The performance of information set Monte Carlo tree search in the game Klaverjas

Bart Aaldering

Supervisors:
Jan N. van Rijn & Peter Koning & Joost Broekens

BACHELOR THESIS

24/07/2023

Abstract

In solving real-world artificial intelligence problems, the significance of game-solving cannot be overlooked. Many real-world problems have imperfect information elements. This makes solving imperfect information games especially relevant. One such game is Klaverjas. Klaverjas is a trick-taking game played with four players in teams of two. The objective of this game is to collect as many points as possible by winning tricks. In this thesis, we introduce a Klaverjas agent. Inspired by the remarkable success of AlphaZero in the realm of game-solving, we have drawn inspiration from its methods and applied them to the development of our Klaverjas agent. We did this to answer the question: How applicable are the techniques used in AlphaZero to imperfect information games? We developed an agent using an imperfect information variant of Monte Carlo tree search. We combined this with three different state value estimators, i.e., a random rollout approach, a neural network approach trained on human-generated data and a neural network approach trained on self-play data. We tested our agents against a rule-based agent. The score we kept track of was the average point difference per round. Against the rule-based agent, the random rollouts agent scored 55 points more than its competitor, the human-data agent scored 50.5 more, and the self-play agent scored 45.8 more. We found that the Monte Carlo tree search variant is an effective approach for solving Klaverjas.

Contents

1 Introduction **1**

2 Background **2**
2.1 Related work 2
2.2 Klaverjas 3
2.3 MCTS 5
2.4 SO-ISMCTS 6
2.5 Reinforcement learning 6
2.6 Existing Klaverjas agents 7

3 Implementation details **8**
3.1 SO-ISMCTS implementation 9
3.1.1 Determinization step 9
3.1.2 MCTS steps 9
3.2 Input representation neural network 12
3.3 Neural network configuration 14
3.4 Self-play agent 14
3.5 Human-data agent 15
3.6 Experimental environment 15

4 Results **16**
4.1 Different amounts of exploration and simulations 16
4.2 Comparing the three agents 18
4.3 Time efficiency 20
4.4 Training the self-play agent 21

5 Discussion **21**

6 Limitations and future work **23**

7 Conclusion **24**

References **27**

Appendix **27**

1 Introduction

Advancements in artificial intelligence have not only grabbed the attention of researchers but also that of the general public. One area where artificial intelligence has shown remarkable progress is in the realm of games. Games have long served as a benchmark for developing and evaluating artificial intelligence algorithms. This is partially due to their well-defined rules, clear objectives, and competitive nature. Through strategic decision-making, pattern recognition, and sophisticated planning, artificial intelligence agents have been able to achieve superhuman performance in various game environments [Tes95, CJH02, BBJT17]. The progress made by studying games enables the development of algorithms that will scale to more complex, real-world problems.

One notable example of the advancements in artificial intelligence through games is AlphaGo. This artificial intelligence agent defeated the world-renowned Go player Lee Sedol in 2016 [SHM+16]. Go is an ancient board game with a higher complexity than chess. The victory of AlphaGo over Lee Sedol was a pivotal moment that showcased the power of artificial intelligence and its potential to surpass humans in highly complex tasks. Building upon the success of AlphaGo, subsequent advancements led to the development of AlphaZero [SSS+17, SHS+17]. Unlike its predecessor, AlphaZero learns solely through self-play, without human-designed heuristics or domain-specific knowledge. Surprisingly this simple approach achieved superior performance. The techniques used in AlphaGo and AlphaZero that helped them become successful are the combinations of Monte Carlo Tree Search (MCTS) and neural networks trained with reinforcement learning.

While the achievements of AlphaZero have undoubtedly pushed the boundaries of artificial intelligence research, it is crucial to recognise the limitations of its approach. A limitation of AlphaZero is its reliance on perfect information games. In perfect information games, all players have complete knowledge of the game state. These games are thus well-suited for algorithmic optimisation and analysis. However, most real-world problems do not have perfect information. Real-world problems are often characterised by imperfect information [HS16], where players have limited or incomplete knowledge of the environment. Because of this hidden information, the decision-making process is more complicated. Therefore, advancements in artificial intelligence research that focus on imperfect information would be beneficial to help solve these hard imperfect information problems. In real-world scenarios, imperfect information problems are not the hardest problems because we know what we do not know (known unknowns). There are also problems in which we do not know what we do not know (unknown unknowns). Unknown unknowns can have a high impact but are less common and harder to reason about. Because of this, we will only focus on imperfect information (known unknowns). As AlphaZero has proven to be such a successful technique, it would be interesting to see how well it would do in imperfect information settings. In the past, this has not been that successful [BBLG20], but recently progress is being made [BBLG20, BCK23]. To further test how applicable the techniques of AlphaZero are to imperfect information games, we have applied them to the game of Klaverjas. Klaverjas is an imperfect information game that is popular in the Netherlands. Despite its popularity there is currently no human-level agent developed for it. This makes it an interesting game for our research. Our goal was to develop an AlphaZero-inspired Klaverjas agent with the aim of optimal performance. The primary research question we aimed to address was: How applicable are the techniques used in AlphaZero to imperfect information games in particular Klaverjas? Additionally, we wanted to identify what parts of AlphaZero contribute the most to our agent's potential strength. Specifically, the extent to which our agent's efficacy stems from MCTS, neural networks, or reinforcement learning.

To answer these questions, we developed a Klaverjas agent that uses an imperfect information variant of MCTS called single-observer information set Monte Carlo tree search (SO-ISMCTS) [CPW12]. We combined this search algorithm with three different state value estimator algorithms. We used a random rollout approach, a neural network approach trained on human-generated data and a neural network approach trained on self-play data. This gave us three different agents, which we will call rollout agent, human-data agent and self-play agent from now on. The rollout agent works by randomly playing moves till the end of the round and taking the end score as a value estimate. The neural networks try to predict the state value and have as input an array representing the current state of the game from the perspective of one of the players. The human-data agent trains this network solely on games played by humans. While the self-play agent only uses games it played against itself to train its neural network.

To compare the performance of our agents, we let them play against a rule-based agent. In addition to this, we also tested them against each other. By doing this we can see if the differences in performance against the rule-based agent are good estimates of how good they would do when playing directly against each other. In addition to this, we tested different exploration rates with the rollout agent. We then used these optimised values for testing the other agents. Furthermore, we test the effect of more searches and more rollouts on the performance and the compute time. This way we have selected the most time-efficient method. We expected that the use of search could create a decently performing agent but that for better performance we needed to use neural networks.

Our thesis will be organised as follows. We will first look at some background knowledge in Section 2. In this section, we take a look at related work and explain Klaverjas, MCTS, SO-ISMCTS, reinforcement learning and two other Klaverjas agents. Then, our methods are explained in Section 3. This section explains the implementations of the SO-ISMCTS, the neural network, the self-play agent, the human-data agent and the testing. After our results in Section 4, a discussion is provided in Section 5. Next, the limitations and future work are discussed in Section 6. Finally, Section 7 concludes.

2 Background

This section contains an overview of related work and the required background knowledge. It contains an explanation of related work on AlphaZero, Klaverjas, MCTS, reinforcement learning and two other Klaverjas agents.

