WARSAW UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering

Monte Carlo Tree Search and Reinforcement Learning methods for multi-stage strategic card game

mgr inż. Konrad Godlewski

Doctoral Thesis
supervised by dr hab. inż. Bartosz Sawicki

October 2022

Abstract

Doctoral thesis

Monte Carlo Tree Search and Reinforcement Learning methods for multi-stage strategic card game

mgr inż. Konrad Godlewski

The thesis presents the machine learning methods used to create agents for The Lord of the Rings card game (LOTRCG). The distinguishing features of LOTRCG is the non-competitive nature and deep strategic dimension of the gameplay. The player makes decisions at five, well-defined moments during a round. Actions are sequential, meaning that a decision made in a particular game phase impacts a decision in the next phase. This mechanism of the round places emphasis on strategic card management. In addition, random events in the form of card discovery occur between decisions. These events prevent simplifications of the gameplay mechanism by combining several decision moments into one.

LOTRCG is a collectable card game, which means a wide selection of cards available to the player. These cards have different statistics and special abilities profiling them for a particular phase of the game. The wide variety of cards translates into the high popularity of this title among card game enthusiasts. This element determining the game's appeal also poses a major challenge for AI agents.

The complex nature of the round is the first object of research of this dissertation. A random agent was implemented to analyze the different phases of the game. The obtained results made it possible to identify key decision moments. In addition, the random agent was used to perform sampling during iterations of the MCTS algorithm.

The dissertation uses two families of computational intelligence techniques. The first is Monte Carlo Tree Search (MCTS). This method is based on a heuristic search that uses random sampling of game states. MCTS stores gameplay as a tree, which is expanded iteratively. The MCTS algorithm was extended to include action reduction, which reduces the tree size. This modification was based on expert knowledge, which allows the elimination of cards with low utility value. In addition, optimization of MCTS hyperparameters was carried out not only in terms of efficiency but also in terms of computational

time. In this way, an optimal set of MCTS agents set was constructed, which achieved 82.8% of winrate.

The second family of techniques is reinforcement learning (RL). Reinforcement learning is based on a trial-and-error method in which the agent interacts with the environment. The agent receives an observation of the state of the game, and then chooses an action, which is executed in the environment. The environment returns the reward and the next observation. The agent's goal is to maximize the sum of rewards throughout the episodes. Due to the agent's operation in an environment with a variable number of actions, the selection of RL algorithms was based on how the actions were encoded. Two types of coding were introduced. The first is macro actions, which allow a fixed number of actions. The second type is direct action choice, which is an agent's playing of individual cards. Q-Learning and Actor-Critic (AC) algorithms were implemented for macro-actions. Q-Learning is a widely used solution in the RL field that approximates the utility value of a game state. Actor-Critic uses an approximation of both utility and strategy functions. The AC agent is programmed for macro-actions as well as direct actions. As a result of the optimization, the best agents RL set showed an efficiency of 95.3%.

The paper compares the best MCTS and RL agents. The conducted experiments show a significant advantage of reinforcement learning, but this solution requires a long learning time, which strongly depends on the available computational power. MCTS seemingly has fewer requirements, but the decision time is much longer.

Streszczenie

Rozprawa doktorska

Metody Monte Carlo Tree Search oraz Reinforcement Learning w wieloetapowej strategicznej grze karcianej

mgr inż. Konrad Godlewski

Rozprawa przedstawia metody uczenia maszynowego wykorzystane do stworzenia agentów do gry karcianej "The Lord of the Rings: The Card Game" (LOTRCG). Cechą wyróżniającą grę jest nierywalizacyjny i strategicznie pogłębiony charakter rozgrywki. Gracz podejmuje decyzje w pięciu, ściśle określonych momentach podczas rundy. Akcje mają charakter sekwencyjny, czyli decyzja podjęta w danej fazie gry wywiera wpływ na decyzję w etapie kolejnym. Taki mechanizm rundy kładzie nacisk na strategiczne zarządzanie kartami. Dodatkowo pomiędzy decyzjami występują zdarzenia losowe, które uniemożliwiają uproszczenie rozgrywki poprzez połączenia kilku momentów decyzyjnych w jeden.

LOTRCG jest grą typu collectible card game, co znacza szeroki wybór kart dostępnych dla gracza. Karty posiadają różne statystki oraz zdolności specjalne profilujące je do konkretnej fazy gry. Duża różnorodność kart nadaje przekłada się na dużą popularność tego tytułu wśród miłośników gier karcianych. Ten element decydujący o atrakcyjności gry stanowi zarazem duże wyzwanie dla agentów AI.

Złożony charakter rundy jest pierwszym obiektem badań tej rozprawy. W celu analizy poszczególnych faz gry zaimplementowano agenta strategii losowej. Uzyskane przy jego pomocy wyniki pozwoliły określić kluczowe momenty decyzyjne. Ponadto agent losowy posłużył do wykonywania losowego próbkowania drzewa algorytmu MCTS.

Rozprawa wykorzystuje dwie rodziny technik inteligencji obliczeniowej. Pierwszą z nich jest Monte Carlo Tree Search (MCTS). Metoda ta opiera się na wyszukiwaniu heurystycznym korzystającym z losowego próbkowania stanów gry. MCTS przechowuje rozgrywkę w postaci drzewa, które rozbudowywane jest w sposób iteracyjny. Podstawowa wersja algorytm MCTS została rozbudowana o redukcję akcji, która pozwala na ograniczenie rozmiarów drzewa. Modyfikację tę oparto o wiedzę ekspercką, która

pozwala na eliminację kart o niskiej użyteczności. Dodatkowo przeprowadzono optymalizację hiperparametrów MCTS nie tylko względem skuteczności, ale także pod kątem czasu obliczeniowego. W ten sposób uzyskano optymalny zestaw agentów MCTS, który osiągnął współczynnik 82.8% wygranych.

Drugą rodziną analizowanych technik jest uczenie ze wzmocnieniem (Reinforcement Learning - RL). Uczenie ze wzmocnieniem opiera się na metodzie prób i będów, w którym agent oddziaływuje ze środowiskiem. Agent otrzymuje obserwację stanu gry, następnie wybiera akcję, która zostaje wykonana w środowisku. Środowisko zwraca nagrodę oraz następną obserwację. Celem agenta jest maksymalizacja sumy nagród na przestrzeni całego episodu. Z uwagi na działanie agenta w środowisku o zmiennej liczbe akcji, selekcję algorytmów RL oparto na sposobie kodowania akcji. W pracy stworzono dwa typy kodowania. Pierwszy z nich to makroakcje, które pozwalają na uzyskanie stałej liczby akcji. Drugim typem są akcje bezpośrednie, czyli zagrywanie poszczególnych kart przez agenta. Dla makroakcji zaimplementowano algorytmy Q-Learning oraz Actor-Critic (AC). Q-Learning to szeroko stosowane rozwiązanie w dziedzinie RL aproksymujące wartość użyteczności stanu gry. Actor-Critic korzysta z aproksymacji zarówno funkcji użyteczności jak i strategii. Agenta AC zaprogramowano do makroakcji jak i akcji bezpośrednich. W wyniku przeprowadzonej optymalizacji najlepszy zestaw agentów RL wykazał się skutecznością na poziomie 95.3%.

W pracy zestawiono najlepszych agentów MCTS oraz RL. Przeprowadzone eksperymenty wskazują na istotną przewagę uczenia ze wzmocnieniem. Jednak rozwiązanie to wymaga dużego nakładu czasowego na etapie uczenia, który silnie zależny jest od dostępnej mocy obliczeniowej. MCTS ma pozornie mniejsze wymagania, jednak czas decyzji jest znacznie większy.

Acknowledgements

I would like to express my very great appreciation to Bartek for his inspirations, endless patience and criticism, which made this thesis come true.

Contents

Abstract i

Acknowledgements v

Minidictionary viii

1 Introduction 1
- 1.1 Background of AI in card games . . . 3
- 1.2 Aim and Motivation . . . 6
- 1.3 Thesis structure . . . 7

2 AI in card games 8
- 2.1 Monte Carlo Tree Search . . . 8
- 2.2 Reinforcement Learning . . . 12

3 The Lord of the Rings: The Card Game 16
- 3.1 Game Description . . . 16
- 3.2 Cards Description . . . 17
- 3.3 Game Rules . . . 18
  - 3.3.1 Win/Lose Conditions . . . 20
- 3.4 Game Model . . . 21
- 3.5 Random Agent . . . 22
- 3.6 Study of Game Complexity . . . 25

4 Development of MCTS Agent 28
- 4.1 Tree Policy . . . 28
- 4.2 Rollout Policy . . . 30
- 4.3 Depth Limited Search . . . 31
- 4.4 Action Reduction . . . 31

5 Development of RL Agent 34
- 5.1 Main Sequence . . . 34
- 5.2 Simulator . . . 35
- 5.3 State Encoders . . . 37
- 5.4 Action Decoders . . . 39
  - 5.4.1 Direct Method . . . 39
  - 5.4.2 Macroactions . . . 40
- 5.5 Q-Learning Agent . . . 41

5.6 Actor-Critic Agent . . . 42
5.7 Hyperparameter optimization . . . 45

6 Experiments **47**
6.1 AI Setup . . . 47
6.2 MCTS . . . 48
6.3 RL . . . 51
6.3.1 Q-learning with macroactions . . . 52
6.3.2 Actor-critic with macroactions . . . 53
6.3.3 Actor-critic with direct card choice . . . 54
6.4 Best Players . . . 56
6.5 Conclusions . . . 58
6.5.1 Future . . . 58

7 Summary **59**
7.1 Overview of Research Effort . . . 59
7.2 Main Achievements . . . 61
7.3 Perspective Research . . . 62
7.4 Final note . . . 63

A Table of cards **64**

Bibliography **65**

Minidictionary

AI setup	combination of AI agents for decisions in one round of the game;
branching factor	average branching degree of vertices in the directed acyclic graph;
CCG	Collectible Card Game - a type of card game, which is available in a series of sets;
core set	complete set of cards allowing to start the game;
decision stage	a moment in the game, in which a decision must be taken;
deck-building	mechanics featured by the game that allows to play using your own deck of cards;
game phase	set of consecutive activities defined by rules of the game;
LOTRCG	The Lord of the Rings: The Card Game;
meta-game	knowledge base derived from statistics of players in a game;
playout	a single simulation of the game until termination state is reached;
progress points	points required to fulfill a scenario;
sphere	unit of card affiliation in LOTRCG;
winrate	number of wins to number of trials ratio;
vanilla MCTS	base version of Monte Carlo Tree Search algorithm;

Chapter 1

Introduction

Since the development of the first computers, their use in the gaming world has begun. The initial motivation was the desire for a short break from the routine of numerical calculations that consumed most of the time of the first electronic machines. The games are meant to provide intellectual entertainment, be the product created for relaxation or curiosity. Alan Turing (1912-54), a British mathematician and computer scientist, took a step further and, in 1953, published an essay about a machine capable of playing chess [1]. This article showed that games could present interesting challenges to computers with prospects of successfully competing with humans, just as happened 50 years later with Chess, Go, and Poker.

Let's look briefly at the history of the games, and what will take us to the modern strategic card game, which is the main subject of the thesis. As a recreational activity, games have accompanied humankind for thousands of years. The earliest archaeological traces of a board game are dated back to Neolithic times and were found in the Middle East. The board was made of flat stone with two or three carved parallel lines. On these lines, small hollows were carved. Unfortunately, the purpose of this game, as well as the rules, remain unknown. Some researchers suggest that this is the ancestor of Mancala, a game contemplated by African societies to this day. Fortunately, several games have survived until our times, the origins of which date back to ancient times. The Royal Game of Ur comes from the Sumerian culture. In Egypt during the Pharaonic Era, Senet was popular. Its rules are still a subject of debate. The Romans played Duodecim Scripta, an alleged prototype of backgammon. Pachisi came from India and it is predesessor of modern *Ludo* and *Mensch argere Dich nicht*. Go was invented in China. Nowadays, the game is played both recreationally and professionally.

Traditional board games can be divided into the following types [2]:

- Race Games - players compete in getting all their pieces across the board. Examples are Royal Game of Ur, Duodecim Scripta, Pachisi,
- Position Games - these games involve placing the pieces on the board to cover a larger area than the opponent. An example here is Go,
- War Games - the goal of the game is to destroy the opponent's most important pawn, like the king in Chess,
- Count and Capture Games - players seed stones on the board in order to capture as many opponent's counters as possible. An example is Mancala.

The first card games probably originated in China in the 9th century AD. They came to Europe with the influence of the Mameluke Empire, which controlled current Spain in the Middle Ages. The earliest European description of a 52-element deck dates from the 14th century, and from this period, the first references to tarot cards appeared in Italy. The Renaissance brought the birth of Whist, the first card game played in tricks. Whist later evolved into Bridge, which dates to the 19th century. Also from that century came early versions of poker, which was played in the United States before the Civil War. The 20th century marked the beginning of digital card games, which still have been intensively developed today.

In terms of game mechanics, we can divide card games into [1]:

- Trick-taking Games - players play one card on each turn, and the card with the highest value wins the trick. Examples: Whist, Bridge, Spades, Hearts, Tarot.
- Shedding Games - the goal is to get rid of cards from the hand as quickly as possible. An example is Crazy Eights.
- Solitaire - a single-player game where the goal is to arrange the cards in a specific order. An example is Klondike Solitaire.
- Comparing Games - Players draw cards from their hands, the most valuable combination of cards wins. Examples: Poker, Blackjack.
- Collectible Card Games (CCG) - are games with a dedicated set of cards much larger than the standard 52-element deck. Games are published in a series of sets. Players build their decks and duel with each other, as in Magic the Gathering or Pokemon. CCGs can also feature cooperative gameplay like The Lord of the Rings: The Card Game (LOTRCG).

[1] David Parlett, https://www.britannica.com/topic/card-game

There are thousands of today's games that can be divided on different criteria. From the point of view of artificial intelligence, it is essential to split the domain into perfect and imperfect information games.

Perfect information means that players can observe a complete game state, for example, the whole board with the opponent's pawns. Thanks to this game formulation, the AI agent can theoretically determine all available moves of the opponent under given circumstances. This information eases the planning of subsequent actions. The most popular games with complete information are Chess, Go, and Gaps Solitaire. For both of these games, artificial intelligence has reached a master level, for Chess in the famous duel with Gary Kasparov in 1997 [3], while for Go in 2016 [2]. Other examples of games with perfect information include checkers.

If there are hidden elements in the game, such as cards in the opponent's hand or stochastic elements like a dice roll, we can talk about imperfect information. The agent planning his moves does not know what answers the opponent may prepare. Classic examples here are Bridge or Poker.

Another aspect of card games is gameplay type. The game can be played in either a cooperative or competitive manner. In cooperative mode, players have to work together in order to achieve goals. Competition means playing against other participants in the game. While most card games belong to the competitive genre, few cooperative ones are available on the market. Examples are Hanabi and LOTRCG. The latter also features solo gameplay.

1.1 Background of AI in card games

Card games pose a significant challenge for artificial intelligence due to hidden information. Hidden information can result in the player not observing the cards in the opponent's hand or the cards in the deck stacked face down. Over the past years, many agents have been developed using different AI methods. These methods can be categorized into Rule-Based Systems, Counterfactual Regret Minimization, evolutionary algorithms, Monte Carlo Tree Search (MCTS), and Reinforcement Learning (RL) [4]. This section briefly describes all of those techniques, the details of MCTS and RL are discussed in Section 2.

[2] https://www.bbc.com/news/technology-35785875

Rule-Based Systems

Rule-based systems implement expert rules from which a strategy is created. These rules can have an empirical origin, such as an agent using the Magic the Gathering meta-game [5]. Meta-game is a knowledge base successively collected by players based on their gameplay experiences. The database contains strategies for completed decks, their effectiveness, the most beneficial synergies between cards, or qualitative ratings of individual elements within the game. Stiegler at al. [6] present an expert system for Hearthstone. This system includes static and dynamic form knowledge. Static knowledge is a set of information that is independent of specific gameplay, for example, game rules. Dynamic knowledge describes the current state of the game, for example, a character takes damage from an enemy attack. Static knowledge, as well as dynamic knowledge, is stored in a database. The database implements semantic relations like abstraction ("is") and aggregation ("has"). This representation method allows the AI agent to make effective use of expert knowledge.

The rule-based system can also use heuristic functions. These functions allow to estimate the usefulness of a card under given circumstances [6], [7]. Based on this estimation, the cards can be sorted [5]. The authors propose three agents that use different card sorting criteria in the decision-making process. The first method is to estimate the card's strength and its match with other cards the agent has available. The second agent uses a function of the probability of a human playing the card. This probability is determined from a meta-game containing over 100,000 recorded games. The database also serves as a learning set for a third agent, a neural network that predicts human movement. The network on the input receives the number of different types of cards available to the player and a specific card in one-hot encoding in the output.

Unfortunately, the meta-game for niche games is very limited or even unachievable, which significantly affects the effectiveness of such an expert system. Additionally, constructing heuristic functions may involve performing many experiments since the most straightforward formulas do not always prove effective.

Rules can also be derived from books, Di Palma and Lanzi [8] present three rule-based agents for the game Scopone based on strategy books. For the cooperative game Hanabi, agents have been created implementing the expert rule set [9], [10]. Expert knowledge combined with a tree-based algorithm produces a competent agent in Skat game [11]. Rule-based agents are often the baseline for MCTS algorithms [12], [13], [14], [7].

Counterfactual Regret Minimization

Counterfactual Regret Minimization (CFR) is a family of methods based on opportunity cost optimization (Neller and Lanctot [15]). When playing a card, it is determined what the advantage would be if another card were used. CFR looks for such a strategy that the cost of the played cards is as low as possible. In recent years, this algorithm has proven itself in poker [16].

Moravick et al. [17] present a neural network-assisted CFR (DeepStack). The network allows for a quick utility estimation of the game state instead of computing the entire strategy until the end of the game. The network's training is performed on randomly generated positions so that a sufficient degree of generalization of the game states can be achieved. The proposed extension allows DeepStack to defeat professional Texas Hold'em players.

Brown et al. [18] [19] combine CFR with Monte Carlo (MCCFR). MC simulation is used to sample actions on every iteration to avoid passing through the entire game tree. MCCFR has proved its effectiveness by defeating professionals in six-player no-limit Texas Hold'em.

Evolutionary Algorithms

Evolutionary algorithms are based on representing the game state as a genotype or a set of genes. Selection, mutation, and crossover operations are performed on the genotypes, allowing for an iterative search for the optimal strategy. Various evolutionary agents for balancing decks in dominion are presented in Mahlmann et al. [20]. Noble [21] describes evolving neural networks that are used to make betting decisions in poker.

