players from the Gameplay Phase into the Deliberation phase repeatedly until only one player remains. GameLoop is also responsible for implementing decay between rounds, removing dead players from the game and keeping track of player's statuses to prevent infinite loops of friends targeting each other by incrementing the consecNoChanges variable.

GameplayPhase()

The GameplayPhase() method is the largest method within the entire program. Whilst good coding practise usually dictates the breaking up of larger functions, it felt appropriate to keep all game rules regarding the Gameplay Phase into a single function, to keep development more modular by methods. The GameplayPhase() function is responsible for selecting a random player to start the round, and calculating what that players intention will be, as well as fetching the card the player will use before generating a statement about that card based on the

player's intention.

Additionally, the GameplayPhase() also contains functionality for the player being targeted to make a guess about the card/statement their opponent has attacked with. The method then deals with the effects on all player's perceived player-models based on whether the statement was true or false. The GameplayPhase() then calculates the health loss/gain of the targeted player. This process loops until all players left in the game have had a turn.

DeliberationPhase()

The DeliberationPhase() method allows for all players to create an argument towards a target, and adds it to a list of all arguments. The arguments are then passed to PlayerListFunctions.ResolveArguments() to be resolved.

5.2.3 PerceivedPlayerModel.cs

PerceivedPlayerModel.cs deals with all queries regarding the data held within the perceived player-model of a particular player. The variables held are:

```
public int playerID //The ID of the player being stored;
public double perceivedTrustfulness = 50 //Player's perceived trustfulness;
public double perceivedDeceitfulness = 50 //Player's perceived deceitfulness;
public double perceivedDeceitAbility = 50 //Player's perceived deceit ability;
public int goodTimes = 0 //Number of times this player has acted truthfully towards me;
public int badTimes = 0 //Number of times this player has acted deceitfully towards me;
```

All perceived variables were set at the initial baseline amount of 50, before being adjusted further in the constructor.

Constructor

Within the constructor of PerceivedPlayerModel.cs, the playerID is assigned to its corresponding player, as well as random noise being added to all perceived variables, varying from -10 to 10. This decision was made to increase uniqueness of actions within the game and to stop all players being treated equally at game beginning. The noise accounts for human elements of social card games like first impressions, knowing someone from before, or body-language analysis which may take place in more competetive games.

GetThreat() & GetTrust()

GetThreat() and GetTrust() are short methods that demonstrate the implementation of calculations of threat and trust as displayed in figures 3.14, 3.15, 3.16 and 3.17. The methods are passed in the list of all players currently in the game, and the player object is then matched using playerID to fetch their remaining health to perform the calculations accordingly.

AddDeceitfulness() & AddTrust() & AddDeceitAbility()

All three Add methods are virtually identical, save for the name of the variable being altered. This method is called after a player has undertaken a particular action, and player's perceived player-models must change accordingly. The player model **pm** is passed as a parameter with the amount to change by as **amt**. An important line within these methods are:

```
perceivedDeceitAbility += amt * (1 - (pm.trust / 100));
```

In this example using perceivedDeceitAbility (note that it can be perceivedTrust or perceivedDeceitfulness), the amount that a particular perceived player-model will be changed by is dampened by the player's own personal trust. As all personal attributes are held in 0-100 ranges, it must be divided by 100 to get a percentage value, which is then applied. As a result, a player who has absolute trust in players will not be effected by negative actions others may take. Similarly, a player who has no trust in their opponents will have their models dramatically influenced by the actions of others.

5.2.4 Player.cs

Player.cs is the largest class within the project, dealing with all decisions that a player must undertake within the entire course of the game. A player is defined with the following attributes:

```
public int playerID; //Player's ID
public int health; //Player remaining health
public PlayerModel playerModel; //Their personal player model
public List<PerceivedPlayerModel> perceivedPlayerModels //List of all perceived
            = new List<PerceivedPlayerModel>(); //player-models of other players
public List<String> traits = new List<string>(); //List of player's traits
public int score = 0; //Keep track of a players score when they are eliminated
```

Constructor

The constructor takes in parameters of a list of traits and assigns them to the traits list, assigns an ID and also sets the default starting health, which will be changed throughout testing to compare its effect on the results of the game.

AddPerceivedPlayerModels()

AddPerceivedPlayerModels adds to the list of perceivedPlayerModels a new perceived player-model for every player in the game excluding themselves.

GetPerceivedPlayerModel()

Returns a perceived player model (PPM) of a particular player passed in as a parameter. The ID's of both the PPM and the player passed in are used to find a match and return the corresponding PPM.

GetIntention()

GetIntention() is used to get the decision about whether a player is going to deceive or act truthfully towards another. The intention is calculated by using a random number to generate a range between 0 and a number calculated relevant to that player (as shown in figure 3.12). One caveat included in the implementation is that if only two players remain, they will always act deceitfully towards each other, as being truthful is no longer beneficial in any circumstance.

GetHighestTrust() & GetHighestThreat()

Both these methods loop through all the players still remaining within the game, and fetches the highest trust (if GetHighestTrust) or the highest threat (in GetHighestThreat) out of all the players. If the player's intention is to act truthfully, then they will target a player whom they trust. Similarly, if a player chooses to act deceitfully, then they will calculate the target that is considered the greatest threat.

GetCard()

This method returns a random number between 0 and 9 for the player to generate statements for, based on the previously calculated intention.

GenerateStatement()

GenerateStatement() is used to create a string which corresponds to what the player wants to say. The statement is used to either make the deception itself, or to try to convince the target that you are being truthful. If the player intents to be deceitful, then the extent of the deceitfulness needs to be calculated, which is made by dividing their deceitAbility by 10 to discover their 'bound':

```
int bound = Convert.ToInt32(Math.Round(playerModel.deceitAbility / 10));
```

As all possible values that deceitAbility can be are in the range 0 to 100, dividing by 10 will always lead to a value between 0-10 that is then rounded to an integer value. This links numerically with the initial range that a card can be, meaning that the higher a player's deceitAbility, the larger the difference between the card stated, and the real value of the card. Bounds checks are made to ensure that the card statement is still within a 0-9 boundary, and returned as a string to create a statement. If the intention is not to deceive, then the real identity of the card is returned as a string.

GuessCard()

GuessCard() is the method used by players to check firstly whether or not they believe the opponent player is acting deceitfully or not, and then make a guess as to what the card may be based on the information they have. First, a check is made to see if the target player believes the statement the opponent has made by assessing their own general trust, and the perceived deceitfulness of the opponent:

```
double upperBound = ppm.perceivedDeceitfulness + p.playerModel.trust;
double lowerBound = ppm.perceivedDeceitfulness - p.playerModel.trust;
```

A number is generated between 0 and the upperBound variable. If this number is below the lowerBound, then the target believes the opponent is being deceitful.

From there, the extent to which the deception has occurred is calculated by querying the targets perceivedDeceitAbility of the opponent in their perceived player-model. Similar to GenerateStatement(), the DeceitAbility is divided by 10 to fetch the estimated real value of the card. However, the player does not know whether the real value of the card has been added to, or taken away from and therefore must guess. Once this number has been found, it is returned

as an integer. If the target believes the opponent, the statement made by the opponent is returned as the guessed card.

GetArgument()

GetArgument() is a method used by all players during the Deliberation phase of the game to calculate which argument they are going to make, and to whom. Similar to the target and intention discovery in the Gameplay phase, an intention is found and a target calculated based upon that intention. If the intention is to act truthfully, then a good argument is randomly selected from a list of 'good' arguments. Similarly, a 'bad' argument is randomly selected if the intention is to act deceitfully/maliciously towards another player.

```
string[] good_statements = { "NotDeceitful", "NotAggressive", "Trustful" };
string[] bad_statements = { "Deceitful", "Aggressive", "NotTrustful" };
```

From there, a complete argument is returned that includes the argument being made, the sender of the argument and its intended recepient.

Decay()

Decay() is used to act as a 'forgiveness' mechanic, and to prevent permanent grudges being held over the course of a long game. In this method, for all perceived player-models that the player has, each is incremented/decremented by ± 0.1 towards 50, depending on whether the current value is above or below this baseline.

5.2.5 PlayerListFunctions.cs

The PlayerListFunctions.cs class is used to perform operations on all players within the game, not just that of one particular player. The class is used as an overall player management module that helps keep the game running smoothly without any logic bugs such as players targeting other players that have no remaining health, for example.

getNewTraits()

getNewTraits() is a method that takes in an integer between 1 and 3 inclusively, **amt**, which represents how many traits a particular player is going to be assigned. The traits are broken into three different trait groups; trust_traits, deceit_traits and deceitAbility_traits:

```
static List<string> trust_traits = new List<string>()
{ "Trusting", "Untrusting", "Suspicious", "Unsuspicious" };

static List<string> deceit_traits = new List<string>()
{ "Deceitful", "Honest", "Calculating", "Fair" };

static List<string> deceitAbility_traits = new List<string>()
{ "Aggressive", "Passive", "Audacious", "Timid" };
```

To avoid situations where obviously conflicting traits are chosen, such as trusting and untrusting, only one trait from each group can be selected and assigned to a player. A random trait from a random traits list is selected and added to an overall list to return, provided a trait already in that trait group is not already present within the list of traits that is being returned. The final compilation of traits is collected, and returned as a list to be added to a player.

ResolveArguments()

ResolveArguments() takes a particular argument **a** and the list of all players, and makes relevant changes to all players' perceived player-model based on what the argument was, the effects of which can be seen in figure 3.8.

GetPlayerByID()

Fetches a player from the list of players by searching for a particular ID, passed in as a parameter.