2.1 Related work

The techniques from AlphaZero have already been applied to a variety of imperfect information games. In [BCK23], the use of perfect information Monte Carlo [LSBF10] was combined with a similar value and policy network as in AlphaZero. This combination proved to be successful in the games Stratego and DarkHex. Both games are two-player board games. Perfect information Monte Carlo is similar to SO-ISMCTS as they both sample perfect information states from the imperfect information state.

Another successful use of search and reinforcement learning is ReBeL [BBLG20]. ReBeL is a general framework for self-play reinforcement learning. It uses counterfactual regret minimization [ZJBP07, GBLS12, BLGS18] to search and also uses a policy and value network. This was successful in heads-up no-limit Texas hold 'em poker a two-player card game that requires bidding. Counterfactual regret minimization is a search algorithm used in game theory to find optimal strategies in imperfect information games. It iteratively updates strategies based on regret values and aims to minimise the overall regret of actions taken during gameplay. It requires complete traversal of the game tree making it impractical for games with large state spaces without some form of abstraction.

In [JLD+19], human expert level is reached for the game Doudizhu, a three-player bidding card game. This is done using reinforcement learning combined with ISMCTS and using a policy-value network that approximates the policy and value in each decision node.

There have not yet been many studies done on the game of Klaverjas. In [vRTV18], an exact approach and a machine learning approach are compared to predict whether or not a combination of hands is winning. This approach used perfect information, meaning that players could see the other player's hands. There have been studies done on other trick-taking games similar to Klaverjas. For example, in [Gin01] expert-level agents were made for the game Bridge. Bridge is a popular four-player trick-taking game. In [BLFS09] an expert-level agent was made for the game Skat. Skat is a three-player trick-taking game. Both of these approaches only used perfect information Monte Carlo.

2.2 Klaverjas

Klaverjas is a trick-taking card game. It is popular in Dutch-speaking regions and is played by four players in teams of two. The four players are usually referred to as North, East, South, and West, with North and South forming one team and East and West forming the other. It is important to note that players are not allowed to communicate with their partners during gameplay. In Klaverjas, cards 7 through Ace of the French card deck are used. A game of Klaverjas consists of multiple rounds. At the beginning of each round, the 32 cards are distributed among the players, giving each player eight cards. The objective of a round in Klaverjas is to score as many points as possible.

Before a new round begins, the trump suit has to be determined. Trump suit cards have a special role in Klaverjas as they are worth more points and are stronger than normal cards. At the start of the rounds, a pre-determined bidding procedure determines the trump suit. This procedure can vary per game variant and is out of the scope of this thesis. If the declaring team fails to bring in more than half of all the points all the points of that round will go to the opponent's team. This is called "nat".

Once the trump suit is established, the starting player starts the first trick by playing a card from their hand. The other players must follow this suit if they have a card of this suit. However, if they do not have any cards of this suit, they must play a trump card if they have any. If a trump card must be played and there has already been a trump card played, if possible, a higher trump card must be played. In "Amsterdams" Klaverjas you do not have to play a trump card if your partner is winning the trick. This exception does not count for "Rotterdams" Klaverjas. If a player can't follow suit nor play a trump card, any card can be played. After a card has been played, it is the turn of the next player in the clockwise direction.

One trick consists of playing four cards, with each player contributing one card in order of position. The player who plays the best card of the trick wins this trick and starts in the next one.

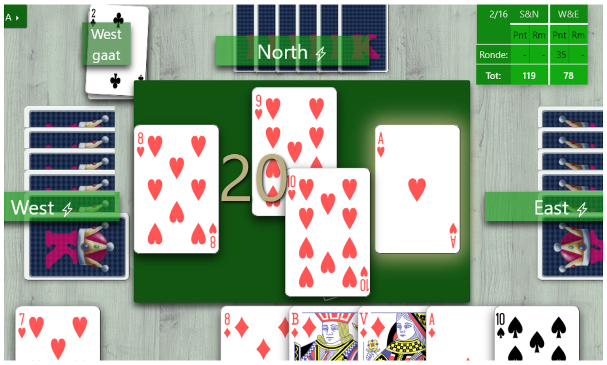

Figure 1: An example of a Klaverjas trick that has 20 meld points for having three cards in a row.[1]

The best card is the highest trump card, or if no trump card has been played, the highest card of the leading suit. The leading suit is the suit of the first played card. What cards are the highest, and thus win over other cards, follows the same ranking as the point value of the cards. The value and, thus, order of each card can be found in Table 1. The team of the winner of the trick receives all the points in that trick. These points consist of the points of the cards and the so-called meld points.

Regular		Trump	
A	11	J	20
10	10	9	14
K	4	A	11
Q	3	10	10
J	2	K	4
9	0	Q	3
8	0	8	0
7	0	7	0

Table 1: The value and order of cards in Klaverjas.

How meld points are earned will now be discussed. Winning the last trick will give you an additional 10 points, and if the declaring team wins all eight tricks, they have obtained a so-called "pit" and get an additional 100 points. Additionally, meld points can be obtained if a trick contains cards following a specific pattern. For these patterns only the cards that have been played matter, the order in which they are played does not matter. Tricks with three cards in a row (normal deck order) are worth 20 extra meld points, tricks with four in a row 50, tricks containing the king and queen of trump 20 and tricks containing four of the same face value 100. An example of a trick with three cards in a row and thus 20 meld points can be seen in Figure 1. These extra meld points make the game more interesting as it is no longer about getting as many valuable cards as possible but also about winning extra points and not giving away any.

Once all eight tricks have been played, the round comes to an end. The typical goal of Klaverjas is to as a team reach 1 500 points as quickly as possible or to get the most points after 16 rounds.

[1]Screenshot from: klaver.live

To play Klaverjas effectively, players employ various strategies and tactics. Implicit communication and teamwork between partners are crucial. Partners should develop a system of signalling and understand each other's card play to maximise their chances of winning tricks. Second, it is important to keep track of the cards played so you know what cards are still in the game. Third, it is important to note when a player is unable to follow suit as this means he can no longer have this suit. Lastly, players should be mindful of choosing trump suits, as failing to win more than half the points will result in losing all points.

2.3 MCTS

To be able to better understand the MCTS variant we used, we will first have a look at MCTS [Cou06, KS06]. MCTS is a search algorithm that uses simulations and statistics to make optimal decisions in decision-making processes or game-playing scenarios. It does this by building a tree structure (the search tree). The nodes of this tree represent the state of the process and the edges represent the action that takes you from one state to the next. The MCTS search starts at the root of the tree, which represents the current state of the process. Subsequently, the following four steps are repeated: selection, expansion, rollout and backpropagation. In Figure 2, these steps are presented visually. One series of these four steps is called a simulation. One simulation adds one node to the tree. These simulations are done until a budget is reached. This can be, for example, a maximum number of simulations or a time limit.