Kowalski and Miernik [22] propose the use of an evolutionary algorithm for card drafting in the game *Legends of Code and Magic* (LOCM). LOCM is a strategy game in which two players battle each other using pre-constructed decks. Each game begins with a draft phase in which players alternate selecting one card from the three drawn that turn. A large card number of 160 makes the decision space challenging for an AI agent searching for an optimal drafting strategy. The authors propose to use an evolutionary algorithm with active genes. In this method, each gene encodes the priority (value between zero and one) of a given card. These values will be used during the draft. The draft consists of a presentation of three cards, and the one with the highest priority is selected. This way, a 30-card deck is obtained, which is then tested in a duel of expert bots. The active gene concept comes into play when mutations and crossovers are performed. These operations are performed only for active genes, i.e., the equivalents of cards that have been selected

during the draft. This selection preserves knowledge from previous generations contained in inactive genes.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a decision-making algorithm using tree representation of game states and Monte Carlo simulations. MCTS has been widely applied to Hearthstone [23], [24], [25], [26]. Diosdado [27] developed a MCTS player for Gwent: The Witcher Card Game. Bjarnason et al. [28] applied MCTS to Klondike Solitaire. MCTS agents have successfully competed against other bots in Poker [29], or Bridge [30]. Applications have also been developed for games derived from Bridge, such as Skat [31] and Doppelkopf [32]. Cowling et al. [14] supported MCTS with a determination for the Magic the Gathering game. The MCTS algorithm can also be supported by expert knowledge as in Whitehouse et al. [33]. A detailed description of MCTS is provided in Section 2.1.

Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning which is based on the trial and error method. The learning process consists of an agent's interaction with the environment. The agent observes the game state and decides which action to take. The action is executed in the environment, which returns a reward. Vieira et al. [34] propose a neural network-assisted RL agent for drafting in *Legends of Code and Magic*. Sturtevant and White [35] analyzes the performance of the RL agent for different state representations of the Hearts game. Barros et al. [36] compare several RL algorithms in Chef's Hat. RL algorithms are reviewed in detail in Section 2.2.

1.2 Aim and Motivation

The thesis aims to verify whether AI agents can play successfully in the strategic card game The Lord of the Rings: The Card Game (LOTRCG).

Despite its popularity, the game has not yet been studied by the research community. The reasons may be the unique features of the game that raise the problem's difficulty. These include the non-competitive nature of the game, the round composed of eight phases with a sequence of five decisions and two random stages, deck of 220 cards with nine different parameters. All of this adds strategic depth to the game and makes it a challenging environment for decision-making algorithms.

Modern card games are an active area of research into artificial intelligence applications. However, the Reinforcement Learning method in such types of problems has a small presence in the literature, so the central part of the research described in the thesis will be devoted to this topic. The Monte Carlo Tree Search algorithm is a more established tool that will create a reference for the results.

1.3 Thesis structure

This dissertation presents selected AI methods applied to a multi-stage fantasy card game. The first chapter outlines the historical context for board games and machine learning in games. The second chapter focuses on the theoretical foundations of two families of algorithms, namely Monte Carlo Tree Search and Reinforcement Learning. Additionally, extensions of these methods towards card games are discussed. The third chapter describes the rules of The Lord of the Rings: The Card Game, covering the various phases of a turn with outlined decision moments. This chapter also includes an implementation of a pseudorandom agent. The fourth chapter presents the MCTS agent with all the steps of the algorithm and the modifications used. Section 5 characterizes the RL agent with emphasis on the encoding/decoding mechanism. Section 6 shows the research results comparing all the agents. The paper concludes with a summary outlining future perspectives.

Specific terms used in this thesis are described in the Minidictionary (on page viii).

Chapter 2

AI in card games

Card games provide an interesting testbed for artificial intelligence due to the turn-based nature and random events that occur during gameplay. In addition to the age-old Poker [37], Solitaire [28], and Bridge [38], the last three decades have seen the development of fantasy games such as Magic the Gathering [1], Pokemon [2], Hearthstone [3], and The Lord of The Rings: The Card Game [4]. These games are released in a collector's edition format (CCG) consisting of a base set and add-ons purchased separately. The base set is a fully playable deck that allows learning the game's rules, while expansions include cards that introduce new characters or mechanics. The following section presents several applications of artificial intelligence to CCGs that have drawn the attention of researchers in the last decade. While the overall review can be found in Section 1.1, this part focuses on the methods applied to LOTRCG, such as Monte-Carlo Tree Search (MCTS) and Reinforcement Learning (RL).

2.1 Monte Carlo Tree Search

In card games, players make decisions in rounds, i.e., strictly defined moments of gameplay specified in the game manual. This property allows the card game to be represented as a tree structure, where edges represent actions taken by players and nodes represent the game's current state. Additionally, the model must consider random events such as drawing a card from the deck. In recent years MCTS has proved to satisfy those requirements [39],[40].

[1] Richard Garfield, 1993
[2] Takumi Akabane and Tsunekaz Ishihara, 1996
[3] Blizzard Entertainment, 2014
[4] Nate French, 2011

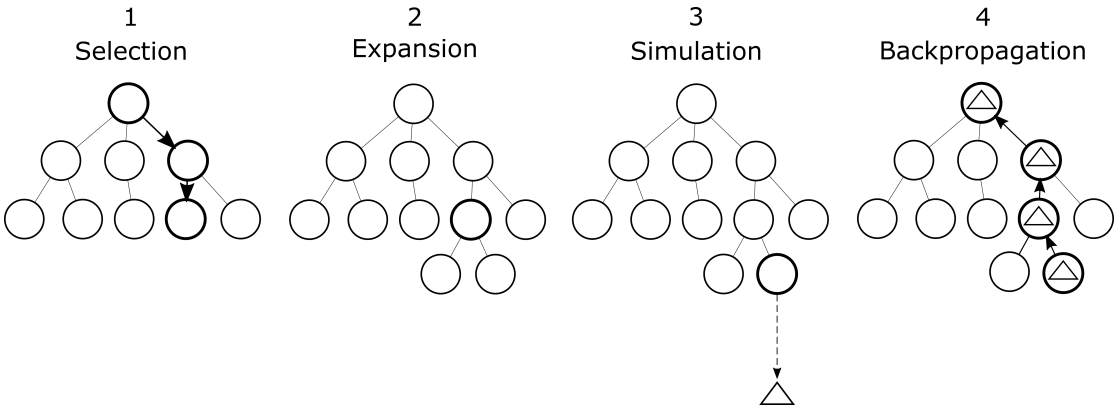

FIGURE 2.1: Subsequent steps of MCTS algorithm.

MCTS is a decision algorithm that searches the action-state space using a tree structure. The algorithm iteratively expands the tree by adding each node containing a game state and outgoing edges (actions) from it. Decisions are taken by estimating the value value of a given state-action pair. For MCTS, this estimate is obtained by sampling, which means repeatedly simulating the game to the end (the terminal state) and obtaining the win/loss probability distribution.

The MCTS workflow consists of four phases presented in Fig. 2.1:

1. **Selection** - a single MCTS iteration starts with selecting a blind node, i.e., one with no descendants. For this purpose, starting from the founder node, the selection is performed based on the Upper Confidence Bound for a node:

$$UCB_i = x_i + C\sqrt{\frac{2\ln n}{n_i}} \tag{2.1}$$

where x_i is the winning probability for a given node i, n is the total number of simulations made from the node's parent i, n_i is the number of simulations from the node i, C is a constant. The node with the highest UCB value is selected, and in the next iteration, it is the parent. The procedure is repeated until a blind node is reached. The fundamental issue in the selection process is maintaining a balance between exploration and exploration of the tree. Exploration selects the nodes with the highest probability of winning among their siblings, while exploration rewards the less frequent nodes. The equation 2.1 allows for a compromise between these two strategies. The first part of the equation accounts for exploration. The higher the probability of winning, the higher the UCB value. The second part of Eq. 2.1 is exploration. This term increases every time another node from the siblings is selected. The constant C allows to give weight to exploration. If a node is not selected even once, then the exploration part significantly increases the probability of selection in the next iterations.

2. **Expansion** - descendants are created for the selected blind node. In this phase, the algorithm specifies all legal actions for the blind node, each leading to a new descendant. The legality of the action must follow the game rules.

3. **Simulation** - a number of trial games are rolled out. Each trial starts in the game state corresponding to the selected node and ends with a win or loss. In the basic version of MCTS, actions taken in these trials are random, allowing for a better sampling of the game space. Based on this statistic, the probability of winning, or x_i, is calculated from the equation 2.1.

4. **Backpropagation** - UCB function update. All nodes lying along the path from the node from phase one to the root receive incrementation of the counter of simulations performed, i.e., the parameter n from equation 2.1. Backpropagation ends the MCTS iteration, and the loop returns to phase one.

In recent years, MCTS has been applied to the collectible card game Hearthstone. In this game, two players combat each other using previously assembled decks. The game is characterized by several stochastic elements, such as random card draw from the deck, random reward for killing a boss, and randomized weapon statistics. All features make modelling gameplay with MCTS result in a tremendous branching factor. A solution to reduce the branching factor can be chance event bucketing [24]. For each chance event, the corresponding chance node has a certain number of possible outcomes. For example, picking a random card from a 10-card deck means ten different outcomes. These results can be grouped into an arbitrary number of buckets. Following the example, if the number of buckets is set to two, each bucket will contain five results. The grouping can base on expert knowledge. In the case of Hearthstone, the cards were grouped based on their strength. A certain number of results are then sampled from these buckets, for example, two out of five, giving four samples from all buckets. In this way, only four results are obtained from the initial ten results by reducing the number of nodes in the tree. Other extensions to MCTS that reduce the branching factor can be found in papers written by Choe and Kim [41] and Santos et al. [23].

In vanilla MCTS, game state evaluation is obtained by performing a number of playouts. For Hearthstone, an alternative approach has been proposed in the form of a deep neural network (Swiechowski et al. [25]). The network takes an encoded game state that, in addition to numerical values, also contains game text representations of each card. The return value is the probability of winning for a given game state. The learning data was generated from bot gameplay with expert knowledge implemented.

Another example of a collectible card game is Pokemon. It is a competitive, two-player game that, like Hearthstone, features randomization-based mechanics or hidden elements.

The Information Set MCTS (ISMCTS) can be a method of handling those features [42]. An information set is a set of states indistinguishable from the perspective of a given player. For example, a player must perform an action followed by a card draw event. This event causes him to be unable to estimate the utility of the state he will be in by performing the action. These states may form single information set containing all draw results.

An interesting example of a card game with spatial elements is *Lords of War*. In this game, players fight each other by placing cards on the board with the possibility of both melee and ranged attacks. Additionally, the game features support units, which heal the cards placed nearby on the board. For this game, an MCTS extended with heuristic functions is proposed to estimate the utility of a state [43]. In addition to card statistics, the functions also use heatmaps for tactical analysis of the spatial distribution of units. For example, if one player has units deployed at strategic locations on the board, then a heuristic function provides a comprehensive estimate of tactical advantage.

The game described in this thesis - Lord of the Rings the Card Game, has already been studied for MCTS (Godlewski and Sawicki [7]). Flat and full MCTS called Upper Confidence for Trees (UCT) were compared. Flat MCTS based on building the tree only up to the first degree of affinity, while UCT was a full implementation of the concept with a selection function (eq. 2.1).

Applications of MCTS can also be found in Poker (Brown et al. [19]). The authors employed tree search with Counterfactual Regret Minimization described in section 1.1. The method performs a depth-first search by exploiting the game tree before the back-propagation phase.

Cowling et al. [14] presented MCTS player for Magic the Gathering (MtG). MtG originated in 1993, and it stands as the oldest collectible card game available on the market. In its vanilla version, MtG is a competitive game for two players, in which they summon creatures and engage in a fight. The authors proposed MCTS with determinization in order to handle stochastic events during the game, like card draws. That technique significantly increased the branching factor, so the authors investigated several pruning strategies. The players with those modified MCTS algorithms proved to be competitive opponents against expert rule-based agents. MCTS agent for MtG was also described in Ward et al. [12].

MCTS player with determinization was also employed for a cooperative card game Hanabi [44]. The game can be played with 2-5 players and features stochastic events like MtG. The authors introduced an extension to MCTS in order to reduce decision time. The extension is based on evaluation functions, which allow the utility estimation of a

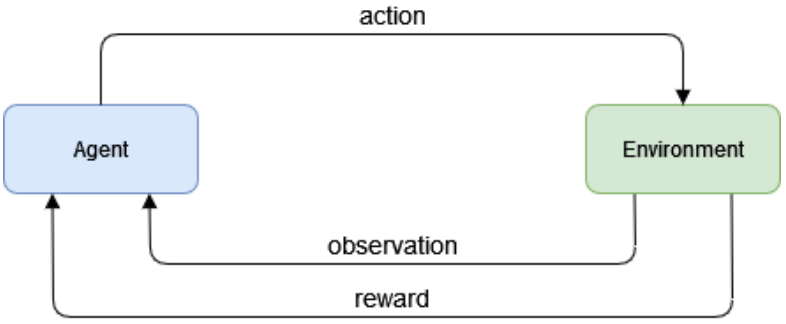

FIGURE 2.2: Information cycle between the agent and the environment.

node instead of simulating whole playouts. The functions were learned offline using the data gathered from previous gameplays.

MCTS was also applied to an Italian four-player card game Scopone (Di Palma and Lanzi [8]) and a mobile game called Spades (Baier et al. [45]).

2.2 Reinforcement Learning

Reinforcement Learning (RL) has been a new trend in the development of artificial intelligence in recent years. The concept is based on the trial-and-error method [46], in which the AI agent and the environment interact, as shown in Fig. 2.2. An AI agent is a decision-making algorithm that takes specific actions based on observations. The environment is the entity where this action will be executed, and a feedback signal (positive, negative, or zero) is sent to the agent. RL differs significantly from other machine learning techniques. RL agent does not operate on a static set of learning data but receives a feedback signal based on which it performs the learning process. The feedback signal itself is irregular, which means positive or negative information may appear at different time intervals. These features are ideally suited to the field of games, where it may not be possible to generate a sufficiently large set of learning data. The win/loss feedback signal also occurs at different times since one game round ends after ten turns and another after four.

The basic concepts used in the context of reinforcement learning are as follows:

- **State** - current game situation including all information coming from the table and hidden elements available to players,
- **Action** - game action resulting from a decision-making process,
- **Policy** - a function that maps state to action or probability distribution over actions $\pi(s)$,

- **Reward Signal** - environment's reaction to action,
- **Value Function** - represents a measure of how beneficial it is for a player to be in a given state or state-action pair. Value function for policy π can be described with the following equations [46]:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.2}$$

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma q_\pi(s', a')] \tag{2.3}$$

where: $v_\pi(s)$ - value function for state s, $\pi(a|s)$ - probability of action a in state s, $p(s', r|s, a)$ - probability of reaching next state s' and reward r by performing action a from state s, γ - discount factor, $v_\pi(s')$ - value function for the next state s', $q_\pi(s, a)$ - value function for the pair state s and action a, $q_\pi(s', a')$ - value function for the next state s' and the next action a'.

The goal of reinforcement learning is maximization of discounted reward in the long term [46]:

$$Q_{n+1} = \frac{1}{n} \sum_{i=1}^{n} R_i = Q_n + \frac{1}{n}[R_n - Qn] \tag{2.4}$$

This formula allows us to iteratively calculate the value function Q_{n+1} given its current value Q_n and reward R_n. This goal is achieved through Generalised Policy Iteration (GPI). GPI consists of two steps: calculating the value function and modifying the strategy (policy improvement). The value function can be calculated from the formulas 2.2 or 2.3 for each state and action for low complexity problems. However, approximation methods such as linear or neural networks are employed for large state spaces. Once the value function is obtained, the strategy is updated according to the formula:

$$\pi(s) \leftarrow argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.5}$$

There are two families of GPI algorithms. The first is **on-policy** method, in which the same strategy is used to compute the value function and then updated. These methods offer stability at the cost of a time-consuming convergence. In **off-policy** algorithms, each GPI step uses different policy. The value function is estimated according to a policy called behavior policy. In the second step of GPI a separate policy called target policy gets updated. With this separation, different types of policies can be used, for example, stochastic behavior policy and deterministic target policy. This arrangement speeds up convergence; however, the learning process for some algorithms may turn out unstable.

The well-known off-policy algorithm is Q-learning. One step Q-learning estimates the value of a state action pair with the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \tag{2.6}$$

where $Q(s_t, a_t)$ - value function for the current state-action pair, α - step size, R_{t+1} reward from taking action a_t, γ - discount factor, $Q(s_{t+1}, a)$ - value function of the action a taken from the next state s_{t+1}. The max function present in Eq. 2.6 is a nonlinear operator taking the maximal value function from the next state. This strategy corresponds to the behavior policy. The target policy is improved as follows:

$$\pi(s) \leftarrow argmax_a Q(s, a). \tag{2.7}$$

The operator $argmax$ assigns the best valued action to the state s. This process forms the deterministic target policy.

There are also non-GPI algorithms in which the value function is not present, and only the strategy is iterated. This solution allows for better strategy optimization for problems with a large state space. Examples of algorithms that iterate only the strategy are REINFORCE or Monte-Carlo Policy Gradient (Sutton et al. [46]).

The third type of RL algorithm is a hybrid approach that approximates both the value function and the strategy. These methods are known as Actor-Critic. The actor is responsible for estimating the probability distributions of actions for a given strategy, while the critic is responsible for the value function. Classic examples here are Advantage Actor-Critic (A2C) or Proximal Policy Optimization (PPO). PPO can be used to build a deck in three variants that differ in the representation of the game state (Vieira et al. [34]). The first variant based on the MLP network includes all previously selected cards into the state vector. In the second, the leading role is played by the LSTM network, which accumulates information about previous card choices only based on the vector of cards currently available to the player. Finally, the third option uses only the MLP network with the representation the same as in the second option.

Three RL algorithms: Deep Q-Learning, A2C, and PPO were compared in the game *Chef's Hat* (Barros et al. [36]). *Chef's Hat* is a competitive four-player card game in which players try to become a chef. The game is played in turns, in which each player decides whether to play cards or fold. Due to the four-player nature of the game, it was possible to experiment with different configurations of agents. RL algorithms were challenged with random agents in direct skirmishes and against a human player.

Proximal Policy Optimization algorithm was applied to the *DouDizhu* game (Guan et al. [47]). *DouDizhu* is a three-player game mixing collaboration with competition. Two players have to cooperate in order to defeat the third player. The authors developed a framework called Perfect-Training-Imperfect-Execution (PTIE) based on centralized training and decentralized execution of RL agents. PTIE allows agents to train their policies on a game that is treated as perfect information. The policies are distilled in order to play the actual game with imperfect information.

Zha et al. [48] developed an open-source platform to learn and test reinforcement learning agents on card games. The platfrom supports standard 52-card games like Blackjack, Texas Hold'em, and also Chinese-originated games such as Mahjong and *DouDizhu*. The authors test three algorithms on their platform, such as Deep Q-Network, Neural Fictitious Self-Play, and one outside RL like Counterfactual Regret Minimization.

