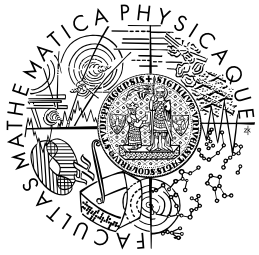

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Azamat Zarlykov

Artificial Intelligence for the Card Game Durak

Department of Software and Computer Science Education (KSVI)

Supervisor of the bachelor thesis: Adam Dingle, M.Sc.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I am filled with a sense of gratitude and admiration for my supervisor Adam Dingle for his unwavering support and guidance throughout this process. His constant patience and dedication to helping in difficult moments have been a constant source of inspiration. His expertise and knowledge along with his willingness to go above and beyond to provide assistance have provided me with valuable insights, especially during the challenging times of writing my thesis. I am truly fortunate and lucky to have had the opportunity to learn from a such remarkable mentor.

Moreover, I would like to thank my family and friends who have always been there for me during the process of writing my thesis. Their encouragement, advice, and participation in playing the Durak game to analyze the strategies have been invaluable to me and I am so grateful for a constant source of strength, and wisdom and for simply being there to provide a shoulder to lean on when I needed it the most.

Title: Artificial Intelligence for the Card Game Durak

Author: Azamat Zarlykov

Department: Department of Software and Computer Science Education (KSVI)

Supervisor: Adam Dingle, M.Sc., KSVI

Abstract: Card games with imperfect information present a unique challenge for many common game-playing algorithms because of their hidden game state. The objective of this thesis is to create a framework for implementing and testing various AI agents in the popular imperfect information card game "Durak" to identify the most effective approach in this environment. This paper presents a theoretical and experimental comparison of agents using various techniques, including rules-based heuristics, minimax search, and Monte Carlo tree search. In our analysis, we found that the Monte Carlo Tree Search agent performed the best among the implemented AI agents, whereas the rule-based heuristic agent and the minimax agent were less effective.

Keywords: artificial intelligence card game Durak

Contents

Introduction

Artificial Intelligence (AI) is a fast-growing field of computer science that focuses on the creation of intelligent machines that can simulate human cognition. In recent years, AI technology has been applied in a wide range of fields, including healthcare, finance, and transportation, with the goal of improving efficiency, accuracy, and decision-making. To gain insights into the capabilities and limitations of AI algorithms and techniques, many researchers have turned to games as a testing platform to evaluate and compare different methods as they provide a convenient and controllable environment to achieve the aforementioned goals [Silver et al., 2016].

In recent decades, computer games have also gained popularity, similar to the growth of AI as a field of study. Due to its utility, the game industry has become one of the many fields that have sought to use AI to their advantage. Being a subject of extensive research, perfect information, a game in which all players have complete knowledge of the game state, in two-player games has been a common focus in game theory, which has allowed the development of algorithms for a greater understanding of games. However, in a manner similar to the real world, situations in which all relevant information is available are not always present. Given the inherent characteristics of their environment, the design of algorithms for imperfect information games, game in which some information about the game state is hidden from some or all of the players, is more challenging. Therefore, this thesis seeks to contribute to this field by developing algorithms for the game "Durak".

Durak is a strategic card game that originated in Russia [McLeod, 2018]. It is played with a deck of cards and typically involves two to six players. Unlike the other games, the aim of Durak is not to find a winner, but to find a loser. Players take turns attacking and defending in a series of rounds. During an attack, the attacking player leads with one or more cards, and the defending player must attempt to beat them by playing a higher-ranked card. If the defending player is unable or unwilling to do so, they must pick up all the cards. The goal of the game is to get rid of all of one's cards, and the player left holding cards at the end is declared the fool.

Given the intricate nature of the game, a key objective of this work is to implement the game correctly with all relevant details. As the game will include various AI agents, it is essential for the game model to provide a suitable interface for the integration of AI agents.

Another goal of this thesis is to implement a range of AI players for the given game model. One of the benefits of introducing the agents for this game is that it will provide an opportunity to examine potential challenges associated with implementing AI for games of this type, as well as verify the suitability and usability of the game's API for this purpose.

After implementing the AI agents, the aim is to compare their performance in mutual play, with the objective of identifying the most effective technique. The AI players must not only win against all other agents but must also make moves quickly, ideally at least several moves per second on average. This requirement reflects the need for AI players to be both effective and efficient in their decision-

making. This comparison will provide valuable insights into the strengths and weaknesses of the various AI approaches and will help to guide future work in this area.

1. Game Description

The objective of this thesis is to develop a simulation of the Durak game, which would serve as an experimental environment for artificial intelligence agents using various techniques. By implementing the full range of gameplay mechanics, our aim is to create a comprehensive simulation that could be used to evaluate the performance of previously mentioned agents.

There are many variations of the Durak game that are played around the world. However, this thesis focuses on the most well-known version of the game, which is called Podkidnoy Durak (also known as "fool with throwing in")[McLeod, 2022]. In this chapter, we will provide a thorough description of this particular variation, providing an in-depth analysis of its rules and gameplay mechanics.

1.1 Terminology

In this section, any unfamiliar or potentially confusing terminology is defined to facilitate understanding of the material.

- **Trump card**
  It is a playing card that belongs to a deck and has a higher rank than any other card from a different suit. This card is typically used strategically during gameplay to defeat the other player's cards and gain an advantage.
- **Bout**
  It is a process of exchange of attacks and defenses between the players. The bout continues until either the attack is successfully defended or the defender is unable to play a suitable card, at which point the attacker wins the bout and the defender is forced to take the played cards into their hand.
- **Discard pile**
  During a bout, if an attack is successfully defended, all of the cards played during this process are placed face down on a discard pile and are not used again for the remainder of the game.
- **Early game**
  refers to any point in the game when there are still cards remaining in the deck
- **End game**
  occurs when the deck is depleted

1.2 Players

While the game of Durak is typically played with a range of two to six players, allowing for the possibility of team play, this work only focuses on the two-player variant of the game. I made this decision in order to maintain a consistent and focused scope for the analysis.

1.3 Cards

The game is played with a 36-card deck, which is divided into four suits: hearts, spades, clubs, and diamonds. The ranks of the cards within each suit are ranked from high to low as follows: ace, king, queen, jack, 10, 9, 8, 7, 6.

1.4 Dealing the cards

At the beginning of the game, cards are dealt to each player until each has a hand of six cards. The final card of the deck is then placed face up, and its suit is used to determine the trump suit for the game. The remaining undealt cards are then placed in a stack face down on top of the trump card.

During the first hand of a session, the player who holds the lowest trump card plays first. If no one holds the trump 6, the player with the trump 7 plays first; if no one holds that card, the player with the trump 8 plays first, and so on. The first play does not have to include the lowest trump card; the player who holds the lowest trump card can begin with any card they choose. If neither player has a trump card, the player who goes first is randomly determined.

1.5 Beating the card

Before discussing the gameplay, it is necessary to establish what it means for an attacking card to be successfully defended. A card that is not a trump can be beaten by playing a higher card of the same suit, or by any trump card. A trump card can only be beaten by playing a higher trump card. It is important to note that a non-trump attack can always be beaten by a trump card, even if the defender also holds cards in the suit of the attack card. There is no requirement for the defender to "follow suit" in this case.

1.6 Game play

The game consists of a series of bouts. During each bout, the attacker begins by placing a card from their hand, face up, on the table in front of the defender. The defender may then attempt to defeat this card by playing a card of their own, face up. Once the attacking card is defeated, the attacker has the option to continue the attack or to end it. If the attack continues, the defender must attempt to defend against this additional card. This process continues until the attacking player is unable or unwilling to attack. Alternatively, if the defender is unable or unwilling to beat the attacking card, they must pick up that card along with other played cards on the table.

1.6.1 Conditions on the attack

Every attacking card except for the first one must meet the following conditions in order to be played by the attacker.

- Each new attacking card played during a bout must have the same rank as a card that has already been played during that bout, whether it was an attacking card or a card played by the defender.
- The number of attacking cards played must not exceed the number of cards in the defender's hand.

The first attacking card can be any card from the attacker's hand.

1.6.2 Successful defense

The defender successfully beats off the entire attack if either of the following conditions is met:

- the defender has successfully beaten all of the attack cards and the attacking player is unable or unwilling to continue the attack.
- the defender has no cards left in hand while defending.

Upon successful defense of an attack, all cards played during the bout are placed in the **discard pile** face down and are no longer eligible for use in the remainder of the game. On top of that, the roles of the players change, i.e. the defender becomes the attacker and the attacker becomes the defender for the next bout.

Furthermore, if the defender decides to take the cards, the attacker may play additional cards as long as doing so does not violate the conditions of the attack. In this case, the defender is required to also accept these supplementary cards.

1.7 Drawing from the deck

Once the bout is over, all players who have fewer than six cards in their hand must, if possible, draw enough cards from the top of the deck to bring their hand size back up to six. The attacker of the previous bout replenishes their hand first, followed by the defender. If there are not enough cards remaining in the deck to replenish all players' hands, then the game continues with the remaining cards.

1.8 Endgame and Objective

Once the deck runs out of cards, there is no further replenishment and the goal is to get rid of all the cards in one's hand. The player who is left holding cards at the end is the loser, also known as the fool (durak). As was mentioned before, this game is characterized by the absence of a winner, with only a loser remaining at the end. However, in a two-player Durak game, it could be said that the opponent of the loser emerges as the winner.

However, it is not always the case. It is possible for the game to end as a draw. In the event that both the attacking and defending player possess the same number of cards and all of the attacking player's cards are successfully defended, the game ends in a draw.

1.9 Illustrative Gameplay Scenario

For the purpose of demonstrating the mechanics of the game, we will consider a scenario in which there are two players, A and B, and it is currently player A's turn. In this particular instance of the game, player A is holding the 6♡, 8♣, 8♢ and A♣, while player B has the 8♡, A♡, 6♠ and K♣ in their hand. It should also be noted that ♠ are the trump suit for this round. Player A initiates the attack with 6♡, which is the lowest value card in their hand. Player B has the option to respond to player A's attack by playing one of the 8♡, A♡, 6♠ from their hand, as these cards conform to the rules outlined in section 1.5. Alternatively, player B can take the card. If player B chooses to defend player A's attack by playing the 8♡, the turn returns to player A. According to the conditions of the attack specified in section 1.6.1, player A has the option to either end the attack or continue the offensive play by playing either the 8♣ or 8♢ from their hand. In case player A decides to finish the attack, all the cards in the bout move to the discard pile. On the other hand, if player A decides to continue the attack by playing one of their remaining cards, such as the 8♢, the bout will continue and the turn will pass to player B. Of the three remaining cards in player B's hand, the only one that can be used to defend against player A's current attack is the 6♠, a trump card. Assume that because of strategic reasons, player B decides to take the cards. Then, the cards in the bout are transferred to player B's ownership, thereby allowing player A to retain the role of attacker in the subsequent bout.

