

Bachelor Thesis

Exploring the applicability of agent-based AI to the game Onitama

Philemon Schulz
(matriculation number 1835051)

December 3, 2024

Submitted to
Data and Web Science Group
Dr. Christian Meilicke
University of Mannheim

Abstract

In this paper, we explore the applicability of agent-based artificial intelligence (AI) to the two-person board game Onitama. We develop five different agents: a random agent, a heuristic agent, and three different versions of Monte Carlo Tree Search (MCTS) - a simple MCTS agent based on the UCT (Upper Confidence bounds applied to Trees) search, combined with a move prioritization during the simulation phase; a heuristic MCTS agent using hardcoded heuristics and simulation cutoffs; and an MCTS agent using Rapid Action Value Estimation (RAVE). We explain how we developed and tested the different agents and test them against an agent of a fellow student of ours. We discover that the most performing agents are the basic MCTS agent and the RAVE agent. However, we expect the RAVE agent to work better than the basic MCTS agent when used in even more complex environments.

Contents

Abstract	ii
1 Introduction	1
2 Rules and strategies for the game Onitama	4
2.1 Onitama - General information and rules	4
2.2 Possible strategies and heuristics	5
2.3 Similar games and their used methods	7
2.3.1 Chess	7
2.3.2 Go	8
2.3.3 Shogi	9
2.4 Game-theoretical analysis of Onitama	10
3 Game AI algorithms and methods	12
3.1 MiniMax Search	12
3.2 Alpha-beta Pruning	14
3.3 Monte-Carlo Tree Search (MCTS)	15
3.4 Rapid Action Value Estimation MCTS	18
3.5 Heuristic MCTS	19
4 Development of the AI	21
4.1 Random agent	21
4.2 Heuristic agent	21
4.3 MCTS agents	23
5 Experimental Evaluation	26
5.1 Parameter testing	26
5.2 Playing strength testing	29
6 Real life testing against a different AI	34
7 Conclusion	36
Bibliography	38
Ehrenwörtliche Erklärung	41

1 Introduction

In recent years, artificial intelligence (AI) has become a powerful force, impacting many areas of life and transforming our economy. AI influences nearly every facet of today's life, from healthcare and education to business and research, and with that, enables a speed up in progress that was hard to imagine before (Nadikattu 2016) (Butson and Spronken-Smith 2024). Therefore, the potential of AI has garnered significant attention from researchers and industry professionals, as can be seen in an article by the OECD (Filippucci et al. 2024), one of the most influential economic forums worldwide.

Historically, games have played an essential role in developing and researching AI. Alan Turing, widely considered the principal inventor of computer science, (re)invented the MiniMax algorithm, one of the most influential AI algorithms in the early phase of AI, and used it to play chess (Georgios N. Yannakakis 2018). Later, Arthur Samuel invented the first form of a reinforcement learning algorithm that mastered the game of checkers by playing against itself.

Today, games continue to serve as ideal testbeds for AI research and development. The strategic importance of games for AI research lies in their controlled environments, complexity and challenge, and rich human-computer interaction (Georgios N. Yannakakis 2018). Games enable researchers to experiment with AI algorithms in structured settings while still offering significant complexity. Moreover, games facilitate the study of AI's ability to engage with human players in dynamic and complex ways, which is crucial for developing AI systems that can understand and respond to human behavior effectively.

The advancements made in game-AI often have far-reaching implications for real-world applications (Georgios N. Yannakakis 2018). Techniques developed for game-AI, such as reinforcement learning and decision-making under uncertainty, can be adapted to solve complex problems in fields like robotics, autonomous vehicles, and strategic planning. Additionally, serious games and gamification techniques leveraging AI are increasingly being used in education, training, and other fields, creating more effective and engaging learning experiences.

However, it's not just their excellent usability for research that makes games so relevant to AI. Today, the gaming industry itself has become a significant economic force, with a recent report of PWC expecting the global game industry to be worth over 320 billion dollars by 2026 ¹. This economic significance has further intensified the focus on AI applications in gaming. AI is now being used not only to create more challenging and adaptive opponents for players but also to generate game content, personalize player experiences, and even assist in game design and development processes.

¹<https://www.pwc.com/gx/en/issues/business-model-reinvention/outlook/insights-and-perspectives.html>, Accessed 05.11.2024

Of all the games used in AI research, Onitama offers a distinct set of challenges that make it especially interesting for testing agent-based AI. Unlike many classic board games, where piece types decide the different movement possibilities, Onitama uses shared movement cards, which means players have to adapt not only to the board layout but also to a constantly shifting set of moves that both players can use. This design demands a careful mix of strategic planning and flexibility, as each player has to think through how both they and their opponent might use the changing cards in upcoming turns. The need to balance immediate tactics with long-term goals makes Onitama an ideal environment for exploring different AI approaches.

Before we continue with this paper, we would like to provide a rough understanding of what AI and specific agent-based AI is.

AI stands for Artificial Intelligence and was first presented by John McCarthy at the Dartmouth conference as "the science and engineering of making intelligent machines" (McCorduck and Cfe 2004). Since then, finding a persistent definition for AI has been challenging; Yet, historically, most attempts oscillated between rationalistic and humanistic approaches (Collins et al. 2021). Rationalistic approaches focus on AI as systems that think and act rationally, emphasizing technical specifications. In contrast, the humanistic approach frames AI in terms of its ability to emulate human intelligence. This perspective dates back to the Turing test, which sets human capabilities and behavior as a reference point. Therefore, the humanistic approach allows for a more flexible understanding of AI, allowing it to adapt to changing expectations over time (Collins et al. 2021).

To bridge the gap, the European Commission’s High-Level Expert Group on AI (AI HLEG) recently proposed a definition that incorporates both technical specifications and social purpose (AI 2019). They define AI systems as "software (and possibly also hardware) systems designed by humans that, given a complex goal, act in the physical or digital dimension by perceiving their environment through data acquisition, interpreting the collected structured or unstructured data, reasoning on the knowledge, or processing the information, derived from this data and deciding the best action(s) to take to achieve the given goal".

Agent-based AI can be allocated to the rationalistic view of AI, as it focuses on the development of autonomous entities called agents that can perceive their environment, reason about it, and take actions to achieve specific goals. In this context, an agent is a software or hardware system that is situated in a particular environment and is capable of flexible, autonomous action to meet its initial objectives (Russell and Norvig 2010). By simulating individual agents, agent-based AI allows to model complex scenarios and observe interactions with the environment based on certain actions. In the case of Onitama, the agents are the different players, Red and Blue, the environment is the game board with its pieces and cards, and the actions are the players moves, through which the gameplay changes.

In this paper, we implement and compare five distinct AI approaches — a random

1 Introduction

agent, a heuristic agent, a simple Monte Carlo Tree Search (MCTS) agent, an MCTS Rave agent, and a Heuristic MCTS agent — to examine how each method performs within Onitama’s unique constraints. By evaluating these agents against one another, we aim to identify the strengths and limitations of each approach in handling Onitama’s blend of deterministic and dynamic elements. These insights can be extended beyond the game, highlighting the adaptability and decision-making efficiency of different AI strategies in scenarios where agents must continuously adapt to shifting variables. The code for our developed agents can be found in our ² GitHub repository. Further information about how to run it can be found in the ReadMe.

The remainder of this paper is structured as follows: In the second chapter, we provide the general concept and rules of Onitama and present possible strategies and heuristics for it. We further present games similar to Onitama and conduct a game-theoretical analysis. In the third chapter, important game AI algorithms are presented. Continuing in the fourth chapter, we explain how we developed the five different AI agents for Onitama. Then, we conduct experiments and evaluate them in chapter five, and we test the most promising agents against a reference AI by a fellow student in the sixth chapter. Lastly, we end this paper by concluding our findings and presenting an outlook on possible enhancements.

²<https://github.com/philemonSchulz/onitama>

2 Rules and strategies for the game Onitama

In this chapter, we will explain the basic rules of the game Onitama and the modifications we implemented for this paper. Further, we will describe possible strategies and heuristics that could potentially work for developing an AI for Onitama. Then, we will cover similar games to Onitama and share which methods worked for them. Lastly, we will conduct a game-theoretical analysis of Onitama. Here, we explain important concepts in the domain of game AI development and present basic statistical information about Onitama.

2.1 Onitama - General information and rules

Onitama is a two-person strategic board game created by the Japanese author Shimpei Sato. It was first published in 2014 with the publisher "conception". The game is set in Japanese martial arts, where two masters compete using different fighting styles.

In the original game, the game board is a 5x5 square grid, with each player playing five pieces. On each player's side, the middle tile is marked as a temple, representing the spot where the master initially stands. The players are distinguished by colors: red and blue. Two types of pieces can be found in the game: so-called "Students" and "Masters." At the start of the game, each player gets four students and one master, which are placed on the board as shown in Figure 2.1. However, to increase the difficulty for the AI, we decided to widen the game board to a 7x7 square grid. This results in each player receiving seven pieces, six students, and one master still. The master continues to stay in its central position.

Additionally, there are sixteen cards available, five of which are randomly chosen at the start of the game. A list of the cards can be found here ¹. Each player gets two of those five cards, with a fifth card placed next to the board between the players. This fifth card determines the starting player by a colored symbol on the card. To win the game, players either capture the opponent's master or reach the opponent's temple with their own master.