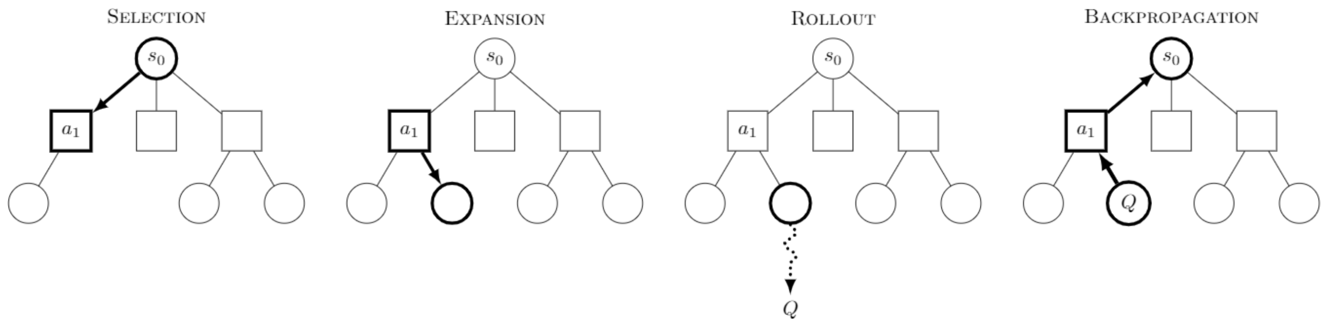

Figure 2: The four steps of MCTS.[2]

1. In the **selection** step, the algorithm starts at the root of the tree. Child nodes are selected repeatedly based on a particular selection policy usually, the upper confidence bound for trees formula [KS06, Ros11]. This selection policy balances the exploration of nodes that have not yet been visited often and the exploitation of nodes that have already received good results. The selection of child nodes is done until a node is reached that is not yet completed. A node is completed when all its child nodes are added to the tree.

2. In the **expansion** step, one of the child nodes not yet added to the tree is created and added to the tree. If there is more than one child node that can be added, one of them is chosen at random.

[2]Source: Wikipedia

3. In the **rollout** step, a value estimate is determined for the node added in the expansion step. To do this, from the added node, one or more paths are explored. The paths are selected using a random policy or another simple policy. The path ends when a leaf node is reached. At the end of the path, the value is determined. The value of the added node will be the average of all the paths explored. The exploration of one of these paths is also called a rollout.

4. In the **backpropagation** step, the results of the rollout step are propagated back up the tree. This is done following the path that was followed during the selection phase. For each node, the statistics that are updated are the number of visits and the total value. The total value is updated by adding the value of the rollout step.

2.4 SO-ISMCTS

SO-ISMCTS is an imperfect information variant of MCTS [CPW12]. In imperfect information games, some information is hidden. This makes it impossible to know what exact state we are in. We do know all the possible ways the hidden information can be filled in. In other words, we do know all the exact states we can potentially be in. These exact states together make a so-called information set. An information set is a collection of game states that a player can not differentiate between. These states are all states the player could be in but because of the hidden information, he does not know in which he is.

In each simulation, SO-ISMCTS does its version of the four MCTS steps. In addition to this, a fifth step is done at the start of each simulation. This is the determinization step and this addresses the problem of imperfect information. Determinization randomly samples one perfect information state from the information set. This perfect information state will then be used to do the MCTS steps.

In MCTS a node in the search tree represents a single state of the game or environment. In SO-ISMCTS, however, nodes represent an information set. Each node can be uniquely identified by the actions that were taken to get to this node. When traversing the tree, the selection, expansion and rollout steps will only select nodes that are possible given the current determinization. As a result of this, in each simulation, nodes can have different child nodes available, making some parts of the tree inaccessible.

2.5 Reinforcement learning

Reinforcement learning is a form of machine learning in which an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or punishments for its actions, allowing it to learn through trial and error. It does this to maximise its long-term cumulative reward. The goal is to find an optimal policy. A schematic representation of a typical reinforcement learning framework can be seen in Figure 3.

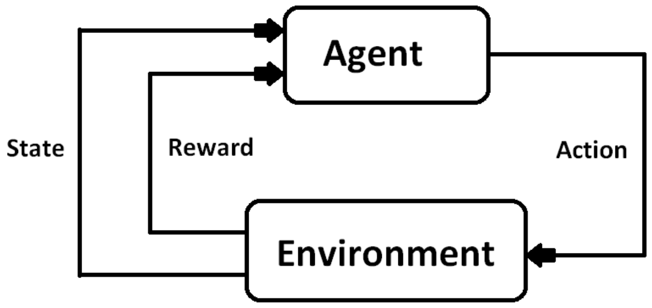

Figure 3: A typical framework of a reinforcement learning scenario.

The most simple form of reinforcement learning learns a policy or value for each possible state. For environments with very large state spaces, this becomes problematic as no longer all states can be stored. This can be solved by using function approximators instead. One of these function approximators is using a neural network. Neural networks are especially powerful as they can learn complex representations. The neural network takes the state as input and outputs the estimated value or policy. Neural networks in combination with reinforcement learning are called deep reinforcement learning. In deep reinforcement learning neural networks are trained using techniques such as stochastic gradient descent and backpropagation, where the gradients are calculated using a reinforcement learning algorithm, such as Q-learning [WD92, MKS$^+$15] or policy gradients [SMSM99].

2.6 Existing Klaverjas agents

In [Hor22], two agents for Klaverjas were developed. The first agent is rule-based. This agent plays cards based on some simple rules. These rules can be seen in Figure 4. We will, from now on, refer to this agent as the rule agent. We tested our agents directly against this agent.

```
if first turn or second turn then
    if Has highest card of cards still in the game of suit then
        Play highest card
    else
        Play card with the lowest value
    end if
else if third turn then
    if Cant follow asked suit then
        Play lowest card
    else if Teammate played the highest card left of suit then
        Play highest card
    else
        Play lowest card
    end if
else
    if Teammate is currently winning trick then
        Play highest card
    else if Has card that wins trick then
        Play this card
    else
        Play lowest card
    end if
end if
```

Figure 4: The pseudocode of the rule-based agent from [Hor22].

The second agent is a random forest [Bre01] agent. This agent uses 42 handmade features. There are features created by evaluating the whole hand, features created by evaluating a single card and features created by evaluating both. We were unable to test our agents directly against this agent as we did not have access to the random forest model or all the code used to create it. We can, however, use the results this agent got against the rule agent to estimate its performance. The random forest scored on average 13 points more per round than the rule agent.

3 Implementation details

We made three different agents. All three agents use the same basis, the SO-ISMCTS algorithm. To use this search algorithm we need to be able to play and reason over the game of Klaverjas. To do this, all the rules and scoring systems of Klaverjas were implemented. In our Klaverjas implementations, we always used the "Rotterdams" version.

When searching we need to estimate the value of a game state. Each agent does this differently. The rollout agent uses the same random rollouts as in MCTS. The human-data agent and the self-play agent both use a neural network that estimates the value of a game state. Both neural networks have the same architecture but the human-data agent's neural network is only trained on data generated by humans while the self-play agent only uses data generated by self-play. To test the performance of these methods, we set up a testing method. With this method, we can test against the rule agent as well as test different agents against each other. The neural network is explained in Section 3.2 and 3.3, the data generation and training are explained in Section 3.4 and 3.5 and the testing of the agents is explained in Section 3.6.

For all our implementations, we used Python version 3.9.15. All the computational expensive work was performed using the compute resources from the Academic Leiden Interdisciplinary Cluster Environment (ALICE) provided by Leiden University.

3.1 SO-ISMCTS implementation

How the five steps of SO-ISMCTS are implemented will now be explained. This will be done from the perspective of the player searching. The pseudocode of the algorithm we used can be found in Algorithm 1.