2. Game Analysis

As described in the chapter 1, Durak is a game that requires players to consider a range of factors in order to play effectively. Given its intricate nature, this chapter will analyze the game from the game-theoretic perspective in order to understand its underlying structure and strategic considerations. This will involve categorizing the game according to relevant criteria, examining the complexity of the game as a whole, comparing the length of the game, introducing new concept and etc.

2.1 The Termination of Durak

It is essential to acknowledge that every game of Durak must eventually conclude, as it is not possible for a game to continue indefinitely. For a game to persist, players would need to repeatedly exchange the same set of cards. However, this scenario is not feasible because cards cannot return to a previous owner until certain cards from the bout are placed in the discard pile. The inclusion of cards in the discard pile allows for the changing of turns, enabling the return of previously exchanged cards to their original owner. As the exchange of cards between players continues, additional cards from the deck and ultimately from the players' hands will be placed in the discard pile until only the cards being exchanged remain.

2.2 Repetition of States in Durak

Another question of interest is whether it is possible for the same game state to occur twice within a single game of Durak. Despite the numerous exchanges of cards between players, it is not possible for the same game state to be replicated. For the same state to repeat, players would need to exchange the same set of cards in a circle, such as player A attacking with an Ace card and player B taking it, then player B attacking with the same Ace on the next turn to player A. However, this scenario is not feasible. As previously mentioned, the intermediate bout must come to an end and cards from it must be placed in the discard pile in order to allow player B to return the Ace to player A. By this point, the game state will have changed due to the presence of additional cards in the discard pile that were not present when the Ace belonged to player A. As a result, it is not possible for the same game state to repeat in a single game of Durak.

2.3 The Advantage of the First Moving Player

It is a common misconception that the first player to move in a game of Durak holds a significant advantage over their opponent. While it is true that the first player has the opportunity to set the tone of the game and establish their strategy from the outset, this advantage is not necessarily decisive. A skilled opponent can effectively counter the strategies of the first player and ultimately achieve

victory. The outcome of a game of Durak is determined by the players' abilities and strategies, rather than the order in which they move.

2.4 Classification

Durak can be classified as a **discrete game**. A discrete game is a type of game in which players have a finite number of choices, or actions, that they can take [Owen, 2008]. This applies in Durak. Players have a limited number of choices that they can make at each turn. They can choose which card to play, and must decide whether to attack or defend. These choices are limited by the cards that the player has in their hand and the rules of the game.

Furthermore, it can be considered a **sequential** game from a game-theoretic perspective. A sequential game is a type of game in which the order in which players make their decisions matters [Owen, 2008]. In Durak, the order in which players play their cards is important, as it determines who is able to attack and who must defend. The sequence of actions is determined by the rules of the game described in chapter 1, and players must consider the potential actions of their opponents as they make their own decisions.

In addition, Durak can be classified as a game of **imperfect information**. In a game of imperfect information, players do not have complete information about the game state or the actions of their opponents [Owen, 2008]. They must make decisions based on incomplete information and must try to infer the actions of their opponents based on their observations and past experiences. As described before, Durak is a game of imperfect information because players do not have complete information about the cards in the hands of their opponents. They must make decisions about which cards to play and when to use their trump cards based on incomplete information, and must adapt their strategies as the game progresses and new information becomes available.

Additionally, Durak is often played in a **deterministic** manner, meaning that the outcome of the game is determined by the initial cards that are dealt and the actions that are taken by the players during the game. However, there is some element of chance in Durak, as the cards are shuffled randomly before the game begins can affect the outcome of the game. Therefore, it is possible to consider Durak to be a **non-deterministic** game to some degree. In game theory, a non-deterministic game is a type of game in which the outcomes are not determined solely by the actions of the players and the rules of the game, but are also influenced by random events or factors [Owen, 2008].

In summary, Durak can be classified as a discrete, sequential, imperfect information, and non-deterministic to some extent game from a game-theoretic perspective, which contribute to its complexity and strategic depth.

2.5 Branching Factor

The branching factor of a game refers to the number of possible moves that a player can make at each turn. In "Podkidnoy Durak", it can be challenging to determine the branching factor as it varies depending on the specific game state. At each turn, the number of possible moves a player can make is influenced by

the cards in their hand and the cards on the table, as well as the defending or attacking rules. To be specific, the attacker can initiate an attack by playing any card from their hand. Therefore, the maximum branching factor is the maximum number of cards that a player can hold in their hand, which is 35 if one player holds all but one of the cards. However, there are also situations in which a player may only have one possible move, such as when they are unable to defend against an attack and must pass and take the card. Therefore, the average branching factor in this game is relatively low, as players often have only a few choices of cards to play in a given situation. This is particularly true for the defender, who may only have a few options for defending against an attack, and for subsequent attacks, where the number of available options may also be limited.

To clarify the branching factor assumption, I have run an experiment to estimate the average branching factor in Durak by simulating 1000 random games played between two greedy agents. The results showed that the average branching factor, as computed using the **geometric mean**, was **2.17**, which suggests that the branching factor of the game is low.

2.6 Duration

The objective of this section is to determine the typical duration of games of Durak in terms of bouts and plies, where ply refers to a single move made by a single player. To address this question, we will compare the average length of games played between two random players and two greedy agents, in order to examine the influence of player strategy on the duration of the game.

To compare the durations of player strategies in Durak, we conducted two experiments. The first experiment involved 1000 games played between two greedy agents, and the second experiment involved 1000 games played between two random agents. The results of the first experiment showed that the average number of bouts per game was 8.3, the average number of plies per bout was 5.3, and the average number of plies per game was 44.0. The results of the second experiment showed that the average number of bouts per game was 24.0, the average number of plies per bout was 2.9, and the average number of plies per game was 68.0. These findings suggest that the behavior of the random agents led to longer games, as evidenced by the higher number of bouts and lower number of plies per bout in the second experiment.

2.7 Weakness Concept

This section introduces the concept of **weakness** and **well-covered weakness** in Durak, which are the concepts that arise from the analysis of the game and may be relevant to various strategies or agents.

The concepts in question are introduced in Edouard Bonnet's paper "The Complexity of Playing Durak". It examines the difficulty of identifying winning strategies in the card game Durak. Bonnet's work demonstrates that, even in a perfect information setting with two players, finding optimal moves is a challenging computational problem. Specifically, Bonnet establishes that determining the presence of a winning strategy in a generalized Durak position is PSPACE-

complete. In my own research, I aim to construct a strong agent capable of playing optimally in both perfect and imperfect information settings. Bonnet's contributions, including the concept of weakness and well-covered weakness, have been invaluable in the development of my rule-based agents.

A weakness for a player, referred to as player P, is defined as a rank r that meets the following criteria:

1. player P's hand contains at least one card of rank r, and
2. for each suit s of rank r in player P's hand, there exists a rank r'$> r$ such that the opponent holds a card of rank r' and suit s [Bonnet, 2016].

To clarify the concept of weakness, consider the following scenario: Player P holds the cards 10♡, 10♠, and K♢, while player O holds Q♡, Q♠, and J♣. In this case, player P has a weakness at rank 10, as it satisfies the two conditions outlined in the definition of weakness. Specifically, player P holds at least one card of rank 10 (10♡ and 10♠), and for each suit of rank 10 in player P's hand (10♡ and 10♠), the opponent holds a card of higher rank (Q♡ and Q♠, respectively).

A well-covered weakness for player P, on the other hand, refers to a weakness card with rank r such that for every card in player P's hand with suit s and rank r, there is a higher card with suit s and rank r' in player O's hand, and player P does not possess any cards with rank r'. Essentially, if player P attacks with a well-covered weakness, player O can defend effectively, preventing player P from playing any other attacking cards during the bout.

In the example provided following the definition of weakness, the cards 10♡ and 10♠ are considered well-covered weaknesses because attacking with these cards allows the opponent, player O, to effectively defend with Q♡ and Q♠ and prevent player P from attacking again.

2.8 Closed World Deduction

In the closed world environment, it is possible for a player to deduce the opponent's remaining cards once the deck is depleted. This is because once the deck is exhausted, the only cards that can be played are those that are held by the players. If a player is able to keep track of the cards that have been played, they can narrow down the possible cards that their opponent still holds and make educated guesses about their strategy based on this information. To give an example, imagine that the deck has been exhausted, and there are only four cards left in the game: A♡, 6♠, 7♢, and 8♣. Player A is able to deduce that their opponent, player B, still holds A♡ because they have played the 6, 7, and 8 cards, but have not yet played the A card. Based on this information, player A can infer that player B is likely trying to hold onto the A card in order to use it as a trump card later in the game. By paying a close attention, player A will know which cards have already been played and which ones are still in play. Player A knows that A♡ has not yet been played because they have not seen it played, and they also know that the A♡ card has not been discarded because the deck has not yet been exhausted. Therefore, player A can deduce that the A♡ card is still in play either in the deck or in the opponent's hand. Once the deck is depleted, it becomes obvious that it is being held by player B. The ability to deduce an opponent's

remaining cards can significantly impact the course of the game, as it allows players to incorporate this information into their strategic decision-making. However, it is important to note that Durak is a game of chance as well as strategy, and even if a player is able to deduce their opponent's remaining cards, they cannot predict with certainty which cards the opponent will play in a given turn.

3. Game and AI Implementation

The primary objective of this study is to design and develop a framework for the implementation and evaluation of artificial intelligence (AI) agents in Durak. Since the framework in this game is tailored to support the application and evaluation of AI algorithms within the game environment, the focus of this chapter is to provide a high-level overview of the framework's architecture, including its capabilities for supporting the development and evaluation of AI agents for Durak.

3.1 OS support

I tested the game framework for Durak on both the Windows 10 and Linux operating systems to confirm compatibility and functionality. While this list is not exhaustive, and the framework may potentially run on other operating systems, these are the only ones that were formally tested.

3.2 A High-Level View of the Framework

The Durak AI framework includes a game model, AI agents, and a command-line interface (CLI) that are implemented using the C# programming language and targeted for the .NET 6 platform. C# was selected as the programming language for the implementation of aforementioned projects in Durak. The language offers benefits from a highly optimized Just-In-Time (JIT) compiler, resulting in enhanced speed. This attribute made C# an ideal choice for this development.