RemoveDeadPlayers()

Loops through every player within the list of current players, and checks if their health is equal to 0 or below. If it is, they are removed from the list of players currently in the game and added to a list of dead players. The list of dead players is kept track of to calculate corresponding scores at the end of the game.

RemovePerceivedModels()

This method checks every player in the game's health, and removes every other player's perceived player-model of them if they have 0 or less health. This method is done using the .NET library LINQ.

5.2.6 PlayerModel.cs

PlayerModel.cs holds the attributes contained within the player model, as shown in figure 3.2. Listed within the PlayerModel class are the modifier values which are used to determine the numerical effect that a particular trait may have on the models.

Constructor

The constructor takes a player object as parameter input, and modifies each attribute with a random number between -20 and 20, before running each attributes Modifiers() method to apply traits to the model.

TrustModifiers() & DeceitModifiers() & DeceitAbilityModifiers()

Each trait has a numerical value assigned to it, as shown in figure 3.4. The Modifiers() methods check if a particular player has a trait, and if so, applies the relevant modifier based on whether it will result in a large/small increase/decrease to that attribute within the player model.

5.2.7 Results.cs

Results.cs is used to log the results of a particular game or set of games. Results.cs has the ability to keep track of which traits have scored highly, the scores of individual players and the average score that a particular trait has achieved. By using a new object, ResultRecord, keeping track and displaying results is simplified and allows for a higher-level analysis.

ResultRecord

The ResultRecord class is used to store the following variables:

```
public string trait;
public int score = 0;
public int amount = 0;
public double average;
```

The ResultRecord is used to store the overall performance of a particular trait within the game.

DisplayResults()

Each unique trait is assigned its own result record. At the end of the game, with each player assigned a score based on their finishing position, they are checked to see if they have a trait. If

they do, then the trait's unique result record is incremented by the score of the player, with the total amount of players that have that trait being kept track of to fetch an argument at the end. The results of the performance of each trait is then printed, with a trait and its corresponding total score, average score and number of players who had that trait.

5.2.8 Options.cs

Options.cs contains variables that are shared globally throughout the application. These variables are used to change parameters of the program to test different types of player, game and user-model values. Upon release of the project, this will be the only file that is recommended to be changed, as it will effect the output of the game without fundamentally changing the mechanics of the game, nor require any programming knowledge to do so.

5.3 Changes from Design

During the course of the implementation, many changes were made to the program that differed from the scope outlined in the background research. Primarily, a change regarding the inclusion of player strategies. Player strategies would allow for a particular action to be gauranteed to occur given the strategies preconditions were met. For example a strategy of "tit for tat", which was outlined in Axelrod's Tournament as a well-perform strategy, would always target the most recent player that attacked them in the previous turn.

However, during testing during the development of the program, it was quickly realised that the inclusion of strategies most times completely overrode the user-models included, which effectively resulted in a recreation of Axelrod's Tournament. In this, the results emerged were that of which strategies were most dominant in winning the game, moving away from the initial aim of discovering which personality traits and types performed best in a game involving social elements.

5.4 Limitations in Accuracy of Models

5.4.1 Attributes

Humans are complicated. Due to the complexity of the human brain, any system that attempts to emulate natural social behaviour will always fall short. As a result, one of the great chal-

lenges this project faced was the device a system/model that can accurately portray human behaviour without the requirement of supercomputer processing or a vast framework that would be beyond the scope of this project. The attributes used in the model (DeceitAbility, Deceitfulness, Trustfulness) were selected as they encompass every possible action a player can take, and are directly related to those actions. However, the decision to deceive someone or act in a trustworthy manner is a much more complicated decision than querying a single Deceitfulness value. A real human may choose to be deceptive towards a player to appease another, to show strength, they may be bored of the game and might want to make enemies to see themselves eliminated quickly, etc. Encapsulating every possible motive behind taking an action is impossible to determine for real-life situations amongst people, let alone AIs.

Striking a balance between attributes that might be relevant (such as Deceitfulness) and attributes that would not add to the results (keeping track of a player's boredom, in case they want to eliminated themselves) was challenging. Whilst many additional variables would have improved the overall realism and decision making of the AIs, it is unlikely to have contributed much to the results, as the increase of variables would further obfuscate conclusions as to what personality traits are best in social card-games.

5.4.2 Traits

Translating human characteristics into numerical values that an AI can understand is oversimplifying. Being deceitful in real life means more than your likelihood to make a particular action in a card game. It can affect your interactions with other players, your long term or short term goals in the game. A deceitful player will never deceive when it is the dominant strategy to not do so in reality, but the probabilistic method of determining intention will undoubtedly lead to instances of nonsensical actions.

With only twelve traits available, the depth and richness of the human mind when playing social card games is not captured. Traits which would have the same effect on the personal player-model attributes, such as Passive and Apathetic, are in fact very different outside of the models when the subtle nuances are taken into account. To create a trait system that would capture all possible traits a human could be described as, as well as assigning different numerical effects in the model to them such that all traits are unique, would prove impossible.

5.4.3 Variables

Scaling variables from 0-100 may result in not enough granularity within the model. While many people in the real-world may share similar levels of Deceitfulness, restricting the values in a 0-100 range implies there are only 101 (inclusive) different levels of Deceitfulness in people, which is unlikely. Additionally, when it comes to issues of trust, it is typically regarded that you either trust someone, or you do not, as a boolean interpretation. Additionally, if you trust most people, it does not necessarily mean you will trust everyone. Even the most truthworthy person can be persuaded to distrust someone. Research into generalised trust [14] has shown that when queried about general trust in *most people*, questions are raised in what you are trusting that person with, the state of the person in whom you are confiding trust into, as well as varying levels of trust.

Using a one-dimensional scaling for the attributes within the player-models may also be perceived as lacking. Trust in itself can be broken down into many constituent parts. For example, Niklas Luchmann argues in *Trust and Power* [15] that trust can be broken down into different dimensions distinct from each other; familiarity and trust. Luchmann argues that trust can also be further broken down into "bad trust and good trust", which effect their optimality of maximisation, rather than desirability. The depth that models of trust can be broken down into were beyond the scope of this project due to resource and time constraints, however it felt appropriate to include trustfulness, despite the limitations listed.

5.5 Balancing Limitations

For each action undertaken by an opponent, every player's perceived player-model (PPM) is updated to reflect the result of that action on their beliefs of that player. As each action is translated into a numerical change to a PPM, the resulting changes are influential to which players will be targeted and which moves will be made against them in both gameplay phases and deliberation phases, as well as being the primarily proponent for the overall outcome of the game and which player claims victory.

However, choosing which values are best appropriate to a corresponding action is difficult. As there exists no consensus of how impactful an argument that a player is not trustworthy on participant's PPMs, appropriate values would have to be chosen for them. For example, in the ResolveArguments() function in the PlayerListFunctions.cs class, all arguments are resolved as to having a ± 1 effect on PPMs. If this change is largely increased, it disadvantages perceived-deceitful players as the argumentation phase of the game will destroy their reputation within the game, making them a target for almost all player's in the next round. Similarly, having scores too low can result in the deliberation phase becoming meaningless, as all arguments made are negligibly impactful.

Similar issues can be seen with the values used in the Decay mechanic, as well as the values used to record impacts of making a deceitful move or a trustworthy move in the gameplay phase. As a result, the values chosen have been selected as it is felt that they will be able to give the most accurate realisation of perception impact of particular moves, however the values that can be selected are subjective to the implementer, and other developers creating an identical system would likely pick others.

5.6 Runtime

The program runs at an $O(n^2)$ time complexity. For every player in the game, each player has a perceived-player model for all players except themselves. This can be roughly modelled as $n(n-1)$ or $n^2 - n$, where n is the number of players in the game. Although many further operations are carried out in the program, the highest polynomial complexity is derived from perceived player-model operations. To demonstrate the change in run-time as different parameters increase, tests were run displaying the change by firstly increasing the number of players (shown in figure 4.2), and then increasing the player health at the start of a game (figure 4.3).

Each program was run on the following specifications:

Component	Specification
Operating System	Windows 10 Home
IDE	Visual Studio 2019
CPU	Intel Core i5-4690K @ 4.5Ghz
Memory	16GB DDR3 RAM @ 1600Mhz

Figure 5.1: Runtime environment specifications

Number of Players	2	4	8	16	32	64	128
Total Run-time (ms)	9	14	38	254	1201	6693	63285

Figure 5.2: Runtime with increasing number of players. Average run-time for 10 games. Player health starting at 20.

Starting Health	2	4	8	16	32	64	128
Total Run-time (ms)	41	99	193	234	364	512	641

Figure 5.3: Runtime with increasing player health at start of game. Average run-time for 10 games. Number of players is 50.

The number of players increasing is without a doubt the greatest cause of increased run-time, for each player added, the run-time increases exponentially, exploding to an intractable size as the number of player's rises about 1000. A more gradual, linear effect on run-time can be observed by increasing player health, with manageable times displayed.

5.7 Optimisation

5.7.1 Search Algorithm Optimisation

The majority of processing time is spent linearly searching through each player's list of perceived-player models in order to fetch the correct model to adjust variables for every action taken by an opponent, for all players. As the number of players increases, the number of searches required increases exponentially. As a result, small tweaks in peformance can have massive benefits in reducing overall run-time.

One of the main benefits of using a library-rich language such as C# is access to uniform query syntax's such as Language Integrated Query (LINQ). LINQ is a Microsoft .NET component that allows development to include native data querying capabilities to C#. In the function GetPerceivedPlayerModel, in Player.cs, a linear search was being performed, with an if statement to see if a match was found and returning it if so. Using LINQ, the same linear search is conducted, but do to LINQ's optimisations, dramatic overall runtime improvements were seen.