To move their piece, players have to play one of their cards. Each card allows for a set of possible moves. For example, the card "Hase" (rabbit) shown in Figure 2.2 allows the player to move one of their pieces, either two boxes to the right, one box to the right and one box up, or one box to the left and one box down relative to their current position. Pieces can jump above other pieces but can't be placed on a field if occupied

¹https://brettspielbox.de/wp-content/uploads/2017/08/IMG_8185.jpg

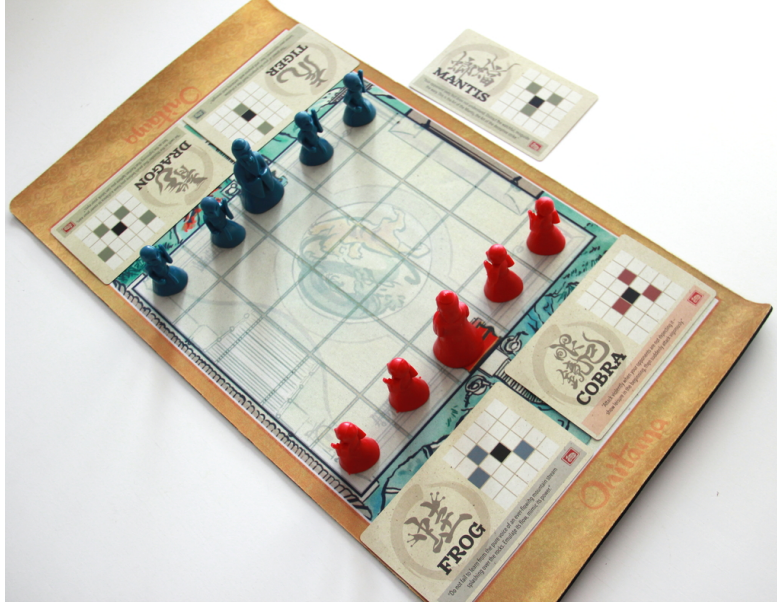


Figure 2.1: The classic Onitama starting state on the 5x5 board. Each player's pieces are aligned on the field, with the masters staying in the middle fields on their temples. Each player also has two player cards, with the fifth card lying next to the board between both players. Image source: ([Brettspiel-news.de](https://www.brettspiel-news.de) 2024)

by an own piece. Pieces can't be placed outside the board, either. If a piece lands on an opponent's piece, its piece is captured and removed from the board, with the capturing piece being placed on the vacant field.

After a card has been selected, and the piece has been moved, the card is exchanged with the fifth card that lies separate next to the board. The second player follows this procedure of selecting a card and moving a piece. As a result the cards are constantly rotated.

It is important to notice that a game of Onitama could go infinitely. That happens when both players don't capture the opponent's master and can't place their master in the opponent's temple. In this case, there is no predetermined procedure to handle a draw, such as in chess.

2.2 Possible strategies and heuristics

When playing Onitama, one can think of multiple strategies to enhance the chances of winning. Hence, we introduce three strategies we came up with, which we will use in two of our AI agents. The first one will only use those strategies to select its moves; the second one uses the strategies in the form of an MCTS agent with a heavy playout. It will later show that using these heuristics in combination with MCTS does allow for a solid AI agent; however, using standard MCTS with move prioritization or MCTS with

2 Rules and strategies for the game Onitama



Figure 2.2: An example game card from the German version of Onitama.

Rapid Action Value Estimation proved to be much stronger. More about our results in section 5.2.

1. Capture Opportunistically:

This first strategy is very simple. Whenever it is possible to capture an opponent's piece, the AI should do so. When an opponent's student is captured, it reduces its mobility and the number of possible moves, leading to a strategic disadvantage. While we have to consider that this aggressive behavior possibly leads us to a position where we are at risk of losing the game (e.g., our master can capture an opponent's piece but now is in danger of being captured by one of the opponent's pieces), it could be especially useful in the early stage of the game to fastly weaken the opponent still.

2. Avoid the Edges

The second strategy aims to determine where the AI prefers to position its pieces. As positioning your piece on one of the border fields reduces your move options, giving you less flexibility in reacting to an opponent's move, the AI should focus on keeping the pieces away from the outermost fields.

3. Preserve the Master

Naturally, the master is the most valuable piece. Hence, the third strategy aims at keeping the own master as safe as possible. He should only move when necessary. Accordingly, the students should be prioritized for moves.

An important task in AI development for games is game state evaluation. For this, heuristics are used to determine how good or bad a certain state is. We came up with four heuristics that could potentially help AI to make better decisions.

1. Victory or Defeat

The first heuristic is all about the game result. Naturally, a move that results in a victory or defeat should be rewarded with the highest positive or negative value.

2. Piece Advantage

As pieces are essential to attack or defend, the second heuristic deals with the number of pieces. More pieces means more options to make moves; therefore, the more one has, the higher the heuristic value should be. Similarly, the higher the number of opponents is, the lower the value should be. This heuristic corresponds with the first named strategy.

3. Positioning on the Board

The third heuristic evaluates the position of your own pieces as well as those of the opponent. It focuses on whether they are located in one of the outmost fields on the board. A positive value should be given to fully movable pieces, with different negative values for pieces standing either at an edge or in a corner. For the opponent's pieces, the opposite counts.

4. Mobility and Flexibility

Finally, the fourth heuristic measures the mobility of the AI's pieces for a given move. As mobility is an important factor, moves that result in an increase of move options should be positively rewarded, while moves that result in a decrease should be negatively rewarded. Here, the size of the reward is related to the relative changes of possible moves, meaning the higher the increase, the higher the reward, and vice versa.

2.3 Similar games and their used methods

The development of AI for games has always been one of the main AI research fields. Games helped to test or invent key concepts of modern AI, like the MiniMax Algorithm of Alan Turing or the first form of machine learning developed by Arthur Samuel, who tested his algorithms on the game checkers (Georgios N. Yannakakis 2018, p.8). In this section, we will examine three games that are similar to Onitama and essential in AI game development. All three are turn-based games that allow for perfect information gameplay. Perfect information games allow each player to be aware of the moves that all other players have made (Roy et al. 2010). Those three games are Chess, Go, and Shogi.

2.3.1 Chess

Chess is a famous two-player board game that is played on a 64-square grid. Each player has 16 pieces; however, contrary to Onitama, there are six different types instead of two. While in Onitama, cards determine the possible moves of each piece, in Chess, each type of piece allows for its unique movement options. Chess has proven to be an intellectual challenge for centuries. It fulfills entertainment purposes and serves as a tool for cognitive development, a simple model to explain decision-making, an educational asset, and a competitive sport with international tournaments.

In AI development, computer Chess is one of the most researched topics, with its story beginning back in the 18th century. Over the years, hundreds of papers have been

written, and thousands of Chess programs have been developed (Ensmenger 2012). Yet, two achievements stand out: IBM’s Deep Blue and the open-source program Stockfish.

Deep Blue became famous in 1997 when it defeated the reigning World Chess Champion, Garry Kasparov (Campbell et al. 2002). The success of Deep Blue mainly came from its immense computational power and specialized hardware. Deep Blue was able to evaluate up to 200 million positions per second. For this, they used 480 single-chip Chess engines, each one capable of evaluating 2 to 2.5 million positions per second. However, Deep Blue was no learning system; it relied heavily on brute-force calculations and an extensive knowledge base. For instance, it used an endgame database representing each possible Chess ending with five or fewer pieces on the board. Further, the system was able to adjust its search strategy depending on the current positions to allow it to give more computational power to important lines of play.

Stockfish was developed in 2008 by Tord Romstad, Marco Costalba, and Joona Kiiski (Chess.com 2024). In contrast to Deep Blue, Stockfish is open-source, allowing for continuous improvements from developers worldwide. Thus, Stockfish became one of the strongest Chess engines. Stockfish uses a combination of modern hardware and alpha-beta pruning to evaluate millions of positions per second. Alpha-beta pruning improves the MiniMax search algorithm to significantly reduce the evaluated nodes in a search tree (Knuth and Moore 1975). More about the MiniMax and Alpha-beta pruning algorithms in section 3.1 and 3.2. In 2017, Stockfish was beaten by AlphaZero, a computer program that uses Monte Carlo Tree Search (MCTS) and neural networks to determine its moves, which led to the development of Stockfish 12, a new version that combined the existing approach of alpha-beta pruning with neural networks. As of 2024, Stockfish is rated as the highest-rated Chess engine in the computer chess rating list (CCRL)².

2.3.2 Go

Go is an ancient two-player board game that originated in China over 2,500 years ago. It is played on a square grid of 19x19 lines. Players place black and white stones on the intersections where adjacent stones build groups, which can be captured if they have no liberties (orthogonally adjacent empty spaces) (Browne et al. 2012). To end the game, both players must pass; the player who controls the most board territory wins.

While the rules of Go are relatively simple, developing sufficient AI for Go proved to be a harder challenge than Chess. This is caused by the high difference of the branching factor (number of legal moves per position) b and depth d (game length) Go has in contrast to Chess. While Chess has a branching factor of roughly 35 and a depth of 80, Go comes with a branching factor of 250 and a depth of 150 (Allis 1994, p.171). This makes using brute-force methods like those used in Deep Blue or Stockfish infeasible. While some programs use alpha-beta search, they only achieved the level of a strong beginner by 1997, with stagnation afterward (Browne et al. 2012). It was not until 2006, through MCTS 3.3, that rapid progress was made. Using MCTS allowed Go programs to beat professional players on a large board (that played with a strong handicap) by

²<http://computerchess.org.uk/ccrl/4040/>

2008 (Lee et al. 2009). Today, the leading Go programs use a combination of MCTS and neural networks. Two programs have stood out in recent years: DeepMind’s AlphaGo and its successor AlphaZero.

AlphaGo was the first computer Go program to beat a human professional player in the full-sized game of Go (Silver et al. 2016). It was introduced in 2016 and achieved a 99.8% winning rate against other Go programs. AlphaGo uses three components: a policy network, a value network, and MCTS. The policy network consists of three parts: a fast rollout policy to evaluate positions quickly, a policy based on supervised learning on expert moves to predict human expert moves, and a policy based on reinforcement learning to improve beyond human expert level. The value function also uses reinforcement learning. A detailed explanation of the methods and functionalities can be found here: (Silver et al. 2016). MCTS is used to search the game tree based on the policy and value network.

DeepMind introduced two new models based on AlphaGo in 2017. The first one was AlphaGo Zero. It still focuses on the game Go but no longer relies on supervised learning on expert moves for the policy network but only on reinforcement learning and MCTS. AlphaGo Zero beat AlphaGo 100:0 (Silver et al. 2017).

Based on AlphaGo Zero, DeepMind introduced AlphaZero (Silver et al. 2017). It uses the same model as AlphaGo Zero, but can be applied in a variety of games like Chess, Shogi, or Go. AlphaZero does not need any domain knowledge except the game rules. Within 24 hours of training, AlphaZero was able to beat world champions in Chess, Shogi, and Go repeatedly. In 2017, AlphaZero also beat the current version of Stockfish; however, through the introduction of Stockfish 12m, Stockfish became the leading Chess engine again.