In order to run and test the program, the project has to be cloned from the repository . The source code for the project is available on GitHub. From the CLI directory of the project, the user can use the command line to enter the following command:

```
$ dotnet run
```

This will launch the command-line interface and provide guidance on how to proceed with experimentation (for additional information, please refer to Section 3.2.3).

An analysis of the source code using Visual Studio's Calculate Code Metrics for Solution feature revealed that the solution consists of a total of 3034 lines of source code. This includes 993 lines in the Agent project, 757 lines in the CLI project, and 1284 lines in the game model.

3.2.1 Project Structure

The game Durak is organized within a solution file, with the file extension ".sln", which is a type of file used to manage projects in Visual Studio. This solution includes three individual projects:

- Model - A C# library project that contains the game logic for Durak.

- Agent - A C# library that contains all of the implemented AI agents.
- CLI - A C# Command-Line Interface (CLI) project that includes parameters for modifying the game model and agents settings in order to perform experiments.

  The aforementioned components will be further discussed in the following subsections.

3.2.2 Model

The game model, which represents the current state of the game, is implemented using object-oriented programming principles. As was mentioned before, the game logic for Durak is contained within the **Model** C# library, which serves as a modular and reusable unit. It includes class objects, such as Player, Card, and Deck, as well as all of the other main components that make up the game.

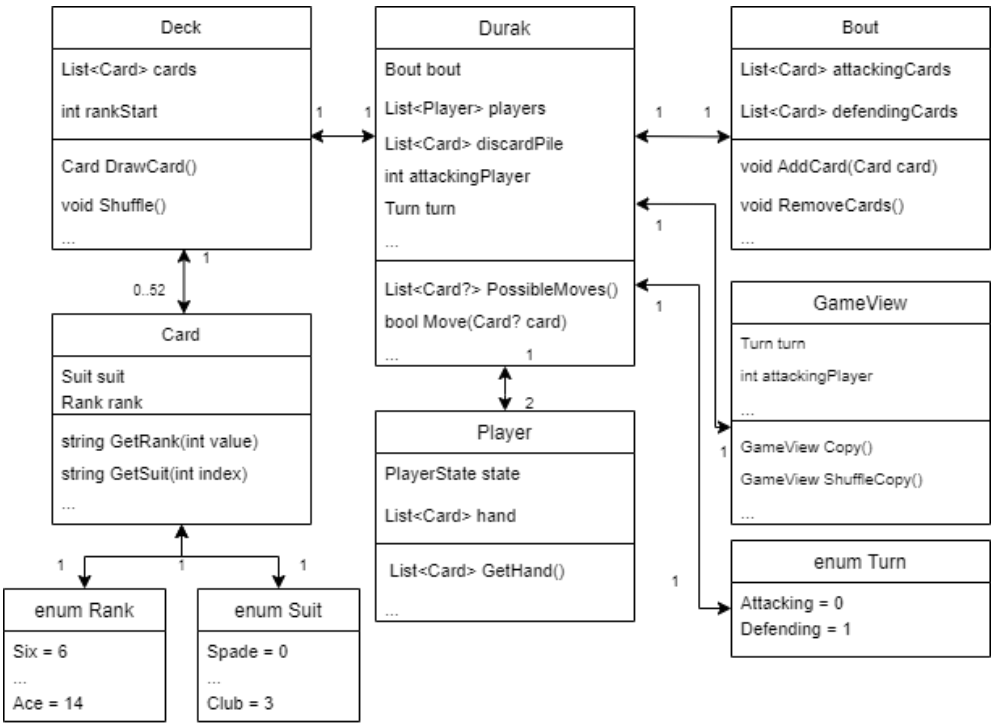

Figure 3.1: A simplified UML diagram showing the relationship between the objects within the Model library.

Before delving into the description of the game state and components of Durak, it is useful to first consider the relationship of all of the objects in the model to that state. A class diagram illustrating the relationships between the objects in the model can be found in Figure 3.1.

The Deck object includes the property `rankStart`, which is an integer value that can be modified from the command-line to alter the starting rank of the cards in the deck. By default, the `rankStart` is set to 6, but it can be changed to any value between 6 and 14. For instance, if the `rankStart` is set to 13, the deck will only consist of 8 cards: 4 Aces (rank value 14) and 4 Kings (rank value 13).

The representation of the game state `Durak` in the model is a key aspect of the overall system. This representation holds all of the necessary information and logic required to play the game of Durak, shown in Figure 3.2, and therefore plays a central role in the functioning of the model. As such, it is important to carefully consider the design and implementation of the game state representation along with its components.

```
// Bout object of the game
private Bout bout;

// Deck object of the game
private Deck deck;

// Trump card of the game that can be assigned or not
private Card? trumpCard;

// Representation of the discard pile in the game
private List<Card> discardPile = new List<Card>();

// Players inside the game
private List<Player> players = new List<Player>();
```

Figure 3.2: A simplified diagram of the Durak class, which encompasses the main properties of the game

The object in question serves as a comprehensive representation of all game states and data throughout a single game of Durak. To facilitate communication and coordination between the Durak model and the agents that interact with it, the game state provides two primary functions: `PossibleMoves`, shown in Figure 3.3 and `Move`, shown in Figure 3.4. These functions serve as the primary means of interaction between agents and the game state, and as such, play a crucial role in the overall operation of the model.

The `PossibleMoves` method determines the list of actions that are available to the current player based on the current game state and the rules of the game. When it is the attacker's turn, the method considers the rules for attacking (details in section 1.6.1) and generates a list of eligible cards that can be played or allows the player to pass if no suitable cards are available. Similarly, when it is the defender's turn (details in section 1.5), the method takes into account the card being attacked and generates a list of cards that can be played to defend or offers the option to take the attack if no suitable defense is available. Because of that the `PossibleMove` function returns `List<Card?>` type. If there are possible moves that can be made in the current state, the function returns a list of the available cards to play. If no moves can be made, the function returns a list containing a single `null` element, which indicates that the current player must pass or take the card or cards from the bout, depending on the current turn. It should be noted that the example provided in Figure 3.3 is a simplified version that omits certain implementation details, such as the process of adding elements to the list.

The `Move` method modifies the current game state by executing the action chosen by the current player. This move is selected by the agent, which performs calculations based on the possible moves generated by the `PossibleMoves`

```
if (turn == Turn.Attacking){
        if (CanAttack() && OpponentCanFitMoreCards()) {
                return GenerateListOfAttackingCards();
        } else {
                // passing the attack
                return null;
        }
}
else {
        Card attackingCard = bout.GetAttackingCards()[^1]
        if (CanDefend(attackingCard)) {
                return GenerateListofDefendingCards(attackingCard);
        } else {
                // taking the cards
                return null
        }
}
```

Figure 3.3: A simplified overview of the PossibleMoves method inside the Durak class

method. The specific nature of these calculations depends on the type of agent being used. For example, a rule-based agent may simply select the lowest value rank card, while a more sophisticated agent, such Monte-Carlo Tree Search (MCTS), may use more complex decision-making processes to determine the optimal move to make. Regardless of the type of agent being used, the `Move` method ultimately updates the game state to reflect the chosen action and advances the game to the next turn. To implement the desired changes, the `Move` method accepts a parameter of type `Card?`. This parameter represents the move made by the agent, which can be a card play (type `Card`), or a pass or draw (type `null`), depending on the role of the agent. Also, it is worth noting that the Move method returns a boolean value. In order to prevent invalid moves, the method verifies the validity of the intended action before making any changes. If the action is deemed valid, the method returns true; otherwise, it returns false.

Additionally, it is important to note that, for efficiency and security purposes, the agents are not provided with the entire `Durak` object. One reason is that the Durak class contains a large amount of data and methods that are not relevant to the agents' decision-making process. By providing a smaller class such as `GameView` that only includes the necessary information, such as methods outlined in Figures 3.3 and 3.4, the agents can more efficiently access the information they need and ignore the rest. Another reason is that the Durak class contains sensitive or proprietary information that should not be shared with the agents. By using a smaller class such as `GameView` to provide the necessary information, it is possible to control what information is exposed to the agents and protect any sensitive data.

Other than `PossibleMoves` and `Move` methods, the `Copy` method is an important feature of the GameView object worth mentioning. This method creates a duplicate of the current game state, which is useful for game tree exploration by agents such as Minimax and MCTS, due to their need to examine multiple

```
// check if move is valid
if (!ValidAction(card, attacker, defender))
        return false;

if (turn == Turn.Attacking){
        // the attacker played a card
        if (card is not null) {
                attacker.GetHand().Remove(card);
                bout.AddCard(card);
        } else {
                bout.RemoveCards();
                return true;
        }
}
else {
        // the defender beat the attacking card
        if (card is not null){
                defender.GetHand().Remove(card);
                bout.AddCard(card);
        } else {
                FillPlayerHand(bout.GetEverything(), defender)
                return true;
        }
}
turn = turn == Turn.Attacking ? Turn.Defending : Turn.Attacking;
return true;
```

Figure 3.4: A simplified overview of the Move method inside the Durak class

potential moves and outcomes. For further information on the use of the Copy() method in Minimax and MCTS, please refer to section 4.3 and 4.4 of the text.

3.2.3 CLI

The command-line interface (CLI) plays a crucial role in the architecture of the application. Through the CLI, the user can interact with the application using a text-based interface, providing parameters and receiving feedback or results. The CLI enables a range of experimental and testing scenarios, including the ability to conduct playouts between different agents within a customizable game environment that can be modified by altering various parameters. In this section, we will examine the various parameters that can be used to manipulate the behavior of the agents and the game environment through the CLI.

Before discussing the organizational structure of the project, it is important to introduce the parameters and their roles within the project (please refer to Table 3.1).