The Hanabi card game has been seen as a challenge to AI in recent years [49]. Hanabi is a cooperative card game for 2-5 players. The game stands out for the way it handles imperfect information. The player does not see his own cards; he can only observe other players' cards. In order to play the right card, the player must get hints from other participants.

Recently Hanabi has drawn the attention of RL researchers. Grooten [50] et al. compare different RL algorithms for Hanabi. The algorithms include Proximal Policy Optimization (PPO), Vanilla Policy Gradient (VPG), and Simple Policy Gradient (SPG). VPG is an actor-critic method. It maintains separate neural networks for both policy and value function approximation. SPG uses only a policy network. The authors analyze various aspects of the algorithm's performance within the game, such as learning curves and policy refinement over episodes.

Yao et al. [51] propose a method of handling large action spaces of the *Axie Infinity* card game. *Axie Infinity* is an online competitive 2-player game in which the player forms a few subsets of cards in order to defeat the opponent. Cards to a subset are chosen based on Q-function approximation. The function indicates the optimal decision from a restricted subset of actions. The method can be applied to other card games with large action spaces like *DouDizhu* or *Hearthstone*.

Chapter 3

The Lord of the Rings: The Card Game

John Ronald Reuel Tolkien (1892-1973) spent his whole life creating a high fantasy world called the Middle-earth. The most popular books from Middle-earth are The Hobbit, The Lord of the Rings, and Silmarillion. The worldwide success of Tolkien's works has always inspired other artists and creators, especially in the game industry. Computer players can enjoy the Middle-earth series: Shadow of Mordor [1] and Shadow of War [2]. Fans of unplugged entertainment can find many board games like War of the Ring [3], The Lord of the Rings: Journeys in Middle-earth [4], Middle-earth Strategy Battle Game [5] and The Lord of the Rings The Card Game [6].

3.1 Game Description

The Lord of the Rings: The Card Game (LOTRCG) is a fantasy-themed, collectible card game created by Nate French. A collectible card game is a game released in a long-term policy that consists of publishing in a series of sets. The base is a core set containing a playable card game. In addition to that, the publisher releases expansion sets, which enrich the gameplay with new cards and mechanics. The core set of LOTRCG features 220 cards and a few hundred in 26 expansions, which have been published over

[1] Monolith Productions Studio, 2014
[2] Monolith Productions Studio, 2017
[3] Roberto Di Meglio, Marco Maggi, and Francesco Nepitello, 2012
[4] Nathan Hajek and Grace Holdinghaus, 2019
[5] Games Workshop Limited, 2018
[6] Nate French, 2011

the years [7]. Since 2019 the digital version of the game called The Lord of the Rings: Adventure Card Game has been available on Steam online platform [8].

The game is based on challenging adventures in order to complete a scenario. During the scenario, a fellowship led by the players encounter many adversities, and if they fail, the scenario is lost. The game can be played in two variants: solo or two-player cooperative. These two modes make the game distinct from other well-known card games like Magic the Gathering [9], and Pokemon [10] since they feature only competitive gameplay.

In LOTRCG, the players can form their fellowship using a variety of decks. By default, the core set features four decks, but a more appealing option is building its own. This deck-building approach offers to deal with cards more experimentally with discovering new synergies between them and better replayability.

3.2 Cards Description

The core set of LOTRCG offers diverse cards the player can choose from. The cards are divided into four types called spheres: Spirit, Lore, Tactics, and Leadership. Every sphere is inclined toward a certain game aspect, e.g., Tactics is combat-minded, whereas Lore concentrates on healing abilities. Each deck contains four card types such as:

- Heroes - basic cards. Every deck contains three different Heroes. They define the general strategy of the deck (see statistics and abilities in Fig. 3.1),
- Allies - auxiliary cards, they support Heroes,
- Events - cards influencing the gameplay, can be played only one time,
- Attachments - equipment cards extending abilities to the attached card. Attachments can be equipped to Heroes and Allies.

An example player's card is shown in Fig. 3.1 (a). Eowyn is a hero card (7) and belongs to the leadership sphere. She has a life pool of 3 hitpoints (4) and can attack with a power of 1 (3) and defend with 1 (3). Her parameter willpower of 4 (1) is used to make progress in the scenario. The game text (6) describes his unique abilities, synergizing with other hero and ally cards.

[7] https://ringsdb.com/search
[8] https://store.steampowered.com/app/509580/
[9] Richard Garfield, 1993
[10] Takumi Akabane and Tsunekaz Ishihara, 1996

FIGURE 3.1: Example of a hero card (a) and an enemy card (b). Card features: 1 - willpower, 2 - attack, 3 - defense, 4 - hitpoints, 5 - name, 6 - game text, 7 - type, 8 - engagement threshold, 9 - threat.

An example enemy card is named Forest Spider (b in Fig. 3.1). It has an engagement threshold of 25, a threat level of 2, an attack of 2, and a defense of 1. Forest Spider can take 4 hitpoints (4) until death, and it also features game text (6) as Eowyn. Parameters of all core set cards can be found in appendix A.

During a scenario, the player will struggle with many encounters like enemies, lands, or adversities. Enemies are creature cards with an attack ability, and the player has to defeat them during the scenario. Lands are neutral cards, which the player can explore as additional quests to the scenario. Adversities are event cards harmful to the player. They can hurt the player's fellowship or hinder the scenario's progress.

3.3 Game Rules

The game's goal is to complete a scenario, which means reaching a specified number of progress points throughout the game. During the scenario, the player encounters objects such as enemies or lands, which hinder gaining progress points. These objects require a player's reaction. If an enemy appears, the player has to defend himself; otherwise, he can lose his heroes or allies. If a land card appears, the player can decide whether explore it. Leaving lands unexplored makes the scenario progress difficult. The encounters can happen in a round presented in Fig. 3.2.

A round of the game consists of eight phases (Fig. 3.2). Each phase contains a sequence of activities, which can be rule-based, a random event, or the player's decision.

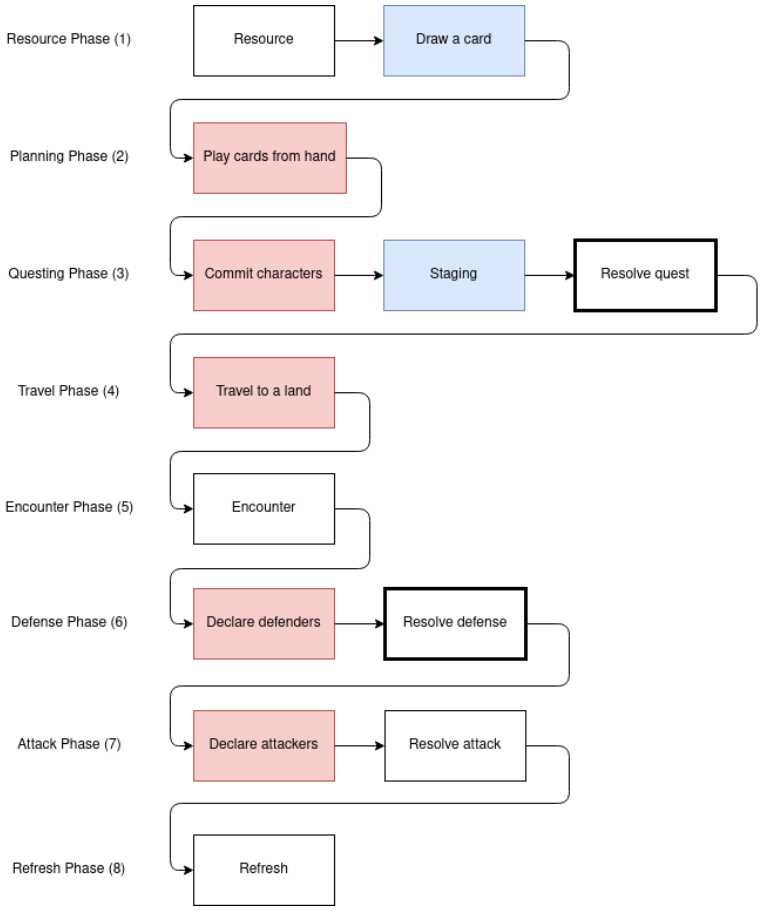

FIGURE 3.2: The sequence of activities which constitute each phase. Activites are ordered in rule based (white), random events (blue) and player's decisions (red). The bolded rects mark the moments of determinig game end conditions.

The first phase of a round is the resource phase (1). At this stage, the player draws a card from the player's deck to his hand. He adds one resource token to the resource pool for every controlled hero (he starts the game with three heroes).

These resource tokens can be spent in the planning phase (2). The player can buy cards from his hand only within the budget. The acquired cards are placed on the table and considered ready to play.

In the questing phase (3), the player commits characters to the quest. To do this, the player taps the selected cards and sums up their total willpower (parameter 1 in Fig. 3.1). Next, a card is drawn from the encounter deck and goes into the staging area. The player calculates the difference between the total willpower and the combined threat of the cards in the staging area. If the result is positive, then the player gains progress points; otherwise, the player's threat level gets increased.

The player can explore a land during the travel phase (4). He selects a land card and removes it from the staging area. Now, all progress points go on this card until it is

fulfilled. A trip to a land delays the scenario progress for a few rounds, but it makes questing less card demanding.

The next phase is the encounter phase (5). It is a rule-based activity in which enemy cards from the staging area engage in a fight with the player.

The player can defend himself in the defense phase (6). Only cards that have not been committed to the quest can be played in this phase. The engaged enemies perform attacks one after another. In order to face these attacks, the player declares cards as defenders and resolves the fight. If the player declares no defenders, then the damage is taken by heroes, which can result in their death.

The last decision taken by the player is a counterattack against the enemies during the attack phase (7). The player sets the target (one or more enemies in the encounter area) and taps characters as attackers. If the target enemy runs its hitpoints below zero due to the counterattack, it dies and goes into the graveyard. A comprehensive description of both defense and counterattack resolution can be found in the official rulebook [11].

In the refresh phase (8), all characters get untapped, and the player's threat level increases by one.

3.3.1 Win/Lose Conditions

There are two moments in the game where the win or lose condition is examined.

The first moment occurs during quest resolution in the questing phase (3). When the number of points exceeds the number required by the scenario, then the game is won immediately. The scenario requires 8 points, so the player must make eight progress through the whole game.

The quest resolve of a round can yield a negative result. In this case, the threat level increases by that number. If the threat level exceeds 50 points, then the game is lost. For future reference that condition is called `lose_by_threat`.

The second moment of determining the game's end condition is the resolution of the defense phase (6). Whenever the last hero controlled by the player dies, then the game is lost. The death of a hero means the moment its hitpoints fall to zero or below as a result of a fight with an enemy. It is called `lose_by_heroes`.

[11] https:/images-cdn.fantasyflightgames.com/ffg_content/lotr-lcg/LOTRRules.pdf

	Easy	Medium	Hard
Scenario setup	no	no	yes
Number of cards	13	25	25
Number of card types	7	15	15

TABLE 3.1: Parameters of the difficulty levels

3.4 Game Model

In order to experiment with AI algorithms, one needs to have an implementation of the game. Even though there is a gameplay simulator of LOTRCG [12], the project has still been experimental since 2019. Moreover, the simulator is unsuitable for scientific research due to license restrictions. In this regard, a new game model of LOTRCG was developed. The model implements the rules presented in Section 3.3. It represents a modified version of the game, which means that some features were not included. Due to the implementation complexity, the game model does not take advantage of the card's unique abilities described in a game text (11 on Fig. 3.1). The length of a scenario was also restricted, which means that the player needs to achieve fewer victory points to win the game.

There are five types of cards in the model class diagram (Fig. 3.3): hero, ally, enemy, land, and quest. Any other such as event or attachment cards were neglected. Every card has its parameters, as shown in Fig. 3.1. Statistics of the cards are presented in Appendix A.

The model expands on three difficulty levels regarding setup and the number of cards in the game (Table 3.1). The easy level provides the encounter deck containing only enemies with a total number of 13 cards. The enemies are of seven types; therefore, some cards are duplicated. The medium and hard levels deal with the complete encounter deck - 13 enemy cards and 12 land cards. These modes differ in the initial setup specified in the rulebook [13] so that the game on the hard difficulty level starts with two extra cards from the encounter deck already in the staging area. In practice, the agents have to engage more cards to make progress in early rounds.

The implementation of the game model follows an object-oriented paradigm in Python language. The source code of the simulator is available on github platform [14]. The class `Card` forms a base, which is extended by other classes (Fig. 3.3). These can be leaf nodes such as `Quest` or `Land` or middleware classes like `Creature` or `PlayerCreature`. The `Deck` component holds `Card` objects and manages operations on cards, for example,

[12] https://ringsdb.com/
[13] https://images-cdn.fantasyflightgames.com/ffg_content/lotr-lcg/LOTR_Rules.pdf
[14] https://github.com/kondziug/LOTR_AI

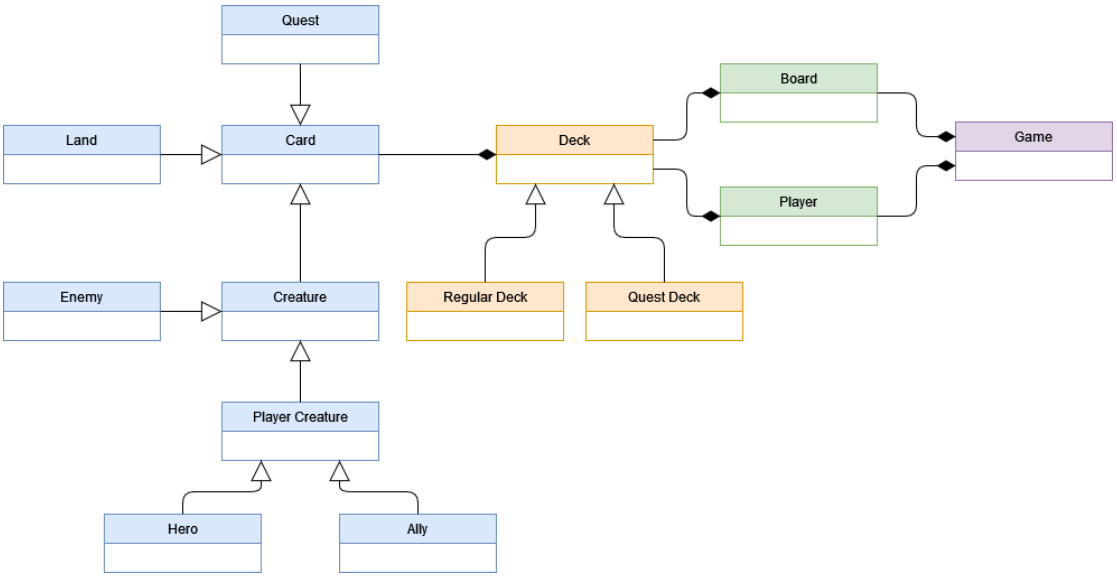

FIGURE 3.3: The game model class diagram.

shuffling. One level up in the hierarchy operate `Board` and `Player` classes. They handle complex game activities (Fig. 3.2) such as an encounter check on the staging area or defense resolution. The `Game` component manages them in order to perform the phase sequence.

3.5 Random Agent

The implementation of agents into the LOTRCG began with a random agent. The agent was meant to be the base of initial studies of the game complexity and used as a baseline for other agents. The random agent was also used for MCTS, where it performed playouts during the simulation phase. The task of this agent was to run the gameplay randomly, i.e., to make decisions based on probability distributions with high entropy. During a turn, there are five decision moments (red rectangles in Fig. 3.2) for which corresponding methods were implemented: `randomPlanning`, `randomQuesting`, `randomTravel`, `randomDefense` and `randomAttack`.

The first decision phase is planning (the first red rectangle in Fig. 3.2), in which cards are purchased with resources accumulated during gameplay. The `randomPlanning` function shown in Listing 3.1 is responsible for this phase. At first, it receives a list of cards available in the player's hand. Then the length `slen` of the subset of cards to be purchased is drawn. If `slen` is zero, the agent will exit the function without purchasing any cards. In the case of non-zero values of `slen`, the `for` loop starts execution. In each loop iteration, a random card is selected from the player's hand (function `random.choice()`).