3.1.1 Determinization step

SO-ISMCTS starts with determinization. A determinization is done for each new node added to the tree. To do this in Klaverjas, a possible card distribution needs to be found for the hands of the other players. When doing this, the other players need to receive the right amount of cards and should not receive cards they could not have if they followed the rules. If, for example, a player has been observed to be unable to follow a suit, this player cannot receive cards of this suit. To keep track of these constraints, a list of cards is maintained for each player, which contains all the cards that this player can still have. At the start of a round, this list contains all the cards not in the hand of the searching player. As the round progresses, cards are removed from this list.

Even with the lists of possible cards finding a determinization is not that easy as every determinization needs to be a random sample from the possible determinizations. Finding a determinization in a short amount of time is especially challenging. Since assigning a card to the wrong player can be hard to notice and doing so can make all subsequent the search unnecessary. The search time of determinizations is important because one is done in each simulation, making it directly proportional to the total search time. We developed an algorithm that was able to find this determinization in a short amount of time. This algorithm can be seen in Algorithm 2.

The input of this algorithm consists of four lists. The first list is `hands` it contains three sets of cards, one for each player. This is what we are trying to fill. All hands are empty at the start. The second list is `possible_cards` this contains the three sets of cards we have kept track of containing the possible cards each player can still have. The third list is `all_cards` this is a list of tuples. It has a tuple for each card that still needs to be distributed. The first element of this tuple consists of the card it represents. The second element is a list with the numbers of all the players that can have these cards. The `all_cards` list is created from the `possible_cards` before the algorithm is first called. The fourth list is `num_cards` this contains an integer for each player representing how many cards his hand still needs. The index of all these lists corresponds directly to their player numbers. From the perspective of the searching player, player 0 is the player to the left of him, player 1 is his teammate and player 2 is the player to the right of him.

3.1.2 MCTS steps

After the determinization, we can do normal MCTS steps. Starting with the selection step. In this step, we keep going deeper into the tree by selecting child nodes using the following upper

Algorithm 1 Finds the best move by searching the search tree using SO-ISMCTS.

```
function SO-ISMCTS(start_state)
    current_state ← start_state
    current_node ← new SO-ISMCTS_Node()
    for simulations iterations do
        // Determination
        current_state.set_determinization()
        // Selection
        while current_state not terminal and current_state has no legal moves that are not
                in the children of current_node do
            current_node ← current_node.select_child_ucb(simulation)
            current_state.do_move(current_node.move)
        end while
        // Expansion
        if current_state not terminal then
            current_node = current_node.expand()
            current_state.do_move(current_node.move)
        end if
        // Simulation
        if current_state not terminal then
            sim_score ← 0
            for rollouts iterations do
                // Do random moves with current_state until round is complete
                current_state ← current_state.random_moves()
                // Save the score
                sim_score ← sim_score + current_state.get_score()
                // Undo the random moves
                current_state ← current_state.undo_random_moves()
            end for
        else
            sim_score = current_state.get_score()
        end if
        // Backpropagation
        while current_node.parent has a parent node do
            current_node.visits ← current_node.visits + 1
            current_node.score ← current_node.score + sim_score
            current_node ← current_node.parent
            current_state.undo_move()
        end while
        current_node.visits ← current_node.visits ← + 1
        current_node.score ← current_node.score + sim_score
    end for
    return the child from current_node with the most visits
end function
```

Algorithm 2 Finds a determinization from all possible determinizations uniformly at random.

```
function FIND_DETERMINIZATION(hands, possible_cards, all_cards, num_cards)
    if all_cards == [] then // Base case if all cards are distributed
        return True
    else
        stop ← False
        all_cards_copy ← all_cards
        choose card from all_cards_copy uniformly at random
        delete card from all_cards_copy
        shuffle card[1]
        for player in card[1] do // One by one try to give the card to one of the players
            num_cards[player] ← num_cards[player] − 1
            if num_cards[player] < 0 then // Check if this player still needs a card
                num_cards[player] ← num_cards[player] + 1
                continue with next iteration in for loop
            end if
            for other_player in card[1] do // Check if this card is not needed by other players
                delete card[0] from possible_cards[other_player]
                if length(possible_cards[other_player]) < num_cards[other_player] then
                    stop ← True
                    break from the for loop
                end if
            end for
            if stop then
                stop ← False
                num_cards[player] ← num_cards[player] + 1
                for other_player in card[1] do
                    add card[0] to possible_cards[other_player]
                end for
                continue with next iteration in for loop
            end if
            add card[0] to hands[player]
            if find_determinization(hands, possible_cards,
                                    all_cards_copy, num_cards) then
                return True
            end if
            num_cards[player] ← num_cards[player] + 1
            for other_player in card[1] do
                add card[0] to possible_cards[other_player]
            end for
            delete card[0] from hands[player]
        end for
        return False
    end if
end function
```

confidence bound formula:

$$A = argmax_a \left(\frac{s_a}{n_a} + C * \sqrt{\frac{\ln(N)}{n_a}} \right).$$

Here *argmax* outputs node A from the set of possible child nodes such that it has the highest score. A is the child node that will be selected, s_a is the total score of node a, n_a is the number of visits of node a, N is the number of simulations already done and C is the exploration rate. This C value balances between the left part of the formula that tells us what we already know (exploitation) and the right part that tells us the potential the node still has (explorations). In our implementations, the C value ranges between 3 and 204 800. Selection is done until, given the current determinization, a node is reached that has a child node that has not yet been added to the tree or until we reach a terminal node (game complete).

After this, we do the expansion. This is only done if we do not reach a terminal node in the selection. In the expansion step, we add a child node to the node reached in the selection step. This node has to represent a move that is not already a child node and is legal given the current determinization. If there are multiple options, one of these is chosen at random.

In the evaluation step, we need to determine the value of the node that was just added. To do this, we have two options: we can use a certain number of random rollouts, or we can evaluate the state of the node using a neural network. What this neural network looks like and how it is trained is explained in Section 3.2, 3.3, 3.4 and 3.5. When the rollouts reach the end of a round the difference between the scores of the two teams is used as the value. If multiple rollouts are done the average of these values is used.

After this, we do the backpropagation step. This is done by adding the value of our expanded node to all the nodes in our action trace. These are all the nodes we visited to get to the added node. These five steps are repeated until the search budget is reached. If this is reached, we see what child node from the root node has the highest visit count and select this node as the best move.

3.2 Input representation neural network

We have implemented a neural network to evaluate the value of a given state from a specific player's perspective. This player we will call the original player. This network will get a state representation as input and will try to predict the difference between the end scores of the Klaverjas round.