The `-open_world` parameter plays a key role in determining the level of information available to players in the game. If the `-open_world` parameter is present, the game environment is fully visible to all players, including the cards in the deck and the cards in players' hands. This results in a perfect information environment, where all players have access to the same information. By comparing agents in this type of environment, it is possible to identify the most effective one. On the other hand, if the `-open_world` parameter is not present,

Parameter	Description
-ai1	The agent for player 1. (String) (Default = random)
-ai2	The agent for player 2. (String) (Default = random)
-d1	Displays # of states & depth for minimax move (Default = False)
-d2	Displays all the moves that minimax considers (Default = False)
-include_trumps	Enable trump cards in the game (Default = True)
-log	Enable logs for writing in the file (Default = False)
-open_world	Make all cards visible to both players (Default = False)
-seed	A seed for random number generation (Int32)
-start_rank	The starting rank of cards in the deck (Int32)(Default = 6)
-total_games	The number of games to play (Int32)(Default = 1000)
-tournament	Runs the tournament with the agents specified.
-verbose	Enable verbose output (Default = False)

Table 3.1: Command-line parameters

the game environment is not fully visible to all players, resulting in an imperfect information environment where players may not have complete knowledge of the game state

The `-tournament` parameter allows for the specified agents to engage in a series of games, the results of which are recorded in a CSV file. The game settings for the tournament can be customized when the `-tournament` parameter is included. These settings will be applied to all games played between the agents. However, if the results of the games between two agents do not show a significant difference according to Wilson's score, the number of games will be increased by 500 and the tournament will be restarted for two equally strong agents in order to ensure that the best player can be accurately determined. There is an upper limit on the number of games that can be played in cases where the agents are evenly matched and unable to produce a clear winner. In such situations, the agents will be listed in a separate table within the CSV file. An example of how to initiate a tournament between the Random, Greedy, and Smart agents is provided below:

```
$ dotnet run -tournament="random,greedy,smart" -total_games=100
```

There are various ways to utilize the parameters in Table 3.1. An example of using the default settings to run a game between RandomAI agent(random) and GreedyAI agent(greedy) is provided below.

```
$ dotnet run -ai1=random -ai2=greedy
```

This command initiates the simulation of 1000 games between the RandomAI and GreedyAI agents in a fully enclosed environment, where players can only see their own cards and not those of other players, (with a starting rank of 6) and provides the following output to the console:

```
==== RUNNING ====

Game 1: Agent 1 (random) won. Total bouts: 21
Game 2: Agent 2 (greedy) won. Total bouts: 19
Game 3: Agent 2 (greedy) won. Total bouts: 13
Game 4: Agent 2 (greedy) won. Total bouts: 18
Game 5: Agent 2 (greedy) won. Total bouts: 18
```

```
...
```

To more thoroughly analyze the results of any specific game, `seed` with the game id and the `-verbose` parameters may be utilized. This provides detailed information about the progression of the game by showing every possible move, the chosen move and other game related details. An example of the first game in which a RandomAI agent defeats a GreedyAI agent using this parameter is shown below:

```
$ dotnet run -ai1=random -ai2=greedy -verbose -seed=1
```

The command above generates verbose output, as shown below. It should be noted that this is only a portion of the full output and the blue colored suits are the indications of the trump suit.

```
==== START ====

Trump card: A♢
Deck's size: 36

Player 1 (random) cards: 9♠  A♠  10♡  Q♡  9♢  K♢
Player 2 (greedy) cards: 6♠  8♠  Q♠  7♡  A♡  6♣

=== New Bout ===

TURN: Player 1 (random) (Attacking)
Can attack
Possible cards: 9♠  A♠  10♡  Q♡  9♢  K♢
Attacks: 9♠

Bout 1:
Attacking cards: 9♠
Defending cards:

TURN: Player 2 (greedy) (Defending)
Can defend
Possible cards: Q♠
Defends: Q♠

Bout 1:
Attacking cards: 9♠
Defending cards: Q♠
...
```

Reproducibility is a critical aspect of scientific experimentation. In this context, reproducibility refers to the ability to obtain the same results by running the program with the same command line parameters. To ensure reproducibility, the program assigns a unique identifier to each game, which serves as a seed for the random number generator that deals cards to players. This allows testing different agents in a controlled and consistent environment, facilitating comparison of performance and error detection.

Upon completion of an experiment, the program presents statistical analysis on all simulations conducted using the specified game and agent configuration. This analysis includes various metrics, such as the average number of rounds played per game, the average number of plies made per bout, the average time

taken per ply by each agent, the win rate, and the **Wilson confidence interval** between the two agents.

```
$ dotnet run -ai1=greedy -ai2=random -open_world -total_games=1000
                         -start_rank=6 -include trumps
...

==== STATISTICS ====
Total games played: 1000

Average bouts played over the game: 17.1
Average moves per bout over the game: 3.1

Average time per move (greedy agent): 0.0201ms
Average time per move (random agent): 0.0181ms

Draw rate: 0.8%
Agent 1 (greedy) won 913 / 1000 games (91.3%)
Agent 2 (random) won 79 / 1000 games (7.9%)

With 98% confidence, Agent 1 (greedy) wins between 89.8% and 93.8%
With 98% confidence, Agent 2 (random) wins between 6.2% and 10.2%
```

Figure 3.5: A representation of command and statistics.

I chose Wilson's confidence interval in order to assess the statistical significance of the observed difference in win rates between the two players. By using this measure, we may determine whether the observed difference in win rates was likely to be a true reflection of the relative abilities of the players, or whether it may have occurred by chance [Moor and G. McCabe, 2017]. Overall, the statistical data generated from the simulations was useful for identifying potential errors and for optimizing the strategies for the rule-based agents. As an example, Figure 3.5 presents the statistical result of a simulation comparing the performance of greedy and random agents in a full open-world game.

Other than that, it is important to consider that when utilizing certain advanced agents, such as Monte-Carlo Tree Search (MCTS) and Minimax, it is necessary to specify their respective parameters in order to effectively utilize their capabilities. Those parameters will be discussed in Section 4.5.1.

3.2.4 Agent

The Agent library is a collection of artificial intelligence agents that are utilized in the experiment. The agents included in this library are: Random, Greedy, Smart, Minimax, and MCTS. The functionality and implementation of these agents will be elaborated upon in the next chapter. This section describes the overall structure and organization of the Agent library.

The abstract class `Agent` represents a common interface for all agents. By implementing the abstract method `Move` inside the `Agent` abstract class, the agents are able to make a move inside the game given the set of possible options or moves. A visual representation of the abstract class `Agent` is provided in Figure 3.6.

```
public abstract class Agent
{
        public string? name;
        public string? GetName() => name;
        public abstract Card? Move(GameView gameView);
}
```

Figure 3.6: A representation of the abstract class 'Agent'

As an example, we can examine the behavior of a Random agent, which selects a move from the list of possible moves at random, can be examined. This process is demonstrated in Figure 4.1. The Move method of the Random agent is responsible for implementing this behavior. The figure illustrates the process of overriding the abstract Move method in order to modify the behavior of the function based on the random nature. This process is followed for all agents that must implement this method, resulting in diverse behaviors depending on the agent in question.

4. Artificial Intelligence Agents

My implementation of Durak features five distinct agents: Random, Greedy, Smart, Minimax, and Monte Carlo Tree Search. In this chapter, we will delve into the characteristics and behaviors of each of these agents in order to gain a better understanding of their nature.

4.1 Random Agent

The random agent simply selects an action at random, as its name suggests. It serves as a foundational reference point for comparison with more advanced algorithms.

```
function Move(gameView: GameView)
        cards = gameView.Actions(excludePassTake: true)

        // cannot attack/defend
        if cards.Count == 1 and cards[0] is null
                return null

        // include the case when the ai can decide to pass/take
        // even if it can defend/attack (20% chance)
        // allow only when the first attack was given
        rn = random number between 0 and 100
        if rn <= 20 and gameView.bout.GetAttackingCards().Count > 0
                return null

        return a random card from cards
```

Figure 4.1: Pseudocode of Move method of the Random agent

Figure 4.1 presents the Move method for the Random agent. When there are no available moves to be made (i.e., the `gameView.Actions(...)` method returns an null at the first index of the list), the agent interprets this as a Pass if it is the attacker, or as a Take if it is the defender. Given that there are multiple options available, the random agent has a 20% probability of randomly deciding to either Pass or Take the card, depending on its role. The choice of a 20% probability is arbitrary, but it represents a reasonable balance between actively selecting a card and opting to pass or take.

4.2 Rule-Based Heuristic Agents

Rule-based heuristic agents are agents that use a set of predefined rules, heuristics or strategies to make decisions [Millington and Funge, 2009]. These rules are designed to capture the key characteristics of a problem or task, and they are used by the agent to determine the best course of action to take in a given situation.

The strategies employed by the agents in this section were developed through the accumulation of experience gained over multiple games. These strategies are

not guaranteed to produce victory in every game, but they increase the chances of winning by making moves based on predefined rules.

4.2.1 Greedy Agent

I have observed through years of playing this game that the best move in any given game state is often to play the card with the lowest value. This approach is advantageous because it allows the player to save strong cards for later in the game, increasing the chances of winning. Conversely, using high rank cards for attack and defense in the early game and reserving weak cards for later can lead to a loss. Additionally, playing the lowest value cards allows the player to draw stronger cards from the deck, thereby building a stronger hand over the course of the game. As such, the greedy agent employs a predetermined strategy in which it selects the lowest ranked card for attack or defense based on the current turn. Specifically, if the agent is attacking, it will choose the lowest valued card to attack with. If the agent is defending, it will select the lowest valued card that can defend against an incoming attack. To provide an example, consider the scenario described in section 1.9, in which Player A attacks with 6♡ as a first turn. Among all the possible options, which includes 8♡, A♡, 6♠, to select to defend against the attacking card, Player B chooses 8♡ which is the lowest rank value in their hand to defend against the attacking card. The 6♠ card was not selected because it is a trump card and therefore has a higher value than non-trump cards, thus, it has higher value than 8♡. This decision-making process aligns with the strategy employed by a greedy agent, which aims to maximize their own short-term gain at the expense of potentially more optimal long-term outcomes [Russell et al., 2022].

```
function Move(gameView)
    moves = PossibleMoves(gameView)
    if HasNull(moves) is true
        return NULL
    else
        return GetCard(moves, gameView)

function GetCard(possibleCards, gView)
    noTrumpCards = SelectNonTrumpCards(possibleCards)
    if size of noTrumpCards is 0
        if EarlyGame(gView) is true
            if size of BoutACards(gView) > 0 and Turn(gView) == ATTACK
                return NULL
            else
                return LowestRank(possibleCards)
        else
            return LowestRank(possibleCards)
    else
        return LowestRank(noTrumpCards)
```

Figure 4.2: Pseudocode of the Move method

The pseudocode shown in Figure 4.2 outlines the steps taken by the greedy agent to evaluate its options and make a decision. Specifically inside the `Move`

method, the agent utilizes the `PossibleMoves` method to obtain a list of all possible actions it can take based on the current game state. `HasNull(moves)` detects if no actions are available. If that's the case, the agent will either pass or take a card, depending on its role in the game. However, if there are available actions, the agent will use the `GetCard` method to identify the card with the lowest value among the possible moves and select it as its next action. To ensure that the greedy agent is able to maximize its gain, we select the non-trump cards from the list of available options when determining the agent's next move. If the agent has a choice between non-trump cards, it will naturally select the one with the lowest rank value. However, if the agent only has trump cards to choose from, we must consider the **stage** of the game: early or end game. If the greedy agent is acting as an attacker and has the option to pass the attack, it is generally better to do so during the early game, as this allows the agent to preserve its trump cards for use in the end game, when they are likely to be more valuable. On the other hand, if the greedy agent is defending, it does not matter whether it uses trump cards or not at any stage of the game, as giving up these cards is preferable to taking the entire bout.