Parameter	Value
Player Health	20
Number of Players	50
Number of Games	10

Figure 5.4: Game Parameters

Method	Run-time
Old Method	5068ms
LINQ Find Method	3865ms
Overall Improvement	+24%

Figure 5.5: Average Run-times

5.8 Testing and Maintenance

Testing was carried out throughout the project, and at the time of submission, there exists no known bugs that can be found under heavy, light or regular workloads. To keep track of version control, bugfixing and issues, GitHub was used throughout the development of this program to ensure a clean development environment that could be easily refactored, tested and fixed when issues did arise. In particular, the use of issue tracking was of paramount importance for regular code reviews to ensure a high quality throughout.

However, ambiguous run-times can be observed with a very high number of players within the game. Due to the Halting Problem, it is impossible to determine whether or not a program has entered into an unintentional infinite loop, or if it is working normally but in intractable time. Whilst it is possible to print all current moves within the program at every step, there is too much information as the number of players climbs high to keep track of.

Chapter 6

Results/Evaluation

Here we will discuss the results of the experiments. In this section, we are looking for trends between certain traits and overall performance within the game. Different methods of evaluation will be used, such as assigning scores to players based on their final position in the game, in order to have a more abstract analysis of overall player behaviour.

Additionally, different parameter variations will be utilised to demonstrate how different traits perform in different games. For example, it could be reasonable to assume that in a shorter game, being aggressive is less punishing as there is less time for a coalition to form against you.

6.1 Evaluation Methods

For the different evaluation methods below, the results are taken from a cumulative 100 games played in a row, with each game starting with new players. All games are run with the same starting parameters that may effect the course of the game as follows:

6.1.1 Final Position Scoring

Final position scoring assigns points to a player based on the order in which they were eliminated in the game, with less points meaning a better performance. In an N-person game, when a player is eliminated they are assigned a score of N - α, where α is the number of players eliminated so far. For example, in a game with 10 players, the first player eliminated will be assigned a score of 10, the next player 9, the next 8 and so on.

Variable	Description	Value
Number of players	How many players start in the game	50
Player Health	Starting health of each player	20
Decay	How much decay affects PPMs per round	±0.1
Argument Resolution	How much weight arguments are assigned	±1
Trait smallModifier	Amount that a trait with a smallModifier affects the player-model	±15
Trait largeModifier	Amount that a trait with a largeModifier affects the player-model	±30
Personal Action Impact	Amount that your PPM of an opponent changes when you are targeted	4
Impersonal Action Impact	Amount that your PPM of an opponent changes when someone else is targeted	2

Figure 6.1: Base-line game parameters

Trait	Total Score	Players With Trait	Average Score
Untrusting	18758	806	23.27
Trusting	20037	852	23.52
Unsuspicious	19550	829	23.58
Suspicious	20373	847	24.05
Calculating	21357	851	25.10
Timid	21212	836	25.37
Fair	20958	812	25.81
Aggressive	22323	862	25.90
Passive	22953	879	26.11
Audacious	21432	813	26.36
Honest	22978	853	26.94
Deceitful	21947	814	26.96

Figure 6.2: Run 1: Final Position Scoring

An interesting takeaway from these results firstly is the order in which the traits appear when trait group is considered. Untrusting, trusting, unsuspicious and suspicious all appear in the top four traits, however all belong to the group of trust traits, of which only one can be selected and assigned to a player. From this, we can note that perhaps having a trait that effects your trust in other players is more important than whether or not you are trusting of others or untrusting. Additionally, of this subgroup of the top four traits, the top half are both traits which use the largeModifier value when traits are translated into the player-models, implying that is beneficial for a player to either be incredibly trusting, or very untrusting, with further deviation from the player-models base of 50 being rewarded in the game.

To confirm this, another game was run with the exact same parameters, except the trait

largeModifier and smallModifier for the trust traits group was changed to 80 for largeModifier, and 30 for smallModifier. In this experiment, the gulf between results was widened:

Trait	Average Score
Unsuspicious	16.05
Trusting	17.94
Passive	25.10
Fair	25.34
Timid	25.50
Calculating	25.57
Honest	25.61
Aggressive	25.72
Audacious	26.29
Deceitful	26.44
Suspicious	36.84
Untrusting	45.31

Figure 6.3: Run 2: Final Position Scoring

With the average scores of all other traits not in the trust trait group remaining fairly stable (Run 1 Average 26.04 vs 25.69 Run 2 Average), the range dramatically increases from 3.69 in Run 1 to 29.26. Interestingly, another trend seems to form at both the top and bottom of the table. Both unsuspicious and trusting are hugely beneficial, positive modifiers to a player's personal trust attributes, whereas suspicious and trusting traits have a negative modifier. High levels of personal trust effect areas of the game such as beliefs that you are being deceived when targeted by an opponent, and how big of an effect an opponent's deceitful actions have on one's perceived-player model of that opponent.

Outside of the trust trait group, deceitful, audacious and aggressive scored in the bottom half of the table for both runs. Both aggressive and audacious result in positive modifiers to a player's deceitfulness, which strongly implies that the more deceitful a player is, the less likely they are to succeed in the Number Guessing game. However, negative modifiers such as Fair and Honest sit around the middle of the scores, showing that whilst it may not pay to be deceitful, being especially honest does not reap many rewards either.

6.1.2 Winner Takes All Scoring

In a winner takes all game, a point is assigned to only the player that has emerged the victor. The traits which have a victorious player have a point assigned to it, resulting in the trait having the most points after 100 games being determined as the strongest.

Trait	Total Score
Timid	23
Unsuspicious	21
Calculating	19
Trusting	17
Fair	17
Aggressive	17
Suspicious	15
Audacious	15
Deceitful	14
Honest	14
Passive	14
Untrusting	13

Figure 6.4: Run 1: Winner Takes All Scoring

One notable takeaway from Run 1 with the Winner Takes all scoring is the distribution of traits with large modifiers and small modifiers to player's personal player-models. In the top half of results, 5 out of 6 of the traits are small modifier traits. That is, traits which use the smaller modifier when translated into numerical effects of the personal model. Curiously, the untrusting trait (which was in 1st place with an unmodified Final Position Scoring method), is now at the bottom of the table, showing that whilst being untrusting may improve chances of not finishing last, it can often hinder a player's chances of becoming the outright victor in the game.

By changing the trait's large and small modifier values to be closer together, we should see more variation between different personality types scoring highly, rather than results influenced by the modifiers that a trait has. With the smallModifier being changed for all traits to 25, and largeModifier kept at 30, different results were observed:

Trait	Total Score
Deceitful	25
Suspicious	24
Trusting	21
Audacious	20
Aggressive	19
Passive	19
Calculating	18
Unsuspicious	16
Timid	14
Fair	13
Untrusting	12
Honest	8

Figure 6.5: Run 2: Winner Takes All Scoring

Instead, with the modifiers for traits being adjusted to be closer together, it is the large modifier traits which are preferred. In Winner Takes All scoring with closer-together trait modifiers, there is a preference to a more ‘extreme‘ personal player-model, with potentially a preferable modifier space located between the previous small and large modifiers.

Another clear pattern emerges from the Run 2 results as well, when trait modifiers are brought to be more similar, there results in a clear polarisation in the top and bottom of the table with the deceitful and honest traits. Both effecting the same attribute in the personal player-model with large modifiers but in different directions, in Winner Takes All scoring it is clearly more beneficial to act deceitfully rather than honestly. Deceitful's almost 3.13x larger average score than honest is vastly different to their identical results in Run 1.

6.1.3 Survivorship Scoring

Not all social card games are about outright victory, however. For many, the objective of playing a game may not be to *win*, but rather to *not come last* in order to save face. To see if there are traits that may not results in outright victory, but that could leave a player in the game for as long as possible, a variable was added to a player to keep track of how many rounds they have been able to stay in the game for. For each trait, the total number of rounds that trait has survived for is added, and an average is displayed:

Trait	Total Score	Players With Trait	Average Score
Untrusting	282836	883	320.31
Trusting	261832	848	308.76
Suspicious	252988	861	293.83
Unsuspicious	238355	822	289.97
Audacious	233697	823	283.96
Calculating	245976	867	283.71
Fair	238992	846	282.50
Aggressive	228649	832	274.82
Passive	226459	831	272.51
Timid	243781	896	272.08
Honest	223646	861	259.75
Deceitful	201970	801	252.15

Figure 6.6: Run 1: Survivorship Scoring

Unsurprisingly, evaluation via survivorship displays similar results to Final Position Scoring, as the higher your position when you were eliminated, the longer you have stayed in the game for. However, players that displayed a low amount of deceit ability, through the passive and timid traits, performed poorly in terms of rounds survived before elimination.

6.2 Parameter Variation

Balancing the game around parameters, variables and formulas was always going to be a challenge in both the design, development and evaluation of this project. As the modelling of human traits and their effects on personalities within a social card game has not been explored much, many subjective decisions were made to simulate the weight of all parts of the game, as well as performing balance changes to ensure the game was reflective of a real-life game.

In order to soften this limitation, many different parameters were changed, and compared against the three evaluation methods of Final Position Scoring (FPS), Winner Takes All Scoring (WTAS) and Survivorship Scoring (SS). However, instead of comparing the scoring methods against each other based on the scores they produce, for legibility we will be converting their sorted scores into ranks (1-12) with 1 being the best. This allows for a much better comparison of traits and their evaluation methods on the changing parameters.