If the player has the required resources, this card is purchased and added to the player's allies list.

```
def randomPlanning(self):
    playerHand = self.player.getHand()
    slen = random.randint(0, len(playerHand))
    for _ in range(slen):
        card = random.choice(playerHand)
        if self.player.getResourcesBySphere(card.sphere) >= card.cost:
            self.player.spendResourcesBySphere(card.sphere, card.cost)
            self.player.addToAllies(card)
```

Listing 3.1: Random Planning

The function `randomQuesting` from Listing 3.2 operates on all player cards available on the table which is `playerCharacters`. The function is designed to select a random subset of cards of length `slen` from this list. As in `randomPlanning`, the function `random.choice()` also plays the main role here by making the card selection. The next step is to update the variable `combinedWillpower` which is the sum of willpower (parameter 1 in Figure 3.1) and tap the card. The final step is to resolve the quest for a given value of `combinedWillpower`.

```
def randomQuesting(self):
    combinedWillpower = 0
    playerCharacters = self.player.getAllCharacters()
    slen = random.randint(0, len(playerCharacters))
    i = 0
    while i < slen:
        card = random.choice(playerCharacters)
        if not card.isTapped():
            combinedWillpower += card.getWillpower()
            card.tap()
            i += 1
        if self.player.checkIfAllTapped():
            break
    self.resolveQuesting(combinedWillpower)
```

Listing 3.2: Random Questing

During `randomTravel` from Listing 3.3 the player can start exploring a new place in the form of a side quest. In order to do this, a list `lands` is created from which a single card is selected using `random.choice()`. This card is now the player's destination. Selecting an empty item means deciding to renounce the side quest.

```
def randomTravel(self):
    activeLand = self.board.getActiveLand()
    if activeLand: return
```

```
    lands = self.board.getAllLands()
    if not lands: return
    lands.append(None)
    land = random.choice(lands)
    if not land: return
    self.board.travelToLocation(land)
```

Listing 3.3: Random Travel

In `randomDefense` (Listing 3.4) opponents located in the engagement area perform an attack on the player. For this purpose, the list `enemiesEngaged` is obtained. Then, for each enemy, the algorithm tries to assign a random defender. The function `declareRandomDefender()` is responsible for this action. This function uses `random.choice()` on the set of heroes and allies that have not been tapped in the questing phase. Once declared as a defender, a card cannot be used again in the same turn. If `declareRandomDefender()` fails to find a defender, then according to the rules, the entire attack value must be taken by one of the heroes, in this case, also chosen at random.

```
def randomDefense(self):
    enemiesEngaged = self.board.getEnemiesEngaged()
    for enemy in enemiesEngaged:
        defender = self.player.declareRandomDefender()
        if defender:
            result = defender.defense - enemy.attack
            if result < 0:
                defender.takeDamage(abs(result))
        else:
            self.player.randomUndefended(enemy.attack)
```

Listing 3.4: Random Defense

`randomAttack` (Listing 3.5) can only be made by characters not tapped in the previous phases from the list `untappedCharacters`. From this group, a subset of `playerCharacters` is drawn at random. Then, each character attacks a random target chosen from the involved enemies (`enemiesEngaged`).

```
def randomAttack(self):
    enemiesEngaged = self.board.getEnemiesEngaged()
    untappedCharacters = self.player.getUntappedCharacters()
    slen = random.randint(0, len(untappedCharacters))
    if not slen: return
    playerCharacters = random.sample(untappedCharacters, k=slen)
    for character in playerCharacters:
        randomTarget = random.choice(enemiesEngaged)
        result = randomTarget.defense - character.attack
        if result < 0:
```

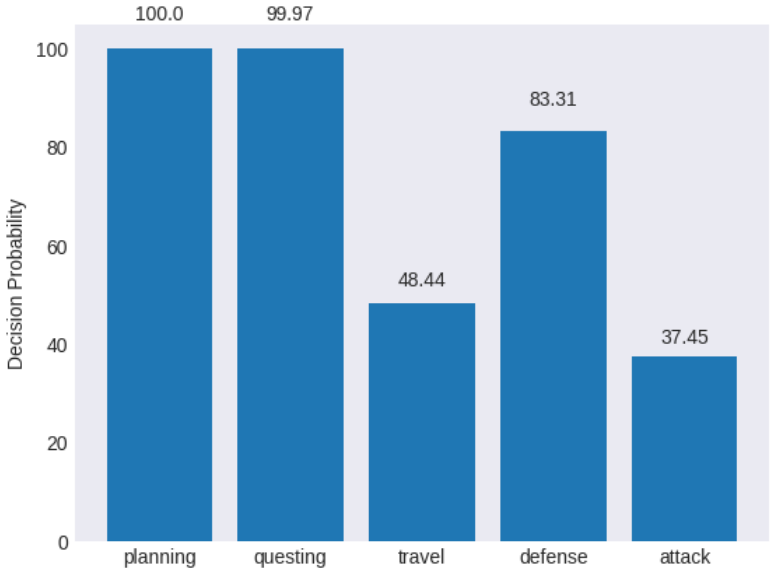

FIGURE 3.4: Probability of taking decisions in each phase of a round.

```
11              randomTarget.takeDamage(abs(result))
12              if randomTarget.isDead():
13                  self.board.removeFromEngagementArea(randomTarget)
14          if not enemiesEngaged:
15              return
```

LISTING 3.5: Random Attack

Review of the above listings points that the construction of a random agent is based on built-in python functions such as `random.choice()` or `random.randint()`. This strategy allows to develop an agent with the highest entropy according to the game rules.

3.6 Study of Game Complexity

The sequence of player moves in a single round crossed with random events indicates a high decision complexity of LOTRCG. Its investigation was performed using the Random Agent, which implementation was presented in the previous section. The analysis allowed to limit the scope of the experiments performed for the other agents, such as MCTS and RL.

The first step was to determine the choices for each decision. There are five decision moments in the game (Fig. 3.2), one for each phase: planning, questing, travel, defense, and attack. These decisions are made one after the other, so cards played in questing cannot be used in the defense phase. Analogous cards assigned for defense cannot occur in the attack phase. In the travel phase, there may be no cards to choose from, depending

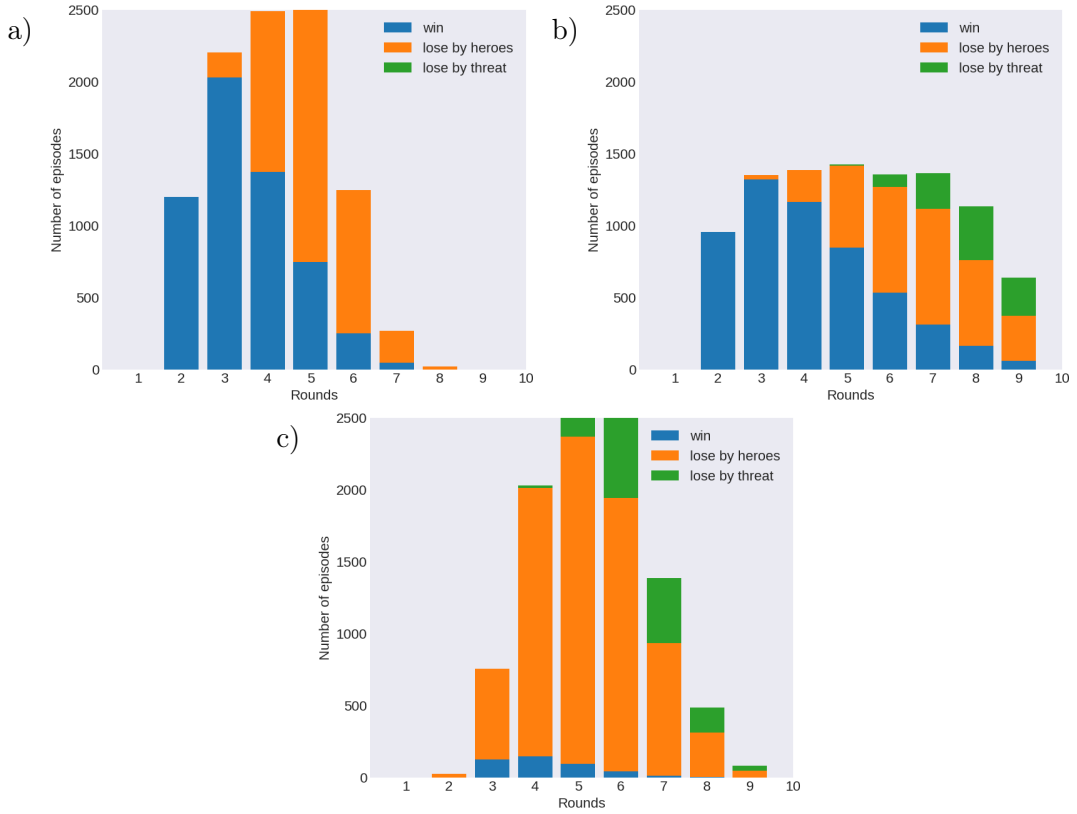

FIGURE 3.5: Histograms of episode length probabilities for easy (a), medium (b) and hard (c) difficulty levels.

on the current game situation. An agent has a choice if there are at least two choices for a given decision phase. In practice, the list of cards to be played cannot be empty; for example, `playerCharacters` in `randomQuesting` must have a non-zero length. Fig. 3.4 shows the probability of having choices for an average episode. It clearly shows that in the planning and questing decisions, the agent always had some choice, while in the other phases, there were rounds with no possibility to move. The phases with the lowest decision probability are travel and attack, so these decisions were not examined further. The development of agents was limited to planning, questing, and defense decision.

The length distribution of episodes is shown in Fig. 3.5. The typical episode length is five rounds, while the agent wins most often after four rounds. Increasing the length of a game results in a loss by exceeding the 50 threshold of threat points, as evidenced by the increasing proportion of `lose_by_threat` (right part of Fig. 3.5).

The effectiveness of the random agent is shown in Table 3.2. It reports the winrate for different game difficulty levels, as defined in Section 3.4. The agent was examined with the 95% confidence interval given by Eq. 6.1. From this table and the histogram from Fig. 3.5, it can be concluded that the hard level is very challenging for the agent. Starting the game with two cards in the staging area is a serious difficulty for the agent.

difficulty	number of episodes		
	1000	10000	100000
easy	55.4 ± 3.08	55.97 ± 0.97	55.71 ± 0.31
medium	52.6 ± 3.09	53.74 ± 0.98	53.83 ± 0.31
hard	4.7 ± 1.31	4.18 ± 0.39	4.53 ± 0.13

TABLE 3.2: Winrate of the random agent for various game difficulties.

The almost equally easy and medium levels indicate that changing the number of cards in the encounter deck does not affect the winrate.

Chapter 4

Development of MCTS Agent

This chapter presents the implementation of the MCTS algorithm, which background was described in Section 2.1. One MCTS iteration performs a sequence of the following phases: selection, expansion, simulation, and backpropagation. The description of this agent begins with tree policy (Section 4.1), the phases associated with tree expansion, and backpropagation. Section 4.2 is rollout policy, which covers the strategies of the simulations themselves, which means executing full games ending in victory or defeat. This division helps to outline the implementation of MCTS-specific elements. Section 4.3 describes parameters tuned for depth restrain of the MCTS tree. Section 4.4 considers action reduction, which was applied to MCTS as a method of reducing branching factor.

4.1 Tree Policy

MCTS starts with node selection (Listing 4.1). This function aims to find a path from the root node through the entire tree up to the leaf node. It first checks if the function argument `node` is a leaf node meaning that it has no children. If not, it iteratively checks its children for the number of visits. If any descendant has never been visited in previous MCTS iterations, it is selected as a leaf node. In the case where all the children have been visited before, the determination is based on the UCB function given by Eq. 2.1. The node with the highest value is passed as an input argument for the recursive call of the selection function.

```
1 def select(self, node, lvl):
2     lvl += 1
3     if not node.getChildren():
4         return node
5     children = node.getChildren()
6     for child in children:
```

```
7           if child.getVisits() == 0:
8               return child
9       score = 0
10      result = node
11      for child in children:
12          newscore = self.uctFn(child)
13          if newscore > score:
14              score = newscore
15              result = child
16      return self.select(result, lvl)
```

Listing 4.1: MCTS selection step implementation

The selected leaf node is extended with all possible descendants. Each node has a unique tag indicating the phase of the game (Fig. 3.2) in which it currently resides. Based on this tag, the corresponding function that adds children is called. For instance, for a leaf node tagged *Attack Phase* an expansion will be performed in the following decision phase (Listing 4.2). This function first creates a game state (`game`) based on the information stored in the selected `node`. It then searches for all possible actions, i.e., cards that can be purchased in that game state (`legalCards`). Each such card is applied to a new game state in a loop, which is a copy of `game`. Based on this copy, the function creates a new child node connected to `node`.

```
1  def expandPlanning(self, node):
2      game = node.createGame()
3      game.resourcePhase()
4      legalCards = self.findLegalsPlanning(game.getPlayer())
5      for name in legalCards:
6          newGame = copy.deepcopy(game)
7          if name != 'None':
8              card = newGame.getPlayer().findCardInHandByName(name)
9              newGame.getPlayer().spendResourcesBySphere(card.sphere, card.
       cost)
10             newGame.getPlayer().addToAllies(card)
11         newNode = Node(newGame.getBoard(), newGame.getPlayer(), node, '
       Planning')
12         node.addChild(newNode)
```

Listing 4.2: MCTS expansion step on the planning phase

Once a node has obtained an extension, the simulation stage described in Section 4.2 follows. This stage returns the value of `score` - the number of successfully completed games. `score` is the argument of the backpropagation function presented in Listing 4.3. Backpropagation updates the number of visits and the UCB value for a node. The

function is called recursively, so the update applies to all predecessors up to the root node.

```
def backpropagate(self, node, score):
    node.incrementVisits()
    node.incrementUtility(score)
    if node.getParent():
        self.backpropagate(node.getParent(), score)
```

Listing 4.3: MCTS backpropagation step

4.2 Rollout Policy

The second aspect of any MCTS algorithm is the rollout policy used during the simulation. This step allows to estimate the utility of a game state. The utility value is represented as a winning probability for a game starting in that state. In the basic MCTS version, this probability is obtained by simulating the specified number of games shown in Listing 4.4. This parameter is specified here by `playoutspernode`. After the auxiliary variables are initialized, the main loop follows. Each iteration is a single playout resulting in a win or loss. In order to preserve information about the game state for which the simulation is executed, the playout uses a copy of the game state (`tmpGame`). The new, copied game state is played to the end in an inner loop, which contains the `makeTurn()` function. This function executes all phases of the game shown in Figure 3.2. Decisions (the red rectangles in Figure 3.2) are made according to a specific rollout policy, for example for random playouts there will be functions from Listings 3.1, 3.2, 3.3, 3.4 and 3.5. If a win/loss condition is met during the execution of these functions, then the appropriate flag is set to true, and the inner loop ends. Finally, the auxiliary variables are set, and the single playout cycle closes. This process yields the number of `wins` in a given number of `playoutspernode`. The wins number is then passed to the next phase of MCTS, which is the backpropagation described in Section 4.1.

```
def simulateComplete(self, game):
    wins = 0
    turns = 0
    for _ in range(self.playoutspernode):
        if self.playoutBudget < 0:
            break
        self.playoutBudget -= 1
        tmpGame = copy.deepcopy(game)
        while 1:
            turns += 1
            self.doTurn(tmpGame)
            if Game_Model.globals.gameOver:
```

```
13                 break
14             if Game_Model.globals.gameWin:
15                 wins += 1
16                 break
17         Game_Model.globals.gameWin = False
18         Game_Model.globals.gameOver = False
19     return wins
```

LISTING 4.4: MCTS simulation step

4.3 Depth Limited Search

An essential issue in MCTS is limiting the size and expansion of a tree. One of the solutions is to restrict the tree depth. This approach consists of expanding the tree only to a certain level of depth to reduce the number of nodes. Fewer nodes mean fewer playouts and consequently lower computational cost of each decision. In this project, the tree depth is described by two parameters `playoutbudget` and `playoutspernode`.

Parameter `playoutbudget` specifies the number of MCTS iterations that will be performed before each game decision is made (red rectangles in Figure 3.2). For example, if the user sets `playoutbudget` to 100, then 100 selection - expansion - simulation - backpropagation sequences will be executed. This process limits the number of calls to the simulation phase, which requires the most computational effort.

Each simulation includes the execution of playouts to estimate the probability of winning (Section 2.1). The parameter `playoutspernode` specifies how many playouts must be executed in the simulation phase.

4.4 Action Reduction

Action reduction is a technique for size reduction of tree-based algorithms. It limits the set of possible moves the agent can take at the decision moment. The reduction applied to the MCTS agent for LOTRCG was based on expert knowledge, which allowed to estimate the utility of every action.

Action reduction for the planning decision consists of purchasing the Gandalf card if the player can afford to. Giving high priority to Gandalf comes from expert knowledge embedded into the algorithm. Despite its high cost of 5 resource points, Gandalf has very powerful stats of 4 willpower, defense, and attack (table of cards in Appendix A). These parameters make Gandalf useful for decision phases such as questing,

defense, and attack. Action reduction for planning decision is represented by the function `findLegalsPlanning` (Listing 4.5). This function determines all available actions, where each action is an affordable card. In the beginning, the list of cards to purchase `legalCards` is initialized. Its first element is always the value `None`, which means no card acquisition. Next, the player checks if a Gandalf card is in hand (line 5). If the card is found and affordable for the player, it is added to the list, and the function terminates. `legalCards` then has only two actions `None` and `Gandalf`. Otherwise, all affordable cards are added to the list meaning no action has been rejected.

```
1  def findLegalsPlanning(self, player):
2      legalCards = []
3      legalCards.append('None')
4      playerHand = player.getHand()
5      gandalf = player.findCardInHandByName('Gandalf')
6      if gandalf and player.getResourcesBySphere(gandalf.sphere) > 4:
7          legalCards.append('Gandalf')
8          return legalCards
9      for card in playerHand:
10         if not card.getName() in legalCards and player.
       getResourcesBySphere(card.sphere) >= card.cost:
11             legalCards.append(card.getName())
12     return legalCards
```

Listing 4.5: Find reduced legal actions for planning decision

In the questing decision, action reduction is performed using game mechanics. The sequence of game activities at the questing phase (Fig. 3.2) shows that, at first, cards are committed to the quest. The cards add up to total willpower. Next, a card is drawn from the encounter deck, and it increases the combined threat. In the end, the quest resolves, and the combined threat is subtracted from the total willpower. The player tries to make this difference positive every round, so the card commitment must be made with sufficient willpower. The task of the function `findLegalsQuesting` from Listing 4.6 is to determine all legal actions in the questing decision. Each action is a subset of the cards in the `playerCharacters` list. If the total willpower of such a subset is greater than the combined threat by a margin of 2 (line 7), then that action goes into the `legalNodes` list. For example, if the player has two cards Eowyn and Eleanor (Fig. 4.1), then the actions will be three subsets: two one-element subsets and a two-element subset. The total willpower for these subsets is 4, 1, and 5, respectively. If the combined threat is 2, then the action with Eleanor is pruned because its total willpower is way much below the combined threat.

```
1  def findLegalsQuesting(self, player, combinedThreat):
2      playerCharacters = player.getAllCharacters()
3      legalNodes = []
```

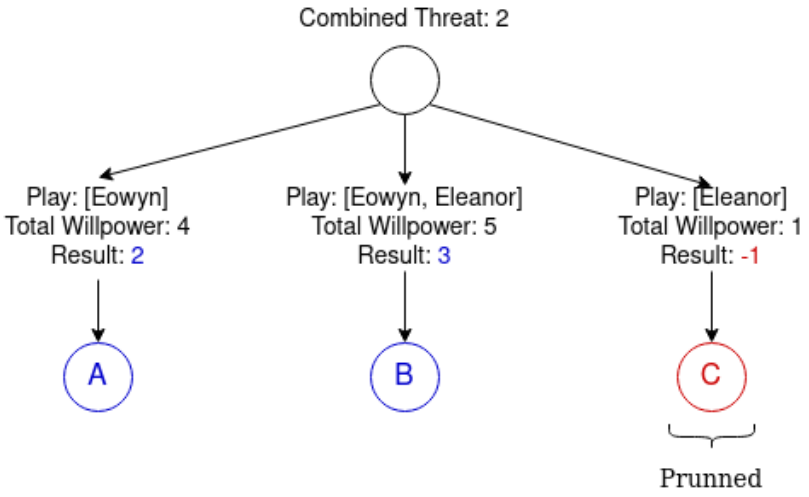

FIGURE 4.1: An example of questing reduction. Result is a difference between total willpower and combined threat. The action C is prunned since its result is below zero.

```
4     for i in range(1, len(playerCharacters)):
5         for subset in itertools.combinations(playerCharacters, i):
6             cardList = list(subset)
7             if self.getSubsetWillpower(cardList) > combinedThreat + 2:
8                 legalNodes.append(self.subsetToNames(cardList))
9     return legalNodes
```

LISTING 4.6: Find reduced legal actions for questing decision

In the defense decision, the best option for the player consists of defending with untapped allies. Using untapped allies minimizes the risk of a hero's loss; the death of all heroes means the end of the game. Therefore, subsets containing tapped allies are discarded. Each subset is checked in line 8 of Listing 4.7. If the subset contains only untapped cards, it is added to the `legalDefenders` list.

```
1 def findLegalsDefense(self, board, player):
2     enemiesEngaged = board.getEnemiesEngaged()
3     if not enemiesEngaged: return
4     playerCharacters = player.getAllCharacters()
5     legalDefenders = []
6     for subset in itertools.combinations(playerCharacters, len(
      enemiesEngaged)):
7         if not self.containsAllyTapped(list(subset)):
8             legalDefenders.append(self.subsetToNames(list(subset)))
9     return legalDefenders
```

LISTING 4.7: Find reduced legal actions for defense decision

Chapter 5

Development of RL Agent

This chapter describes the software components of the RL agent. Theoretical introduction of RL methods can be found in Section 2.2. The general workflow starts with the agent, which communicates with the environment. The agent receives an observation from the environment and selects an action. The environment executes the action and returns a new observation. This process repeats until the game succeeds or fails.