2.3.3 Shogi

Shogi is a Japanese chess variant that has been played since the 16th century. Like Chess, Shogi is played by two players, Black and White, and the goal is to capture the opponent’s king. However, unlike Chess, Shogi is played on a 9x9 board instead of the classical 8x8 Chess board, with each player having 20 pieces instead of 16. Further, in Shogi, there are eight different types of pieces, compared to six in Chess. While in Chess, only the pawn is allowed to be promoted, and only on the last rank, in Shogi, there are six types of pieces that allow for a promotion, needing to be placed in the back three ranks on the enemy camp. The biggest difference to Chess is the possibility to reuse captured pieces. When a piece is captured, it is held by the capturing player. When it is a player’s turn, the player can either make a move with a piece on the board or use one of the captured pieces and place it nearly anywhere on the board.

This so-called ”drop rule” and the other differences to Chess, like the increased board size, make shogi much more challenging for computer programs than classical Chess. Shogi has a state-space complexity of 10^{71} , compared to 10^{43} in Chess, and a game-tree complexity of 10^{226} compared to 10^{123} in Chess (Iida et al. 2002). Due to its similarity to Chess, early Shogi programs often had a structure that was similar to that of Chess programs (Takizawa 2005). They mostly used mini-max trees combined with an iterative

alpha-beta search.

An early Shogi program was YSS, which uses alpha-beta search in an iterative process together with a "half-ply" extension. A similar extension was used later by Deep Blue. Half-ply extensions are applied for each best move when extending trees. Here, the program searches the memorized best move from a previous search and increases the search depth by 0.5 ply for this move only. If the best move is consistently chosen, the cumulative effect of multiple 0.5 ply extensions can result in a deeper search (Takizawa 2005).

In 2005, the Shogi program Bonanza was developed by Kunihiro Hoki, which won the 2006 world computer Shogi championship. His Bonanza was the first Shogi program that introduced machine learning. Bonanza used a mini-max tree and a learning method for its evaluation function. Based on a set of grandmaster game records, a search function, and a linear weighted evaluation function, it iteratively adjusted the weights of the evaluation function based on a procedure of numerical minimization techniques (Takizawa et al. 2015).

In recent years, programs that use reinforcement learning have also become more and more dominant in Shogi. As we stated earlier, DeepMind's AlphaZero did not only succeed in Chess and Go but also in Shogi. It was able to beat the 2017 world champion in computer Shogi, the program Elmo, after just 2 hours of training (Silver et al. 2017).

2.4 Game-theoretical analysis of Onitama

Onitama can be seen as a 2-agent environment. In game theory, an agent is seen as anything that perceives its environment through sensors and acts upon that environment through actuators (Russell and Norvig 2010, p.34). In our case, the two agents represent the two players, Red and Blue. Here, both agents act rationally, meaning they perform a sequence of actions that results in a desired outcome, in our case, making moves to win the game. Thereby, desirability is measured through a performance measure that evaluates any given sequence of environment states. Therefore, a rational agent always selects an action that is expected to maximize its performance measure.

Aside from the initial random drawing of cards, Onitama is a zero-sum game of perfect information.

Zero-sum games are characterized by the fact that the utility value at the end of the game is equal but symmetrical. In Onitama, this means when player Blue wins, player Red loses. Perfect information means that (1) the game is fully observable. Both players can see all the information about the game, like the different cards or piece positions, at any time in the game. (2), it is deterministic. That means that there are no stochastic influences that could change the state of the game. Every action leads to a pre-known state.

Further, Onitama is non-terminating. It is possible that both players move their pieces in such a way that they never capture the opponent's master or place their master on the opponent's temple. Therefore, it is possible to play the game infinitely.

2 Rules and strategies for the game Onitama

In game theory, the branching factor b is of essential meaning. It refers to the number of choices or legal moves available to a player at each decision point in a game. It essentially measures the complexity of a game by quantifying the number of options a player has to consider at any given turn (Lantz et al. 2017).

To calculate the branching factor for Onitama, we first make an estimation by finding the game states with the lowest (lower boundary) and highest (higher boundary) possible number of moves. For the lowest game state, a player needs to have at least one piece left (otherwise, the game would be finished). Therefore, we need to find the two cards with the lowest number of moves. In the card deck we introduced in chapter 2.1, there is one card that allows for two moves, ten cards that allow for three moves, and five cards that allow for four moves. Therefore, as a player always has two cards at hand, the game state with the lowest possible moves allows for $1 * (2 + 3) = 5$ moves. Accordingly, the higher boundary is calculated as follows: $7 * (4 + 4) = 56$. This gives us the following boundaries for the branching factor:

$$5 \leq b \leq 56 \quad (2.1)$$

A card allows for an average of $(2 + 10 * 3 + 5 * 4) / 16 = 3,25$ moves. As a player always has two cards to choose from, the average branching b factor is

$$b = 7 * 2 * 3,25 = 45,5 \quad (2.2)$$

Therefore, the average branching factor is slightly higher than in Chess (35) but significantly lower than Go (250) (see section 2.3).

3 Game AI algorithms and methods

In this chapter, we describe multiple important algorithms that are important in the domain of game-AI.

We first explain one of the oldest and most simple yet significant algorithm in game-AI, the MiniMax algorithm. Then, we present alpha-beta pruning, one of the most common methods to enhance the usability of MiniMax. Finally, we dive deep into Monte-Carlo-Tree Search (MCTS), one of the most influential AI algorithm in the last decades, and explain two extensions of it: Rapid Action Value Estimation MCTS and Heuristic MCTS. Because MCTS agents have proven to be very successful in games similar to Onitama (see section 2.3), we decided to develop MCTS agents based on the proposed three MCTS concepts.

3.1 MiniMax Search

Almost every AI problem can be cast as a search problem (Georgios N. Yannakakis 2018, p.39), where the goal is to find the best state for an according measurement. Given a game state in Onitama, where, e.g., it is player Red's turn, this means finding a sequence of moves to get to a state in which player Red wins. To find such a sequence, search algorithms are used to systematically search the given search space.

This is done by creating a search tree, where the root node represents the game's current state. Edges are then used to represent actions an agent can take to get from one state to another. Those states are then added to the search tree as children nodes.

To find the best possible sequence of moves in a two-agent search problem, the most basic algorithm one can use is called MiniMax. MiniMax is very successful in playing classic perfect-information two-player board games like Checkers or Chess.

The core loop of MiniMax switches between player 1 and player 2 - in our case, that would be player Red and player Blue - which are called the *min* and the *max* player (Georgios N. Yannakakis 2018, p.43). Given a game state, we specify that the current player is *max*. The algorithm then starts by exploring all possible moves for this state. For each of the resulting states, all possible moves of the *min* player are also explored. This loop continues until all possible combinations of moves are found. After this process, a complete search tree has been generated from the root node down to the leaves. Each level of the tree can be assigned alternately to the *min* or *max* player, starting with the *max* player at level 0. Afterward, each leaf gets assigned a value of a utility function. Then, each leaf value is passed up the tree to determine which player would have chosen which move in which state. By doing so, the algorithm assumes that every player tries to play optimally, meaning that every player tries to choose the move with the best value.

3 Game AI algorithms and methods

This means that from the standpoint of the *max* player, it tries to choose the maximum value, while from the standpoint of the *min* player, it chooses the minimum value. An example of a MiniMax tree with an explanation can be seen in Figure 3.1. A detailed description of MiniMax and its algorithm can be found here: (Russell and Norvig 2010, chapter 5.2)

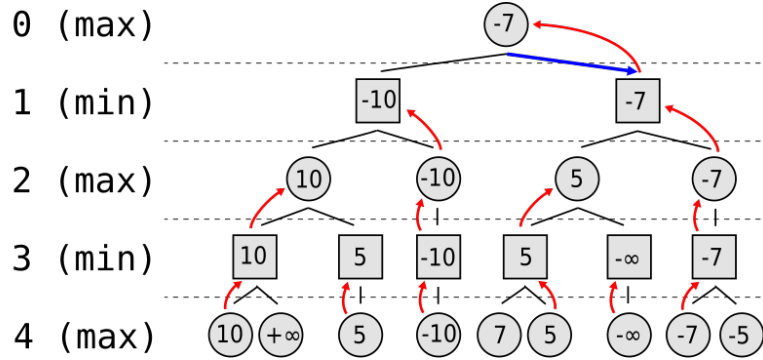


Figure 3.1: An example MiniMax game tree. At first, each node value is empty except those of the leaves (level 4). As on level 3, the player is the *min* player, and it chooses the lowest possible value for each node. On level 2, the player is the *max* player, resulting in it choosing the maximum value for each node. After a value has been determined for each node, the player at level 0 (always the *max* player) decides on the move with the highest value, in our case, the move leading to the node with the value -7. Image source: (Nogueira 2024)

However, the big problem with the plain MiniMax algorithm is that it needs to explore every possible move to create a complete search tree. This is infeasible for almost every search problem, as the search tree grows exponentially based on the branching factor. Whilst a game like tic-tac-toe has a game tree size of $9! = 362,880$, searching the entire game tree of chess with an average branching factor of 35 and an average game length of 80 plies would create a search tree the size of 10^{123} (Allis 1994). This is more than 100 orders of magnitude higher than what would be possible with modern computer technology (Lai 2015). Therefore, almost all implemented MiniMax algorithms cut off the search at a given depth d and use a state evaluation function to determine how desirable a certain game state is. Possible evaluation functions for game states in Onitama are presented in 2.2.

Further, numerous improvements have been made to the basic MiniMax algorithm itself. For example, Gradual Depth Increase gradually increases the search depth to benefit from depth-first search's space efficiency with breadth-first search's completeness. Pruning techniques like alpha-beta pruning allow to reduce the number of nodes to be evaluated. Lastly, endgame tables (essentially databases of pre-calculated, perfect endgame positions) allow for better game-state evaluation functions and a reduction of the necessary game tree depth. Combining those methods leads to very competent AI programs for many classic games, e.g., IBM's Deep Blue (Russell and Norvig 2010, p.43).

3.2 Alpha-beta Pruning

As we stated in the previous chapter, the biggest problem of the MiniMax search is that the number of states to be examined is exponential in the depth of the search tree. Alpha Beta pruning is a method to effectively cut the exponent in half by using a node-elimination technique called pruning.

Pruning is a technique that eliminates branches of the search tree that cannot possibly influence the final decision without examining all the nodes within those branches. It allows the algorithm to discard large portions of the search space, significantly reducing the number of nodes evaluated and thus improving efficiency. This means that it returns the same move as MiniMax would do but with a significantly smaller search tree.

Let's take a look at these game trees 3.2. The upward-pointing triangles (e.g., A) represent Player 1. Since Player 1 is at the root node, this player would be the *max* player in a classic MiniMax tree. Accordingly, the downward-pointing triangles (e.g., B) represent Player 2, making this the *min* player.