To establish a baseline to compare the results of the parameter variation against, we have translated the initial runs of each evaluation scoring method into a table below, with each trait's final position recorded as a rank.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	8	6	8	7.3
Audacious	10	8	5	7.6
Calculating	5	3	6	4.6
Deceitful	12	9	12	11
Fair	7	5	7	6.3
Honest	11	10	11	10.6
Passive	9	11	9	9.6
Suspicious	4	7	3	4.6
Timid	6	1	10	5.6
Trusting	2	4	2	2.6
Unsuspicious	3	2	4	3
Untrusting	1	12	1	4.6

Figure 6.7: Trait Evaluation For All Evaluation Methods

6.2.1 Single Trait

In reality, people have more than a single trait, however having multiple traits can often obfuscate the impact that a particular trait may be having on the course of the game. Some results, for example, may show great promise in terms of great synergy between traits, which may artificially inflate the impact that a trait may actually have. In order to explore this further, each player will have the number of traits they have reduced from a random amount from 1-3 to have only 1 exactly.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	2	3	9	4.6
Audacious	7	4	6	5.6
Calculating	8	5	5	6
Deceitful	12	2	12	8.6
Fair	6	11	10	9
Honest	11	1	11	7.6
Passive	9	6	8	7.6
Suspicious	1	9	2	4
Timid	10	7	7	8
Trusting	4	10	1	5
Unsuspicious	3	8	4	5
Untrusting	5	12	3	6.6

Figure 6.8: Single Trait Evaluation

These figures show a different story as to which traits are most beneficial to a player. One result that stands out is the performance of the honest trait. In Final Position Scoring (FPS) and Survivorship Scoring (SS), honest is at the bottom of the traits along with deceitful, both large modifiers of the deceitfulness player-model attribute. However, when it comes to pure number of wins, both are the top two. This demonstrates that large modifiers to the deceitfulness attribute can go one way or the other in terms of overall performance with the game in the sense that you will either win decisively, or be wiped out by the other players quickly. This also shows that the deceitfulness attribute is working as intended; it's a risk that may win you the game, but may also be the cause of your demise.

By this data, the most effective singular trait to have is the suspicious trait, which performs well in FPS and SS, but lags behind in Winner Takes All Scoring (WTAS), along with the aggressive trait, which scores in the middle of the rankings for multi-trait runs, but according to its average rank is the second-best trait to have singularly. This shows that, when more traits are introduced, aggression often syngergises poorly, resulting in a more average performance than just baseline aggression on its own.

6.2.2 Player Health

The amount of health that each player starts with also has a large influence on the game. On paper, a game that has players with less starting health should reward more aggressive behaviour, as opponents should become eliminated before they can realise how much of a threat a particular player is. Similarly, a game with more health should reward more trustworthy players overall. In these two experiments, we run the game with player health starting at 40, and player health starting at 10.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	9	7	10	8.6
Audacious	10	4	7	7
Calculating	8	8	5	7
Deceitful	11	5	11	9
Fair	5	11	6	7.3
Honest	12	3	12	9
Passive	6	2	9	6
Suspicious	3	10	3	5.3
Timid	7	12	8	9
Trusting	1	1	1	1
Unsuspicious	4	9	4	5.6
Untrusting	2	6	2	3.3

Figure 6.9: Run 1. Player Health Starting at 10.

Surprisingly, reducing player health results in the more aggressive traits (aggressive, audacious) performing similarly to baseline runs for all evaluation methods. However, whilst trusting performed typically well in FPS and SS scoring methods, it shot up to first place for WTAS. The implication the results show might be that lower player health at game starting does not favour aggression, but rather it favours impartiality. If you are more aggressive towards players, you will be seen as such and therefore targeted. However if you are trusting, then a player is less likely to target the greatest threats in the game, reducing the risk of them being targeted in response to their aggression. With less player health, opponents are knocked out of the game faster, increasing the likelihood that you will be targeted by the diminishing pool of remaining players if you were to attack large threats, rather than remaining impartial and attacking players of more variety.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	7	4	10	7
Audacious	6	6	9	7
Calculating	8	2	5	5
Deceitful	12	11	12	11.6
Fair	9	8	6	7.6
Honest	11	9	11	10.3
Passive	4	3	7	4.6
Suspicious	10	5	4	6.3
Timid	5	10	8	7.6
Trusting	2	7	1	3.3
Unsuspicious	3	1	3	2.3
Untrusting	1	12	2	5

Figure 6.10: Run 2. Player Health Starting at 40.

Confirming the theories gathered from run 1, being trusting and remaining impartial quickly loses its dominance when player health is increased, however overall there exists no strong patterns emerging. In the space of the game, it is likely that increasing the player health above the baseline of 20 may begin to have diminishing returns and a decreasing overall impact on the trends within the game as to which traits perform better. Whilst it is likely to increase the run-time of the game, and the number of rounds each game has, the length of a game with 40 health is similar to a game with 20 health in terms of gameplay between players and their personalities.

6.2.3 Number of Players

As discussed in the implementation, the number of players in a game can dramatically increase the run-time of the program. As a result, it becomes intractable to increase the number of players to values such that the random assignment of traits reaches an equal distribution due to the law of large numbers [16]. To test the effect of changing number of numbers, we are confined to a range well below desired. In these examples, player numbers shall we changed to 5 (to simulate a small, tight-knit group of friends playing together) and to 100. In these tests, it should be expected that more aggressive behaviour is more beneficial in the game with less players compared to a more 'keep your head down' approach in a larger game to avoid being targeted.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	6	4	5	5
Audacious	8	6	8	7.3
Calculating	10	5	10	8.3
Deceitful	11	12	11	11.3
Fair	5	3	9	5.6
Honest	12	9	12	11
Passive	7	1	8	5.3
Suspicious	4	8	4	5.3
Timid	9	10	7	8.6
Trusting	1	7	1	3
Unsuspicious	3	2	3	2.6
Untrusting	2	11	2	5

Figure 6.11: Run 1. Game with 100 players

A non-aggressive style of play is rewarded in a game with higher players. This is shown by the WTAS score of the passive trait. The direct opposite to aggressive, passive was able to claim the most victories in an average of 100 games with 100 players (10,000 total). Compared

to the average rank of the baseline evaluation scoring runs, passive's 5.3 average is much higher than the previous 9.6. This strongly confirms the previous assumption that higher players incentivises a less aggressive personality.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	4	7	10	7
Audacious	6	8	4	6
Calculating	9	12	7	9.3
Deceitful	3	2	6	3.6
Fair	11	5	9	8.3
Honest	12	11	12	11.6
Passive	1	1	11	4.3
Suspicious	5	4	3	4
Timid	8	6	5	6.3
Trusting	7	9	2	6
Unsuspicious	10	10	8	9.3
Untrusting	2	3	1	2

Figure 6.12: Run 2. Game with 5 players

In a smaller game, aggression and deceitfulness is rewarded. Where the deceitful trait previously had an average rank of 11, in a 5-player game being deceitful now has an average rank of 3.6. This large improvement is most likely due to the amount of time it takes for players to build an accurate perceived player-model of each of their opponents. By the time they have realised that a player is particularly deceitful, and therefore a threat, they have themselves already been eliminated by said deceitful opponent. To counteract this, traits which compound the negative behaviours of opponents and allow for a faster build-up of their perceived player-model remain strong. Untrusting, for example, sees an improvement of its previous average ranking of 4.6 improve to an average rank of 2. This, when compared with a previously very strong trait in unsuspicious (average rank 3) and trusting (average rank 2.6) fall down the table to 9.3 and 6 respectively.

6.2.4 Weight of Argumentation

The weight of arguments is used to determine the effects that an opponent's argument has against another player. At the moment, each argument about a target effects all other player's perceived player-models of that target by ± 1. In these parameter variations, we explore changing argument weights to ± 0.1 and ± 5.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	10	12	10	10.6
Audacious	7	6	7	6.6
Calculating	6	2	5	4.3
Deceitful	12	9	12	11
Fair	8	8	8	8
Honest	11	5	11	9
Passive	9	10	9	9.3
Suspicious	3	4	3	3.3
Timid	5	1	4	3.3
Trusting	2	3	1	2
Unsuspicious	4	7	6	5.6
Untrusting	1	11	2	4.6

Figure 6.13: Run 1. Argument Weight of 0.1

Overall, results are largely inconclusive. The similarity to the baseline results of figure 5.7 to run 1 show that perhaps the baseline values of ± 1 are already too weak. If the argument weight is already negligible, then reducing it further only maintains the argument weight effect of negligability. Whilst some traits perform slightly better/worse in some evaluation methods, it is likely that these differences are likely due to the randomness of trait assignment at player creation. Repeating this experiment leads to similar outliers, but no overall patterns emerge.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	9	9	10	9.3
Audacious	7	1	7	5
Calculating	10	12	8	10
Deceitful	11	7	11	9.6
Fair	5	2	5	4
Honest	12	8	12	10.6
Passive	8	11	9	9.3
Suspicious	1	3	3	2.3
Timid	2	5	4	3.6
Trusting	4	4	2	3.3
Unsuspicious	6	6	6	6
Untrusting	3	10	1	4.6

Figure 6.14: Run 1. Argument Weight of 5

Higher argument weight penalised all trust-based traits. Trusting, untrusting, suspicious and unsuspicious all fell lower down in the table. This is likely due to the argument weight being so large, the extent at which a player believes it makes no difference as the effect on their perceived player-model is already big. As a result, whilst no patterns emerge amongst the other trait groups, there is an overall performance increase, likely to fill the vacuum left by the trust-based traits.