In order to perform the action the environment calls the game model. The action is decoded, then executed by the game model. The second task of the environment is rolling out turns to the end. For this purpose, pseudorandom methods are called for the travel and attack decision phases shown in the Listings 3.3 and 3.5.

5.1 Main Sequence

The adaptation of the RL learning model to LOTRCG required the creation of several modules shown in the sequence diagram (Fig. 5.1). Along with the principal environment and agent classes, there are also auxiliary classes such as simulator and encoder. The simulator class handles the exchange of information between the agent and the environment. It obtains the `observation` from the environment (function `getObservation()`), which is passed to the agent. The agent chooses `action`, which is returned to the simulator. The simulator applies this action to the environment using the `step()` function. This function returns `reward` and `nextObservation`. Those entities form a vector containing all the elements needed for the learning process, namely `observation`, `action`, `reward` and `nextObservation`. The simulator passes that vector to the agent, which performs the learning process in the `learn()` function.

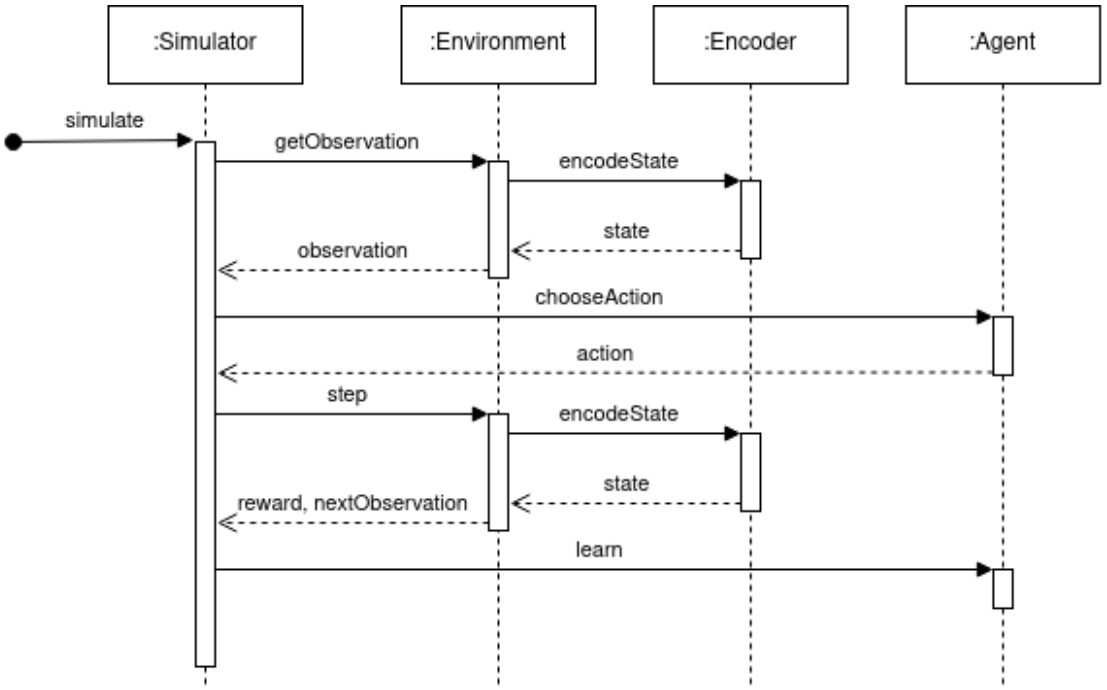

FIGURE 5.1: Sequence diagram modelling the RL workflow.

In order for the agent to know at which moment of the game he currently is, it was necessary to develop a module that encodes the game state into a vector. The encoder class is responsible for this process. Encoding of the game state is done with every query, that is, during execution of functions `getObservation()` and `step()`.

5.2 Simulator

The simulator is the main class of the RL agent. It enables the exchange of information between the agent and the environment. The communication is handled within the function `objective()`.

```
    # parameters:
    # lr - learning rate
    # n_neurons - number of neurons
    self.setAgents(params)
    best_local_avg = -1
    score_history = []
```

LISTING 5.1: Setting agents in the objective function

The first step is to initialize the agents that make decisions in each phase presented in Listing 5.1. The agents are initialized with the optimization hyperparameters reported in Section 5.7. The auxiliary variables `best_local_avg` and `score_history` are initialized for optimization process.

```
    Game_Model.globals.gameWin = False
    Game_Model.globals.gameOver = False
    episode_done = False
    score = 0
    if rlMode[0] == 'l': self.agent_planning().reset()
    if rlMode[1] == 'l': self.agent_questing().reset()
    if rlMode[3] == 'l': self.agent_defense().reset()
    pobservation = self.env.reset()
```

LISTING 5.2: Reset agents and environment in objective function

The episodes are simulated in a for loop. Each iteration begins with a reset of all agents and auxiliary variables (Listing 5.2). An observation is extracted from the environment (Fig. 5.1). The observation is embedded in a feature vector (Fig. 5.3) containing selected game state elements relevant to the phase.

```
        while not episode_done:
            reward = 0
            planning_action, next_pobservation = self.simulatePlanning(
    pobservation)
            qobservation, questing_action, next_qobservation, reward,
    episode_done = self.simulateQuesting()
            dobservation, defense_action, next_dobservation, reward,
    episode_done = self.simulateDefense()

            self.env.endRound(rlMode)

            if Game_Model.globals.gameWin:
                reward = 1
                episode_done = True
            if Game_Model.globals.gameOver and not Game_Model.globals.
    gameWin:
                reward = -1
                episode_done = True

            if rlMode[0] == 'l': self.learnPlanning(pobservation,
    planning_action, reward, next_pobservation, episode_done)
            if rlMode[1] == 'l': self.learnQuesting(qobservation,
    questing_action, reward, next_qobservation, episode_done)
            if rlMode[3] == 'l': self.learnDefense(dobservation,
    defense_action, reward, next_dobservation, episode_done)

            pobservation = self.env.encoder.encodePlanning('critic')
            score += reward
```

LISTING 5.3: Simulation of one episode

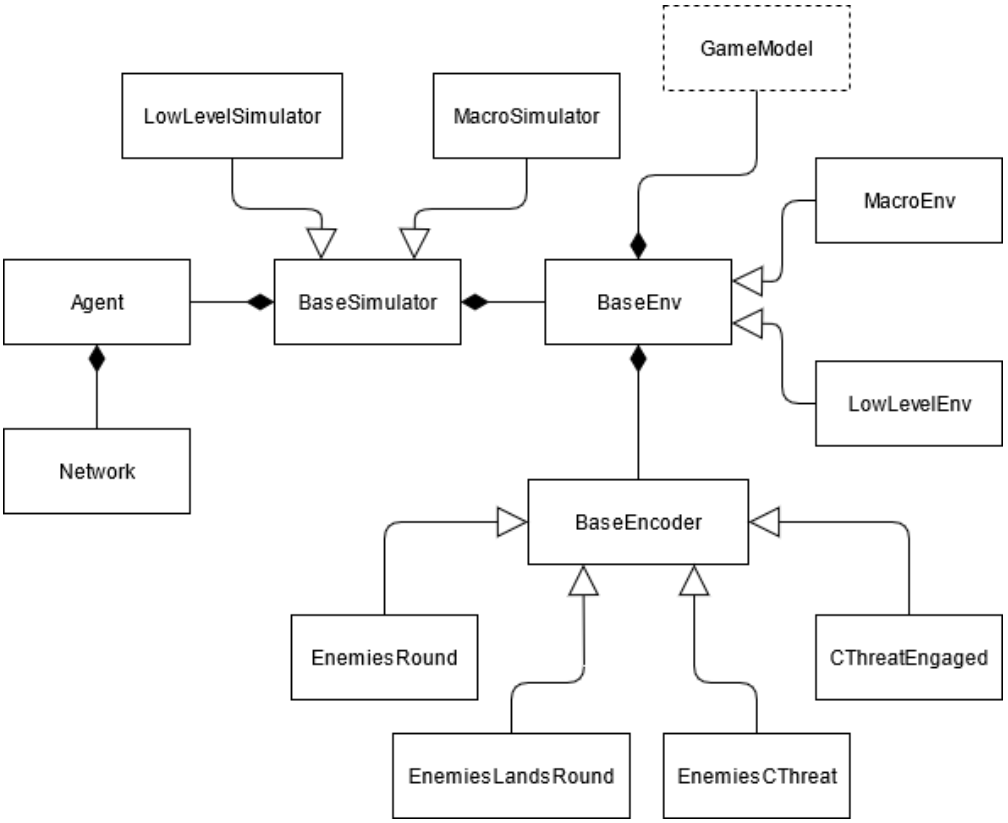

FIGURE 5.2: The simulator class diagram.

The while loop simulates the turns of a single episode (Listing 5.3). The abstract functions `simulatePlanning()`, `simulateQuesting()` and `simulateDefense()` are involved, where actions are taken. The types of actions are described in Section 5.4. These actions are executed in the environment resulting in a reward and the next observation. The round is then played to the end. If the game succeeds (fails), the reward is set to one (minus one). The last stage of the episode is the learning process. It uses the observations and the reward collected during the turn. After simulating all episodes, a hard reset of the agents is performed, and the average reward returns to the calling function.

5.3 State Encoders

Encoders handle the data flow from the environment to the agent. They fetch data from the game model and embed it in a feature vector. This vector then feeds the agent as a neural network input. The process is performed for planning and questing separately.

The feature vector for planning consists of 28 binary variables and an integer (Fig. 5.3). The binary features point to ally cards in the player's hand and enemy cards in the staging area. The integer stands for the resource pool available to the player.

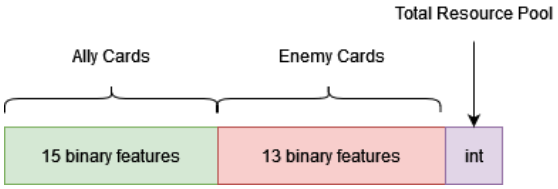

FIGURE 5.3: Feature vector for the planning phase.

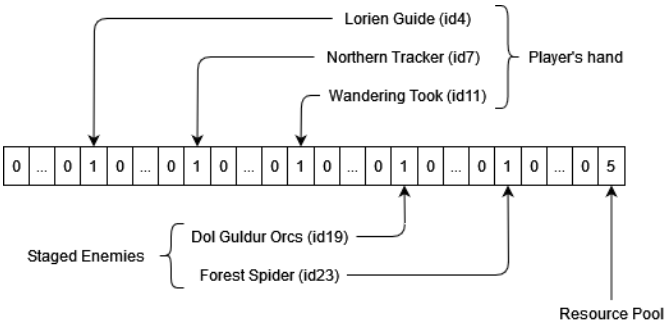

FIGURE 5.4: Example of the encoding scheme for the planning phase.

An example encoding for planning is presented in Fig. 5.4. There are three cards in the player's hand: *Lorien Guide* (id:4), *Northern Tracker* (id:7), and *Wandering Took* (id:11). The encoder iterates over the cards and sets bits of the vector according to the ids. The same process is exercised on the staged enemies - *Dol Guldur Orcs* (id:19) and *Forest Spider* (id:23) in this example. IDs of all cards are reported in Appendix A. The last bit is set to the current resource pool, 5 in this example.

The feature vector for questing also relies on binary and integer variables, but it points to different cards and statistics regarding a situation on the board as follows:

- Encoding type 0 - enemies in the staging area and round number,
- Encoding type 1 - lands, enemies in the staging area and round number,
- Encoding type 2 - enemies in the staging area and combined threat,
- Encoding type 3 - enemies in the engagement area and combined. threat.

Every vector has 18 binary variables representing the player's cards: three for hero cards and 15 for allies. The remaining features depend on a particular encoding scheme. An example feature vector of Encoding type 1 is shown in Fig. 5.5.

Vector of State encoding for defense decision consists of binary features of hero, ally, and enemy cards.

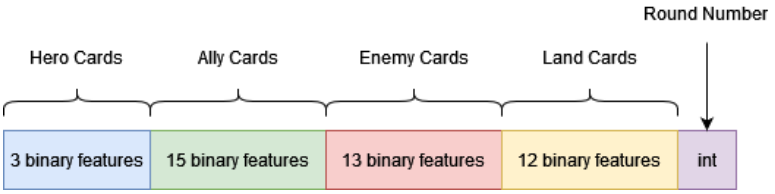

FIGURE 5.5: Feature vector of Encoding type 1 for the questing phase.

5.4 Action Decoders

Action Decoders serve as a middleware between the agent and the environment. They receive an action from the agent and translate it to an executable form suited for the environment. The form of action can be either a macroaction (abstract) or direct card choice.

The process is as follows. After receiving a game state, the agent responds with action. The environment then executes the action by playing cards within the game model. These cards can be assigned either directly or using macroactions. The selection of these two action types is motivated by letting the agent select cards freely. Direct actions can achieve this process. Macroactions were applied because they offer some level of abstraction instead of picking exact cards. Adding abstraction allows the agent to operate on a fixed number of actions. The drawback of this approach is losing a degree of freedom of choice during the decision process.

5.4.1 Direct Method

The direct method restricts the agent's response to a two-point distribution of whether a card should be played or not. Those decisions are aggregated into a binary vector, where one (zero) means the card to be played (omitted). The vector is used in the learning process of actor-critic modules. The decision aggregation starts with setting the agent's state (Fig. 5.6). Next, the program executes a loop until there are no cards affordable for the agent. The loop passes the current state to the agent and applies its response to the environment. When the breakpoint is reached, the algorithm sets the agent's cumulative action, which has been recorded with every iteration of the loop.

Table 5.1 presents an example execution of the loop. The agent has four cards (ids: 3, 10, 7, 15) in hand and five tokens in the resource pool. There are two enemies in the staging area. He decides to play card (id:10) with a cost of 2 - it shows up on the table, and the resource pool gets updated. Then the agent purchases card (id:3) with a cost of 3. The resource pool drops to zero, and the program breaks the loop. The agent's cumulative action consists of cards with id:10 and id:3 played in those two iterations.

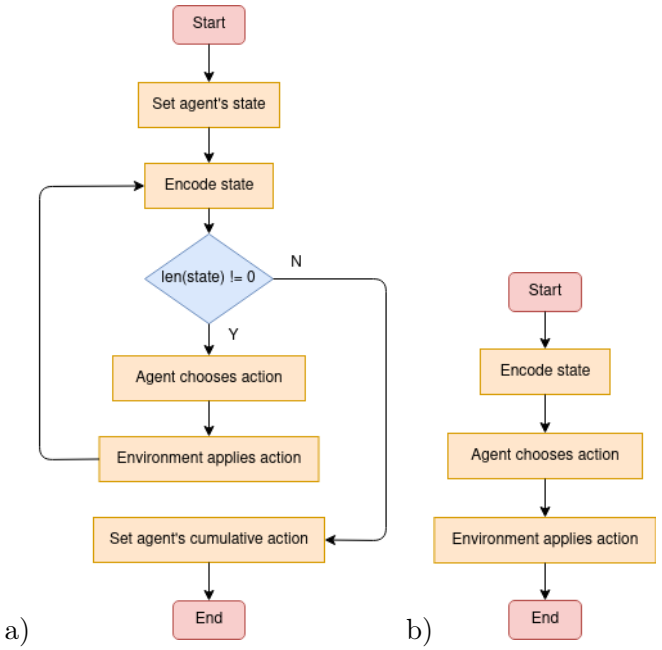

FIGURE 5.6: Control flow for the planning phase (a) and the questing phase (b).

Hand	Enemies	Res. Pool	Table	Action
3, 10, 7, 15	22, 28	5	10	Play 10
3, 7, 15	22, 28	3	10, 3	Play 3
7, 15	22, 28	0	10, 3	No resources

TABLE 5.1: An example planning action loop.

Direct actions at the questing phase are processed in a single workflow (Fig. 5.6). It begins with sending an encoded game state to the agents, taking action, and applying it to the environment. Following the previous example, now the agent has three heroes (ids: 0, 1, 2) accompanied by two allies (ids: 10, 3) acquired in the planning stage. There are still two enemies (ids: 22, 28) in the staging area. The agent decides to commit (ids: 1, 10, 3) to the quest, leaving the rest (ids: 0, 2) for later phases such as the defense or attack phase (Fig 3.2).

5.4.2 Macroactions

Macroactions base on an abstract reaction. The idea behind them is to segregate cards occurring in a particular game state. The cards are ordered according to a simple formula:

$$\frac{\beta * cw + (1 - \beta) * cd}{cc} \tag{5.1}$$

where β - a weighting coefficient, cw - card willpower, cd - card defense and cc is card cost. An action for planning means taking a particular weighting coefficient from 0 to

1 with 0.2 step. When the ordering process is done, cards are acquired until the total resource pool is depleted. For the questing decision, a similar formula applies involving willpower and defense statistics of a card. Segregated cards are committed to the quest up to the combined threat level of the staging area.

5.5 Q-Learning Agent

A key element in the decision-making process of the RL agent is the neural network. This network, along with the learning process described in Section 5.6 has been implemented using the Tensorflow (TF) library. The study includes two types of methods such as Q-Learning and Actor-Critic.

The agent using Q learning (Eq. 2.6) estimates q function in `Qnetwork` class:

```
1  class Qnetwork(keras.Model):
2      def __init__(self, name, n_actions, fc1_dim=100):
3          super(Qnetwork, self).__init__()
4          self.fc1 = Dense(fc1_dims, activation='relu')
5          self.q_values = Dense(n_actions, activation=None)
6
7      def call(self, state):
8          value = self.fc1(state)
9          q_values = self.q_values(value)
10         return q_values
```

Listing 5.4: Q Network class

The class initializes neuron layers in the constructor using the TF Dense function. The function's arguments are the number of `fc1_dim` outputs and the activation function. The hidden layer of the neural network has an rectified linear (relu) activation function ($g(x)$):

$$g(x) = \max(0, x), \tag{5.2}$$

The call method inherited from the `keras.Model` class allows calling the feedforward neural network. It takes `state` as an input argument and returns a vector of q function estimates.