Tree (a) shows the initial state of the search tree. As one can see, the two nodes that are not leaf nodes have been assigned a two-dimensional array. This array is assigned to every node that will be expanded and is not a leaf node. The first value below triangle B is 3. Since B is a *min* node, this node can have, at most, the value 3, so 3 is recorded as the upper bound in the corresponding array.

In steps (b) and (c), the values 12 and 8 are discovered. However, both are ignored because they are greater than 3; the *min* player would thus choose the move leading to the node with the value 3. Since node B has now been fully explored, the upper bound of B is now set as the lower bound of A (d). This means that no matter which of the previously discovered moves (in this case, only the move from node A to B) A chooses, A can expect a resulting move with at least the value of 3.

In (d), node C is now expanded. The first value is smaller than the lower bound of C's parent node, A. Therefore, no further nodes from C need to be explored, as C will at most have the value 2, making it a worse choice for A than node B (which has at least the value 3).

In trees (e) and (f), we can see that node D must be fully explored since both 14 and 5 are greater than 3. Only through the last node, 2, can D's upper bound be clearly determined. Since D's values are also smaller than 3, A gets the final values [3, 3], meaning that A would choose the move to node B.

Alpha Beta Pruning gets its name from the two values that describe the upper and lower limits of each node (Russell and Norvig 2010):

- α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for *max*.
- β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for *min*.

3 Game AI algorithms and methods

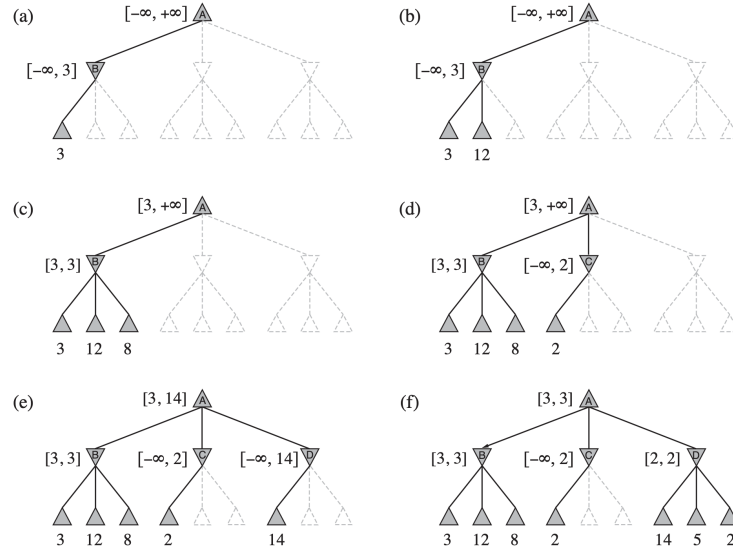


Figure 3.2: Exemplary generation of an alpha-beta search tree. Image source: (Russell and Norvig 2010, p.168)

A detailed description of Alpha-Beta pruning and its algorithm can be found here: (Russell and Norvig 2010, chapter 5.3)

3.3 Monte-Carlo Tree Search (MCTS)

Monte Carlo tree search is a search method "that combines the precision of tree search with the generality of random sampling" (Browne et al. 2012). The term was first introduced in a conference paper in 2005/2006 by Remi Coulom (Coulom 2007, p.74). In the paper, he references an adaptive multi-stage simulation-based algorithm for Markov decision processes (Chang et al. 2005). Today, MCTS is one of the most intriguing algorithms in the field of AI due to its success with various problems like Computer Go.

Browne et al. provide a general overview of the MCTS process:

The basic MCTS process is conceptually very simple... A tree is built in an incremental and asymmetric manner. For each iteration of the algorithm, a tree policy is used to find the most urgent node of the current tree. The tree policy attempts to balance considerations of exploration (look in areas that have not been well sampled yet) and exploitation (look in areas that appear to be promising). A simulation is then run from the selected node, and the search tree is updated according to the result. This involves the addition of a child node corresponding to the action taken from the selected node and an update of the statistics of its ancestors. Moves are made during this simulation according to some default policy, which, in the simplest case,

is to make uniform random moves. A great benefit of MCTS is that the values of intermediate states do not have to be evaluated, as for depth-limited MiniMax search, which greatly reduces the amount of domain knowledge required. Only the value of the terminal state at the end of each simulation is needed. (Browne et al. 2012)

In the basic algorithm for MCTS, the search tree is built iteratively. When a predefined computational budget is reached, such as a time limit or an iteration cap, the search stops, and the best root action is returned. Each node in the search tree represents a state of the respective domain, which, in our case, is a game state in Onitama. Furthermore, nodes are connected by directed links to child nodes, which correspond to actions that lead to subsequent states. In the case of Onitama, these are the moves that lead to a new game state.

In each search iteration of MCTS, the following four steps are applied (Browne et al. 2012):

1. **Selection:**

In the selection phase, starting from the root node, the search tree is recursively traversed using a child selection policy until the next node to be expanded is found. Potential nodes to expand are characterized by not being terminal states and having unvisited, i.e., unexpanded, children.

2. **Expansion:**

During the expansion phase, one or more children are added to the corresponding node to extend the search tree.

3. **Simulation:**

In the third step, during simulation, a simulation is executed starting from the new node(s), and its outcome is evaluated using a default policy.

4. **Backpropagation:**

In the final step, the value of the simulation is propagated back through the search tree, and the statistics of the visited nodes are updated accordingly.

Figure 3.3 shows the four steps of one iteration.

A general MCTS function 3.1 is described by Browne et al. (Browne et al. 2012, p.5). It takes an initial state s_0 as input. Then, a root node v_0 is created with state s_0 . While within computational budget, a sequence of three steps is repeated. First, with the TreePolicy function, an expandable node is selected (Selection) and expanded (Expansion). Then, in the DefaultPolicy function, a simulation (Simulation) is played, and the reward is returned. In the final function, Backup, the reward, together with the expanded node, is backpropagated (Backpropagation).

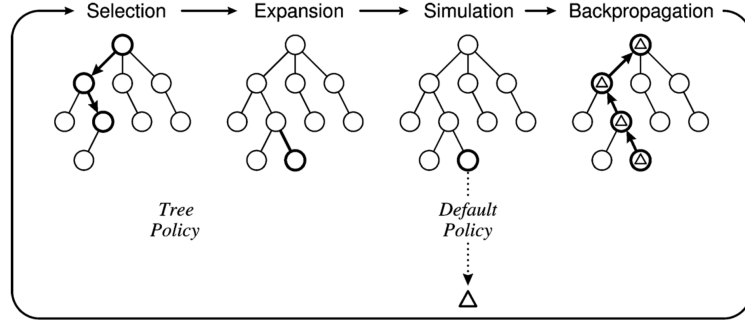


Figure 3.3: The four steps in each iteration of the general MCTS approach (Browne et al. 2012, p.6)

Algorithm 3.1 General MCTS approach

```

1: function MCTSSEARCH( $s_0$ )
2:   create root node  $v_0$  with state  $s_0$ 
3:   while within computational budget do
4:      $v_l \leftarrow \text{TreePolicy}(v_0)$ 
5:      $\Delta \leftarrow \text{DefaultPolicy}(s(v_l))$ 
6:      $\text{BACKUP}(v_l, \Delta)$ 
7:   end while
8:   return  $a(\text{BestChild}(v_0))$ 
9: end function
    
```

Whilst there are many variations of the MCTS algorithm, the most popular is called upper confidence bound for trees (UCT) (Browne et al. 2012), which uses a modified version of the upper confidence bound policy (UCB) used for bandit problems. Bandit problems are decision problems, where one has to choose among K actions (representing the K arms of a multiarmed slot machine) to maximize the cumulative reward. As the underlying reward distributions are unknown, choosing the right action is a difficult task, because of which possible rewards must be estimated based on past observations. This is called the exploitation-exploration dilemma; "one needs to balance the exploitation of the action currently believed to be optimal with the exploration of other actions that currently appear suboptimal but may turn out to be superior in the long run" (Browne et al. 2012, p.4). In MCTS, the same problem occurs when selecting a node action within the existing tree.

One way to balance this problem was presented by Auer et al. (Auer 2002) in the form of the UCB1 policy, where one should choose to play arm j that maximizes the UCB value:

$$UCB1 = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}} \quad (3.1)$$

Here, \bar{X}_j is the average reward from arm j , n_j is the number of times arm j was played,

and n is the overall number of plays so far. "The reward term \bar{X}_j encourages the exploitation of higher reward choices, while the right-hand term $\sqrt{\frac{2 \ln n}{n_j}}$ encourages the exploration of less visited choices" (Browne et al. 2012, p.5).

UCT uses the UCB1 policy to select the child node j that maximizes:

$$UCT = \bar{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}} \quad (3.2)$$

n is the number of times the parent node has been visited, n_j represents the number of times child j has been visited. C_p is a constant.

UCT has proven to be very useful as it balances very well between exploitation (first term in the equation) and exploration (second term in the equation). "As each node is visited, the denominator of the exploration term increases, which decreases its contribution. On the other hand, if another child of the parent node is visited, the numerator increases, and hence, the exploration values of unvisited siblings increase." (Browne et al. 2012, p.7) Further, the constant C_p allows the adjustment of the amount of exploration depending on the specific domain and situation. The full UCT MCTS algorithm can be found here (Browne et al. 2012, Algorithm 2).

3.4 Rapid Action Value Estimation MCTS

RAVE is an enhancement algorithm to Monte Carlo Tree Search that stands for Rapid Action Value Estimation. It addresses the limitations of standard MCTS regarding the slow convergence of action value estimates (Gelly and Silver 2011).

Standard MCTS estimates the value of each state and each action in a search tree separately, and is therefore unable to generalize between related positions and related moves. Hence, many simulations must be played from all states and for all actions. However, having unlimited computational power and time, standard MCTS will eventually converge on the optimal search tree (Kocsis and Szepesvári 2006). RAVE, on the other hand, shares action values across each subtree of the search tree, allowing for a very fast and rough estimate of the action value. This can help to accelerate learning and improve decision-making, especially during the opening phases of a game like Go.

