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.

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 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 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, 2022a]. It is played with a deck of cards and typically involves two to six players. Unlike the other games, the aim of the 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 is to ensure its correct development 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, 2022b]. 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.

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. This decision is made 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 it was mentioned before, this game is characterized by the absence of a winner, with only a loser remaining at the end.

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 multifunctional game that requires players to consider a range of factors in order to play effectively. This complexity is a key characteristic of the game, and contributes to its strategic depth and appeal. 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 and examining the complexity of the game as a whole.

2.1 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 [Guillermo, 2013]. 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 [Guillermo, 2013]. 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 [Guillermo, 2013]. 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 can be classified as a **non-deterministic** game due to the presence of elements of randomness that can affect the outcome of the game. 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 [Guillermo, 2013]. In case of the game, the distribution of cards at the beginning and the order in which cards are played can both be considered elements of randomness that can affect the outcome of the game.

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

2.2 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.

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 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

The game framework for Durak was tested on both 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. It is possible that the framework will operate correctly on other platforms, but this has not been verified.

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 Core 6 platform. C# was selected as the programming language for the implementation of aforementioned projects in Durak due to its suitability for writing back-end systems. The language offers a range of reliable libraries and benefits from a highly optimized Just-In-Time (JIT) compiler, resulting in enhanced speed. These attributes 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 and opened with an Integrated Development Environment (IDE) of user's choice. From the root directory of the project, the user can utilize the command line to enter the command

```
dotnet run --project CLI
```

, which will launch the Command-Line Interface (CLI) and provide guidance on how to proceed with experimentation. The CLI will provide further instructions on how to use and test the program (for additional information, please refer to Section 3.2.3.).

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 contains the game logic for Durak.

- Agent - A C# library contains all of the implemented AI agents.
- CLI - A C# Command-Line Interface (CLI) project 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 Game Model

The game model, which represents the current state of the game, is implemented using object-oriented programming principles. As it 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 each of which is equipped with the necessary methods to support the game's functionality. These objects and their methods are designed to reflect the key components and features described in the game description.

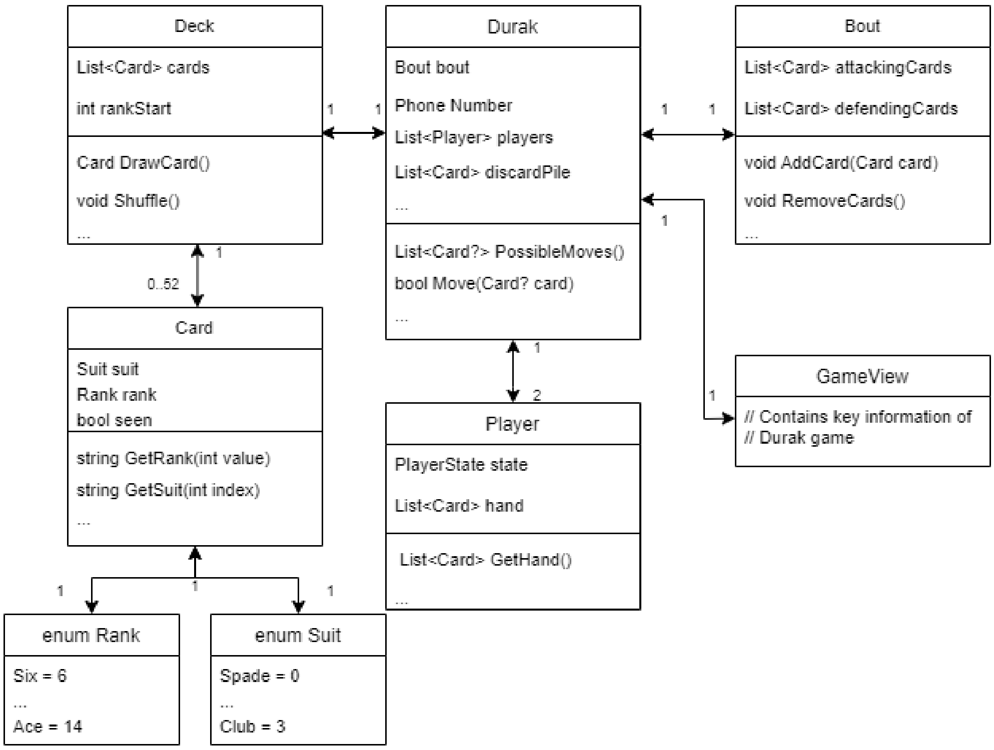

Figure 3.1: A simplified UML diagram showing the relationship between the objects within the Model library.