4.2.2 Smart Agent

A smart agent using rule-based heuristics can exhibit improved performance compared to a greedy agent. This is because the smart agent utilizes rules that take into account the opponent's hand, providing a strategic advantage. One specific rule that may be implemented during the defensive stage is the selection of a defending card with a rank that the opponent does not possess. If it is not possible to utilize the aforementioned defensive strategy due to the opponent holding cards of the same rank as the available defending cards, the agent may choose to defend with the card of the lowest rank. The implementation of the method is represented in Figure 4.3. This rule aims to prevent further attacks in the same bout by mitigating the opponent's options. While this rule may not be the most effective in every scenario, it has demonstrated successful outcomes in certain situations.

```
Card? DefendingStrategy(List<Card> oHand, List<Card> cards)
        for each card∈cards do
                if (oHand) contains any cards with the same rank as (
                    card) then
                        return (card)
        return GetLowestRank(cards);
```

Figure 4.3: Pseudocode of the `Move` method of the Greedy agent, where `oHand` is a list of cards in the opponent's hand and `cards` is a list of possible cards to defend with

Moreover, a smart agent may employ a strategy that leverages the concept of **weaknesses** discussed in Section 2.7 in order to gain an advantage. *If player P only possesses a single weakness at rank r during their turn, then they have a winning strategy* [Bonnet, 2016]. In the example presented in Section 2.7, player P possesses a weakness of rank 10. According to the aforementioned statement,

this implies that player P has a winning strategy and can outmaneuver their opponent. This is indeed the case. To initiate the attack, player P can utilize all non-weakness rank cards, such as the K♢. The opponent must take the attacking card, as they are unable to defend against it due to K being a non-weakness rank. With the new bout, it is player P's turn to initiate an attack. They choose to attack with the 10♡, which has a weakness rank. In response, player O defends with the Q♡. The turn then returns to player P, who attacks with their final card, the 10♠, which also has a weakness rank. Although player O is able to defend against this attack, they become the loser because they are the only player remaining with cards.

One limitation of this strategy is that it is only applicable in open-world environments, where the player is able to gather information about their opponent's hand in order to identify weakness in their hand. Nonetheless, the weakness strategy can still be utilized in closed-world games, but only in the endgame when the deck is exhausted. This is because when players have used up the entire deck, it implies that certain cards played during the bout have been placed in the discard pile. This allows the players to deduce the opponent's cards and successfully use the strategy for their advantage. A smart agent adheres to this principle, which, while not being a common occurrence in the game, is highly effective and ensures victory for the smart agent.

4.3 Minimax Agent

In this section, we will discuss an agent that uses the minimax algorithm to play Durak. Specifically, we will examine the importance of the parameters in the minimax algorithm and the heuristic functions included in the Durak Command Line Interface (CLI) for evaluating the game state.

The minimax algorithm is a decision-making algorithm often used in two-player turn-based perfect information games, such as chess, tic-tac-toe, and Go [Russell et al., 2022]. It is called minimax because it tries to minimize the maximum loss that a player can incur. It operates by constructing a game tree, with each node representing a potential game state, and the branches representing the possible moves that can be made from that state. The minimax algorithm then traverses the tree, evaluating the value of each node recursively. When applied to the entire tree, the minimax values produced by this algorithm accurately reflect the outcome of the game with perfect play by both players. The pseudocode of the minimax algorithm is illustrated in Figure 4.4. To explore the potential game states, the minimax algorithm creates the copy, `game.Copy()`, of the state to ensure that the original game state cannot be changed after making possible moves. It is important to note that the following discussion of the algorithm assumes an open Durak environment, in which all players have access to the visible cards. This allows the game state to be freely cloned without letting the agent to see the hidden information. Section 4.5 will address the algorithm's handling of the closed world environment, in which there is hidden information.

Due to the exponential increase in the size of the tree as the search depth increases, it is often infeasible to search the entire tree. As a result, the minimax algorithm must be terminated at some depth (`MAX_DEPTH`), and the minimax value for a given node is approximated using a heuristic evaluation function. The

```
function minimax(game, depth, out bestMove)
        if depth = MAX_DEPTH or game is in terminal state
                return the heuristic value of state

        bestVal = game.turn == 1 ? -infinity : +infinity

        for each card of game.PossibleMoves()
                stateCopy =     game.Copy()
                stateCopy.Move(card)
                v = minimax(stateCopy, depth + 1, _)

                if game.turn == 1 && v > bestVal or game.turn == 2 && v
                    < bestVal then
                        bestVal = v;
                        bestMove = card;

        return bestVal
```

Figure 4.4: Pseudocode for the minimax algorithm

heuristic function assigns a score to the current game state based on a set of predefined criteria, which will be discussed further in this section. While the use of a heuristic evaluation function introduces some error into the minimax values, it allows the algorithm to produce reasonable results in a reasonable amount of time.

4.3.1 Basic Heuristic

The basic heuristic function utilizes predetermined criteria to evaluate the game state, using the contents of the players' hands, the size of the players' hands, and the presence of any weaknesses. It should be noted that this is not the only method of evaluating the state, and there are potentially numerous other criteria that could be incorporated into the heuristic function. However, the criteria mentioned are considered to be the primary ones used in the basic heuristic function.

In order to evaluate the state of a card game based on the cards held by each player, the heuristic function begins by computing the individual hand value of each player. Prior to beginning this calculation, the value of each card must be determined. This can be done by assigning the value of a card as its rank. For example, if player P holds the cards 6♡ and A♡, the value of their hand would be 20 (6 + 14). Trump cards are powerful, so we assign them a higher value. To achieve this, we calculate the value of a trump card by adding its rank to the total number of ranks in the deck. For example, the value of a trump card with rank 6 would be calculated as 6 + the total number of ranks in the deck (6, 7, ..., K, A). After the individual hand value of each player has been calculated, a basic heuristic function may be used to evaluate the game state. This can be done by subtracting the hand value of player P from that of player O. The result of this calculation can then be used to represent the game state. How this is achieved can be viewed in Figure 4.5. It should be noted that this method of evaluating the game state is only suitable for use when the players have an equal number of

```
function ConvertHandToValue(hand: List of Card, gw: GameView)
        TOTALCARDS = 9
        value = 0

        for each card in hand
                if gw.includeTrumps is true and card.suit == gw.
                    trumpCard.suit
                value = value + card.rank + TOTALCARDS
                else
                value = value + card.rank

        return value

function EvaluatePlayerHandToValue(gw: GameView)
        return ConvertHandToValue(gw.players[0].GetHand(), gw) -
            ConvertHandToValue(gw.players[1].GetHand(), gw)
```

Figure 4.5: Pseudocode of partial basic heuristic function evaluating the state based on the players' hand value

cards. In past experiments, using this evaluation method with players who have an equal number of cards has been found to produce more accurate and reliable results.

```
function EvaluateHandSize(gw: GameView)
        pHandSize = gw.players[0].GetNumberOfCards()
        oHandSize = gw.players[1].GetNumberOfCards()

        if gw.isEarlyGame is true
                pHandSize = pHandSize
        else
                pHandSize = pHandSize * 15
        if gw.isEarlyGame is true
                oHandSize = oHandSize
        else
                oHandSize = oHandSize * 15

        return (pHandSize - oHandSize) * -1
```

Figure 4.6: Pseudocode of partial basic heuristic function evaluating the state based on the players' hand size value

Another criterion that may be used to evaluate the state of a card game is the size of the players' hands. The number of cards held by each player can provide insight into the likely outcome of the game. This can be done by subtracting the size of player P's hand from that of player O's hand. In the end game, the value of the hand size is given additional weight by being multiplied by a factor, such as 15, which is arbitrarily selected. This is because the number of cards held at the end of the game can be particularly important in determining the outcome. It is important to note that the resulting value is inverted, by being multiplied by -1, because a player with more cards is generally considered to be at a disadvantage. The method being described is illustrated in a figure, labeled as Figure 4.6.

```
function AttackingStrategy(gw: GameView, noTrumpCards: List of Card,
    possibleCards: List of Card)
        oHand = gw.GetOpponentCards()
        pHand = gw.playerHand

        weaknesses = GetWeaknesses(possibleCards, oHand)

        if size of weaknesses is 1
                weakRank = weaknesses[0]

                if (pHand) contains all cards with rank (weakRank)
                return first card in pHand
                return lowest rank (card) in (pHand) where (card)'s rank
                     is not weakRank

        if size of weaknesses is greater than 1
                nonweakness = cards in pHand where each card's rank is
                    not a weakness

                if size of weaknesses is less than or equal to size of
                    nonweakness ranks
                        weakRank = GetBadlyCoveredWeakness(gw, oHand,
                            nonweakness, weaknesses)
                        if weakRank does not exist
                                return lowest rank in noTrumpCards
                        return first card in pHand such that its rank is
                             weakRank
        return lowest rank in noTrumpCards
```

Figure 4.7: Pseudocode of partial basic heuristic function evaluating the state based on the weaknesses

As a final criterion for evaluating the game state, it is possible to consider the existence of weakness in the state. This concept was discussed in more detail in section 4.2.2. Scenarios of this type are relatively rare, occurring only in the endgame when the defending player, the opponent, does not possess any trump cards. This is because the strategy involving weaknesses is only effective in a perfect information game where trumps are not included. Figure 4.7 presents the pseudocode for this strategy and demonstrates the precise manner in which cards are selected in this specific environment. If the game state is in the environment described above, we invoke the strategy. The pseudocode checks the number of weakness cards in player P's hand. If there is only one weakness card, it can be inferred that player P has a winning strategy by playing a non-weakness card first and then following it up with the weakness card (this strategy is discussed in further detail in Section 4.2.2). In the event that there is more than one weakness card, a different approach is employed in an effort to reduce the number of weaknesses and return to the previously mentioned strategy. This method involves the concept of **badly-covered weaknesses** (the opposite of well-covered weaknesses, as discussed in Section 2.7). Essentially, if player P attacks with a badly-covered weakness, the defense card played by player O will allow player P to play an additional attacking card. This ensures that the number of weakness cards held by player P decreases and the aforementioned strategy can be implemented

when there is only one weakness card remaining. If a player does happen to have only one weakness in a particular game state, it can provide a strong advantage and increase the likelihood of winning the game. As a result, a basic heuristic function may assign a high value to this type of game state.

```
function EvaluateState(gw: GameView)
        score = 0
        // 1) get the value of the hand only if the hand sizes are the
            same
        if gw.players[0].GetNumberOfCards() == gw.players[1].
            GetNumberOfCards()
                score = score + EvaluatePlayerHandToValue(gw)
        // 2) size of the hand: smaller -> better
        score = score + EvaluateHandSize(gw)
        // 3) weaknesses
        if gw.isEarlyGame is false
                score = score + EvaluateWeaknesses(gw)
        return score
```

Figure 4.8: Pseudocode of `basic` heuristic function

Figure 4.8 illustrates the basic evaluation function, which combines the values obtained from the various criteria discussed earlier to assign a single value to the game state. The function adds together the values calculated for each criterion, resulting in a single value that represents the estimated value of the state. This value is then returned to the minimax function, which can use it to evaluate the state and guide decision-making.