The learning process for the Q-Learning Agent begins with tensor conversion of `observation` and `next_observation` variables (lines 1-2 of Listing 5.5. Those variables are a vector encoded states described in Section 5.3. Next lines (from 4 to 7) compute the q-values for those observations by calling the `Qnetwork` from Listing 5.4. The network returns vectors containing q-values for the current state and the next state - `q_current` and

`q_next` respectively. In line 8 TensorFlow function `reduce_max` selects maximal value q of `q_next` vector. That value is used for the target update (line 10) for currently played action. The update corresponds to the Equation 2.6 namely to the part:

$$R_{t+1} + \gamma \max_a Q(s_{t+1}, a), \tag{5.3}$$

where variable `maxQ1` stands for $\max_a Q(s_{t+1}, a)$. The updated vector `q_target` is then subtracted from the q-value vector of the current state (line 11). That yields the loss function.

```
1  state = tf.convert_to_tensor([observation], dtype=tf.float32)
2  next_state = tf.convert_to_tensor([next_observation], dtype=tf.float32)
3  with tf.GradientTape(persistent=True) as tape:
4      q_current = self.q_network(state)
5      q_current = tf.squeeze(q_current)
6      q_next = self.q_network(next_state)
7      q_next = tf.squeeze(q_next)
8      maxQ1 = tf.reduce_max(q_next)
9      q_target = tf.Variable(q_current)
10     q_target[action].assign(reward + self.gamma * maxQ1)
11     loss = q_target - q_current
```

Listing 5.5: Learning process for the Q-Learning Agent

The gradient of the loss function is applied to the network variables (Listing 5.6). It is done by the TF functions `gradient()`, `apply_gradients()`, which operates on the trainable variables of the `Qnetwork`.

```
1  q_network_gradient = tape.gradient(loss, self.q_network.
       trainable_variables)
2  self.q_network.optimizer.apply_gradients(zip(q_network_gradient, self.
       q_network.trainable_variables))
```

Listing 5.6: Gradient application for the Q-Learning Network

5.6 Actor-Critic Agent

The Actor-Critic (AC) agent consists of two classes `ActorNetwork` and `CriticNetwork` shown in Listings 5.7 and 5.8. The goal of `ActorNetwork` is policy approximation [46]:

$$\pi(a|s, \varphi) = [R_{t+1} + \gamma V(s_{t+1}, \omega) - V(s_t, \omega)] \nabla \ln(\pi(a|s, \varphi)), \tag{5.4}$$

where φ is trainable variables of the actor; ω - trainable variables for the critic; R_{t+1} - reward, γ - discount factor, $V(s_{t+1}, \omega)$ - value function of the next state, $V(s_t, \omega)$ - value function of the actual state. The actor class sets up neuron layers, which have rectified linear (Eq. 5.2) and softmax activation functions. The softmax function is defined as follows (Eq. 4.1 from [52]):

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}, \tag{5.5}$$

where x_i, x_j are elements of the input vector. The `call()` method returns the probability distribution of the actor's action.

```
class ActorNetwork(keras.Model):
    def __init__(self, name, n_actions, fc1_dim=100):
        super(ActorNetwork, self).__init__()
        self.fc1 = Dense(fc1_dims, activation='relu')
        self.policy = Dense(n_actions, activation='softmax')

    def call(self, state):
        value = self.fc1(state)
        probs = self.policy(value)
        return probs
```

Listing 5.7: Actor Network class

The `CriticNetwork` estimates the value function of the state according to the equation [46]:

$$V(s_t, \omega) = V(s_t, \omega) + \alpha[R_{t+1} + \gamma V(s_{t+1}, \omega) - V(s_t, \omega)]\nabla V(s_t, \omega). \tag{5.6}$$

The class consist of neural dense layers with relu activation function (Eq. 5.2). The overridden `call()` method returns a single value corresponding to the estimated value function.

```
class CriticNetwork(keras.Model):
    def __init__(self, name, fc1_dim=100):
        super(CriticNetwork, self).__init__()
        self.fc1 = Dense(fc1_dims, activation='relu')
        self.v = Dense(1, activation=None)

    def call(self, state):
        value = self.fc1(state)
        v = self.v(value)
        return v
```

Listing 5.8: Critic Network class

The agent features learn function (Listings 5.9, 5.10 and 5.11) in which the learning process is performed. The learning process consists of computing the partial derivatives of the loss function over the trainable variables of the neural network. The loss function δ is in the form of temporal difference error taken from [46] p. 258, which follows:

$$\delta = R_{t+1} + \gamma V(s_{t+1}) - V(s_t). \tag{5.7}$$

The input arguments of the learn function are all elements present in the RL pipeline (Fig. 2.2): `observation` vector on which the action was performed, the obtained `reward` and `next_observation` vector. After performing the conversion of the vectors to tensor format, the loss function is computed according to Eq. 5.7:

```
delta = reward + self.current_discount * next_state_value * (1 - int(done
    )) - state_value
```

Listing 5.9: Loss function

The argument `done` serves as an end-of-episode flag. The variable `delta` is used for actor and critic loss functions:

```
critic_loss = delta**2
probs = self.actor(self.state)
action_probs = tfp.distributions.Categorical(probs=probs)
log_prob = action_probs.log_prob(self.action)
actor_loss = -self.current_discount * delta * log_prob
```

Listing 5.10: Loss function specific for Actor and Critic

Actor loss function requires calculating logarithm of probabilities for every action (Eq. 5.4). This process is provided by TensorFlow functions `tfp.distributions.Categorical()` and `log_prob()`. The gradients for all network weights are computed using the built-in TF function. The values of the weights are updated according to these gradients for the actor and the critic networks separately. The process is executed using the TensorFlow `tape.gradient()` and `apply_gradients()` functions:

```
    critic_network_gradient = tape.gradient(critic_loss, self.critic.
    trainable_variables)
    self.critic.optimizer.apply_gradients(zip(critic_network_gradient,
    self.critic.trainable_variables))

```

hyperparameter	decision phase	values
learning rate	planning	[3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7] 10^{-4}
learning rate	questing	[6, 6.5, 7, 7.5, 8, 8.5, 9, 9.5, 10] 10^{-4}
learning rate	defense	[3.5, 4, 4.5, 5, 5.5, 6] 10^{-4}
number of neurons	planning	[50, 75, 100, 125, 150]
number of neurons	questing	[50, 75, 100, 125, 150]
number of neurons	defense	[50, 75, 100]

TABLE 5.2: Hyperparameter space.

```
4     actor_network_gradient = tape.gradient(actor_loss, self.actor.
    trainable_variables)
5     self.actor.optimizer.apply_gradients(zip(actor_network_gradient, self
    .actor.trainable_variables))
6
7     self.current_discount *= self.gamma
```

LISTING 5.11: Gradient application

At the line 7 `current_discount` is updated according to the discount factor from Eq. 5.7.

5.7 Hyperparameter optimization

Each RL agent has two hyperparameters (Tab. 5.2) for each decision moment. The first is the learning rate used during the learning process, and the second is the number of neurons in the hidden layer (`fc1dim` in Listings 5.7 and 5.8). The decision for a single hidden layer in the neural networks is motivated by the stability of the learning process. Preliminary experiments proved that two hidden layers cause a sudden drop in average reward at the end of the learning process.

The goal of the optimization is to find a point in the hyperparameter space such that the average reward from 1000 episodes is as high as possible.

The *Hyperopt* library was used for the optimization of neural networks (Bergstra et al. [53]).*Hyperopt* is a python library aimed for optimization over high scale spaces. The library handles continuous as well as discrete parameter spaces. It features three algorithms such as Random Search, Tree of Parzen Estimators (TPE) and Adaptive TPE (Bergstra et al. [54]). The library provides a simple interface, making it easy to incorporate. *Hyperopt* already proved successful with RL optimization in the card game (Vieira et al. [34]).

```
best = fmin(
    fn=simulator.objective,
    space=space,
    algo=tpe.suggest,
    max_evals=100
)
```

LISTING 5.12: hyperopt call function

Listing 5.12 presents the call script for *Hyperopt*. `space` is defined according to the Table 5.2. It is arranged as a key-value dictionary where keys are strings and values are a *Hyperopt* function `hp.choice()`. In Listing 5.12 the main optimization function is called `fmin`. The function takes the following arguments: `fn` - objective function (Listings 5.1, 5.2 and 5.3), `space` - hyperparameter space, `algo` - type of optimization algorithm (TPE in this case) and `max_evals` - the number of trials.

Chapter 6

Experiments

The following section presents the results of experiments regarding the MCTS and RL agents. The experiments were conducted in terms of the achieved winrate within the interval defined as a binomial proportion confidence interval for 95% confidence level:

$$\pm z\sqrt{\frac{p(1-p)}{n}}, \tag{6.1}$$

where z - 1.96 for 95% confidence level, p - winrate probability n_s/n, n - total number of episodes, n_s - number of wins. The agents ran on three difficulty levels such as easy, medium and hard. The levels are defined in Section 3.4. The simulations ran on a local workstation with Intel Core i9-9960X CPU, 128 GB RAM and GPU RTX 2060 Super.

6.1 AI Setup

The game consists of five decisions: planning, questing, travel, defense and attack. According to experiments shown in Fig. 3.4, three crucial decisions were identified: planning, questing and defense. For those three, there is the highest probability that an AI agent would have a choice between two or more actions.

AI agents acting at those decisions are combined into setups. For example, a MCTS-random-random setup stands for a MCTS agent for planning, a random agent for questing and a random agent for defense. In the case of reinforcement learning, for instance, random-rl-rl setup involves a random agent for planning, an RL agent for questing, and a RL agent for defense decision.

AI setup			MCTS method	
planning	questing	defense	vanilla	action reduction
MCTS	random	random	2.1 ± 0.8	4.0 ± 1.2
random	MCTS	random	7.0 ± 1.5	21.8 ± 2.5
random	random	MCTS	15.9 ± 2.2	21.0 ± 2.5

TABLE 6.1: Winrate of vanilla MCTS compared with MCTS with action reduction on hard game difficulty. Simulation budget set to 40.

6.2 MCTS

Examination of the MCTS agent began with a comparison between the standard (vanilla) algorithm and MCTS enhanced with action reduction (Tab. 6.1). The table presents three decision moments that comprise an AI setup: planning, questing and defense. Among all combinations of agents acting in those decision moments, only three setups containing one MCTS agent were investigated. For the MCTS-random-random setup the impact of action reduction is negligible since the results overlap 95% confidence interval (Eq. 6.1). The second setup containing a MCTS agent in questing and the third with MCTS in defense benefit from action reduction. The random-MCTS-random setup yields the highest winrate growth, from about 7% for the standard algorithm to almost 22% for the modified version. The third setup with standard MCTS already performs the best among other setups (about 16% winrate) and with action reduction it reaches 21% winrate. Those results suggest that the random-random-MCTS setup takes less advantage of action reduction than the random-MCTS-random setup.

The marginal impact of action reduction for the MCTS-random-random setup points that it needs a more complex mechanism of reduction. Despite being a logical strategy, the rule of prioritizing Gandalf at card purchase does not work as an action reduction method to improve winrate. Cards can only be purchased within an unpredictable budget, varying across rounds. This specificity requires adding more expert knowledge to the rule, which may prove challenging to implement.

The best winrate growth achieved for the second setup resulted from a simple expert rule in the questing decision. The rule discards subsets of cards with total willpower below a certain threshold. Section 4.4 provides a detailed description of the action reduction with a use case.

Fig. 6.1 presents a distribution of episode lengths for those three setups. The distribution regards games with success or loss. Despite the broad range of the test, no setup achieved games of one episode length. The distributions start from two episodes and last up to nine, depending on the setup. The histogram (a) shows that the MCTS-random-random

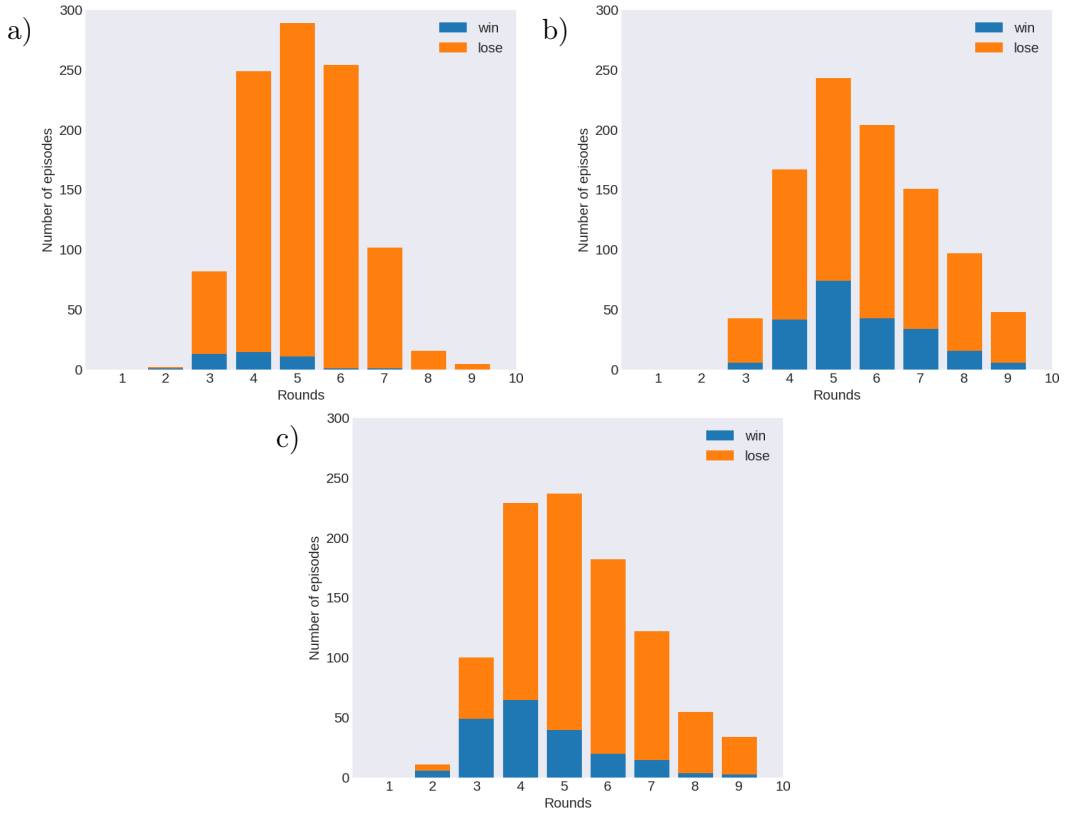

FIGURE 6.1: Win/lose distribution for planning (a), questing (b) and defense (c) decisions with MCTS agent.

setup played games between 4-6 rounds on average and those games mostly ended with a fail. Successful games last shorter, and the highest probability of winning falls within three to five rounds. The random-MCTS-random adds more entropy to the episode distribution (b). The average episode runs within a broader range of four to seven rounds regardless of their success or fail. The distribution leans toward longer episodes, and the games last more likely above five rounds. The setup with an MCTS agent in the defense decision (c) produced a similar distribution inclined to longer episodes. The skewness suggests that both setups did not play all the cards in one round but slowly built up the progress from round to round.

The next step was to test different MCTS setups with respect to different difficulty levels (Tab. 6.2). The test includes all possible setups of a MCTS agent. Among setups with one MCTS, the random-MCTS-random setup achieved the highest winrate (above 90%) for easy and medium difficulties. Changing the game to hard difficulty affects the results. The best setups (random-MCTS-random and random-random-MCTS) reach 19% of winrate.

The setups with two MCTS agents: MCTS-MCTS-random and random-MCTS-MCTS

AI setup			game difficulty		
planning	questing	defense	easy	medium	hard
MCTS	random	random	50.7 ± 3.1	50.1 ± 3.1	5.1 ± 1.4
random	MCTS	random	93.1 ± 1.6	94.7 ± 1.4	19.0 ± 2.4
random	random	MCTS	80.0 ± 2.4	77.6 ± 2.6	19.3 ± 2.5
MCTS	MCTS	random	89.3 ± 1.9	90.4 ± 1.8	14.3 ± 2.2
MCTS	random	MCTS	87.5 ± 2.0	51.3 ± 3.1	12.0 ± 2.0
random	MCTS	MCTS	94.1 ± 1.4	95.8 ± 1.2	32.7 ± 2.9
MCTS	MCTS	MCTS	97.5 ± 1.0	88.0 ± 2.0	24.3 ± 2.7

TABLE 6.2: Winrate of Random and MCTS Agents at different decision stages. Simulation budget set to 40.

achieve winrates of about 90% for easy and medium difficulties (Tab. 6.2). The random-MCTS-MCTS setup plays the game on the top level with a 32% winrate, which is the best result among all setups for that difficulty. Those outcomes indicate that the crucial stage to apply an MCTS agent is the questing decision. Adding MCTS to other decisions (planning and defense) does not increase the performance significantly except the hard level. For that difficulty employing an MCTS agent to the defense results in the highest performance boost.

The setup with three MCTS agents performs with 97.5% winrate for easy and 88% for medium levels (Tab. 6.2). For the game on the hardest difficulty, the setup does not overtake the best two MCTS setup. The winrate considerably below 30% points indicates that the setup does not take any advantage of employing a MCTS agent in the planning decision. That fact follows the previously mentioned conclusion that the currently applied action reduction to planning requires more elaborated expert rules.

The vast majority of setups achieved efficiencies of 90% or more for the easy and medium levels. These two difficulty levels do not pose a problem for MCTS agents with action reduction. Only the starting bias in the hard level, which requires the game to start with two cards in the staging area (described in Sec. 3.3), causes significant difficulty for MCTS agents.

The next phase of experiments was parameter tuning. MCTS exploits two parameters: simulation budget and playouts per node (ppn), which are described in Sec. 4.3. Figure 6.2 shows the effectiveness of MCTS algorithms with respect to different values of ppn. The simulation budget is fixed at 200. Starting from one playout per node, the winrate increases until it peaks at 20 ppn for MCTS with action reduction and 30 ppn for the standard algorithm. The difference between those peaks proves the positive impact of action reduction on the winrate. After 30 ppn both algorithms constantly decline below their starting values.