We represent the game state as an array of 299 float16 values. This array consists of two parts, a part placing every card in a (potential) position and a part giving extra information about the state. The first part can be seen as a flattened 2D array that has 32 rows and nine columns. Each row represents a card, and each column a possible card position. An example of this can be seen in Table 2. Column one represents the player's hand, columns two, three and four represent the other players in clockwise order; columns five, six, seven and eight are the centre positions starting from the player's position and continuing in clockwise order, and column nine is for cards already played in a previous trick. The row's first seven cards are always the trump cards from 7 through ace, with the other suits coming after that in random order but also going from 7 through ace. The position of the players is always from the perspective of the original player and the trump cards are always first because then similar games are not represented differently. The second part of the array consists of the following 11 values:

Card \ Location	own hand	left opponent hand	teammate hand	right opponent hand	own centre	left opponent centre	teammate centre	right opponent centre	previous trick
trump 7	0	0	0	0	0	0	0	0	1
trump 8	0	0	0	0	0	0	0	0	1
trump 9	0	0	0	0	0	0	0	0	1
trump 10	0	0	0	0	0	0	0	0	1
trump jack	0	0	0	0	0	0	0	0	1
trump queen	0	0	0	0	0	0	0	0	1
trump king	0	0	0	0	0	0	0	0	1
trump ace	0	0	1	0	0	0	0	0	0
not trump1 7	0	0	0	0	0	0	0	0	1
not trump1 8	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump1 9	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump1 10	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump1 jack	1	0	0	0	0	0	0	0	0
not trump1 queen	0	0	0	0	0	0	0	1	0
not trump1 king	0	0	0	0	0	0	1	0	0
not trump1 ace	0	0	0	0	0	0	0	0	1
not trump2 7	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump2 8	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump2 9	0	0	0	0	0	0	0	0	1
not trump2 10	1	0	0	0	0	0	0	0	0
not trump2 jack	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump2 queen	1	0	0	0	0	0	0	0	0
not trump2 king	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump2 ace	1	0	0	0	0	0	0	0	0
not trump3 7	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump3 8	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	0
not trump3 9	0	0	0	0	0	0	0	0	1
not trump3 10	0	0	0	0	0	0	0	0	1
not trump3 jack	0	0	0	0	0	0	0	0	1
not trump3 queen	0	0	0	0	0	0	0	0	1
not trump3 king	0	0	0	0	0	0	0	0	1
not trump3 ace	0	0	0	0	0	0	0	0	1

Table 2: Example of the first part of the array representation we use with our neural network. In this part all cards in the game are given a probability of being in a certain position.

- 4 values that represent the one hot encoded starting player,
- 4 values that represent the one hot encoded current player,
- 1 value that represents a Boolean value for whether the original player is in the declaring team or not,
- 2 values contain the current scores for each team with the first value being the score of the original player's team.

The one-hot encoded arrays are from the perspective of the original player. This means the first value represents himself, the second value represents the player left of him, the third opposite of him and the fourth right of him.

3.3 Neural network configuration

The neural network itself is a `Sequential` model from Tenserflow's Keras. It consists of the following layers:

1. a dense layer with 1 024 nodes and `rectified linear unit` as activation function,
2. a dense layer with 512 nodes and `rectified linear unit` as activation function,
3. a dense layer with 128 nodes and `rectified linear unit` as activation function,
4. a dense layer with 16 nodes and `rectified linear unit` as activation function,
5. a dense layer with 1 node and `rectified linear unit` as activation function.

Adam was used as the optimiser, and the mean squared error was used as the loss metric.

3.4 Self-play agent

The training of the neural network consists generation of the training data using self-play and the training of the neural network. These two steps (one cycle) are repeated until the neural network stops improving. The performance of the neural network is tracked by testing it against the rule-based agent and tracking the score difference. This is done every 20 cycles. We only do this every 20 cycles to prevent spending too much time measuring the loss. We test the network using the test described in section 3.6. The two steps will now be explained.

Self-play is the longest of the two steps. Because of this, we did this step in parallel. Each cycle we did 60 self-play rounds. We divided these rounds between the multiple processes and started doing self-play with each process. For each round, we initialise a random starting configuration. We choose a random starting player, suit and declaring team. After this, we let each player determine what card they want to play and save that player's perspective of the game to an array representation. When the game is finished, all array representations get paired with the difference between the final scores of the two teams. These values then become the training data, with the array representations being the input and the scores the output. All the training data of all the processes are concatenated and added to a buffer. This so-called replay buffer contains If the size of this buffer exceeds 43 200 training pairs, the last added data pairs are removed so the size stays within this maximum. After this, we are ready for the train step.

In the training step, we train the neural network on a subset of training data in the replay buffer. The size of this training data is the same as the amount of data generated by one self-play step. The data is then randomly sampled from the replay buffer. To train the network, we use the fit function with a learning rate of 0.01. After 50 training cycles, the learning rate was lowered to 0.001. We use 1 epoch and a batch size of 2 048.

3.5 Human-data agent

All the data that was used to train the human-data agent was provided by [Kon]. This is a website where everyone can play online games of Klaverjas. The data set consisted of 87 218 rounds of Klaverjas played by humans. These rounds contained all the information needed to know how a round went. From each of these rounds, we have 33 different game states, one for each card played and one for the start of the round. For each of these game states, we have four perspectives on the game, one for each player. This provided us with a total of $87\,218 * 33 * 4 = 11\,512\,776$ perspectives. These perspectives were transformed into the array representation for our neural network. These array representations together with the score difference of the outcome of the games become the training data.

With this training data, the same neural network as explained in Section 3.2 and 3.3 was trained. This was done by first splitting the data in a train and test set, with 10% of the data going to the test set. When training a batch size of 2 048 was used. The training was stopped once an epoch had a lower validation loss (loss on the test set) than that of the previous epoch.

3.6 Experimental environment

To benchmark our agents and neural networks, an experimental environment was developed. In this environment, different agents can play against each other. In all the tests we did, there were always two agents of the same type in a team. To keep testing more consistent, tests were always done on the same initial game configuration. In addition to this, all rounds were done twice, once normal and once with the starting cards and declaring team switched between the teams. This was done to even out imbalances in the starting configuration. The starting configurations of the rounds are from the same dataset as the human-data agent was trained on. These starting configurations contain the starting cards, the declaring team and the These configurations are more realistic than random ones because humans have chosen if they want their team to be the declaring team by choosing a trump suit. Most of the tests done were against the rule agent described in Section 2.6. When testing, two metrics were kept track of. The difference between the scores of the two teams for each round and the time it takes a player to find a move. With the score differences, the average score difference can be determined, as well as the estimated standard error of this average $se = s/\sqrt{n}$, where s is the score difference standard deviation and n is the number of rounds. With this value, the 95% confidence interval can be determined.

4 Results

In this section, we present the results of our experiments. In Section 4.1 the results of using different amounts of exploration (C values) and simulations for the rollout agent are presented. In Section 4.2 results of our three agents are compared. To clarify, the number of simulations is the number of nodes added to the search tree and can be seen as the search budget. The results of using different amounts of rollouts and simulations on computation time are presented in Section 4.3. In Section 4.4 the results of different amounts of self-play time are presented. In these sections, results are shown using the score difference. This is the difference in scores between the two teams at the end of a round. This score difference is always an average over multiple rounds and is from the perspective of the agent that is tested. Thus a positive score means the agent scores more points than its opponent.