The main assumption in RAVE is that in many games, the value of a move is often similar, regardless of when it is played in a given subtree. While this assumption often fails, it provides a useful heuristic, allowing for a quick move value estimation. RAVE leverages this idea through a heuristic called All-Moves-As-First (AMAF). AMAF estimates the value of an action based on its results whenever it occurs in a simulation. Hereby, the AMAF value of an action a in a state s is defined as:

$$\tilde{Q}(s, a) = \frac{1}{\tilde{N}(s, a)} \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s, a) z_i, \quad (3.3)$$

It represents the "mean outcome of all simulations in which action a is selected at any turn after s is encountered" (Gelly and Silver 2011). It is $\tilde{\mathbb{I}}_i(s, a)$ an indicator function returning 1 if state s was encountered at any step t during the i th simulation, and action a was selected at any step $u \geq t$. Otherwise, it is 0. $\tilde{N}(s, a) = \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s, a)$ counts the total number of simulations. z_i represents the outcome of a simulation. A good visualization of the AMAF updates can be found here (Helmbold and Parker-Wood 2009, Fig. 1), while further information about AMAF in general can be found here (Browne et al. 2012, chapter V.C) and here (Gelly and Silver 2011).

While the basic RAVE algorithm is able to learn very quickly, it often chooses wrong actions. This is because the core assumption (a particular move has the same value across an entire subtree) frequently fails. In Onitama, for example, moving your Master one step forward at the start of the game has a completely different value compared to moving the Master one step forward to a position where an opponent's piece can capture it. To resolve this issue, MC-RAVE is used, which combines the "rapid learning of the RAVE algorithm with the accuracy and convergence guarantees of Monte-Carlo tree search" (Gelly and Silver 2011).

MC-RAVE uses a weighted sum $Q_*(s, a)$ of the MC value $Q(s, a)$ and the AMAF value $\tilde{Q}(s, a)$ to estimate the overall value of an action a in state s .

$$Q_*(s, a) = (1 - \beta(s, a)) \cdot Q(s, a) + \beta(s, a) \cdot \tilde{Q}(s, a) \quad (3.4)$$

The important addition in MC-RAVE is the weighting parameter $\beta(s, a)$ for state s and action a , which allows the combination of the MC and AMAF values. Here, the weighting parameter decreases relative to the number of simulations of a node. This means that when a node has only a few simulations played, the β value is relatively high ($\beta \approx 1$), weighting the AMAF value higher. But when many simulations have been played on that node, the β value gets low ($\beta \approx 0$), thus weighting the Monte-Carlo value higher.

Further, MC-RAVE can be combined with the UCT search to incorporate the idea of an exploration bonus (Gelly and Silver 2011). The UCT-RAVE algorithm uses the following value to select its actions during the tree policy:

$$Q_*^\oplus(s, a) = Q_*(s, a) + c \sqrt{\frac{\log N(s)}{N(s, a)}} \quad (3.5)$$

Actions are selected to maximize the augmented value, $a = \arg \max_b Q_*^\oplus(s, b)$.

3.5 Heuristic MCTS

A key strength of standard Monte Carlo Tree Search lies in its domain-independent nature, requiring no specific heuristics for evaluating states. That allows MCTS to be applicable to any domain that could be modeled as a search tree (Browne et al. 2012).

However, adding domain-specific knowledge can often lead to significant improvements in performance.

One of the most common uses of this "heuristic MCTS" is during the Simulation phase. Here, standard MCTS uses uniform random playouts, called light playout. Strategies using custom heuristics to select certain moves during simulation are called heavy playout. While the random simulations from the light playout are not a completely correct representation of an actual game situation (in reality, there would almost never be a situation where two players play only random moves in a zero-sum game), they allow for a high speed in move selection, therefore leading to a large number of simulations played. In contrast, using domain knowledge usually drastically reduces the number of possible simulations (Browne et al. 2012) but may also reduce the variance of simulation results. However, depending on the heuristic, heavy playout allows for depth-limited MiniMax. This allows the simulation to be cut at a certain depth and the current state to be evaluated with a given heuristic, therefore reducing the time of the simulation and enhancing the number of simulations played. Still, the degree to which one uses game-specific knowledge highly depends on the specific domain and the heuristics used and needs to be tested through experiments.

Further, heuristics in MCTS can not only be used for state evaluation (heavy playout) but also for move-ordering. Move-ordering heuristics aim to prioritize the exploration of more promising moves earlier in the search process. This can be particularly effective in the Selection phase of MCTS, where it can guide the algorithm towards more promising areas of the search space more quickly. For example, in the game of Go, Chaslot et al. (Chaslot et al. 2008) found that using move groups to create an extra decision layer for move selection led to improved performance. However, the effectiveness of move ordering can vary depending on the domain. In the game of Havannah, Teytaud and Teytaud (Teytaud and Teytaud 2010) observed that progressive widening without heuristic move ordering had little effect on playing strength.

The effectiveness of heavy playouts versus move ordering can also vary between domains. Drake et al. (Drake and Urtamo 2007) demonstrate that heavy playouts are generally more effective than move ordering for applying heuristics in Monte Carlo Go. This finding suggests that focusing heuristic efforts on improving the simulation phase may be particularly beneficial for games like Go. However, it's important to note that the optimal approach may depend on the specific characteristics of the game or problem being addressed.

4 Development of the AI

For Onitama, we developed a total of five different AI agents. We decided to implement a Random agent as a basic reference AI. Further, we implemented a heuristic agent that bases its move decisions on the proposed heuristics in section 2.2. Finally, we implemented three MCTS agents, since they showed great success in games similar to Onitama, as presented in section 2.3. We implemented a basic MCTS agent and a Heuristic MCTS agent to evaluate whether a heavy playout would perform better than the standard MCTS agent. Further, as RAVE proved to be very successful in the Onitama like game Go, we decided to implement a MCTS RAVE agent as well.

4.1 Random agent

The first agent is a random AI called **RandomAi**. This AI is, as the name suggests, very simple, since it selects its moves completely randomly from a given set of possible moves. However, we added a modification to the RandomAi that allows it to focus on winning moves and on those moves that capture opponents' pieces. For this, both variations use a function that returns an object containing three lists: one for winning moves, one for capturing moves, and one for the remaining moves. Depending on which type of RandomAi is selected, the agent either combines those three lists and chooses one move at random (Base RandomAi) or iteratively checks the three given lists in the presented order (Advanced RandomAi). If there are moves in the winning moves list, it selects the first move; if not, it checks the capturing moves. Here, if the list is not null, it chooses one of the moves at random. Finally, if this list is empty as well, it chooses one of the normal moves from the remaining moves list. We expect that both agent types will mostly lose against the more advanced agents.

4.2 Heuristic agent

The next implemented agent is a rule-based AI called **HeuristicAi**. It uses the hard-coded heuristics from section 2.2 to select its moves. For this, the AI uses a HashMap that maps each move to a certain heuristic value. The base for the move selection is again formed by the object containing the three types of moves: winning moves, capturing moves, and remaining moves. First, it is checked whether there is a move in the winning move, and if so, the first move of the list is chosen, and the algorithm stops. If the winning moves list is empty, each move gets evaluated based on the following heuristics. For this, each move is given an initial value of 0. A pseudocode can be found here: 4.1

Algorithm 4.1 Pseudocode for the evaluation of a move (heuristics 2,3 and 4)

```

function EVALUATEMOVE(Move m)
  value = 0

  /* Check if move leads to a direct loss */
  if m leads to loss then
    value = NegativeInfinity
    break
  end if

  processMoveWithGameLogic(m)

  /* Checking for piece advantage */
  value += (currentPlayerPiecesSize - opponentPlayerPiecesSize)

  /* Check for piece positioning */
  for Piece p in currentPlayerPieces do
    if p is in corner then
      value -= 2
    else if p is on edge then
      value -= 1
    else
      value += 1
    end if
  end for

  /* Check for mobility */
  value += (numberOfPossibleMovesForCurrentPlayer -
numberOfPossibleMovesForOpponent)

  return value
end function

```

1. Losing Move: If there are no winning moves at hand, the algorithm first checks whether the move leads to a direct loss. For this, the move is processed by the game logic, and the possible move object for the opponent is computed. Then, the algorithm checks if the opponent's winning moves list is not empty. If this is the case, the value of the move is set to Negative Infinity, and the move is not further evaluated.
2. Piece advantage: If a move doesn't lead to a direct loss, the current piece ratio is checked. For this, the number of opponent pieces is subtracted from the number of pieces by the current player. This value is then added to the initial value.

3. Positioning on the board: Next, the position of each piece of the current player is evaluated. If a piece is in a corner of the game board, a value of 0 is given; if it is on one of the edges, a value of 1 is given; and if the other two conditions don't apply, a value of 2 is given. Then, those values are added up and added to the initial value.
4. Mobility of pieces: Lastly, the number of possible moves is weighted. For this, the current move is processed by the game logic. Then, the number of moves possible for the opponent is subtracted from the number of moves possible for the current player. This value is again added to the initial value.

4.3 MCTS agents

The last three types of AI are all MCTS agents. All MCTS agents have a time limit of two seconds for their search.

The first one, **MCTS**, is a basic MCTS agent primarily based on the UCT algorithm we explained in section 3.3. However, we made a few important changes.

First, we use the RandomAi for the *defaultPolicy*. Since the RandomAi agent always returns the object containing the three lists of moves, we decided to use the function to prioritize capturing/winning moves (advanced RandomAi), as it would not give the agent any time advantage when not using it. Therefore, our standard MCTS could be seen as an MCTS with a slightly heavy playout. However, the use of the advanced RandomAi increased the playing strength of the MCTS immensely. Our experiments showed that this advanced MCTS was, together with the MCTSRave agent, the strongest of our developed AI agents. We will go into more detail about the comparison in section 5.2.

Second, as a result of a simulation in our MCTS agent only returns 1 (for a win of the current player) or 0 (for a loss of the current player), our *Backup* function does not switch between δ and $-\delta$ but between 1 and 0.

Further, we discovered during our experiments that the MCTS player struggled to choose its next move when there was a strong difference in the remaining pieces. This mostly occurred when testing against weaker AIs. Because all possible moves had been assigned very high and similar values (like 0.9999), this resulted in the MCTS agent rotating its pieces and not attacking. That's why we decided to implement an additional check in the *bestChild* function. Every time the *value* of a child is over 0.99, we instantly return this child, making it obsolete to compute the remaining values and preventing the agent from playing the same pattern over and over again. This problem also applies to the MCTSRave agent.

Lastly, as the simulation result of a given node declares the winner based on the initial current player of the simulation and not based on the player that made a particular move that led to this node, the given reward is flipped initially.