6.2.5 Weight of Actions

When a player performs an action each player, targeted or not, has their perception of their opponent changed based on the action they undertook. When a player is deceived for example, their perceived player-model's deceitfulness of the player targeting them is increased by 4. This is their personal action weight. Other players observing this deception take place will have their perceived player-model's deceitfulness changed by 2. This is their impersonal action weight. By changing the weights of actions on the players, patterns may emerge that further punish very deceitful players and reward very trustworthy, kind ones.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	10	1	9	6.6
Audacious	9	3	8	6.6
Calculating	5	5	5	5
Deceitful	11	11	11	11
Fair	7	9	7	7.6
Honest	12	12	12	12
Passive	8	6	10	8
Suspicious	3	7	3	4.3
Timid	6	10	6	7.3
Trusting	2	8	2	4
Unsuspicious	4	4	4	4
Untrusting	1	2	1	1.3

Figure 6.15: Run 1. Personal Action Weight 2, Impersonal Action Weight 1

With action weights reduced, aggressive shows an interesting improvement in Winner Takes All Scoring. In the baseline runs, aggressive scored rank 6 for WTAS, whereas now it is top of the pack of traits. However, aggressive does not see significant improvements in FPS or SS scoring. The conclusion drawn from these results however is a much higher risk:reward for the aggressive and audacious traits. With these parameters, very aggressive players are able to seize victory much more often, but often find themselves knocked out first as well. The untrusting trait also sees a large ranking increase in WTAS scoring, moving from position 12 in baseline tests to position 2 in Run 1. Whilst it may be difficult to hypothesise why these changes are occurring, it may be because if aggressive players are more powerful with a lesser argumentation weight, then being untrusting and not seeing their actions as threatening will result in one targeting them less, therefore making the player less of a target in return.

Trait	FPS	WTAS	SS	Average Rank
Aggressive	8	6	10	8
Audacious	5	11	4	6.6
Calculating	12	9	7	9.3
Deceitful	11	8	12	10.3
Fair	7	3	8	6
Honest	10	2	11	7.6
Passive	9	1	9	6.3
Suspicious	1	5	2	2.6
Timid	6	10	6	7.3
Trusting	4	4	1	3
Unsuspicious	2	12	5	6.3
Untrusting	3	7	3	4.3

Figure 6.16: Run 2. Personal Action Weight 8, Impersonal Action Weight 4

As expected, aggressive, deceitful players are no longer rewarded in Run 2 as they were in Run 1. With a larger action weight, it is instead the 'kinder' traits, such as passive, fair and honest that see significant improvements to overall victories by WTAS scoring. With mucher higher scrutiny from other players that will react to your actions, it would be expected that passive, for example, would also survive longer and have a better final position in terms of ranking. However, similar to Run 1, it appears that being passive can either make a player largely ignored until the moment they clinch victory, or seen as too meek to perform well and is quickly wiped out by the more deceitful players, as shown by its FPS and SS scores.

6.3 Overall

Overall, the results display the most important aspect of succeeding in a social card game is traits affecting trust in others. The traits listed within the trust traits group (trusting, untrusting, suspicious, unsuspicious) were most often the highest performance traits across both FPS and SS evaluation methods, with an average rank across all parameter variation experiments between the four traits of 4.16, firmly within the top half of trait performance. However, it should be noted that while the trust traits thrived in FPS and SS, it frequently scored much lower in WTAS. Trust traits therefore give great strength to overall performance in the game, but as it does not affect actions undertaken to your opponents, it rarely makes the difference in terms of pure number of victories.

Depending on the game rules and parameters set, pure number of games one was most often achieved by the deceit ability trait group traits, most notably passive and aggressive. For

example, out of the ten different parameter variation runs, the passive trait achieved a top 3 rank score for WTAS in 50% of all runs. This is similar to aggressive, achieving a top four rank in in 40% of runs. This disparity between aggressive and passive indicates an overall favouring towards a less deceitful (by extent of deception) type of player.

Finally, the deceit traits (deceitful, honest, calculating, fair) showed an overall preference to deceitful play. With the average rank of the honest trait across all parameter variation runs being 8.9, and the fair trait scoring a middling 6.3, the implication is that the more honest a player is, the worse overall performance is. The trend is similar towards more aggressive styles of play, with deceitful scoring an average rank of 8.3 across all runs, compared to audacious with 5.77. The overall takeaway from these findings are that it is better to be as close to neutral as possible, neither deceitful nor honest. However, if a player is to be non-neutral, then it is better to be slightly deceitful, with greater punishment to the player dealt the further either side from neutrality they go.

6.4 Improvements

Improvements to the overall quality of the results could be improved firstly if better optimisation of the program was carried out. With greater optimisation, the less run-time which could result in a higher number of players. Due to the random assignment of traits to players, as well as the random buffers added to each personal player-models attributes, a higher number of players would allow for all combinations of models to be created, improving the results dramatically.

Additionally, improvements to the mechanics of social interaction between players could be improved, but were not added due to resource constraints within the project. Mechanics such as changes in targeting, which for example could see strong players attacking weaker players, would add to the overall realism and emulation of a real-life social card game. Additionally, seeing features such as stronger, more hard-coded coalitions forming could see aggressive, deceitful players teaming up together. At the moment, aggressive players will often see other similar play-style opponents as threats, rather than potential allies due to a shared ideology in the way they play.

Finally, more tests could be brought in to improve the overall quality of data. Additional evaluation scoring methods, for example, would be highly beneficial to get an insight into the more technical, subtle details of how a particular trait interacts with the overall performance of a player within a game. Different match structures could also be implemented to see the effects of traits over a longer period of time, with instead of new players created each game, the same players from previous games could begin, foregoing the initial period in the game where players are building their perceived player-models of each other. This would allow for, firstly, the simulation of many real-life card games where the same group of friends play each other each time, meaning that actions from previous games would affect their interactions. This change would also allow for long-term trait analysis and how they effect performance.

Chapter 7

Legal, Social, Ethical and Professional Issues

7.1 Representing Neurodiversity

People are more complicated than three variables, and the simplification of the human personality is representative of neurotypical individuals, which may fail to correctly encapsulate neurodiversity of a person, for example, who may be on the autistic spectrum. Creating an abstract model of human decision-making that accounts for people from all backgrounds is a challenge, and something that I would love to expand further on to create the most inclusive project possible.

7.2 British Computing Society Code of Conduct

The British Computing Society's (BCS) Code of Conduct and Code of Good Practise were considered throughout the development of this project. Whilst many sections were not relevant, care has been taken to ensure that our results are accurate, true and development was conducted in a way such that the professional and academic integrity were maintained throughout.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The empirical evaluation of personality types within a social card game using Artificial Intelligence produces a solid framework for further testing. Whilst the complexities of real-life social interaction are difficult to completely encapsulate, we have created a framework that displays player behaviour consistent with a realistic social card game.

The results accurately portray the challenge in social card games of having a balanced playstyle whilst maximising likelihood of victory. From testing, a deviation from neutral baseline values is preferred, with more risk being undertaken with stronger traits to improve chance of winning a game. Whilst and overall trend to deceitful play was observed, a more important factor may be trait synergy, with different traits complimenting each other. The most important traits for a player to have syngergised with in a social card game are traits that affect one's trust in others. Believing/Not believing arguments made by others and actions undertaken by opponents provide a critical change in style of play that can massively benefit and improve player's performance.

Additionally, different trait synergies performed better/worse depending on the nature of the game. When the parameters are tweaked to have a shorter game, such a less players or less player starting health, more aggressive styles of play were favoured. The contrary is also true, that longer games reward players who are less aggressive and make efforts to deem themselves as less of a threat to others.

Whilst not based on a real, established card game, the Number Guessing game and its simplicity lends itself well to creating a basic framework to analyse trait performance using user models and argumentation. Deliberation, bluffing, deceit and makeshift, informal coalitions are implemented and have a noticeable effect on the flow of the game and the way players interact, which successfully partially negates the difficulties that AIs have with quantifying social attributes.

8.2 Future Work

In the future, I would love to be able to firstly apply this framework to a real-world game like Hanabi, Werewolf, etc. The framework at the moment is a proof-of-concept and being able to verify it against a widely played game would prove interesting. To take the framework to other games may display previously unthought strategies and styles of play, similar to how DeepBlue changed the strategum of how we play Chess.

Optimisation is another area I would like to focus on. By pushing the source code to the limit, including threading and refactoring would allow for more players, more traits, and ultimately more mechanics. The additional features that could be added would massively improve the overall quality of the results.

Finally, mechanical improvements are something that could be added. Different ways that players can target each other, deliberate with one another and form teams/coalitions would add great complexity to the project, but would hopefully produce even more meaningful and realistic results.

References

[1] Leon Ciechanowski, Aleksandra Przegalinska, Mikolaj Magnuski, and Peter Gloor. In the Shades of the Uncanny Valley: An Experimental Study of Human-Chatbot Interaction. 2019. MIT Center for Collective Intelligence.

[2] Allied Market Research. Artificial Intelligence (AI) Market by Technology, 2018. URL `https://bit.ly/3iQCPIc`. Last Accessed 17th August 2021.

[3] DeepMind. AlphaStar: Grandmaster Level in StarCraft II using multi-agent reinforcement learning. 2019. URL `https://bit.ly/3g94ylJ`. Last Accessed 17th August 2021.

[4] Starcraft II World Championship, 2017. URL `https://pro.eslgaming.com/tour/sc2/`. Last Accessed 17th August 2021.

[5] Michael Cale. OG Make History by Beating Team Liquid and Winning the International 2019, 2019. URL `https://bit.ly/37Mr7rP`. Last Accessed 17th August 2021.

[6] P. M Dung. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-Person Games. 1995.

[7] Robert Axelrod. Effective Choice in the Prisoner's Dilemma. 1980.

[8] Cristiano Castelfranchi. Modelling social action for ai agents. 1998.

[9] Daniel Rousseau and Barbara Hayes-Roth. Personality in Synthetic Agents. 1996.