4.1 Different amounts of exploration and simulations

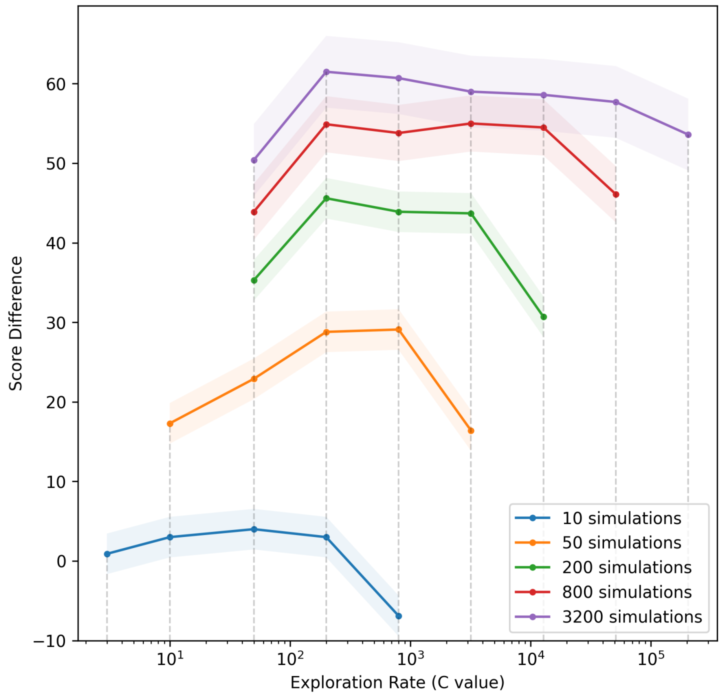

Figure 5: This figure shows the average score difference of the rollout agent versus the rule agent. It does this for different exploration rates (C values) and different tree depths (simulations). The bar around the lines shows the 95% confidence interval of the score averages. The exact C values that were tested are 3, 10, 50, 200, 800, 3 200, 12 800, 51 200 and 204 800.

Simulations	10	50	200	800	3 200
Standard error	1.3	1.3	1.3	1.8	2.3
Rounds	10 000	10 000	10 000	5 000	3 000

Table 3: The table shows the details of the data points in Figure 5. It shows the standard error and the number of rounds done for each simulation amount.

We wanted to find out what the effect of deeper search was on the performance of our rollout agent. We also wanted to find out what the effect of different exploration rates was so we could use these values to test the much more computationally expensive agents with neural networks. To do this, we tested this agent against the rule agent. All the tests were done with one random rollout per simulation. For each amount of simulation, we tested different exploration rates until we found the optimum. The result of these tests can be seen in Figure 5. The standard error and the number of rounds done for each amount of simulations can be seen in Table 3. We found that with ten simulations, the best exploration rate is around 50, with a best score of 4. With 50 simulations, the best exploration rate is 800, although 200 also performs well. This setup scores around 29 points. With 200 simulations, the best exploration rate is around 200, with a score of 45.6. With 800 simulations, the 200, 800, 3 200, and 12 800 are all very close, having a score of around 54.5 points. With 3 200 simulations, the best exploration rate seems to be 200, although the large confidence interval makes this less clear. 3 200 simulations get a best score of 61.5. With most of these settings, the rollout agents outperform the random forest agent 2.6, which got a score of 13 against the rule agent. Lastly, we can see that an exploration rate of 200 performs well for all numbers of simulations.

4.2 Comparing the three agents

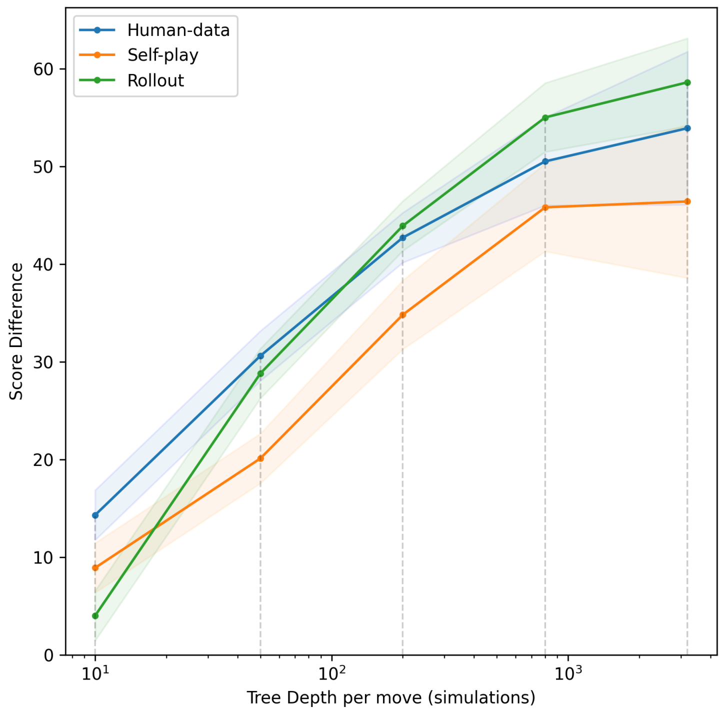

Figure 6: This figure shows the effect of more simulations on the average score difference against the rule agent. The rollout agent has one random rollout per simulation. The bar around the lines shows the 95% confidence interval of the score averages.

Simulations	10	50	200	800	3 200
Human-data	10 000 - 1.3	10 000 - 1.3	10 000 - 1.3	3 000 - 1.3	1 000 - 2.3
Self-play	10 000 - 1.3	10 000 - 1.3	5 000 - 1.8	3 000 - 2.3	1 000 - 4

Table 4: This table shows details of the data points in Figure 6. It first shows the number of rounds done and then the standard error for each agent. It does this for different agents and different numbers of simulations (search budget).

Simulations	10	50	200	800	3 200
C value	50	200	800	3 200	12 800

Table 5: This table shows the C value (exploration rate) for the data points in Figure 6.

To see the individual performance, to compare and to see the effect of a deeper search of the three different agents we made, we tested them against the rule agent.

The results of these tests can be seen in Figure 6 and the test details can be seen in Table 4. In this figure, we can see that increasing the number of simulations increases the score for all the agents. This effect becomes less the more simulations are done. We can also see that with low amounts of simulation, the agents with neural networks outperform the rollout agent. At higher amounts of simulations, the rollout agent is better. For the self-play agent, this switch point is at around 17 simulations. For the human-data agent, this is around 110 simulations. The performance of the human-data agent and the self-play agent grows at about the same rate, but the human-data agent outperforms the self-play agent consistently with around 8 points.

To test if these performance differences would hold up in a heads-on competition we tested the agents directly against each other. We did this using 10 and 200 simulations. Ten thousand rounds were used for each test giving us a standard error of 1.3 for all of them. The rollout agent used one random rollout per simulation. The results of these tests can be seen in Table 6.

Agents \ Simulations	10	200
Rollout vs Human-data	-3.5	3.9
Rollout vs Self-play	-2.9	13.1
Human-data vs Self-play	-0.2	5.9

Table 6: This table shows the average score difference when our three agents play directly against each other. The score is from the perspective of the first agent, meaning that positive values are in its favour and vice versa. The rollout agent uses one random rollout per simulation (search budget).

We can look at the difference in score difference against the rule agent and see if they are similar to the heads on score differences. To do this we have put the difference in score difference from Figure 6 in Table 7. The difference between Table 6 and 7 is thus that the first table shows the direct score difference while the second table shows the difference in score difference when playing against the rule agent.

Agents \ Simulations	10	200
Rollout vs Human-data	-1.3	1.2
Rollout vs Self-play	-4.9	9.1
Human-data vs Self-play	5.4	7.9

Table 7: This table shows the score difference of the average score difference of each agent against the rule agent. The scores are from the perspective of the first agent, meaning that positive values are in its favour and vice versa.