To give an example we have a parent node n_1 and a child node n_2 , which was reached by moving a Red piece through a move called R_1 . This means that at n_1 , Red was at play, and through move R_1 , n_2 was created where now Blue is at play. Assuming that

4 Development of the AI

for the simulation at n_2 , the winner was Red, the simulation is given a reward of 0, as Blue was the initial player and Red won the game. However, through the initial flipping of the reward in the *Backup* function, node n_2 is actually given the reward 1, as it was the Red move R_1 that led to this node, and Red won the simulation.

The second agent is called **MCTSHuristic**. This agent is again based on the UCT MCTS algorithm (Browne et al. 2012, Algorithm 2) but uses a heavy playout instead of a classic random playout. The heavy playout is based on the heuristic considerations of the HeuristicAi. This means that during playout, the moves of the player who made the move that led to this node are using the move generation of the HeuristicAi, while the RandomAi still simulates the moves for the opponent. Therefore, the moves of the player that made the move that led to this node are generated by the HeuristicAi, while the RandomAi generates the moves of the opponent. Further, we limited the iteration count of a single simulation to 30. Afterward, the current gamestate is evaluated again based on the heuristics of the HeuristicAi.

The last MCTS agent is called **MCTSRave** and is an agent based on the Rapid Action Value Estimation process and the all-moves-at-first heuristic presented in section 3.4. Again, the algorithm uses the basic UCT algorithm (Browne et al. 2012, Algorithm 2) as a base and the advanced RandomAi for the *defaultPolicy*. Further, in the *defaultPolicy*, not only the game result but also a list of all the moves played are returned, which is then used in the *backup* function to update the AMAF value. For this, in each iteration of the backpropagation loop, after the "normal" UCT values are updated, it is checked for each child of the current node whether the incoming move of the child appears in the *playedMoves* list. If this is the case, a *raveVisit* counter of the child is increased by one, and a *raveReward* counter is increased by the resulting reward of the simulation. Again, this value is flipped depending on the player of the incoming move. Afterward, the incoming move of the current node is added to the *playedMoves* list.

Then, when the *bestChild* function is called, the best child of a node is then selected as follows: First, the classic UCT value is calculated 4.1.

$$uctValue = \frac{childReward}{childVisits} + cValue * \sqrt{\frac{2 * \ln(nodeVisits)}{childVisits}} \quad (4.1)$$

Next, the rave win rate is calculated by dividing the rave value of the child node by the rave visits. 4.2:

$$raveWinRate = \frac{child.raveReward}{child.raveVisits} \quad (4.2)$$

Afterward, the rave bias is calculated based on a fixed parameter *Rave_Bias* 4.3:

$$raveBias = \frac{Rave_Bias - nodeVisits}{Rave_Bias} \quad (4.3)$$

Finally, the resulting value is computed as follows 4.4:

$$finalReward = raveBias * raveWinRate + (1 - raveBias) * uctValue \quad (4.4)$$

4 Development of the AI

As stated when explaining the MCTS agent, the MCTSRave agent had the same problem of rotating its pieces when having a lot more pieces left than its opponent. Hence, we added the same check as for the MCTS agent, returning the first child in the *bestChild* function whose *finalReward* is over 0.99.

5 Experimental Evaluation

In this chapter, we will conduct multiple experiments to test our developed agents. We start by testing the MCTS and MCTSRave agent for their C value and bias value b . We then let the different agents play against each other to assess their playing strength. All tests were conducted on a 2020 Macbook Air with M1 chip and 8GB of RAM.

5.1 Parameter testing

Before we can test our different agents against each other, we must first test the parameters for the MCTS and MCTSRave agents. We decided to test the C value for the standard MCTS and use this C value as a baseline for the MCTSRave agent, for which we only tested for the bias b .

It is important to mention that at this stage, we still worked with the "standard" versions of the MCTS and MCTSRave agent without them checking whether a child value is above 0.99. This change was only introduced during playouts in the next section 5.2.

Regarding the standard MCTS agent, we started by testing the agent with three initial values for C , 0.7, 1, and $\sqrt{2}$ against the advanced RandomAi. Each value was tested for two hours. The results can be found in table 5.1. The different C values did not make a big difference. All three versions beat the opponent AI with no losses. The average number of matches played was 266, with an average match duration of 28 seconds. Interestingly, all games were won by capturing the opponent's master, and none by reaching its temple. It can be assumed that this lies in the weak playing strength of the advanced RandomAi, which has no integrated mechanism to secure its master.

As the C value did not seem to have a big impact on the playing strength against the advanced RandomAi, we secondly conducted tests by letting different versions of the MCTS agent play against each other, this time for eight hours each (Except the last comparison, 0.3 against 0.7, which has been tested for only two hours). As a game in Onitama can go infinitely, we added a time constraint of 20 minutes for each match. If the limit was exceeded, we aborted the simulation and counted it as aborted. The results can be seen in table 5.2.

Through the increase in the opponents' player strength, naturally, the games took much longer. Further, the different C values now had a real impact. As the first comparison between 0.7 and 1 suggested that a low C value is more performant, we continued to lower the C value in 0.2 decrease steps. **The sweet spot was found at a C value of 0.3.** When comparing the 0.3 value with the 0.7 value from the first comparison, a clear win rate of 88% validates this value. Contrary to the simulations against the

5 Experimental Evaluation

C-Value	Wins MCTS	Wins RandomAi	Number of matches	Avg. duration	Avg. winner pieces	Avg. loser pieces	Avg. Temple wins
0.7	253	0	253	28.63 sec	5.62	4.58	0
1	253	0	253	28.56 sec	5.57	4.83	0
$\sqrt{2}$	261	0	261	27.61 sec	5.49	4.77	0
0.3	229	0	229	31.45 sec	5.69	4.50	0

Table 5.1: Results for the C -Value tests played by the standard MCTS agent against the advanced RandomAi. The simulation duration was two hours for each test.

C-Value Red	C-Value Blue	Wins Red	Wins Blue	Number of matches	Avg. duration
0.7	1	91	40	131	203.00 sec
0.5	0.7	80	45	125	218.38 sec
0.3	0.5	80	44	124	228.01 sec
0.1	0.3	2	141	143	184.65 sec
0.3	0.7	30	4	34	216.94 sec

C-Value Red	C-Value Blue	Avg. winner pieces	Avg. loser pieces	Avg. temple wins	Avg. Starting player wins
0.7	1	2.67	1.27	61.33 %	38.11 %
0.5	0.7	2.71	1.28	62.29 %	49.62 %
0.3	0.5	2.64	1.23	63.83 %	48.38 %
0.1	0.3	3.48	0.61	18.95 %	54.22 %
0.3	0.7	2.75	1.06	47.06 %	35.29 %

Table 5.2: Test Results for different C -Values for the MCTS agent

advanced RandomAi, temple wins now played an important role for the agents, with an average temple win rate of 62 % for the first three simulations. This aligns with the relatively low number of remaining pieces, indicating that the MCTS agent is very good at protecting its own master against potential attacks from the opponent, therefore forcing the opponent to focus more on the temple to win. For the fourth simulation, temple wins weren't that important as the MCTS agent with a C value of 0.3 was so much stronger than the agent with 0.1 that it was easier to just capture the opponent's master. Again,

5 Experimental Evaluation

this now aligns with the comparatively high number of remaining winner pieces and the extremely low number of remaining loser pieces. We also measured the average winning rate for the starting player to observe if starting in Onitama makes a big difference. As the chance to be the starting player in Onitama is 50%, the expected outcome should be around this number for a game where the starting player doesn't have an advantage. With an average starting player win rate of 45.12%, this was the case.

After we set the C value for the standard MCTS agent to 0.3, we continued to test the bias b for the MCTSRave agent. As the range of the bias can vary strongly depending on the domain, we started with short tests to determine a general range for the bias. For this, we ran four tests of 60 min each while comparing the following bias values: 1 vs. 500, 500 vs. 1000, 1000 vs. 1500, and 1500 vs. 2000. Results showed that a good Rave bias would lie between 1500 and 2000. Consequently, we conducted three tests of six hours each, with the following bias values: 1500 vs. 1700, 1700 vs 1900, and 1900 vs. 2100. The results can be seen in table 5.3.

b -Value Red	b -Value Blue	Wins Red	Wins Blue	Number of matches	Avg. duration
1500	1700	62	46	106	201.00 sec
1700	1900	53	52	105	207.16 sec
1900	2100	42	62	104	207.92 sec
b -Value Red	b -Value Blue	Avg. winner pieces	Avg. loser pieces	Avg. temple wins	Avg. Starting player wins
1500	1700	2.43	1.31	64.81 %	44.44 %
1700	1900	2.49	1.34	70.48 %	49.52 %
1900	2100	2.45	1.43	73.08 %	50.00 %

Table 5.3: Test Results for letting MCTSRave agents play against each other with different bias values b . C value set on 0.3, each comparison went for 6h

This time, there is no clear winning bias value, contrary to the tests for the C parameter. Both 1500 and 2100 seem to perform better than 1700 and 1900. However, as the MCTSRave agent is based on the classic MCTS agent, it is no wonder that the remaining values are similar to those of the classic MCTS agent. Interestingly, temple wins played an even more important role, since around 69% of all matches were won through reaching the opponent's temple. Again, there was no noteworthy starting advantage, with an average winning rate for the starting player of 47.97%.

Because there was no clear winning bias value, we decided to continue the tests by letting the Rave agent play against the standard MCTS agent. Again, we let the agents play for six hours each and used the four bias values from the previous test. Results are presented in table 5.4.

<i>b</i> -Value	Wins MCTS	Wins MCT-SRave	Number of matches	Avg. duration
1500	49	49	99 (1 aborted)	212.44 sec
1700	43	56	99	219.53 sec
1900	45	54	99	220.40 sec
2100	59	40	99	219.91 sec
<i>b</i> -Value	Avg. winner pieces	Avg. loser pieces	Avg. temple wins	Avg. Start-player wins
1500	2.43	1.30	73.73 %	44.44 %
1700	2.43	1.33	75.76 %	45.46 %
1900	2.51	1.25	64.65 %	60.60 %
2100	2.45	1.33	73.73 %	44.44 %

Table 5.4: Test Results for letting the MCTS agent play against the MCTSRave agent. For each test, a different Rave bias was used. C value was set at 0.3 in both agents, and each test ran for six hours.