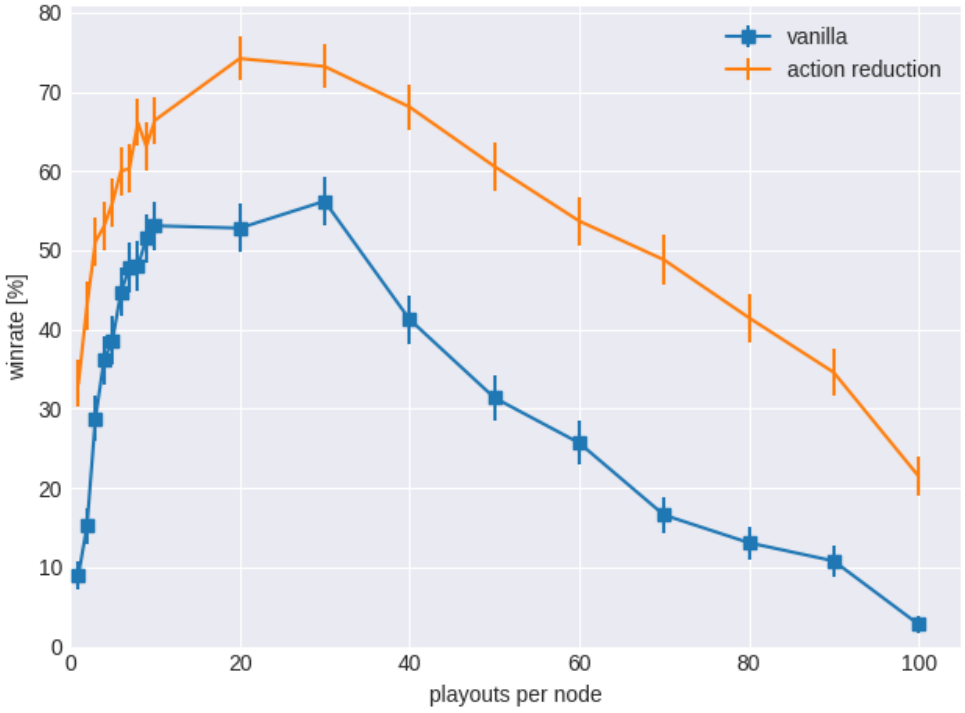

FIGURE 6.2: Playouts per node for the best MCTS setup, simulation budget set to 200, game difficulty: hard.

simulation budget	optimal ppn	winrate	time of one episode [s]
200	20	73.6 ± 2.7	1.49
400	40	78.7 ± 2.5	2.19
600	50	80.3 ± 2.5	3.63
800	80	82.8 ± 2.3	3.77

TABLE 6.3: Winrate and time of one episode for different simulation budgets for optimal playouts per node (ppn).

Higher values of the simulation budget result in a better performance of MCTS with action reduction (Tab. 6.2). Optimal playouts per node increase with the simulation budget. It reaches about 10% of the simulation budget. Setting the budget of 800 simulations with 80 playouts per node allows the agent to reach 82% winrate for hard difficulty level, which is the best result achieved by MCTS. As a drawback, increasing the budget affects the simulation time of one episode (the fourth column of Tab. 6.2).

6.3 RL

The second set of experiments was dedicated to the RL agent. The agent executed two types of actions: macroactions and direct actions. The experiments involved several

AI setup			game difficulty		
planning	questing	defense	easy	medium	hard
random	random	random	56.0 ± 1.0	53.7 ± 1.0	4.2 ± 0.4
rl	random	random	69.1 ± 0.9	58.1 ± 1.0	6.9 ± 0.5
random	rl	random	92.2 ± 0.5	95.3 ± 0.4	29.9 ± 0.9
random	random	rl	65.2 ± 0.9	62.6 ± 1.0	7.1 ± 0.5
rl	rl	random	94.6 ± 0.4	96.9 ± 0.3	60.1 ± 1.0
rl	random	rl	77.6 ± 0.8	80.3 ± 0.8	18.6 ± 0.8
random	rl	rl	91.5 ± 0.6	95.5 ± 0.4	29.4 ± 0.9
rl	rl	rl	94.2 ± 0.5	97.2 ± 0.3	59.6 ± 1.0

TABLE 6.4: Winrate of Q-learning with macroactions (rl) at different difficulty levels. Encoding type 0.

RL algorithms such as Q-learning and Actor-Critic (AC). The agent with Q-learning employed macroactions. The AC agent exploited macroactions and direct card choice.

6.3.1 Q-learning with macroactions

The first RL algorithm with macroactions was Q-Learning (Tab. 6.4). All possible combinations of RL and random agents in different decisions were verified. For an RL agent in only one decision, it is most advantageous to apply it to the questing decision; all other setups barely outperform the random-random-random setup. Better results are obtained for two RL agent setups. The rl-rl-random setup reaches 60% of winrate, which is the best result for the hard difficulty level for all setups. Comparing that result with the random-rl-random setup indicates a considerable advantage when changing from random to RL agent in the planning decision.

The random-rl-rl setup achieved a winrate close to the random-rl-random setup (about 29%). That observation leads to the conclusion that an RL agent benefits the most when applied in the questing decision. Switching a random agent to RL in the defense decision does not impact the effectiveness.

The rl-rl-rl setup performed close to the rl-rl-random setup. It follows the previous constatation that a RL agent in the defense decision does not bring any advantage since the difference between those setups falls within the confidence level.

The rl-rl-rl setup was examined for the distribution of episodes shown in Fig. 6.3. The average episode length is shorter than MCTS (Fig. 6.1), as it falls between four and five rounds. The distribution has a lower variance than MCTS. It suggests that RL agents employed one strategy, which won the game in four to five rounds. The strategy was

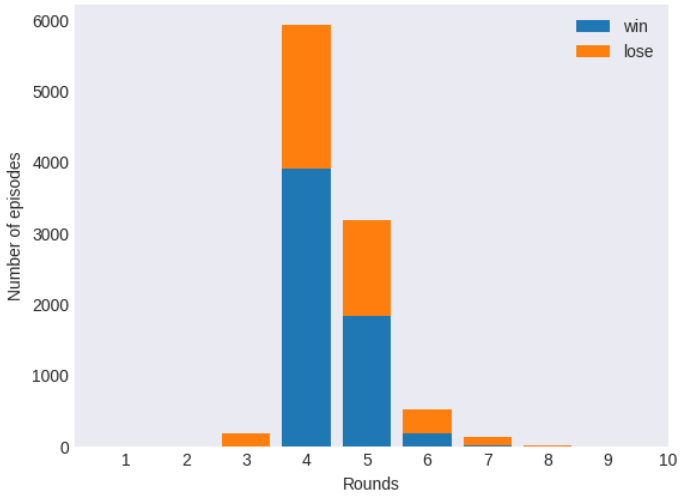

FIGURE 6.3: Episode result distribution for actor-critic with macroaction (rl-rl-rl).

AI setup			game difficulty		
planning	questing	defense	easy	medium	hard
rl	random	random	69.0 ± 0.9	66.3 ± 0.9	11.2 ± 0.6
random	rl	random	91.2 ± 0.6	91.4 ± 0.6	28.6 ± 0.9
random	random	rl	63.7 ± 0.9	54.9 ± 1.0	7.0 ± 0.5
rl	rl	random	95.0 ± 0.4	95.9 ± 0.4	66.2 ± 0.9
rl	random	rl	77.1 ± 0.8	72.1 ± 0.9	16.3 ± 0.7
random	rl	rl	91.3 ± 0.6	93.7 ± 0.5	33.6 ± 0.9
rl	rl	rl	94.8 ± 0.4	96.5 ± 0.4	64.4 ± 0.9

TABLE 6.5: Winrate of Actor-Critic (rl) with macroactions at different difficulty levels. Encoding type 0.

likely all-in, i.e., playing all cards available only in the questing decision, which turned out to be crucial as mentioned in previous tests (Tab. 6.4).

6.3.2 Actor-critic with macroactions

Macroactions were also implemented for the Actor-Critic algorithm. As seen in Tab. 6.5, the random-rl-random setup performed the best of all setups containing one RL agent. For two rl setups, adding a RL agent to the planning decision gives the advantage because the rl-rl-random setup achieved the highest winrate (66%) among all setups at the hard difficulty level. That result remains unbeaten even by the setup with a RL agent in all decisions.

Tables 6.4 and 6.5 conclude the importance of applying an RL agent to the questing decision. Every setup standing out from those tables involves a RL algorithm playing

the game in the questing phase. The setups with the questing decision driven by a random agent perform worse for all game difficulties.

AI setup			state encoding type			
planning	questing	defense	0	1	2	3
random	rl	random	34.2 ± 0.9	27.5 ± 0.9	92.9 ± 0.5	80.7 ± 0.8
rl*	rl*	random	47.8 ± 1.0	61.5 ± 0.9	94.7 ± 0.4	83.2 ± 0.7
rl**	rl**	random	42.8 ± 1.0	40.6 ± 1.0	86.1 ± 0.7	95.3 ± 0.4

TABLE 6.6: Winrate for Actor-Critic (rl) with macroactions on different stages. For hard and medium levels three encoding schemes were examined. Difficulty hard. Parameter policy: shared (*) and distinct (**).

The next experiment investigated the effect of game state encoding type on efficiency (Tab. 6.6). Two best setups were tested: one with a single RL agent (random-rl-random) and two with two RL agents (rl-rl-random). The rl-rl-random setups include two parameter policies used during optimization: shared (*) and independent (**). Shared parameters mean that neural networks for planning and questing agents are described by two variables, such as learning rate (lr) and the number of neurons (nn). In the independent case, each agent has its own lr and nn.

All setups gained an advantage when using the encoding type of 2 or 3. Both types observed the combined threat, which is a sum of threat (the parameter 9 in Fig. 3.1) of all cards in the staging area. The importance of the combined threat indicates that the RL agent is learning the game based on a condensed observation since the combined threat is a composite value that depends on the cards in the staging area, either lands or enemies.

6.3.3 Actor-critic with direct card choice

number of neurons	learning rate	encoding	winrate
100	6.5e-4	2	92.9 ± 0.5
100	8e-4	2	88.7 ± 0.6
70	7.5e-4	3	83.7 ± 0.7

TABLE 6.7: Winrate for setup: direct AC agent at planning, direct AC at questing and random agent at defense. Difficulty hard.

The network structure optimization problem was performed for the Actor-Critic agent using direct actions (Tab. 6.7, Tab. 6.8 and Tab. 6.9). Table 6.7 presents the optimized networks, which achieved the best results for setups with one RL agent. Two of them used the encoding type of 2, which allowed the winrate above 90%. The setups with two RL agents sharing the same network parameters are shown in Tab. 6.8. The best winrate

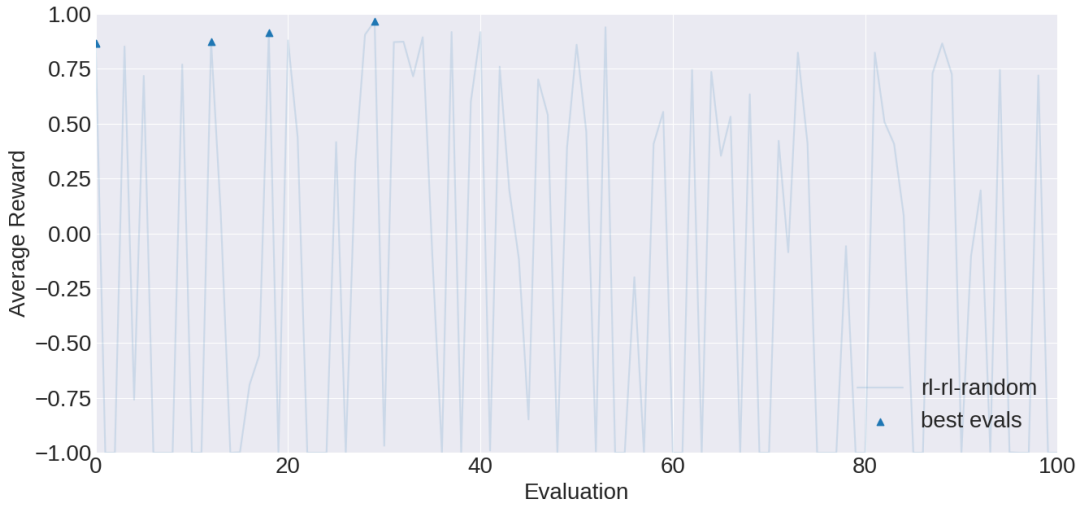

FIGURE 6.4: Hyperparameter optimization for AI setup: RL agent at planning, RL at questing and random agent at defense. Difficulty hard.

planning		questing			winrate
num. of neurons	learning rate	num. of neurons	learning rate	encoding	
130	9e-4	130	9e-4	2	94.7 ± 0.4
70	4.5e-4	70	4.5e-4	2	87.6 ± 0.7
80	7e-4	80	7e-4	2	82.7 ± 0.7

TABLE 6.8: Winrate for setup: direct AC agent at planning, direct AC at questing and random agent at defense. Difficulty hard.

planning		questing			winrate
num. of neurons	learning rate	num. of neurons	learning rate	encoding	
130	3e-4	120	9e-4	3	95.3 ± 0.4
130	5e-4	90	5.5e-4	3	89.7 ± 0.6
130	6e-4	80	8e-4	3	87.1 ± 0.7
120	5.5e-4	90	9e-4	3	85.5 ± 0.7

TABLE 6.9: Winrate for setup: direct AC agent at planning, direct AC at questing and random agent at defense. Difficulty hard.

also exceeds 90% for the encoding type 2. Tab. 6.9 depicts rl-rl-random setups with distinct network parameters as the number of neurons and learning rate are independent for each agent. Two network setups reached 90% winrate or above. Both of them used the encoding type 3.

Tables 6.7, 6.8 and 6.9 conclude that only networks containing 100 neurons in the hidden layer prove effective. Increasing the number of neurons might result in overfitting, which means the network remembers actions for given observations instead of generalizing them. Such a large number of neurons also causes instabilities in the learning process.

The tables indicate the influence of encoding type, which dominates the rl-rl-random

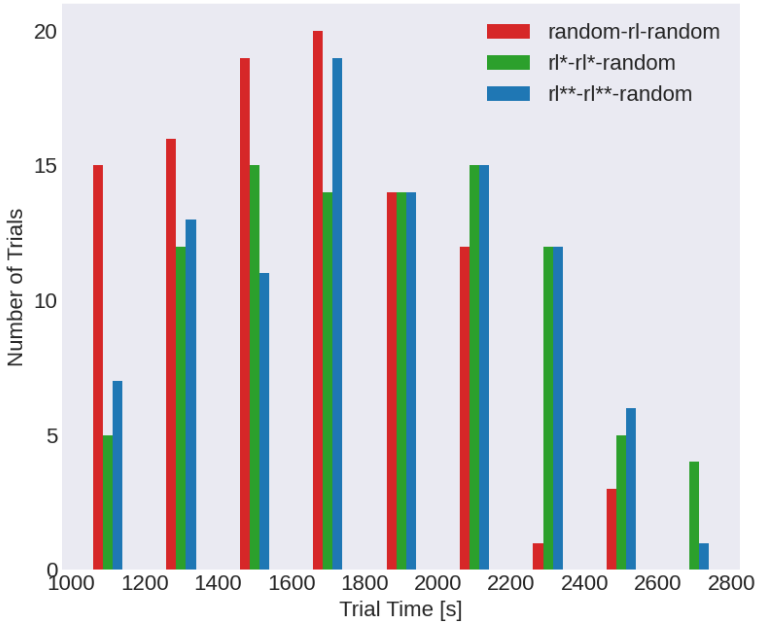

FIGURE 6.5: Histogram of trial times for random-rl-random and rl-rl-random setups. Two rl agent setups were optimized with shared (*) and distinct (**) parameters.

setups (type two and three in Tab. 6.8 and Tab. 6.9 respectively). The single RL agent setups show some inconsistency in terms of encoding types since two and three appear in Tab. 6.7.

Hyperparameter optimization consists of 100 evaluations (Fig. 6.4). Each evaluation samples the search space, which means that the AI setup plays 10000 episodes in order to learn the RL agents. The evaluation function tracks the best average reward from 1000 episodes. It is recorded as the value of the evaluation function (faded blue plot on Fig. 6.4). The triangle markers reflect the best trials achieved during the optimization process. The first point is obtained in trial number one, which arises from a preprocessing stage performed by the optimization algorithm (Sec. 5.7).

6.4 Best Players

This section compares the MCTS and RL algorithms for the best agent configurations. The first algorithm was chosen for the setup of the random agent in the planning decision, MCTS in questing, and defense decisions (Tab. 6.2). This setup operating on 800 simulation budget achieved 82.8 % winrate at hard level (Tab. 6.3).

The RL algorithm was chosen for two configurations. The first is the RL agent only in the questing decision. This choice was motivated by the fact that using RL in questing yields the best results compared to the other single setups, as can be seen by comparing the first

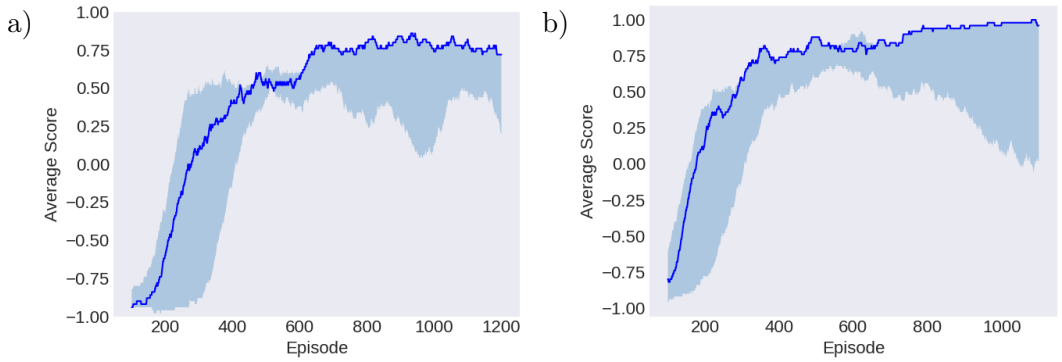

FIGURE 6.6: Learning curve for random-rl-random (a) and rl-rl-random (b) setups.

AI setup			Avg. time			
planning	questing	defense	game [s]	learning [s]	hyp. opt. [h]	winrate
random	mcts	mcts	3.77	-	20.9	82.8 ± 2.3
random	rl	random	0.017	1595	44.8	92.9 ± 0.5
rl*	rl*	random	0.021	1812	50.1	94.7 ± 0.4
rl**	rl**	random	0.021	1780	49.3	95.3 ± 0.4

TABLE 6.10: Final comparison of four best AI setups. Game difficulty hard. MCTS Simulation budget set to 200. Entries with shared (*) and distinct (**) parameters.

three rows of Tables 6.4 and 6.5. The random-rl-random setup stands out by achieving less than 30% winrate at the hard difficulty level. The second setup is the best among those containing two RL agents. The rl-rl-random setup achieved a 60% winrate, which turned out to be the best result on the hard difficulty level before optimization(Tab. 6.4).

The learning curves for rl setups are shown in Fig. 6.6. For random-rl-random, the network reaches avg_score above 0.75 already around the 1200th episode; after that value, the increment is smaller. Due to the large fluctuations of the learning curve, the network is saved after reaching the best avg_score of the last 1000 episodes. The rl-rl-random setup achieves winrate close to one after 800 episodes, then the learning curve saturates.