When comparing these two tables, we can see that they match quite well. What can be noticed is that with ten simulations, the performance of the two neural network agents is identical when playing against each other, but the human-data agents score more points than the self-play agent against the rule agent. This effect can, however, not be seen in 200 simulations.

4.3 Time efficiency

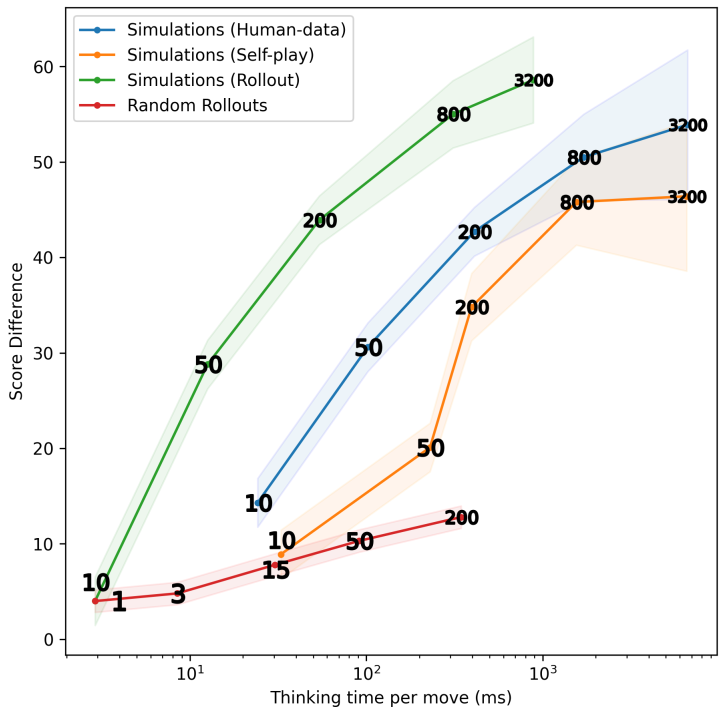

Figure 7: This figure shows the time efficiency of different agents and agent settings. The score is the average score difference against the rule agent. The numbers on the line represent either the number of simulations (blue, orange and green lines) or the number of random rollouts (red line). The bar around the lines shows the 95% confidence interval of the score averages.

To see the effect of increasing the number of simulations and using neural networks on the time it takes to come up with a move, we measured how long different agent configurations took to come up with a move while testing against the rule agent. When the rollout agent is tested with different simulations, one random rollout is used, and when its random rollouts are tested, ten simulations are used. The result of this can be seen in Figure 7. We found that using extra random rollouts does increase performance but not as much as increasing the number of simulations. We also found that using the neural network made finding a move take around eight times longer. This can be seen in Figure 7 because for the same number of simulations, the points of the neural network agents lie about eight times to the right of those of the rollout agent.

4.4 Training the self-play agent

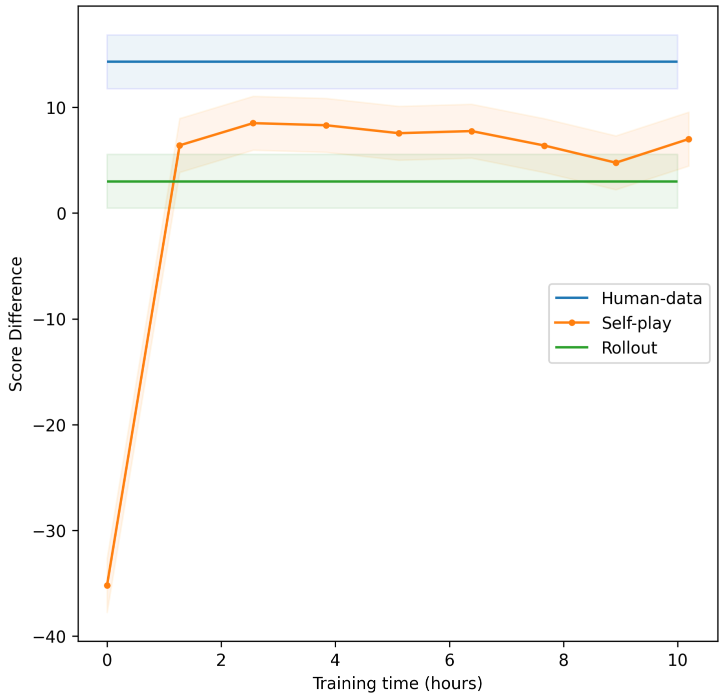

Figure 8: This figure shows the learning curve of the self-play agent when training on self-play data. The score is the average score difference against the rule agent, with each agent having ten simulations (search budget) per move. The bar around the lines shows the 95% confidence interval of the score averages.

To see the effect of the training times of the self-play agent on its performance, we tested this agent while training. This was again done against the rule agent with 10 000 rounds per test, resulting in a standard error of 1.3. The result of these tests can be seen in Figure 8. We found that at the start of the training, the score increased very fast. After this, the score levels off and even decreases.

5 Discussion

The number of simulations that are done is the number of nodes that are added to the search tree using the 5 steps of SO-ISMCTS. This can be seen as the search budget. We have seen that more of these simulations increase the score against the rule agents in a much more time-efficient manner than increasing the number of random rollouts. This can be explained by the fact that each node that is added with an extra simulation will also do extra random rollouts. This rollout will, however, be in an unexplored node and a different determinization, thus providing more new information.

We also found that more simulations allow for more exploration to be done, although the optimal exploration rate seems to stay around a C value of 200. This is expected as extra simulation allows for more time to find interesting nodes, thus reducing the need to exploit the current best. Although an optimal upper confidence bound C value of 200 seems high compared to optima found in other situations, the scoring system of Klaverjas needs to be taken into account. In Klaverjas, the outcome of a game is not just 1, 0 or -1 for a win, draw or loss but a score that can go up to 300. The left side of the upper confidence bound formula uses the score and is thus much larger than the right side which uses the visit count. Thus the C value needs to be large to compensate for this. We can undo this meaningless increase of the C value by normalising it. This is done by dividing the C value of 200 with a typical maximum score of 162. This results in a C value of 1.25. This comes close to the typical default C value of 1.41. Thus supporting the fact that 1.41 is a good default value.

We also found that with low amounts of simulations, the rollout agent performs the worst, followed by the self-play agent and then the human-data agent. At high amounts of simulation, the random rollouts outperform the neural networks. We think this is the case because search, in this case, random search, is less effective when faced with large search trees compared to estimates of neural networks. This is because, in larger trees, leaf nodes are less correlated to the root node. Because the probability that a leaf node is a node that will eventually be reached is smaller. More simulations would result in smaller trees because more simulations result in deeper search trees. Thus reaching further into to total tree, making the search tree left for the random rollouts closer to the end and thus smaller. This finding that, search is less effective when faced with large search trees compared to estimates of neural networks, supports the finding that hard-to-search (large search tree) instances of Klaverjas are the instances on which a machine learning approach (neural network) classified correctly [vRTV18].