While the results of the tests are still very close, a bias value of 1700 seemed best with a winning rate of 56.57%. **Therefore, we decided to set the bias b of the MCTSRave to 1700.**

5.2 Playing strength testing

After we dialed in the different parameters for MCTS and MCTSRave, we started to let the different agents play against each other. We let the standard RandomAi play against the advanced one and then the advanced RandomAi against the HeuristicAi. Because of an average duration of under zero seconds for a simulated game, we decided to use an iteration limit of 500 instead of a general time constraint. The results can be seen in table 5.5. To no surprise, the standard RandomAi was clearly beaten by the advanced RandomAi. Continuing, the advanced RandomAi was clearly beaten by the HeuristicAi. While those results don't come by surprise, we can already make some interesting discoveries. First, while the advanced RandomAi beat the standard RandomAi with a clear advantage of remaining pieces (6.74 vs. 3.66), the HeuristicAi beat the advanced RandomAi with a disadvantage in piece number (3.68 vs. 4.45). Since this difference lies at around one piece, a possible explanation could be that because of the implemented heuristics for the HeuristicAi, it was willing to sacrifice a piece of itself to gain a winning position afterward. Second, expectingly, temple wins did not play an important role in both simulations. However, a clear increase of temple wins can be witnessed during the simulations (4.44% vs. 16.2%), indicating that the HeuristicAi uses its game knowledge more to create more well-founded moves, therefore playing more tactically.

After those tests, we decided to use the HeuristicAi as a baseline to test the three

5 Experimental Evaluation

agent 1	agent 2	Wins agent 1	Wins agent 2	Avg. winner pieces	Avg. loser pieces	Avg. temple wins
Random Ai	advanced Random Ai	10	490	6.74	3.66	4.44 %
advanced Random Ai	Heuristic Ai	21	479	3.68	4.45	16.20 %

Table 5.5: Playout results for RandomAi vs. advanced RandomAi and advanced RandomAi vs. HeuristicAi. Each test runs 500 games.

MCTS agents against it. As we expected that all three MCTS agents would surpass the HeuristicAi’s playing strength heavily, we decided to let each test run for only two hours. However, while testing the HeuristicAi against the MCTS and MCTSRave agents, we encountered the problem of the agents rotating their pieces, which we explained in section 4.3. Both agents played the same pattern of moves over and over again, therefore being stuck in a loop and not attacking. Hence, we aborted the test run and implemented the mentioned change of checking whether a child *finalReward* would be above 0.99 and started the tests again. Results can be seen in table 5.6.

agent 2	Wins agent 1	Wins agent 2	Avg. duration	Avg. winner pieces	Avg. loser pieces	Avg. temple wins
MCTS	1	105	55.06 sec	5.41	2.10	0.01 %
MCTS Rave	2	92	56.29 sec	5.16	2.03	6.19 %
MCTS Heuris- tic	13	49	93.84 sec	3.64	2.17	34.38 %

Table 5.6: Playout results for HeuristicAi vs. all three MCTS agents. Agent 1 was always the HeuristicAi. Each test runs for two hours.

As the results show, the MCTSheuristic agent was clearly the weakest of the three agents. It achieved the lowest win rate with 79.03%. It also had significantly lower remaining pieces for the winner compared to the MCTS and MCTSRave agents. This also aligns with the comparatively higher rate of temple wins. Interestingly, the MCTSheuristic agent shows a much higher rate of temple wins compared to the MCTS and MCTSRave agents. A possible explanation for this behavior could be the substantially higher playing strength of the MCTS and MCTSRave agents. While those agents achieved wins with an average of 5.21 pieces left, the MCTSheuristic agent only had an average of 3.64 pieces left after a match. As more of its own pieces were captured, it was

5 Experimental Evaluation

harder to tactically capture the opponent’s master, hence making it more important to try to win by reaching the opponent’s temple.

Comparing the MCTS and MCTSRave agents, both agents share almost similar win rates (99.06% MCTS vs. 97.87% MCTSRave). They also share almost the same values regarding the average duration and the average remaining winner and loser pieces. However, the MCTSRave does win slightly more often through temple wins. It appears that through using the AMAF heuristic, the agent values moves that lead to a temple win higher than using the normal UCT move evaluation.

Still, both MCTS and MCTSRave perform almost identically, so we consider both agents the winners of this test run.

Lastly, we started comparing the different MCTS agents against each other. Because of the relatively even performance of both the MCTS agent and MCTSRave agent, we decided to modify a version of the MCTS agent not to use the move prioritizing of the RandomAi for the simulations to have a true ”standard” MCTS agent for comparison. For this test, we will call this agent MCTSLight and the normal MCTS agent, with move prioritizing, just MCTS. We therefore conducted the following tests: MCTSLight vs. MCTS, MCTSLight vs. MCTSRave, MCTSLight vs. MCTSheuristic, MCTS vs. MCTSRave, MCTS vs. MCTSheuristic and MCTSRave vs. MCTSheuristic. Each test ran for six hours; results can be seen in table 5.7.

As expected, the MCTSLight agent is by far the weakest of the four MCTS agents. While it had no chance against the normal MCTS and the MCTSRave agent with a win rate of 0.00%, it achieved a win rate of 7.04% against the MCTSheuristic agent. Because the difference in playing strength is so extreme, temple wins played almost no role in all three test runs. This is also reflected in the big gradient of winner pieces to loser pieces. Interestingly, this high gradient also counts for the MCTSheuristic agent. Still, the duration of the matches against it took notably longer compared to the standard MCTS and MCTSRave agent. This, and the higher win rate of the MCTSLight agent, indicate that the MCTSheuristic agent is capable of capturing a lot of pieces in the early phase of the game but then isn’t capable of using this piece advantage to finish off the match.

In general, it shows that the MCTSheuristic agent is, by far, the worst in playing strength when compared to the MCTS and MCTSRave agents. It achieved win rates of 0.00% or 0.59% respectively. That also matches the much lower average iterations during a move search. In all three tests, it had lower iterations than its opponents, achieving only an extreme nearly 1/6 of its opponent’s iterations. This indicates that the usage of an extensive hardcoded heavy playout is just too computationally heavy and that the heuristics considered in section 3.5 don’t compensate for this.

Finally, the best agents of the four tested MCTS variants seem to be the standard MCTS and MCTSRave agents. As already indicated by the test results for testing the Rave bias in table 5.4, those two agents always shared similar win rates of around 50%. Regarding the average duration of a game and average iterations per move search, the MCTSRave agent seems to be superior to the MCTS agent when playing against weaker agents. However, while the MCTS agent takes longer to finish a game, it seems to act

5 Experimental Evaluation

agent 1	agent 2	Wins agent 1	Wins agent 2	Avg. win- ner pieces	Avg. loser pieces
MCTS light	MCTS	0	90	5.35	0.53
MCTS light	MCTS Rave	0	105	5.09	0.43
MCTS light	MCTS Heuristic	5	66	4.29	0.32
MCTS	MCTS Rave	47	43	2.41	1.20
MCTS	MCTS Heuristic	124	0	4.66	0.95
MCTS Rave	MCTS Heuristic	168	1	4.67	1.03
agent 1	agent 2	Avg. du- ration	Avg. tem- ple wins	Avg. It- erations agent 1	Avg. It- erations agent 2
MCTS light	MCTS	157.36	0.00 %	167,452	204,474
MCTS light	MCTS Rave	149.31 sec	0.00 %	158,229	208,728
MCTS light	MCTS Heuristic	225.24 sec	5.13 %	73,739	41,012
MCTS	MCTS Rave	231.38 sec	67.03 %	221,146	166,498
MCTS	MCTS Heuristic	150.06 sec	4.69 %	189,775	37,865
MCTS Rave	MCTS Heuristic	128.51 sec	6.51 %	207,923	35,359

Table 5.7: Playout results for letting different MCTS agents play against each other. Each test ran for six hours. When a match took longer than 10 minutes, the run was aborted and not counted.

a bit more profoundly than the MCTSRave agent, therefore achieving a slightly better win rate when compared to agents with similar strength. However, contrary to what we expected the MCTS agent achieved a significantly higher average iteration number compared to the MCTSRave agent in their matches. This indicates that the storage and processing of the moves played during the simulations just takes too much time when playing against a similar strong player, therefore leading to this disadvantage.

When letting the MCTS agent play against the MCTSRave agent, we discovered an interesting behavior. Because of their equal playing strength, it often happened that both agents would align all their students along the central line of the board while keep-

5 *Experimental Evaluation*

ing their masters in the back, repeatedly in a corner. The agents would then reposition their students along this central line, and whenever one agent captured an opponent's piece, its piece would be captured by the opponent in the next round accordingly. Therefore, both agents tended to capture pieces very equally until one agent would have a piece advantage of 1 or 2. This agent then won the game almost every time. This behaviour however only occurred when playing against similar strong agents, not against weaker ones.

After all the experiments, we have established the following order with regard to the playing strength of the developed agents:

1. MCTS
2. MCTSRave
3. MCTSHuristic
4. (MCTSLight)
5. HeuristicAi
6. advanced RandomAi
7. normal RandomAi

6 Real life testing against a different AI

Parallel to this bachelor thesis, a fellow student of ours developed an AI for Onitama as well. Hence, after we finished testing our own AI agents, we let selected agents of ours play against his agents. Our colleague developed two AI agents. A standard MCTS agent with light playout and an MCTS agent with heavy playout. Both MCTS agents use a C value of 0.25. In its heavy-playout version, the agent uses Move prioritization for capturing and winning moves in his simulation phase. Further, simulations are cut after 50 iterations and evaluated based on the remaining pieces for each player. More about the implementation and development of the fellow students AI agents can be found in his paper (Peeck 2024). As we only wanted to test the strong versions of our agents, we decided to let our MCTS, MCTSRave, and MCTSHeuristic agent play against his MCTS heavy agent. Each test ran for six hours, and the results can be seen in table 6.1.

agent 2	Wins agent 1	Wins agent 2	Avg. duration	Avg. temple wins
MCTS	6	70	250.97 sec	43.04 %
MCTS Rave	10	73	239.80 sec	45.88 %
MCTS Heuristic	91	5	200.51 sec	13.13 %

Table 6.1: Playout results for the heavy MCTS agent of a fellow student vs. all three MCTS agents. Agent 1 was always the heavy MCTS agent. Each test ran for six hours.