4.3.2 Playout Heuristic

In the minimax algorithm, the playout heuristic is an alternative evaluation function that is used to determine the potential outcome of a given game state. It is inspired by playouts as used in Monte Carlo tree search, which will be covered in a later section. Essentially, it involves creating a copy of the current state and simulating the play of the game between two greedy agents. While it is not problematic to simulate the game using random playouts, it is better to use greedy playouts due to their ability to more accurately mimic the optimal strategy and provide more reliable estimates of the expected outcome of a given state. The result of the simulation is used to assign a heuristic value to the game state, which can be used to guide the search process in minimax. The use of greedy agents in the playout heuristic leads to improved performance compared to basic heuristics, due to their ability to make effective and efficient decisions. An illustration of the playout heuristic function can be found in Figure 4.9. The outcome of the simulation, `Winner()`, can be represented by a value of 1 if player 1 wins, -1 if player 2 wins, or 0 if it is a draw. By incorporating this value with the current depth of the game state, it is possible to assign a heuristic value to the game state. This heuristic value is calculated by subtracting the current depth from 1000 and multiplying the result by the outcome of the game. This approach effectively prioritizes a shorter win over a longer one, as the size of the depth is taken into consideration.

```
function Evaluate(gw: GameView, depth: int)
        innerGameView = gw.Copy()
        agents = List of Agent containing GreedyAI("greedy") and
            GreedyAI("greedy")

        while innerGameView.status is GameStatus.GameInProcess
                turn = innerGameView.Turn()
                card = agents[turn].Move(innerGameView)
                innerGameView.Move(card)
        result = innerGameView.Winner()
        score = 1000 - depth
        return result * score
```

Figure 4.9: Pseudocode of playout heuristic function

4.3.3 Alpha-Beta Pruning

The minimax algorithm is capable of finding the optimal move in a two-player game by searching the entire game tree and considering all possible moves by both players. However, for some games, such as Durak, the game tree can be extremely large, making it computationally infeasible to fully explore. In these cases, the **alpha beta** pruning technique is used in conjunction with minimax to optimize the search process. Alpha beta pruning involves eliminating branches of the game tree that cannot affect the final decision, allowing the algorithm to focus its search on more promising areas of the tree and significantly reducing the overall computational burden [Russell et al., 2022]. Through comparison of the average time of the minimax algorithm with and without the implementation of alpha-beta pruning at various fixed depths in the perfect information environment, it is evident that the incorporation of alpha-beta pruning leads to a marked increase in efficiency (as depicted in Figure 3). At a depth of 6, the minimax algorithm incorporating alpha-beta pruning required an average of 10.6 milliseconds to explore the game tree, while the minimax without this optimization technique required an average of 258.2 milliseconds to do so. As anticipated, the use of alpha-beta pruning results in significantly faster performance compared to the minimax without this optimization technique.

The implementation of this technique, given in Figure 4.11, involves adding two additional parameters, “alpha” and “beta”, to the minimax function. These parameters are initialized to negative and positive infinity, respectively, indicating that any minimax value is acceptable. The algorithm then compares the values of alpha and beta to the current minimax value during the search, and if the value exceeds a certain threshold, the search is terminated as a game-winning move has been identified. Overall, the incorporation of alpha-beta pruning into the minimax algorithm involves only a minor modification to the existing code, making it a useful tool for optimizing the performance of minimax in certain situations.

Another method for optimizing the search of a game tree using the minimax algorithm was to implement state caching. This technique involves serializing the current game state with its current depth and storing its corresponding heuristic value in a cache. During the search process, the minimax agent can then check the cache before evaluating a given state. If the state has already been explored,

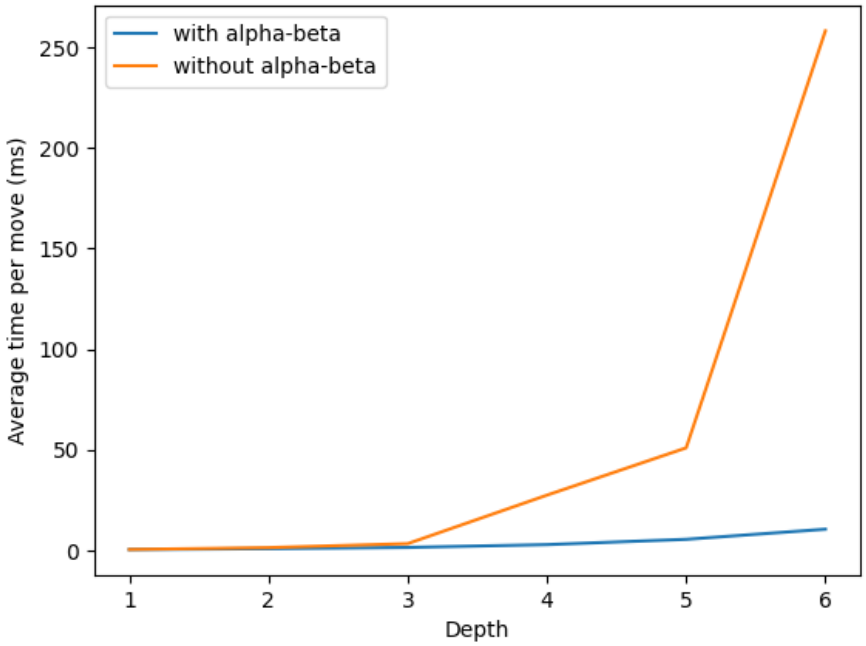

Figure 4.10: Comparison of the average time taken to make a move by the minimax algorithm with and without alpha-beta pruning at various depths in the perfect information environment.

the agent can retrieve the stored heuristic value and avoid the need to recalculate it, thereby reducing the overall computational complexity of the search. It was observed during the development of the program that the same game state may be encountered multiple times through different paths in the tree, making state caching an effective optimization technique for the minimax algorithm.

4.4 Monte Carlo Tree Search Agent

Monte Carlo Tree Search (MCTS) is a search algorithm that is commonly used in games such as chess, Go, and other board games to find the best move to make. It is based on the idea of using random sampling to estimate the value of different moves, rather than explicitly searching through the entire game tree [C.Browne et al., 2012].

The MCTS algorithm consists of four main steps at every iteration:

1. Selection: The algorithm starts at the root of the game tree and selects successive child nodes by using a heuristic evaluation function to guide the search towards promising areas of the tree.

2. Expansion: Once a leaf node (a node without any children) is reached, the algorithm creates one or more child nodes for that leaf node and performs a simulation from that point.

3. Simulation: The simulation involves playing out a random sequence of

```
function minimax(gameView, alpha, beta, depth, bestMove)
...
        bestVal = game.turn == 1 ? -infinity : +infinity

        possibleMoves = getPossibleMoves(gameView)
        for each move in possibleMoves
                gameViewCopy = copy(gameView)
                gameViewCopy.Move(move)
                v = minimax(gameViewCopy, alpha, beta, depth + 1, null)

                if game.turn == 1 && v > bestVal or game.turn == 2 && v
                    < bestVal then
                        bestVal = v
                        bestMove = move

                        if game.turn == 1 then
                                if v >= beta then
                                        return v
                                alpha = Max(alpha, v)
                        else
                                if v <= alpha then
                                        return v
                                beta = Min(beta, v)
...
```

Figure 4.11: Part of the pseudocode of alpha-beta pruning technique inside the minimax `Move` method

moves from the current position to the end of the game, using a simple evaluation function to determine the outcome.

4. Backpropagation: The results of the simulation are then propagated back up the tree, updating the values of the nodes visited during the selection and expansion steps.

The MCTS algorithm repeats these steps until a sufficient number of iterations (in this program 'limits') have been performed, at which point it selects the move with the highest estimated value as the best move to make. Considering the possibility of regrouping the four main steps in MCTS, this version of the algorithm works with two distinct policies:

1. Tree Policy: From the nodes contained within the search tree, either select a leaf node or create a new one by expanding the tree (selection and expansion).

2. Default Policy: Evaluate a non-terminal state in the domain by simulating its progression to a terminal state and generating an estimate of its value (simulation).

The backpropagation step does not involve the use of a policy itself, but rather adjusts the internal parameters of node statistics that inform future tree policy. These steps are summarized in the pseudocode in Figure 4.12.

```
function MCTSSearch(s0)
        create root node v0 from the tree with the state s0
        while limit has not reached do
                vl = TreePolicy(v0)
                reward = DefaultPolicy(s(vl)
                BackPropagation(vl, reward)
        return a(BestChild(v0, 0))
```

Figure 4.12: A general pseudocode for MCTS

v_0 is the root node that corresponds to the state s_0. `TreePolicy` returns the last node, v_l, reached either by expanding or selecting the node, which corresponds to the state s_l. `DefaultPolicy` method simulates the whole game given the last node and returns the outcome of the playout from the state s_l, which is `result`. Given the `result` from the simulated game, the `BackPropagation` method adjusts the values from v_l up until the root node v_0. Once the limit of computation is reached, the algorithm provides the result of the overall search `a(BestChild(`v_0`))` where `a` is the action that provides the best move in the state s_0 of the root v_0.