[10] Joel Nicklaus, Michele Alberti, Rolf Ingold, Markus Stolze, and Thomas Koller. Challenging Human Supremacy: Evaluating Monte Carlo Tree Search and Deep Learning for the Trick Taking Card Game Jass. 2020. URL `http://aaai-rlg.mlanctot.info/papers/AAAI20-RLG_paper_21.pdf`.

[11] Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dun-

ning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. The Hababi Challenge: A new frontier for AI resarch. 2020. URL https://bit.ly/3g9K5w0.

[12] T.J.M Bench-Capon and Paul E. Dunne. Argumentation in artificial intelligence. 2007. URL https://www.sciencedirect.com/science/article/pii/S0004370207000793. University of Liverpool.

[13] Filipa Correia, Particia Alves-Oliveira, Nuno Maia, Tiago Ribeiro, Sofia Petisca, Francisco S. Melo, and Ana Paiva. Just follow the suit! Trust in Human-Robot Interactions during Card Game Playing. 2016. 25th IEEE International Symposium of Robot and Human Interactive Communication.

[14] Jan Delhey, Kenneth Newton, and Christian Welzel. How General Is Trust in 'Most People'? Solving the Radius of Trust Problem. 2011. American Sociological Association - SAGE.

[15] Nilas Luchmann. *Trust And Power (Vertrauen)*. 1973. ISBN 978-1-5095-1945-3.

[16] Michiel Hazewinkel. *Encyclopedia of Mathematics*. 1994. ISBN 1-55608-010-7. Kluwer Academic Publishers.

Appendix A

User Guide

A.1 Instructions

To get the software, you can fetch it from the GitHub repository located at:

```
https://github.com/monerofglory/MastersThesis
```

To run the program you will need a C# compiler of at least version C# 3.0, although as up-to-date as possible is recommended. After building the source code, it will be available to run.

Running the software is simple. As it is a simulation, the only parts that require amendment are all conveniently located in the Options.cs file. Changing the values of the variables within will tweak the program to run differently. Outputted to a console are all three evaluation methods, sorted by ranking along with additional information as to how the values reached have been calculated. For example, to change the number of players within a game change line 9 of Options.cs from:

```
public static int numberOfPlayers = 50;
```

to:

```
public static int numberOfPlayers = FOO;
```

Where FOO is the number of players you would like in the game. Rebuild the source code and re-run the project to see the results with the new parameters.