It showed that the MCTS and MCTSRave agents are much stronger than the fellow student’s MCTS heavy agent, which is clearly superior to our MCTSHeuristic agent. This result is especially interesting, as the foundation for our MCTS and his MCTS heavy agent are nearly identical. Both agents use a relatively low C value (0.3 vs. 0.25) and prioritize winning and capturing moves during the simulation phase. It appears that the main reason for this gradient lies mostly in the way we implemented our agents, as we tested our light versions of the MCTS agents as well, and the strong difference in playing strength remained. Further, our colleague tested whether the cutoff in his simulations would negatively affect the playing performance of his AI. However, it showed that the usage of cutoff actually increased the playing performance, as the cutoff agent was able to run almost three times more simulations than the non-cutoff agent. For his cutoff, he used the piece difference to determine which player won a simulation. As we mentioned in section 5.2, when letting equal MCTS agents play against each other, this difference almost always leads to a win/lose, therefore being well suited as a heuristic to evaluate

the game state. Our colleague witnessed the same behavior with his MCTS variants during testing as well.

Aside from this, the results are more or less as expected. Similar to the ones shown in our own experiments, the MCTSHeuristic agent is again the weakest of our three. Because of its weakness, the average duration of a match is also the lowest. When comparing the MCTS and MCTSRave agents, the same schema as in our experiments shows: While the MCTSRave agents play a bit faster, therefore achieving more games in the six-hour time constraint, the MCTS agent plays a bit slower but earns a slightly higher win rate.

All in all, the results clearly prove the applicability of MCTS and its variants in Onitama's domain. Both our own agents and the agent of our fellow student achieved promising results during testing. It is expected that efficiency improvements to the heavy MCTS agent of the fellow student will result in a similar level of playing strength as in our own MCTS agent. Lastly, we believe that using a cutoff in the simulations for our own MCTS agents could potentially increase the performance of our AI even further.

7 Conclusion

In this paper, we explored the applicability of agent-based AI to the game Onitama.

We started by explaining the game of Onitama and presenting possible heuristics with which decisions could be made. We also presented similar games like Onitama and explored which algorithms and approaches were used to develop AI agents for them. Further, we conducted a game-theoretical analysis of the game, described its game characteristics, and calculated the average branching factor.

Because of the strong success of MCTS approaches in games similar to Onitama, we decided to focus on developing three MCTS agents. The first MCTS agent was a simple agent based on the UCT MCTS approach but used to prioritize capturing and winning moves during simulation playouts. Because of its success in Computer-Go, we also decided to develop an MCTS agent based on Rapid Action Value Estimation (RAVE). The last MCTS version used a heavy playout to use hardcoded heuristics and simulation cutoff. We also developed a random agent as a baseline and a hardcoded heuristic agent to assess the applicability of simple rule-based agents to the Onitama domain.

After we explained the development of our five agents, we configured them based on experiments. Afterward, we conducted multiple experiments to assess their playing strength and let them play against the AI developed by a fellow student of ours.

Overall, it showed that using MCTS approaches is best suited to the domain of Onitama compared to using hardcoded heuristics. However, when comparing the three MCTS approaches with each other, using a heavy playout with hardcoded heuristics proved to not be as performing as using the move prioritization or the Rapid Action Value Estimation. Those two agents also surpassed the playing strength of the best agent of our fellow student.

In general, we achieved our goal of developing a strong AI agent for the game Onitama. However, there are still improvements to be made.

First off, the RAVE agent was not as strong as expected, not surpassing the playing strength of the MCTS agent. A first possible explanation for this could lie in the relatively simple constraints that Onitama offers. While Onitama is not a simple game and offers an interesting research field because of its unique playing style through the rotating cards, the overall branching factor of the game is still very low compared to, e.g., Go (45.5 vs. 250). Therefore, even with the relatively low time constraint of two seconds per move, the MCTS agent still achieved nearly as many iterations as the MCTSRave agent when compared to weaker agents. In the case of the direct comparison, the MCTS agent even clearly surpassed the MCTSRave's iterations, as the additional processing of played moves during simulations took just too long for the MCTSRave agent. Hence,

7 Conclusion

we think that in a more complex environment, e.g., an Onitama board with 9x9 tiles, the MCTSRave agent might surpass the MCTS’s playing strength.

A second explanation could be the usage of the all-moves-first heuristic of RAVE in general. It seems that the assumption that a move always has the same value regardless of when it is played is not as strong as expected. Yet, in a more complex environment, this also could prove to be less important. Especially when having a bigger game board, like 13x13, e.g., we expected MCTSRave to work notably better than our MCTS approach, especially during the early phase of the game.

Further, the heuristics of the MCTSHuristic agent could be tested and explored more. In our approach, we only used very simple and basic strategies. However, with the use of other heuristics, better weights, and different cutoff strategies, higher playing strength may be achieved. Of course, here, the options are almost endless. Yet we think that with our proposed heuristics, we laid a notable foundation for further work, as the MCTSHuristic agent at least exceeded the MCTS light agent.

Bibliography

- AI, H. (2019). High-level expert group on artificial intelligence. *Ethics guidelines for trustworthy AI* 6.
- Allis, L. (1994, January). *Searching for solutions in games and artificial intelligence*. Ph. D. thesis, Maastricht University.
- Auer, P. (2002). Finite-time analysis of the multiarmed bandit problem.
- Brettspiel-news.de (2024). Starting board of onitama. <https://www.brettspiel-news.de/images/Bilderverzeichnis/onitama/onitama2.jpg>.
- Browne, C. B., E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1), 1–43.
- Butson, R. and R. Spronken-Smith (2024). Ai and its implications for research in higher education: a critical dialogue. *Higher Education Research & Development* 43(3), 563–577.
- Campbell, M., A. Hoane, and F. hsiung Hsu (2002). Deep blue. *Artificial Intelligence* 134(1), 57–83.
- Chang, H. S., M. C. Fu, J. Hu, and S. I. Marcus (2005). An adaptive sampling algorithm for solving markov decision processes. *Operations Research* 53(1), 126–139.
- Chaslot, G. M. J., M. H. Winands, H. J. v. d. Herik, J. W. Uiterwijk, and B. Bouzy (2008). Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation* 4(03), 343–357.
- Chess.com (2024). Stockfish.
- Collins, C., D. Dennehy, K. Conboy, and P. Mikalef (2021). Artificial intelligence in information systems research: A systematic literature review and research agenda. *International Journal of Information Management* 60, 102383.
- Coulom, R. (2007). Efficient selectivity and backup operators in monte-carlo tree search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers (Eds.), *Computers and Games*, Berlin, Heidelberg, pp. 72–83. Springer Berlin Heidelberg.
- Drake, P. and S. Uurtamo (2007). Move ordering vs heavy playouts: Where should heuristics be applied in monte carlo go. In *Proceedings of the*. Citeseer.

Bibliography

- Ensmenger, N. (2012). Is chess the drosophila of artificial intelligence? a social history of an algorithm. *Social Studies of Science* 42(1), 5–30. PMID: 22530382.
- Filippucci, F., P. Gal, C. Jona-Lasinio, A. Leandro, and G. Nicoletti (2024). The impact of artificial intelligence on productivity, distribution and growth. (15).
- Gelly, S. and D. Silver (2011). Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence* 175(11), 1856–1875.
- Georgios N. Yannakakis, J. T. (2018). *Artificial Intelligence and Games*. SpringerLink.
- Helmhold, D. P. and A. Parker-Wood (2009). All-moves-as-first heuristics in monte-carlo go. In *IC-AI*, pp. 605–610. Citeseer.
- Iida, H., M. Sakuta, and J. Rollason (2002). Computer shogi. *Artificial Intelligence* 134(1), 121–144.
- Knuth, D. E. and R. W. Moore (1975). An analysis of alpha-beta pruning. *Artificial Intelligence* 6(4), 293–326.
- Kocsis, L. and C. Szepesvári (2006). Bandit based monte-carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (Eds.), *Machine Learning: ECML 2006*, Berlin, Heidelberg, pp. 282–293. Springer Berlin Heidelberg.
- Lai, M. (2015). Giraffe: Using deep reinforcement learning to play chess.
- Lantz, F., A. Isaksen, A. Jaffe, A. Nealen, and J. Togelius (2017). Depth in strategic games. In *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*.
- Lee, C.-S., M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong (2009). The computational intelligence of mogo revealed in taiwan’s computer go tournaments. *IEEE Transactions on Computational Intelligence and AI in Games* 1(1), 73–89.
- McCorduck, P. and C. Cfe (2004). *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. AK Peters/CRC Press.
- Nadikattu, R. R. (2016). The emerging role of artificial intelligence in modern society. *International Journal of Creative Research Thoughts*.
- Nogueira, N. (2024). Minimax Algorithm — Wikipedia, The Free Encyclopedia. [Online; accessed 06-Oct-2024].
- Peeck, K. (2024, December). Developing a strong ai for the game onitama. Bachelor’s thesis, University of Mannheim, Mannheim, GER.
- Roy, S., C. Ellis, S. Shiva, D. Dasgupta, V. Shandilya, and Q. Wu (2010). A survey of game theory as applied to network security. In *2010 43rd Hawaii International Conference on System Sciences*, pp. 1–10.

Bibliography

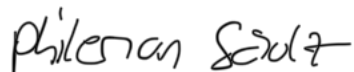
- Russell, S. J. and P. Norvig (2010). *Artificial intelligence: a modern approach*, Volume Third Edition. Pearson.
- Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis (2016). Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587), 484–489.
- Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm.
- Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis (2017). Mastering the game of go without human knowledge. *Nature* 550(7676), 354–359.
- Takizawa, T. (2005, 04). Computer shogi 2000 through 2004.
- Takizawa, T., T. Ito, T. Hiraoka, and K. Hoki (2015). *Contemporary Computer Shogi*, pp. 1–10. Cham: Springer International Publishing.
- Teytaud, F. and O. Teytaud (2010). Creating an upper-confidence-tree program for havannah. In H. J. van den Herik and P. Spronck (Eds.), *Advances in Computer Games*, Berlin, Heidelberg, pp. 65–74. Springer Berlin Heidelberg.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Bachelor-, Master-, Seminar-, oder Projektarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und in der untenstehenden Tabelle angegebenen Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Declaration of Used AI Tools

Tool	Purpose	Where?	Useful?
DeepL	Translation	Single sentences	+
GitHub Copilot	Code generation	Single code snippets	++



Unterschrift

Mannheim, den 03.12.2024