The UCT (Upper Confidence bounds applied to Trees) algorithm is a heuristic method for finding the optimal move in a two-player game by considering both the exploration of new nodes in the search tree and the exploitation of known information about the value of certain nodes. It is a method that allows to find the best node in the `TreePolicy` method and best move given the root node `BestChild` method. It is commonly used in situations where it is necessary to balance the trade-off between exploration and exploitation in order to make the most informed decision. By using this approach, it is possible to effectively navigate the search tree and identify the best possible move, taking into account the inherent uncertainty and complexity of the game at hand [C.Browne et al., 2012]. To calculate the UCT value for each node,

$$UCT(v) = 1 - \frac{Q(v')}{N(v')} + c * \sqrt{\frac{2 * lnN(v)}{N(v')}}$$

is used. Every node has the value of *Q(v)* that corresponds to the total reward of all playouts that passed through this state and the number *N(v)* of times it has been visited. The c value is the exploration constant that balances the exploration exploitation trade off. The reason why the 1 subtracts from fraction of *Q(v)* and *N(v)* is because we want to invert the win rate, i.e. the average reward, which will always be between 0 and 1. As an example, consider a scenario in a two-player game if a given move leads to an average reward of 0.3 for player 1, then the same move leads to an average reward of 0.7 for player 2.

It is important to note that the calculation mentioned above assumes that moves strictly alternate between the two players, such that the first player makes a move, followed by the second player, and then the first player again, and so on. However, it is important to note that the alternating move structure mentioned previously does not always hold true in the game of Durak. Specifically, in Durak, the moves do not necessarily alternate between players. For example, if the first player makes an attack as their first move and the second player decides to take

the cards, the first player may consecutively attack more until they no longer have the necessary card or simply choose not to attack any further. In this scenario, the move turn would not follow the alternating pattern described earlier. Because of that in cases where the moves do not strictly alternate between players, the win rate is not inverted as previously described. Instead, it is only inverted in cases where the turns alternate between players.

Now that we have a general understanding of MCTS and the Upper Confidence Bound applied to Trees (UCT) algorithms, it is time to explore the implementation of the TreePolicy, DefaultPolicy, and BackPropagation in these algorithms. The pseudocode of MCTS algorithm is depicted in Figure 4.14.

For each node v, there are four pieces of data associated with it: the state s(v) corresponding to the node, the action a(v) that led to the node, the total simulation reward Q(v), and the number of visits N(v) to the node represented as a nonnegative integer. The term `reward` indicates the outcome of the simulated game and `f(s(v), a)` represents the function that takes state of the node and action as arguments and produces the new state.

The return value of the overall search in this case is `a(BESTCHILD(`v_0`, 0))` which will give the action `a` that leads to the child with the highest reward, since the exploration parameter `c` is set to 0 for this final call on the root node v_0. [C.Browne et al., 2012].

It should be noted that the simulation process in the `DefaultPolicy` follows a random path through the search tree. While this approach allows for a broad exploration of potential moves, it may result in slower convergence and lower accuracy compared to using a more informed strategy for selecting simulation moves. In the program, it is possible to choose the simulation type when running experiments using MCTS. The options include using a random simulation (simulation='random') or using a more informed strategy (simulation='greedy'). The informed strategy employs a greedy agent to make decisions, which tends to guide the search more effectively towards areas of the tree that are more promising.

```
private void Backpropagation(Node nodeToExplore, int playoutResult)
{
        while(tempNode != null)
        {
                tempNode.IncrementPlayouts();
                // draw
                if (playoutResult == -1)
                {
                        tempNode.AddScore(0.5);
                }
                else if (tempNode.GetGame().Player() == playoutResult)
                {
                        tempNode.AddScore(1.0);
                }
                tempNode = tempNode.GetParent();
        }
}
```

Figure 4.13: A snippet of Backpropagation method

Upon receiving the reward result from the `DefaultPolicy`, the program passes

this information to the `Backpropagation` function in order to update the values of the nodes in the path traversed during the simulation. These values include the visit count and the reward count, also known as the win count. In zero sum games, where one player's win corresponds to the other player's loss, the `Backpropagation` method increments the score of the winning player. However, in the card game Durak, there is the possibility of a draw between two players. To account for this possibility and ensure consistency in the scores, the `Backpropagation` method has been modified to include the case of a draw in its update process. The figure 4.13 illustrates the version of `Backpropagation` function that includes the case of draw. Specifically, the algorithm now awards a score of 0.5 to nodes in the case of a draw and a score of 1.0 in the case of a win.

4.5 Closed World Sampling

It is well-known that algorithms such as Minimax and Monte Carlo Tree Search (MCTS) can provide optimal play in games by thoroughly exploring the game tree. However, in imperfect information games such as Durak, this is not considered fair play because it allows the agent to access hidden information that it would not normally have access to. One way this can be done is through the use of a cloning method, which allows the agent to create a copy of the current game state and explore the potential outcomes of different moves. While this is acceptable in games with perfect information, such as chess or checkers, it is considered cheating in games with hidden information.

One way to address the issue of hidden information in imperfect information games is to use a **sampling** method that randomly generates a game state based on the information that the player can currently observe. This allows the agent to explore and evaluate potential moves without access to hidden information. The agent can then repeat this process a number of times, using algorithms such as Minimax or MCTS to determine the best course of action in each sampled game state. By considering the outcomes of these randomized scenarios, the agent can make a decision that is likely to be successful across a range of potential game states. This approach is called "Perfect Information Monte Carlo" [Long et al., 2010].

To implement a closed world sampling method in the game of Durak, a shuffled copy, `ShuffleCopy`, method was developed. In this game, the hidden information in the early stages of play includes the cards in the deck and the opponent's hand. To generate a randomly shuffled game state, the `ShuffleCopy` method combines these two sources of hidden information and shuffles them, before redistributing the same number of cards to the player and the deck. This process is represented in the Figure 4.15. This shuffled game state is then returned to the calling agent, which can be either a Minimax or a MCTS. Thanks to that the shuffling process preserves the hidden information while allowing the agent to determine the best move to make in the current game state.

The number of samples to be generated by the agents is determined by the user. A larger number of samples can lead to a more accurate result, but it also increases the time required to find the best move. To determine the optimal move after a certain number (n) of samples have been generated, the agents can keep track of the best moves identified in each shuffled game state. As the agents,

such as Minimax and MCTS, evaluate each shuffled game state, they can store the results in a cache that tracks the frequency of each move. Once all n samples have been played, the agent can select the move that was identified as the best option in the greatest number of these scenarios.

4.5.1 Parameter Specification

Minimax

In order to conduct an experiment using the minimax agent, it is necessary to specify certain parameters, which have been informed by our understanding of the minimax agent. Specifically, it is necessary to specify the value of the `depth` parameter, which determines the depth to which the search tree will be explored in order to identify the optimal move. In addition to the depth parameter, it is also necessary to specify the `eval` parameter, which specifies the heuristic function used to evaluate the state when the maximum depth has been reached. The `eval` parameter can take on either the value `basic` or `playout`. It is worth noting that, in a closed world scenario, the `samples` parameter is optional and has a default value of 20. The `samples` parameter serves to prevent agents such as Minimax and MCTS from cheating by accessing information about future states of the game (details in Section 4.5). This illustration presents an example of a simulation between the Minimax and greedy agents with arbitrary parameter values in the open and closed world:

Open-world:

```
dotnet run -ai1=minimax:depth=4,eval=basic -ai2=greedy -open_world
```

Closed-world:

```
dotnet run -ai1=minimax:depth=6,eval=basic,samples=15 -ai2=smart
```

MCTS

Regarding MCTS,it is necessary to specify the value of the `limit` parameter, which determines the computational budget allocated for the algorithm to build the search tree. The search is halted and the best-performing root action is returned once this budget is reached. The `c`, which is called exploration constant, parameter also needs to be specified because it determines the balance between exploitation and exploration in the UCB equation. This parameter is set to a default value of $\sqrt{2}$ (≈ 1.41) and can be adjusted to fine-tune the performance of the algorithm. The `simulation` parameter should also be specified, indicating whether to use a smart simulation (`greedy`) or a random simulation (`random`). Lastly, in a closed world setting, just like in Minimax, it is necessary to specify the `samples` parameter to prevent the MCTS agent from cheating by considering future states of the game. This example demonstrates a simulation between MCTS and greedy agents with arbitrary parameter values in open and closed world settings:

Open-world:

```
dotnet run -ai1=mcts:limit=100,simulation=greedy -ai2=greedy
                                                          -open_world
```

Closed-world:

```
dotnet run -ai1=mcts:limit=100,simulation=greedy,samples=10
                                                          -ai2=greedy
```

```
function MCTSSearch(s_0)
        create root node v_0 from the tree with the state s_0
        while limit has not reached do
                v_l = TreePolicy(v_0)
                reward = DefaultPolicy(s(v_l)
                BackPropagation(v_l, reward)
        return a(BestChild(v_0, 0))

function TreePolicy(v)
        while v is nonterminal do
                if v is not fully expanded then
                        return Expand(v)
                else
                        v = BestChild(v, c)
        return v

function Expand(v)
        get  a ∈ first untried action from A(s(v))
        add a new child v' to v
                with s(v') = f(s(v), a)
                and a(v') = a
        return v'

function BestChild(v, c)
        return    arg max_{v' ∈ childrenofv}  UCTValue(v, v', c)

function UCTValue(v,v',c)
        avgReturn =  Q(v')/N(v')
        if Turn(v) != Turn(v') then
                avgReturn = 1 - avgReturn
        return  avgReturn + c * sqrt(2*lnN(v)/N(v'))

function DefaultPolicy(s)
        while s is non-terminal do
                choose  a ∈  A(s) uniformly at random
                s = f(s,a)
        return reward for state s

function Backpropagation(s, reward)
        while v is not null do
                N(v) = N(v) + 1
                Q(v) = Q(v) + reward
                v = parent of v
```

Figure 4.14: A MCTS Pseudocode

```
public Durak ShuffleCopy()
{
        // perform copy
        Durak copy = (Durak)this.MemberwiseClone();
        ...

        Player opponent = copy.players[GetNextTurn()];
        var opponentHiddenCards = opponent.GetHand().Where(c => !c.
            GetSeen());
        int totalHiddenCards = opponentHiddenCards.Count();

        // add hidden cards from opponent's hand to the hidden deck
            cards
        copy.deck.GetCards().AddRange(opponentHiddenCards);
        // remove hidden cards from the hand
        opponent.GetHand().RemoveAll(card => !card.GetSeen());

        // shuffle the unseen cards
        copy.deck.Shuffle();
        ...

        return copy;
}
```

Figure 4.15: A snipped of code that demonstrate the shuffling of the cards from the deck and opponent's hand inside the ShuffleCopy method

5. Experiments

After providing an overview of the game of Durak and introducing the agents under consideration, we will now proceed to conduct experiments to determine the optimal agent for this game. We will conduct experiments in both open and closed world environments. Through these experiments, we will gain a better understanding of the adaptability and effectiveness of the agents in both environments, and determine whether the agents that performed best in perfect information scenarios are able to achieve similar success in imperfect one.