Appendix B

Source Code

B.1 Contents

1. Argument.cs
2. Game.cs
3. Options.cs
4. PerceivedPlayerModel.cs
5. Player.cs
6. PlayerListFunctions.cs
7. PlayerModel.cs
8. Results.cs

```
namespace MastersThesis
{
    //Class for player arguments
    class Argument
    {
        public string statement;
        public Player sender;
        public Player receiver;

        public Argument(string stat, Player p1, Player p2)
        {
            statement = stat;
            sender = p1;
            receiver = p2;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

namespace MastersThesis
{
    class Game
    {
        static Stopwatch watch = new System.Diagnostics.Stopwatch();

        public static List<Player> players = new List<Player>();
        public static List<String> winningTraits = new List<String>();
        public static List<Player> deadPlayers = new List<Player>();
        public static List<Player> allDeadPlayers = new List<Player>();
        private static Random rd = new Random();
        //Game details
        public static int gameLength = 0;
        public static int numberOfPlayers = Options.numberOfPlayers;

        public static int consecNoChanges = 0;
        static void Main(string[] args)
        {
            watch.Start();
            for (int j = 0; j < 100; j++) //Loop for running a new game
            {
                //Clear variables for new game start
                gameLength = 0;
                deadPlayers.Clear();
                players.Clear();
                //Initialising players
                for (int i = 0; i < numberOfPlayers; i++)
                {
                    players.Add(new Player(i, PlayerListFunctions.getNewTraits(rd.Next(1,
1))));
                }
                for (int i = 0; i < numberOfPlayers; i++)
                {
                    players[i].AddPerceivedPlayerModels(players);

                }
                GameLoop();
                winningTraits.AddRange(players[0].traits);
                allDeadPlayers.AddRange(deadPlayers);
            }
            //Outputting results of ALL games that have happened.
            Console.WriteLine("ALL GAMES RESULTS");
            Results.DisplayResults_FinalPosition(allDeadPlayers);
            Console.WriteLine();
            Results.DisplayResults_WinnerTakesAll(winningTraits);
            Console.WriteLine();
            Results.DisplayResults_Survivor(allDeadPlayers);
            watch.Stop();
            Console.WriteLine($"Execution Time: {watch.ElapsedMilliseconds} ms");
        }

        static void GameLoop()
        {
            while (players.Count > 1) //Whilst players are still in the game
            {

                gameLength++;
                //GAMEPLAY PHASE
                GameplayPhase();
                int deadPlayerCount = deadPlayers.Count;
```

```
                //Removing perceivedModels that are out
                PlayerListFunctions.RemoveDeadPlayers(players);
                if (deadPlayers.Count == deadPlayerCount) //If no players have been
eliminated this round.
                {
                    consecNoChanges++;
                }
                else
                {
                    consecNoChanges = 0;
                }
                //DELIBERATION PHASE
                if (players.Count > 1)
                {
                    DeliberationPhase();
                }
                //Decay each players perceived player models
                foreach (Player p in players)
                {
                    p.Decay();
                    p.roundsSurvived++;
                }
            }
            players[0].score = 1;
            deadPlayers.Add(players[0]); //Adding the winning player to the dead players.
        }

        //Deliberation phase where arguments are created and resolved.
        static void DeliberationPhase()
        {
            List<Argument> arguments = new List<Argument>();
            foreach (Player p in players)
            {
                arguments.Add(p.GetArgument(p, players)); //Fetch arguments and add them
to list of arguments.
            }
            foreach (Argument a in arguments) //Loop through all the arguments and resolve
them (add to ppms).
            {
                PlayerListFunctions.ResolveArguments(a, players);
            }
        }
        static void GameplayPhase()
        {
            int startingPlayer = rd.Next(0, players.Count);
            int playersDone = 0;
            while (playersDone < players.Count)
            {
                Player p = players[startingPlayer];
                if (p.health > 0) //Check the player isnt out
                {
                    Random rd = new Random();
                    int score = players.Count;
                    bool intentionToDeceive = p.GetIntention(consecNoChanges,
players.Count);
                    //Get the target ID with the highest perceived threat
                    int targetID;
                    if (intentionToDeceive)
                    {
                        targetID = p.GetHighestThreat(players);
                    }
                    else
                    {
                        targetID = p.GetHighestTrust(players);
                    }
```

```
                    //Fetch that player object as target
                    Player target = PlayerListFunctions.GetPlayerByID(targetID, players);
                    //Get the card the player is gonna say
                    int card = p.GetCard();
                    //Get the statement about the card (truth or lie)
                    string statement = p.GenerateStatement(card, intentionToDeceive);
                    //Target makes guess about card
                    int guess = target.GuessCard(statement, p);
                    if (guess == card) //If correct, both gain 1 health and add 1 trust to
the model.
                    {
                        p.health++;
                        target.health++;

target.GetPerceivedPlayerModel(p).AddDeceitfulness(target.playerModel,
Options.personalActionImpact * -1);
                        foreach (Player p3 in players)
                        {
                            if (p3.playerID != p.playerID)
                            {

p3.GetPerceivedPlayerModel(p).AddDeceitfulness(p3.playerModel,
Options.personalActionImpact * -1);
                            }
                        }
                    }
                    else //If incorrect
                    {
                        //Lose health based on difference
                        int diff = Math.Abs(card - guess);
                        target.health -= diff;
                        //If out of game, assign score
                        if (target.health <= 0)
                        {
                            target.score = score;
                            score--;
                        }
                        //If the opponent DID NOT lie
                        if (guess == Convert.ToInt32(statement))
                        {
                            target.GetPerceivedPlayerModel(p).goodTimes++;

target.GetPerceivedPlayerModel(p).AddDeceitfulness(target.playerModel,
Options.personalActionImpact * -1);
                            foreach (Player p2 in players)
                            {
                                if (p2.playerID != p.playerID)
                                {

p2.GetPerceivedPlayerModel(p).AddDeceitfulness(p2.playerModel,
Options.impersonalActionImpact * -1);
                                }
                            }

                        }
                        else //If opponent DID lie
                        {
                            target.GetPerceivedPlayerModel(p).badTimes++;

target.GetPerceivedPlayerModel(p).AddDeceitfulness(target.playerModel,
Options.personalActionImpact);

target.GetPerceivedPlayerModel(p).AddDeceitAbility(target.playerModel, diff);
                            foreach (Player p2 in players)
                            {
```

```
                                if (p2.playerID != p.playerID)
                                {

p2.GetPerceivedPlayerModel(p).AddDeceitfulness(p2.playerModel,
Options.impersonalActionImpact);

p2.GetPerceivedPlayerModel(p).AddDeceitAbility(p2.playerModel, diff / 2);
                                }
                            }
                        }
                    }
                    //If the player believed the statement
                    if (guess == Convert.ToInt32(statement))
                    {
                        foreach (Player p4 in players)
                        {
                            if (p4.playerID != p.playerID)
                            {
                                p4.GetPerceivedPlayerModel(p).AddTrust(p4.playerModel,
Options.impersonalActionImpact);
                            }
                        }
                    }
                    else
                    {
                        foreach (Player p4 in players)
                        {
                            if (p4.playerID != p.playerID)
                            {
                                p4.GetPerceivedPlayerModel(p).AddTrust(p4.playerModel,
Options.impersonalActionImpact * -1);
                            }
                        }
                    }
                }
                if (startingPlayer == players.Count - 1)
                {
                    startingPlayer = 0;
                }
                else
                {
                    startingPlayer++;
                }
                playersDone++;

            }
        }
    }
}
```

```
namespace MastersThesis
{
    class Options
    {
        public static int numberOfPlayers = 50;
        public static int startingPlayerHealth = 20;
        public static double decayAmount = 0.1;
        public static int argumentWeight = 1;
        public static int traitSmallModifier = 15;
        public static int traitLargeModifier = 30;
        public static int personalActionImpact = 4;
        public static int impersonalActionImpact = 2;
    }
}
```

```
using System;
using System.Collections.Generic;

namespace MastersThesis
{
    class PerceivedPlayerModel
    {
        public int playerID;
        public double perceivedTrustfullness = 50;
        public double perceivedDeceitfulness = 50;
        public double perceivedDeceitAbility = 50;
        //Number of times this player has acted deceitfully or kindly towards you.
        public int goodTimes = 0;
        public int badTimes = 0;

        public PerceivedPlayerModel(int id)
        {
            playerID = id;
            Random rd = new Random();
            perceivedTrustfullness += rd.Next(-10, 10);
            perceivedDeceitfulness += rd.Next(-10, 10);
            perceivedDeceitAbility += rd.Next(-10, 10);

        }
        public double GetThreat(List<Player> players)
        {
            double threat = perceivedDeceitfulness + perceivedDeceitAbility +
PlayerListFunctions.GetPlayerByID(playerID, players).health + (badTimes - goodTimes);
            return threat;
        }

        public double GetTrust(List<Player> players)
        {
            double trust = perceivedTrustfullness -
PlayerListFunctions.GetPlayerByID(playerID, players).health + (goodTimes - badTimes);
            return trust;
        }
        public void AddDeceitfulness(PlayerModel pm, double amt)
        {
            perceivedDeceitfulness += amt * (1 - (pm.trust / 100));
            if (perceivedDeceitfulness > 100)
            {
                perceivedDeceitfulness = 100;
            }
            if (perceivedDeceitfulness < 0)
            {
                perceivedDeceitfulness = 0;
            }
        }

        public void AddDeceitAbility(PlayerModel pm, double amt)
        {
            perceivedDeceitAbility += amt * (1 - (pm.trust / 100));
            if (perceivedDeceitAbility > 100)
            {
                perceivedDeceitAbility = 100;
            }
            if (perceivedDeceitAbility < 0)
            {
                perceivedDeceitAbility = 0;
            }
        }
        public void AddTrust(PlayerModel pm, double amt)
        {
```

```
            perceivedTrustfullness += amt * (1 + (pm.trust / 100));
            if (perceivedTrustfullness > 100)
            {
                perceivedTrustfullness = 100;
            }
            if (perceivedTrustfullness < 0)
            {
                perceivedTrustfullness = 0;
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;

namespace MastersThesis
{
    class Player
    {
        public int playerID;
        public int health;
        public PlayerModel playerModel;
        public List<PerceivedPlayerModel> perceivedPlayerModels = new
List<PerceivedPlayerModel>();
        public List<String> traits = new List<string>();
        public int score = 0;
        public int roundsSurvived = 0;

        private Random rd = new Random();

        //Player constructor
        public Player(int id, List<string> new_traits)
        {
            health = 20;
            playerID = id;
            traits.AddRange(new_traits);
            playerModel = new PlayerModel(this);
        }

        //Add perceived model of each player
        public void AddPerceivedPlayerModels(List<Player> players)
        {
            foreach (Player p in players)
            {
                if (p != this) //Make sure a player doesnt add themselves
                {
                    perceivedPlayerModels.Add(new PerceivedPlayerModel(p.playerID));
                }
            }
        }

        //Fetch a perceived player model of a particular player P
        public PerceivedPlayerModel GetPerceivedPlayerModel(Player p)
        {
            return perceivedPlayerModels.Find(o => o.playerID == p.playerID);
        }

        //Get the intention
        //Intention is whether a player is going to deceive or act truthfully.
        public bool GetIntention(int consecNoChanges, int remainingPlayers)
        {
            if (remainingPlayers == 2) { return true; } //Always deceive if two players
remain.
            int intention = rd.Next(0, Convert.ToInt32(playerModel.trust +
playerModel.deceitfulness + consecNoChanges));
            if (intention <= playerModel.trust)
            {
                return false; //Do NOT deceive
            }
            else
            {
                return true; //Deceive
            }
        }

        //Get the player out of all perceived models whom you trust the most
```

```
        public int GetHighestTrust(List<Player> players)
        {
            double highest = -9999999;
            int highest_player = -1;
            foreach (PerceivedPlayerModel ppm in perceivedPlayerModels)
            {

                if (PlayerListFunctions.GetPlayerByID(ppm.playerID, players).health >= 1)
//Make sure the player is still alive.
                {
                    double trust = ppm.GetTrust(players); //Get the trust calculation
                    if (trust > highest)
                    {
                        highest = trust;
                        highest_player = ppm.playerID;
                    }
                }
            }
            return highest_player; //Return highest trust.
        }

        //Get the player with the highest threat that is still alive.
        public int GetHighestThreat(List<Player> players)
        {
            double highest = -9999999;
            int highest_player = -1;
            foreach (PerceivedPlayerModel ppm in perceivedPlayerModels)
            {
                if (PlayerListFunctions.GetPlayerByID(ppm.playerID, players).health >= 1)
                {
                    double threat = ppm.GetThreat(players);
                    if (threat > highest)
                    {
                        highest = threat;
                        highest_player = ppm.playerID;
                    }
                }
            }
            return highest_player;
        }

        //Get the card that you're going to say.
        public int GetCard()
        {
            return rd.Next(0, 9);
        }

        //Generate a statement based on how deceitful you are, and by how much.