The number of RL agents in a setup has a strong influence on the length of the learning process presented in Fig. 6.5. It shows the time distribution of trials performed during the optimization process. Each such trial means executing a number of episodes fixed at 10000. The setup with one RL agent (random-rl-random) achieves significantly shorter times than rl-rl-random. This phenomenon is because adding a second agent RL causes a doubling of the number of weights in neural networks to be optimized. For rl-rl-random two parameter policies were optimized such as shared or independent. The results from Fig. 6.5 show that those policies have no impact on trial times, since number of neurons is roughly the same for both cases.

The final comparison is presented in Tab. 6.10. It shows the advantage of the RL algorithm over MCTS at the hard difficulty level. The equal winrate between the configuration with one and two RL agents is also worth noting. At this point, the thesis can be put forward that one RL agent in one critical decision is enough for the system to master the game to a degree above 90%. Adding a second RL agent does not significantly increase efficiency.

When comparing algorithms, it is also essential to discuss their computational cost. These are presented by the average time in Table 6.10. The time of a single game for MCTS is 3.7s, which is incomparably higher than the time required for a pre-learned RL agent to decide. However, learning in the RL algorithm is time-consuming and takes approximately 30 minutes. In comparison, the selection of optimal hyperparameter values took about 50 hours. The differences highlight the fundamental difference between the MCTS and RL algorithms.

6.5 Conclusions

The experiments show that both MCTS and RL methods can play the LOTRCG at a satisfactory level. The crucial aspect is to identify the game decisions in which the application of AI agents benefits the most. For MCTS, the decisions were questing and defense, and for RL, planning, and questing.

Along with multiple agents, the setup consisting one RL agent also proved well in the experiments. The setup of random-rl-random significantly outperformed the best MCTS setup reaching the winrate of two RL agent setups.

6.5.1 Future

The learning process of RL agents can be performed in a hybrid way in the future. That means learning a single agent setup and then making up the final setup, which will consist of RL agents only. For example, two separate setups can be learned: rl-random-random and random-rl-random. After learning, those RL agents will be combined into one setup, which will be employed for testing.

Chapter 7

Summary

This thesis presents two AI methods applied in the fantasy card game "The Lord of the Rings: The Card Game". The last chapter's overview of the whole research process highlights the author's main achievements. The finite simulation resources force limits on the research area, but further work directions will be proposed.

7.1 Overview of Research Effort

The first part of the work was the implementation of a LOTRCG game model. The game mechanics plays in rounds in which decisions are taken sequentially. The game model was used by both MCTS and RL. In MTCS for playouts and in RL as a learning environment. The model was slightly modified compared with the original game rules. The implementation extends the game by three difficulty levels, but some aspects of the cards have been simplified.

The first AI agent developed was the vanilla MCTS. All of its phases were implemented, which means that the game state tree is built in real-time as phases are executed in rounds. It is worth noting that the tree could not contain illegal actions, so the MCTS iteration was done under the strict control of the game rules. Then vanilla MCTS was modified with action reduction enhancement. The modification makes the decision process upon a reduced set of actions. This reduction was based on expert knowledge closely related to the mechanics of LOTRCG.

The second AI agent is based on the reinforcement learning approach, in which the basic idea is unsupervised learning using trial and error iterations. In order to enable agent-environment communication, it was necessary to create domain-specific RL middleware. One of the tasks of this middleware is to encode the game state, that is, to embed the

current game situation into a vector that was fed to the neural network. Four encoding types were proposed, covering different game elements relevant to the decision-making process.

Another process performed by the RL middleware was action decoding, which converted the one-hot output vector obtained from the neural network into actions executed by the game simulator. Two decoding methods were proposed direct actions and macroactions. Direct actions mean that the agent's action is translated into specific cards to be played. Macroactions mean that the agent chooses from a set of abstract actions. Each such action sorted the available cards. The segregation was based on a heuristic function containing expert knowledge.

Two popular RL algorithms were used: Q-learning and Actor-Critic (AC). In the former, the decision was to use only macroactions because only they provide a fixed number of actions. AC method comes in two variants, macro and direct actions. Since the AC algorithm combines both value function and policy approximation, it was possible to develop a learning process with a varying number of actions in each episode.

In testing the MCTS algorithm, different setups were considered for the three decisions. The best setup turned out to be a setup with agent random in planning, MCTS in questing, and defense. For this configuration, the effect of action reduction was investigated. It proved to be a significant improvement of MCTS in terms of winrate. After parameter tuning, it was possible to achieve a winrate for the best configuration of 82%. Experiments were carried out to show the significant impact of the number of playouts implemented for a single decision node on the final efficiency of the algorithm.

The study of the RL methods started with testing macro-actions for different agent setups. Two setups achieved the best winrate: a single RL agent in questing (random-rl-random) and two RL agents in planning and questing (rl-rl-random).

These two setups were then tested for the effect of the game state encoding on winrate. In both cases, the coding appeared to have little reflection on the efficiency of the agents' mastery of the game.

The next step was to optimize the neural networks used in the RL agents. The parameters of these networks were the learning rate and the number of neurons in the hidden layer.

In the final stage, the thesis compared the best MCTS and RL configurations. RL methods achieved better results; however, it was followed by a learning process lasting about 30 minutes. The thesis also examined the decision time of those setups. In this aspect, RL algorithms also outperformed MCTS. That prevalence arose because a neural network query is shorter than iterating MCTS during the game.

All the conducted experiments prove the thesis about the practical application of artificial intelligence to LOTRCG. Both the MCTS algorithm and the RL methods, with modifications involving expert knowledge, achieved results indicating a mastery of the game.

7.2 Main Achievements

The main aim of the thesis was to verify whether AI methods can play successfully in the strategic card game The Lord of the Rings: The Card Game. Until now, this multi-stage fantasy card game has not been subject to such scientific investigation. The results of the experiments showed that the developed algorithms could win in more than 95% of the cases. This clearly confirms that the goal has been achieved.

Moreover, in the author's opinion, three elements deserve to be recognized as the main achievements concerning the current state of knowledge.

- Action-reduction extension for MCTS method is well known and popular approach [40]. However, applications of that technique in card games with complex mechanics seem to be barely found. Zhang and Buro [24]) proposed chance event bucketing for the Hearthstone card game. The technique allows limiting the combinatorial complexity of the MCTS search. A similar approach was presented by Choe and Kim [41]. The authors introduced sparse sampling of chance nodes to restrict the decision tree. The method defines the maximum number of visits for a chance node. The number is tuned empirically specific to the game.

  The method described in the thesis (Sec. 4.4) enables to limit the decision tree based on expert rules. The technique showed a positive impact in terms of game winrate and efficiency of the agent. However, one needs to have game experience in order to apply the approach. Another important observation is that the optimal number of playouts performed for a single decision node should be about 10% of the total computational budget.

- The second achievement is the application of Reinforcement Learning algorithms in a partially observable environment with a varying number of actions. The RL agent acting with a floating number of actions remains barely considered by academia. A card game of that type is Hanabi. The game was used as a testbed for different RL algorithms by Grooten et al. [50]. The authors used a simplified game model to avoid floating action space. Guan et al. [47] proposed a different solution for the problem by applying abstract actions.

The thesis compares two approaches for a variable number of actions: macroactions and direct choice (Sec. 5.1). Both demonstrated a similar level of winrate, even though direct card choice offers more discretion.

- Game with multiple decision stages significantly increases the complexity of the problem for AI players. Therefore, it is beneficial to identify the stages that affect the winrate meaningfully. Popular solutions are simplifications based on expert knowledge. Cowling et al. [14] present such an approach for the strategic card games, where authors omitted decisions other than the combat phase.

  The methodology proposed by this thesis makes use of random agents to identify the key decision stages in the game (Fig. 3.4). Then different AI agent setups for those critical decisions were tested to optimize winrate.

7.3 Perspective Research

LOTRCG offers a cooperative mode in addition to the solo mode described in this paper. This mode seems to be a challenging direction for further project development because it allows interaction between an AI agent and a human. Due to practical limitations, the learning process must be carried out with two players controlled by Artificial Intelligence. When they reach an appropriate level of winrate, it is possible to switch to playing with a human.

Implementing a more accurate game model is also one of the future directions. Physical cards have game text that describes additional mechanics to be used during gameplay. In order to make the game text understandable for AI, it must be serialized, which can be a complex process for a large set of cards.

Another aspect is creating hybrid algorithms. They will be based on making decisions with MCTS or RL in a mixed fashion, for example, planning with MCTS and questing with RL. Then it is possible to obtain the best configuration while reducing the learning time of such an arrangement of agents.

The learning process can also be carried out in stages, after which the algorithm will create the final setup. First, the single RL agents in each decision will be learned: rl-random-random, random-rl-random, and random-random-rl. Then, a final one containing three RL agents will be assembled from these three setups. By breaking up the learning process in this way, better control and elimination of instabilities can be achieved when optimizing the neural network's weights.

RL agents can form a hierarchical structure in which there will be a unit analyzing the cards played in previous rounds. With such an architecture, it is possible to generalize entire sequences of rounds, in which some have been played defensively and others focused on eliminating enemies. In addition to classical feedforward networks, such architecture may also include, for example, recurrent networks of LSTM type, which will accumulate time dependencies between whole rounds.

7.4 Final note

Over the years, researchers have focused their attention on algorithms based on MCTS or CFR, while RL methods have received little attention in the context of card games. This thesis clearly shows that the RL approach does not have to be niche but can provide motivation to apply trial-and-error methods to other card games. Those games do not have to be based on a standard 52-element deck but can feature complex round mechanics that give them strategic depth, as in LOTRCG.

LOTRCG presents many development directions for Artificial Intelligence, particularly Reinforcement Learning. It can be valuable in the scientific community. However, it can also answer the problem of a human player being unable to find another person to cooperate in a card game themed in Tolkien's Middle-Earth Universe.

Appendix A

Table of cards

att. - attack; def. - defence; hit. - hitpoints; will. - willpower;

name	id	type	att.	def.	hit.	will.	cost	threat
Dunhere	0	Hero	2	1	4	1	-	-
Eowyn	1	Hero	1	1	3	4	-	-
Eleanor	2	Hero	1	2	3	1	-	-
Lorien Guide	3-5	Ally	1	1	2	0	3	-
Northern Tracker	6-8	Ally	2	2	3	1	4	-
Wandering Took	9-11	Ally	1	1	2	1	2	-
Rider of Rohan	12-14	Ally	2	0	2	2	3	-
Gandalf	15-17	Ally	4	4	4	4	5	-
Dol Guldur Orcs	18-20	Enemy	2	0	3	-	-	2
Dol Guldur Beastmaster	21	Enemy	3	1	5	-	-	2
Forest Spider	22-25	Enemy	2	1	4	-	-	2
East Bight Patrol	26	Enemy	3	1	2	-	-	3
Black Forest Bats	27	Enemy	1	0	2	-	-	1
King Spider	28-29	Enemy	3	1	3	-	-	2
Ungoliants Spawn	30	Enemy	5	2	9	-	-	2
Necromancers Pass	31	Land	-	-	2	-	-	3
Enchanted Stream	32-33	Land	-	-	2	-	-	2
Old Forest Road	34-35	Land	-	-	1	-	-	3
Forest Gate	36-37	Land	-	-	4	-	-	2
Great Forest Web	38-39	Land	-	-	2	-	-	2
Mountains of Mirkwood	40	Land	-	-	3	-	-	2

Table A.1: LOTRCG simulator cards statistics.

Bibliography

[1] Alan M Turing. Digital computers applied to games. *Faster than thought*, 1953.

[2] Irving L Finkel. *Ancient board games in perspective: papers from the 1990 British Museum colloquium with additional contributions*. British Museum Press, 2007.

[3] Bruce Pandolfini. *Kasparov and Deep Blue: The historic chess match between man and machine*. Simon and Schuster, 1997.

[4] Joel Niklaus, Michele Alberti, Vinaychandran Pondenkandath, Rolf Ingold, and Marcus Liwicki. Survey of artificial intelligence for card games and its application to the swiss game jass. In *2019 6th Swiss Conference on Data Science (SDS)*, pages 25–30. IEEE, 2019.

[5] Henry N Ward, Daniel J Brooks, Dan Troha, Bobby Mills, and Arseny S Khakhalin. AI solutions for drafting in Magic: the Gathering. *arXiv preprint arXiv:2009.00655*, 2020.

[6] Andreas Stiegler, Keshav P Dahal, Johannes Maucher, and Daniel Livingstone. Symbolic reasoning for hearthstone. *IEEE Transactions on Games*, 10(2):113–127, 2017.

[7] Konrad Godlewski and Bartosz Sawicki. Optimisation of MCTS Player for The Lord of the Rings: The Card Game. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, pages e136752–e136752, 2021.

[8] Stefano Di Palma and Pier Luca Lanzi. Traditional wisdom and monte carlo tree search face-to-face in the card game scopone. *IEEE Transactions on Games*, 10(3): 317–332, 2018.

[9] Mark JH van den Bergh, Anne Hommelberg, Walter A Kosters, and Flora M Spieksma. Aspects of the cooperative card game hanabi. In *Benelux Conference on Artificial Intelligence*, pages 93–105. Springer, 2016.

[10] Hirotaka Osawa. Solving hanabi: Estimating hands by opponent's actions in cooperative game with incomplete information. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[11] Stefan Edelkamp. Challenging human supremacy in skat. In *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[12] Colin D Ward and Peter I Cowling. Monte Carlo search applied to card selection in Magic: The Gathering. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 9–16. IEEE, 2009.

[13] Denis Robilliard, Cyril Fonlupt, and Fabien Teytaud. Monte-carlo tree search for the game of "7 wonders". In *Workshop on Computer Games*, pages 64–77. Springer, 2014.

[14] Peter I Cowling, Colin D Ward, and Edward J Powley. Ensemble determinization in monte carlo tree search for the imperfect information card game Magic: The gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, 4 (4):241–257, 2012.

[15] Todd W Neller and Marc Lanctot. An introduction to counterfactual regret minimization. In *Proceedings of Model AI Assignments, The Fourth Symposium on Educational Advances in Artificial Intelligence (EAAI-2013)*, volume 11, 2013.

[16] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

[17] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[18] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

[19] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019.

[20] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Evolving card sets towards balancing dominion. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[21] Jason Noble. Finding robust texas hold'em poker strategies using pareto coevolution and deterministic crowding. In *in ICMLA*. Citeseer, 2002.

[22] Jakub Kowalski and Radostaw Miernik. Evolutionary approach to collectible arena deckbuilding using active card game genes. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[23] André Santos, Pedro A Santos, and Francisco S Melo. Monte Carlo Tree Search experiments in Hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 272–279. IEEE, 2017.

[24] Shuyi Zhang and Michael Buro. Improving Hearthstone AI by learning high-level rollout policies and bucketing chance node events. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 309–316. IEEE, 2017.

[25] Maciej Świechowski, Tomasz Tajmajer, and Andrzej Janusz. Improving Hearthstone Ai by combining MCTS and supervised learning algorithms. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[26] Mateo García Pérez and Jorge Sainero Valle. Jugador automático de hearthstone usando árboles de monte carlo y algoritmos genéticos. 2020.

[27] Daniel Diosdado López. *Sistema de Decisión Inteligente. Teoría de Juegos. Diseño, aplicación y evaluación.* PhD thesis, Universitat Politècnica de València, 2018.

[28] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding Klondike solitaire with Monte-Carlo planning. In *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[29] Johannes Heinrich and David Silver. Smooth uct search in computer poker. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[30] Matthew L Ginsberg. GIB: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303–358, 2001.

[31] Sebastian Kupferschmid and Malte Helmert. A Skat player based on Monte-Carlo simulation. In *International Conference on Computers and Games*, pages 135–147. Springer, 2006.

[32] Silvan Sievers and Malte Helmert. A doppelkopf player based on UCT. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 151–165. Springer, 2015.

[33] Daniel Whitehouse, Peter I Cowling, Edward J Powley, and Jeff Rollason. Integrating monte carlo tree search with knowledge-based methods to create engaging play in a commercial mobile game. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[34] Ronaldo Vieira, Anderson Rocha Tavares, and Luiz Chaimowicz. Drafting in collectible card games via reinforcement learning. In *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 54–61. IEEE, 2020.

[35] Nathan R Sturtevant and Adam M White. Feature construction for reinforcement learning in hearts. In *International Conference on Computers and Games*, pages 122–134. Springer, 2006.

[36] Pablo Barros, Ana Tanevska, and Alessandra Sciutti. Learning from learners: Adapting reinforcement learning agents to be competitive in a card game. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 2716–2723. IEEE, 2021.

[37] Jonathan Rubin and Ian Watson. Computer poker: A review. *Artificial intelligence*, 175(5-6):958–987, 2011.

[38] Stephen J Smith, Dana Nau, and Tom Throop. Computer bridge: A big win for ai planning. *AI magazine*, 19(2):93–93, 1998.

[39] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[40] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte Carlo Tree Search: A review of recent modifications and applications. *Artificial Intelligence Review*, pages 1–66, 2022.

[41] Jean Seong Bjorn Choe and Jong-Kook Kim. Enhancing Monte Carlo Tree Search for playing Hearthstone. In *2019 IEEE Conference on Games (CoG)*, pages 1–7. IEEE, 2019.

[42] Hiroyuki Ihara, Shunsuke Imai, Satoshi Oyama, and Masahito Kurihara. Implementation and evaluation of information set Monte Carlo Tree Search for Pokémon. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2182–2187. IEEE, 2018.

[43] Nick Sephton, Peter I Cowling, Edward Powley, and Nicholas H Slaven. Heuristic move pruning in Monte Carlo Tree Search for the strategic card game Lords of War. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–7. IEEE, 2014.

[44] James Goodman. Re-determinizing MCTS in Hanabi. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[45] Hendrik Baier, Adam Sattaur, Edward J Powley, Sam Devlin, Jeff Rollason, and Peter I Cowling. Emulating human play in a leading mobile card game. *IEEE Transactions on Games*, 11(4):386–395, 2018.

[46] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[47] Yang Guan, Minghuan Liu, Weijun Hong, Weinan Zhang, Fei Fang, Guangjun Zeng, and Yue Lin. Perfectdou: Dominating doudizhu with perfect information distillation. *arXiv preprint arXiv:2203.16406*, 2022.

[48] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019.

[49] Nolan Bard, Jakob N Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, et al. The hanabi challenge: A new frontier for ai research. *Artificial Intelligence*, 280:103216, 2020.

[50] Bram Grooten, Jelle Wemmenhove, Maurice Poot, and Jim Portegies. Is vanilla policy gradient overlooked? analyzing deep reinforcement learning for hanabi. *arXiv preprint arXiv:2203.11656*, 2022.

[51] Zhiyuan Yao, Tianyu Shi, Site Li, Yiting Xie, Yuanyuan Qin, Xiongjie Xie, Huan Lu, and Yan Zhang. Towards modern card games with large-scale action spaces through action representation. *arXiv preprint arXiv:2206.12700*, 2022.

[52] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[53] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[54] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.