To conduct our experiments, we will use the `-tournament` parameter available in the command line interface (as described in Section 3.2.3). This facilitates our experimentation by allowing us to configure the participating AI agents and their respective parameters, as well as the environment for the tournament. In order to compare the agents in both open and closed world scenarios, we will present the results of these experiments in separate sections. The tournament will consist of full games with trump cards, and so the respective parameters `-include_trumps` and `-start_rank=6` will be set accordingly. In order to strike a balance between accuracy and efficiency, we will set the `-total_games` parameter to 500. This number of games allows us to identify the strongest and weakest agents with a sufficient level of confidence, while not unduly prolonging the experiment. However, as previously mentioned in Section 3.2.3, if the total number of games is not sufficient to confidently determine the winner, the program will increment the total number of games by 500 until the maximum number of games, 10 000, is reached. This process ensures that we are able to accurately identify the top performing AI agent while still minimizing the overall length of the experiment.

5.1 Open world

With the game environment parameters set to be in the perfect information world, we will proceed to configure the parameters for the various agents. Some agents, such as Random, Greedy, and Smart, do not require any additional parameters as their strategies are simple and do not involve game tree search. However, Minimax and MCTS, as described in Sections 4.3 and 4.4 respectively, do require the configuration of certain parameters in order to perform at their best. In order to determine the optimal values for these parameters, we will run a set of games to identify the values that best suit the tournament environment before comparing with other agents.

5.1.1 Configuring Minimax Parameters

In order to optimize the performance of the Minimax algorithm in an open world environment, it is necessary to determine the optimal values for the `depth` and `eval` parameters. To find the best value for one of the parameters, the other one has to be set with the value that intuitively seems better suited. Because of that we will use the playout heuristic for `eval` parameter as it is generally superior to the basic heuristic. To find the best value for the `depth` parameter, I run an 1000 game experiment that tries different values for it against the Greedy agent.

Figure 5.1 illustrates the win rates and the average time taken per ply by the minimax agent with different depth parameters. The plot shows that the win rate of the minimax agent improves as the depth of the game tree exploration increases. The same trend can be observed for the average time taken to make a move. However, it is surprising to see that lower depth values such as 2 and 3 have higher win rates than the highest depth value of 6, which has the lowest win rate. This observation warrants further investigation. Besides this, a horizontal shaded line is included in the plot to represent the optimal average time that the agent should take to make a move. This allows for the evaluation of both the effectiveness and efficiency of the agent. It can be observed that the grey shaded line intersects with the average time of the minimax agent at a depth of 9. This suggests that, in an open environment, this value of the parameter is the most suitable and will be used in tournament.

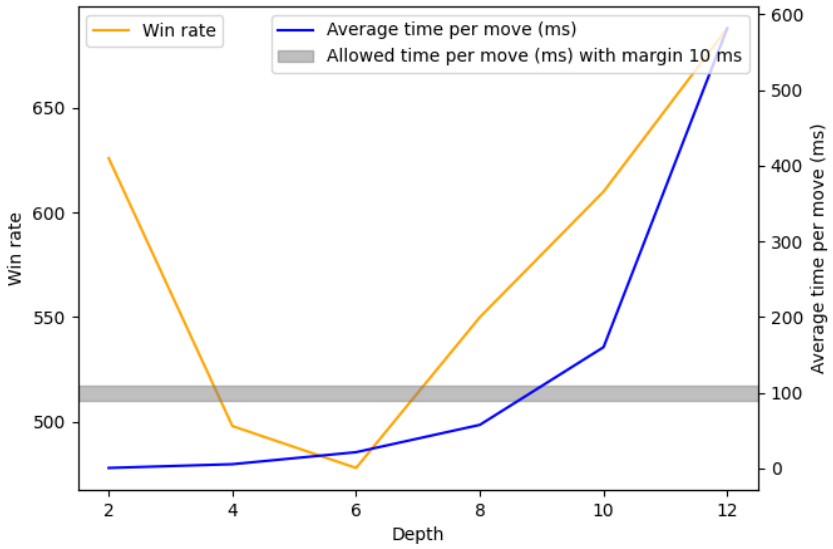

Figure 5.1: Comparison of win rates of different fixed depths parameters in minimax algorithm along with average time taken per ply

Figure 5.2 illustrates the comparison between two heuristic evaluation functions, namely `basic` and `playout`, for the minimax algorithm with a fixed depth of 9. The left subplot shows the win rates for the two evaluation functions, and the right subplot displays the average time (ms) taken to make a move. It can be observed that the playout heuristic consistently outperforms the basic heuristic in terms of win rate, achieving almost twice the win rate of the basic heuristic. However, the playout heuristic is slower than the basic heuristic, as shown by the right subplot. Despite this, the playout heuristic still performs within the optimal time range of 100ms, making it the preferred choice for the tournament. As a result, the minimax agent will use the depth 9 and playout heuristic for the tournament.

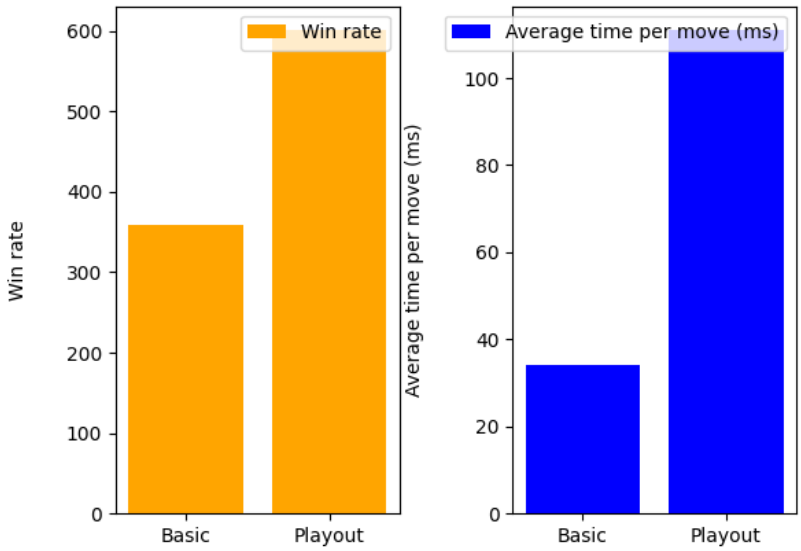

Figure 5.2: Comparison of win rates and average time taken per ply of `basic` and `playout` evaluation functions in minimax algorithm

5.1.2 Configuring MCTS Parameters

In order to compare MCTS agents with other agents in the open world, it is necessary to determine the optimal values for the parameters `limit`, `c`, and `simulation`. To begin, the exploration constant was set to 1.41, which is typically the optimal value, and the simulation was set to greedy. With these parameters in place, an experiment was conducted using 1000 games to find the value of `limit` by trying various values of iterations to identify the optimal value. The results of this experiment are depicted in Figure 5.3.

Similar to the minimax algorithm, it can be seen that an increase in the number of `iterations` generally results in a corresponding increase in the average time taken per move, as well as an increase in the win rate. By examining the shaded region in Figure 5.3, which represents the optimal time per move, it can be determined that the optimal value for the `iterations` parameter is 800. This value results in the highest win rate and the lowest average time taken per move.

With the `iterations` parameter set to 800 and the `simulation` value remaining at greedy, a new experiment was conducted to determine the optimal value for the `c` exploration parameter in MCTS. The results of this experiment are shown in Figure 6. From the figure, it can be observed that the optimal value for the `c` parameter is 1.00, as it yields the highest win rate while still operating within the optimal average time per move.

5.2 Closed world

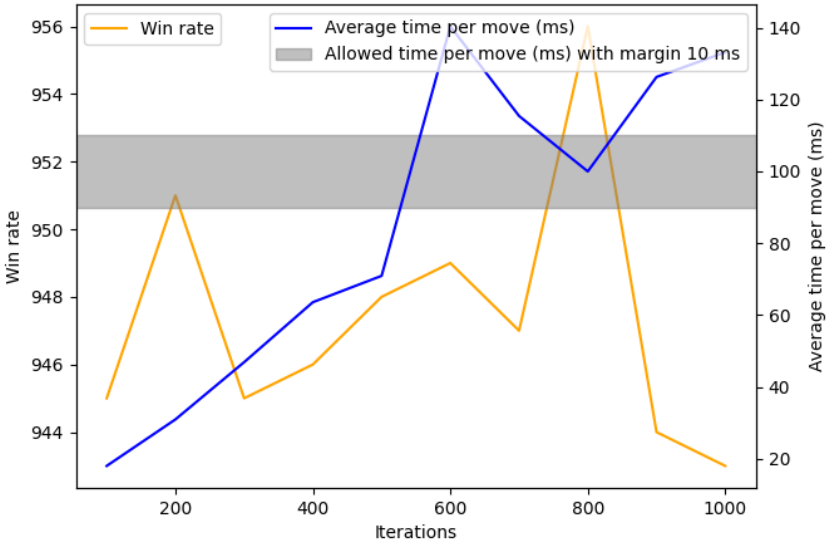

Figure 5.3: Comparison of win rates and average time taken per ply of different fixed iterations in MCTS algorithm

Conclusion

Future Work

a

Bibliography

É. Bonnet. The complexity of playing durak. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2016.

C.Browne, E. Powley, and et al. D. Whitehouse. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/tciaig.2012.2186810.

J. Long, N. Sturtevant, M. Buro, and T. Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):134–140, 2010. doi: 10.1609/aaai.v24i1.7562.

J. McLeod, Jun 2018. URL `https://www.pagat.com/beating/durak.html`.

J. McLeod, Mar 2022. URL `https://www.pagat.com/beating/podkidnoy_durak.html`.

I. Millington and J. Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, 2009.

D. Moor and B. Craig G. McCabe. *Introduction to the practice of Statistics*. W. H. Freeman, 2017.

G. Owen. *Game theory*. Academic Press, 2008.

S. Russell, P. Norvig, and M. Chang. *Artificial Intelligence: A modern approach*. Pearson, 2022.

D. Silver, A. Huang, and et al. J. Maddison. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

List of Figures

List of Tables

List of Abbreviations

A. Attachments