        public string GenerateStatement(int card, bool intentToDeceive)
        {
            int new_card = card;
            if (intentToDeceive) //Check how deceitful this player is
            {
                //I am going to deceive
                //Find out by HOW MUCH by
                //Convert deceitAbility / 10, rounded. E.g deceitAbility 68 => 6.8 => 7 =>
random number between -7 and 7
                int bound = Convert.ToInt32(Math.Round(playerModel.deceitAbility / 10));
                //Fetch new deceit card
                new_card += rd.Next(-1 * bound, bound);
                if (new_card > 9) { new_card = 9; } //If greater than 9, set to 9
                else if (new_card < 1) { new_card = 1; } //If <1 set to 1
                return (new_card).ToString();
            }
            else
```

```
            {
                //I am going to tell the truth
                return new_card.ToString();
            }
        }

        //Guess the card based on the statement made by the opponent, along with their
perceived model.
        public int GuessCard(string statement, Player p)
        {
            //Fetch perceivedDeceit of opponent
            PerceivedPlayerModel ppm = GetPerceivedPlayerModel(p);
            double upperBound = ppm.perceivedDeceitfulness + p.playerModel.trust;
            double lowerBound = ppm.perceivedDeceitfulness - p.playerModel.trust;
            if (rd.Next(0, Convert.ToInt32(upperBound)) <= lowerBound)
            {
                //I believe them to be deceiving me
                //bound is how much they are deceiving me by
                int bound = Convert.ToInt32(Math.Round(ppm.perceivedDeceitAbility / 10));
                int guessed_card = Convert.ToInt32(statement);
                while (guessed_card == Convert.ToInt32(statement)) //If deceived, dont
guess card made in statement.
                {
                    int skew = rd.Next(-1 * bound, bound);
                    guessed_card = guessed_card + skew;
                    //Check to break infinite loop]
                    if ((bound == 1) && (guessed_card == 1))
                    {
                        return 2;
                    }
                    if (bound == 0) { return guessed_card; }
                    if (guessed_card > 9) { guessed_card = 9; }
                    else if (guessed_card < 1) { guessed_card = 1; }
                }
                return guessed_card;
            }
            //I think they are telling the truth
            return Convert.ToInt32(statement);
        }

        //Get an argument from a particular player
        public Argument GetArgument(Player p, List<Player> players)
        {
            bool intention = p.GetIntention(Game.consecNoChanges, players.Count);
            Player target = PlayerListFunctions.GetPlayerByID(p.GetHighestThreat(players),
players);
            //Get statement
            string[] good_statements = { "NotDeceitful", "NotAggressive", "Trustful" };
            string[] bad_statements = { "Deceitful", "Aggressive", "NotTrustful" };
            Argument a;
            if (intention)
            {
                a = new Argument(good_statements[rd.Next(good_statements.Length)], p,
target);
            }
            else
            {
                a = new Argument(bad_statements[rd.Next(bad_statements.Length)], p,
target);
            }
            return a;
        }

        //Decay is the natural tendency over time to 'forgive' players for their
transgressions.
```

```
        //Perceived player models tend towards 50 (the middle) over time.
        public void Decay()
        {
            foreach (PerceivedPlayerModel ppm in perceivedPlayerModels)
            {
                //Decay for perceivedTrustfulness
                if (ppm.perceivedTrustfullness > 50) { ppm.perceivedTrustfullness -=
Options.decayAmount; }
                else if (ppm.perceivedTrustfullness < 50) { ppm.perceivedTrustfullness +=
Options.decayAmount; }
                //Decay for perceivedDeceit
                if (ppm.perceivedDeceitfulness > 50) { ppm.perceivedDeceitfulness -=
Options.decayAmount; }
                else if (ppm.perceivedDeceitfulness < 50) { ppm.perceivedDeceitfulness +=
Options.decayAmount; }
                //Decay for perceivedDeceitAbility
                if (ppm.perceivedDeceitAbility > 50) { ppm.perceivedDeceitAbility -=
Options.decayAmount; }
                else if (ppm.perceivedDeceitAbility < 50) { ppm.perceivedDeceitAbility +=
Options.decayAmount; }
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;

namespace MastersThesis
{
    class PlayerListFunctions
    {

        public static List<List<string>> traitsList = new List<List<string>>();
        //Trait groups
        //Trust trait group
        static List<string> trust_traits = new List<string>() { "Trusting", "Untrusting",
"Unsuspicious", "Suspicious" };
        //Deceit trait group
        static List<string> deceit_traits = new List<string>() { "Deceitful", "Honest",
"Calculating", "Fair" };
        //Deceit ability trait group
        static List<string> deceitAbility_traits = new List<string>() { "Aggressive",
"Passive", "Audacious", "Timid" };
        private static Random rd = new Random();

        //Gets new traits for a new player
        public static List<string> getNewTraits(int amt)
        {
            List<string> new_traits = new List<string>();
            //Create a traits master list from the group traits
            if (traitsList.Count == 0)
            {
                traitsList.Add(trust_traits);
                traitsList.Add(deceit_traits);
                traitsList.Add(deceitAbility_traits);
            }
            while (new_traits.Count < amt)
            {
                //Get new trait
                int traitListToQuery = rd.Next(0, traitsList.Count);
                //Check if a trait from that list already exists
                bool found = false;
                foreach (string t in new_traits)
                {
                    if (traitsList[traitListToQuery].Contains(t))
                    {
                        found = true;
                    }
                }
                //If not, add it
                if (!found)
                {
                    new_traits.Add(traitsList[traitListToQuery][rd.Next(0,
traitsList[traitListToQuery].Count)]);
                }
            }
            return new_traits;
        }

        //Resolve argument made by player
        public static void ResolveArguments(Argument a, List<Player> players)
        {
            double trustChange = 0;
            double deceitChange = 0;
            double deceitAbilityChange = 0;
            //Loop through all players
            foreach (Player p in players)
            {
```

```
                //Check what the argument is, and add it to variable based on weight.
                switch (a.statement)
                {
                    case "Deceitful":
                        deceitChange = Options.argumentWeight;
                        break;
                    case "NotDeceitful":
                        deceitChange = Options.argumentWeight * -1;
                        break;
                    case "Aggressive":
                        deceitAbilityChange = Options.argumentWeight;
                        break;
                    case "NotAggressive":
                        deceitAbilityChange = Options.argumentWeight * -1;
                        break;
                    case "Trustful":
                        trustChange = Options.argumentWeight;
                        break;
                    case "NotTrustful":
                        trustChange = Options.argumentWeight * -1;
                        break;
                }
                if (p != a.receiver)
                {
                    //Assign values to player's PPMs
                    if (trustChange != 0) {
p.GetPerceivedPlayerModel(a.receiver).AddTrust(p.playerModel, trustChange); }
                    else if (deceitChange != 0) {
p.GetPerceivedPlayerModel(a.receiver).AddDeceitfulness(p.playerModel, deceitChange); }
                    else {
p.GetPerceivedPlayerModel(a.receiver).AddDeceitAbility(p.playerModel,
deceitAbilityChange); }
                }
            }
        }

        //Get player by ID
        public static Player GetPlayerByID(int id, List<Player> players)
        {
            return players.Find(o => o.playerID == id);
        }

        public static void RemoveDeadPlayers(List<Player> players)
        {
            //Remove PPMs of dead players
            RemovePerceivedModels(players);
            foreach (Player p in players)
            {
                if (p.health <= 0)
                {
                    Game.deadPlayers.Add(p); //Add dead player to list of dead players
                }
            }
            //Removing players from the players list that are out
            players.RemoveAll(item => item.health < 1);
        }

        //Remove PPMs of dead players
        public static void RemovePerceivedModels(List<Player> players)
        {
            foreach (Player p in players)
            {
                if (p.health < 1)
                {
                    foreach (Player p2 in players)
```

```
                    {
                        if (p2 != p)
                        {
                            p2.perceivedPlayerModels.RemoveAll(item => item.playerID ==
p.playerID);
                        }
                    }
                }
            }
        }
    }
}
```

```
using System;

namespace MastersThesis
{
    class PlayerModel
    {
        //PlayerModel Modifier Values
        private int largeMod = Options.traitLargeModifier;
        private int smallmod = Options.traitSmallModifier;
        private int negLargeMod = Options.traitLargeModifier * -1;
        private int negSmallMod = Options.traitSmallModifier * -1;

        //Baseline variables are 50
        public double deceitfulness = 50;
        public double deceitAbility = 50;
        public double trust = 50;

        public PlayerModel(Player p)
        {
            Random rd = new Random();
            //Initialising trust
            trust += rd.Next(-20, 20);
            trust += TrustModifiers(p);
            //Initialising deceitfulness
            deceitfulness += rd.Next(-20, 20);
            deceitfulness += DeceitfulnessModifiers(p);
            //Initialising deceitAbility
            deceitAbility += rd.Next(-20, 20);
            deceitAbility += DeceitAbilityModifiers(p);
        }

        private double TrustModifiers(Player p)
        {
            double trustModifier = 0;
            if (p.traits.Contains("Trusting"))
            {
                trustModifier += largeMod;
            }
            if (p.traits.Contains("Untrusting"))
            {
                trustModifier -= negLargeMod;
            }
            if (p.traits.Contains("Unsuspicious"))
            {
                trustModifier += smallmod;
            }
            if (p.traits.Contains("Suspicious"))
            {
                trustModifier -= negSmallMod;
            }
            return trustModifier;
        }


        private double DeceitfulnessModifiers(Player p)
        {
            double deceitfulnessModifier = 0;
            if (p.traits.Contains("Deceitful"))
            {
                deceitfulnessModifier += largeMod;
            }
            if (p.traits.Contains("Honest"))
            {
                deceitfulnessModifier -= negLargeMod;
```

```
            }
            if (p.traits.Contains("Calculating"))
            {
                deceitfulnessModifier += smallmod;
            }
            if (p.traits.Contains("Fair"))
            {
                deceitfulnessModifier -= negSmallMod;
            }
            return deceitfulnessModifier;
        }

        private double DeceitAbilityModifiers(Player p)
        {
            double deceitAbilityModifier = 0;
            if (p.traits.Contains("Aggressive"))
            {
                deceitAbilityModifier += largeMod;
            }
            if (p.traits.Contains("Passive"))
            {
                deceitAbilityModifier -= negLargeMod;
            }
            if (p.traits.Contains("Audacious"))
            {
                deceitAbilityModifier += smallmod;
            }
            if (p.traits.Contains("Timid"))
            {
                deceitAbilityModifier -= negSmallMod;
            }
            return deceitAbilityModifier;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace MastersThesis
{
    class ResultRecord
    {
        public string trait;
        public int score;
        public int amount;
        public double average;

        private List<ResultRecord> resultsList_WinnerTakesAll = new List<ResultRecord>();
        public ResultRecord(string t)
        {
            trait = t;
            score = 0;
            amount = 0;
        }
    }
    class Results
    {
        public static void DisplayResults_WinnerTakesAll(List<String> winningTraits)
        {
            Dictionary<string, double> results = new Dictionary<string, double>();
            Dictionary<string, double> adjustedResults = new Dictionary<string, double>();
            foreach (List<string> tL in PlayerListFunctions.traitsList)
            {
                foreach (string t in tL)
                {
                    results[t] = 0;
                }
            }
            foreach(String trait in winningTraits)
            {
                results[trait]++;
            }
            var sortedResults = results.OrderByDescending(o => o.Value);
            foreach(KeyValuePair<string, double> k in sortedResults)
            {
                Console.WriteLine(k.Key + " = " + k.Value);
            }
        }

        public static void DisplayResults_Survivor(List<Player> players)
        {
            List<ResultRecord> resultsList = new List<ResultRecord>();
            foreach (List<string> tL in PlayerListFunctions.traitsList)
            {
                foreach (string t in tL)
                {
                    resultsList.Add(new ResultRecord(t));
                }
            }
            foreach (ResultRecord rr in resultsList)
            {
                foreach (Player p in players)
                {
                    if (p.traits.Contains(rr.trait))
                    {
                        rr.score += p.roundsSurvived;
                        rr.amount++;
                        rr.average = (double)rr.score / (double)rr.amount;
```

```
                    }
                }
            }
            List<ResultRecord> rrL = resultsList.OrderByDescending(o =>
o.average).ToList();
            foreach (ResultRecord rr in rrL)
            {
                Console.WriteLine(rr.trait + " with average of: " +
rr.average.ToString("F") + " (" + rr.score + "/" + rr.amount + ")");
            }
        }
        public static void DisplayResults_FinalPosition(List<Player> players)
        {
            Console.WriteLine("Total Players: " + players.Count);
            List<ResultRecord> resultsList = new List<ResultRecord>();
            foreach (List<string> tL in PlayerListFunctions.traitsList)
            {
                foreach (string t in tL)
                {
                    resultsList.Add(new ResultRecord(t));
                }
            }
            foreach (ResultRecord rr in resultsList)
            {
                foreach (Player p in players)
                {
                    if (p.traits.Contains(rr.trait))
                    {
                        rr.score += p.score;
                        rr.amount++;
                        rr.average = (double)rr.score / (double)rr.amount;
                    }
                }
            }
            List<ResultRecord> rrL = resultsList.OrderBy(o => o.average).ToList();
            foreach (ResultRecord rr in rrL)
            {
                Console.WriteLine(rr.trait + " with average of: " +
rr.average.ToString("F") + " (" + rr.score + "/" + rr.amount + ")");
            }
        }
    }
}
```