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 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. Because of that it is utilized to communicate this information to other components within the system, such as the command-line interface (CLI) or artificial intelligence (AI) scripts. 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 through which changes can be made to the game state, and as such, play a crucial role in the overall operation of the model.

```
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

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. This enables the method to adapt to the specific circumstances of the game and provide appropriate options for the current player to make a move.

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` 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.

```
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;
        }
}
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;
        }
}
turn = turn == Turn.Attacking ? Turn.Defending : Turn.Attacking;
```

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

Additionally, it is important to note that, for security purposes, the agents are not provided with the entire `Durak` object. Instead, they are given access to a `GameView` class representation, which allows them to obtain essential information about the current game state and make changes through the methods outlined in Figures 3.3 and 3.4. This approach ensures that the agents are not able to manipulate the game in an unauthorized manner.

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. The flexibility and versatility of the CLI is crucial for exploring the capabilities of the system and evaluating the performance of the AI agents. In this chapter, we will examine the various parameters that can be used to manipulate the behavior of the agents and the game environment through the CLI. This will provide insight into the capabilities of the system and enable a more thorough evaluation of the AI agents' performance.

Before discussing the organizational structure of the project, it is important to introduce the parameters and their roles within the project (Please refer to Figure 3.5). This will facilitate a better understanding of the various components and how they interact with one another.

```
/ai1             : The agent for player 1. (String)(Default = random)
/ai2             : The agent for player 2. (String)(Default = random)
/config          : Used for grid search parameter configuration.(Default = False)
/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)
```

Figure 3.5: A text-based 'Help' statement that describes the parameters that can be used

There are various ways to utilize the aforementioned parameters. An example of using the default settings to run a game between RandomAI agent and GreedyAI agent is provided below:

```
dotnet run --project CLI -ai1=random -ai2=greedy
```

This command initiates the simulation of 1000 games between the RandomAI and GreedyAI agents in a fully enclosed environment (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 the 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 --project CLI -ai1=random -ai2=greedy -verbose -seed=1
```

The command above generates a 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♠
...
```

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.

In the context of the Minimax algorithm, 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" (additional information on these parameters can be found in Section 4.3). 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). 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 --project CLI -ai1=minimax:depth=4,eval=basic -ai2=greedy -open_world
```

Closed-world:

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

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(detailed information is provided in Section 4.4). 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 --project CLI -ai1=mcts:limit=100,simulation=greedy -ai2=greedy
                                                         -open_world
```

Closed-world:

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

3.2.4 AI Agents

4. Artificial Intelligence Agents

4.1 Random Agent

4.2 Rule-Based Heuristic Agents

4.2.1 Greedy Agent

4.2.2 Smart Agent

4.3 Minimax Agent

4.4 Monte-Carlo Tree Search Agent

5. Experiments

5.1 Title of the first subchapter of the second chapter

5.2 Title of the second subchapter of the second chapter

Conclusion

Bibliography

O. Guillermo. *Game Theory*. Emerald Group Publishing Limited, 4 edition, 2013. ISBN 978-1781905074.

J. McLeod. Durak (fool), March 2022a. URL `https://www.pagat.com/beating/durak.html`.

J. McLeod. Podkidnoy durak, March 2022b. URL `https://www.pagat.com/beating/podkidnoy_durak.html`.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 User Documentation

The purpose of this attachment is to provide a user-friendly introduction to the game and its features. It is written in a way that is accessible to individuals without a technical background, enabling them to test the game without necessarily needing to develop AI for it. This attachment aims to furnish these users with the necessary knowledge and understanding of the game and its functions.

A.1.1 Requirement

In order to run various commands in Durak, the Windows 10 or Linux operating system is a necessary requirement. Additionally, the following software must also be installed:

A.2 Developer Documentation