In terms of performance, we found that all three agents were able to beat the rule agent with a large score difference. Especially, our rollout agent performed beyond our expectations. We thought it might be able to score a similar amount of points as the random forest, but it turned out it could do far better than that, reaching a score of 61.5, even doing this in a time-efficient manner. This score of 61.5 is very high, especially when compared to the random forest agent. This suggests that our agent is a large step forward compared to the previous best agent, the random forest agent. It might also perform well against some better opponents, such as humans.

We did not expect the rollout agent to perform so much better than the self-play agent and the human-data agent. We knew that finding the right setup for training a neural network using reinforcement learning could be tricky, but we did expect it to beat one random rollout if it had enough simulations and exploration.

The fact that the human-data agent outperformed the self-play agent is also interesting and surprising. This is surprising because AlphaZero uses pure self-play while AlphaGo also uses human-play data but, AlphaZero is stronger than AlphaGo. Furthermore, we can learn that the limited performance of our self-play agent was not caused by reaching the maximum performance of our state representation or neural network. It was caused by the quality of the self-play data since the training and the neural network were the same. To improve the quality of the data we tried more exploration in the search, more simulations, regularization in the neural networks, added randomness when doing moves, bigger neural networks, longer training and using data from rollout agents playing each other. However, these things did not increase performance by much or at all. For the data to be of high quality it needs to have a large variation of game states and the games must be played by as strong as possible players so it has the most accurate y value. The reason our self-play approach did not perform as well as the human-data approach is that we were unable to make both of these things better than in the human-data approach. Since when we added randomness to the moves the quality of the moves got lower.

Our findings imply that search is still very important when solving imperfect information games. And although the results from the neural network value function instead of the rollouts are promising, they were unable to live up to the expectations.

6 Limitations and future work

We will now list some limitations of our work and suggest ways to continue in future work.

- To see how our agents perform, it is not enough to only test them against a simple rule-based agent. It is a good indicator of performance, but beating it with a large score difference does not mean that it is a strong agent. In future work, this can be improved by not only testing against the rule agent but also against humans or other strong opponents.

- In our testing, we did not incorporate the bidding process that is part of Klaverjas. This is, however, not an insignificant part of the game, as being able to pick a trump suit at the right time will increase your chances of winning.

- In Klaverjas, knowing the strategy of your opponent or even teammate is valuable information. If you know in what situations a player would play a move, you can gather information about cards they have or do not have. This information can then be used to optimise your strategy. AlphaZero does take this into account by using a policy network. We however were unable to implement this as our state representation does not take the current determinization into account. Making a network policy network not feasible. In future work policy network or another technique can be used to solve this issue. By doing this the search can be directed to actions the opponent is most likely to play. Another way this issue can be avoided is by implementing an upgraded version of SO-ISMCTS called Multiple Observers Information Set Monte Carlo Tree Search [CPW12]. This technique does not only search over a tree from the player's perspective but also trees from the perspective of the other players. Another variant of ISMCTS that takes into account the strategy of the other players is Semi-Determinized Monte Carlo Tree Search [BK17]. This approach utilises a predictive model of the unobservable portion of the opponent's actions.

- After we have done MCTS, the search tree is thrown away. Although at the next turn, only a small portion of this tree could be reused, it would still increase the speed of finding a move.

- As our self-play agent did not reach the same performance as the human-data agent, more ways can be tried to improve it. Hyperparameters can be further optimised, new training techniques can be implemented, or longer training times can be tested.

- In future work some completely other techniques can be tried to create an even stronger Klaverjas agent. Let's take a look at some of the most promising techniques. First, the most promising technique is counterfactual regret. This technique has proven to be highly effective in imperfect information games, such as poker [ZJBP07, BBJT17]. Counterfactual regret tries to minimise regret by iterating over an abstracted game tree. With this, it can find a Nash equilibrium for very large game instances. Another technique is the use of perfect information Monte Carlo. This has proven successful in the games Stratego and DarkHex [BCK23].

7 Conclusion

We have developed a Klaverjas agent using SO-ISMCTS as the search algorithm. We combined this with three different state value estimators, i.e., a random rollout approach, a neural network approach trained on human-generated data and a neural network approach trained on self-play data. The agents were tested against a rule-based agent. All agents were able to beat this agent with the random rollouts agent performing the best and the self-play agent performing the worst.

We have shown that the techniques used in AlphaZero can successfully be applied to imperfect information games. We found that applying these techniques results in a good Klaverjas agent that can outperform a rule-based agent. It does this better than a previously developed agent. We also found that SO-ISMCTS with random rollouts was very successful while using neural networks is less effective. Furthermore, we found that neural networks are better than random rollouts at predicting state values in states far from the end. While in states close to the end random rollouts are better. With these findings, progress is made in developing a good Klaverjas agent. Progress is also made in the development of techniques used for solving imperfect information games. These techniques can in turn be used for solving real-world problems.

References

[BBJT17] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Commun. ACM*, 60(11):81–88, 2017.

[BBLG20] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[BCK23] Jannis Blüml, Johannes Czech, and Kristian Kersting. Alphaze: Alphazero-like baselines for imperfect information games are surprisingly strong. *Frontiers in artificial intelligence*, 6:1014561–1014561, 2023.

[BK17] Moshe Bitan and Sarit Kraus. Combining prediction of human decisions with ISMCTS in imperfect information games. *CoRR*, abs/1709.09451, 2017.

[BLFS09] Michael Buro, Jeffrey Richard Long, Timothy Furtak, and Nathan R. Sturtevant. Improving state evaluation, inference, and search in trick-based card games. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1407–1413, 2009.

[BLGS18] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. *CoRR*, abs/1811.00164, 2018.

[Bre01] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[CJH02] Murray Campbell, A. Joseph Hoane Jr., and Feng-Hsiung Hsu. Deep blue. *Artif. Intell.*, 134(1-2):57–83, 2002.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[CPW12] Peter I. Cowling, Edward Jack Powley, and Daniel Whitehouse. Information set monte carlo tree search. *IEEE Trans. Comput. Intell. AI Games*, 4(2):120–143, 2012.

[GBLS12] Richard G. Gibson, Neil Burch, Marc Lanctot, and Duane Szafron. Efficient monte carlo counterfactual regret minimization in games with many player actions. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1889–1897, 2012.

[Gin01] Matthew L. Ginsberg. GIB: imperfect information in a computationally challenging game. *J. Artif. Intell. Res.*, 14:303–358, 2001.

[Hor22] Lennard Hordijk. How to create an agent for the game of klaverjas using random forests. Bachelor's thesis, LIACS, Leiden University, 2022.

[HS16] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, abs/1603.01121, 2016.

[JLD$^+$19] Qiqi Jiang, Kuangzheng Li, Boyao Du, Hao Chen, and Hai Fang. Deltadou: Expert-level doudizhu AI through self-play. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1265–1271. ijcai.org, 2019.

[Kon] Peter Koning. klaver.live. Live & Online klaverjassen met vrienden. 2023.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[LSBF10] Jeffrey Richard Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010.* AAAI Press, 2010.

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533, 2015.

[Ros11] Christopher D. Rosin. Multi-armed bandits with episode context. *Ann. Math. Artif. Intell.*, 61(3):203–230, 2011.

[SHM$^+$16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[SHS$^+$17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[SMSM99] Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1057–1063. The MIT Press, 1999.

[SSS$^+$17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nat.*, 550(7676):354–359, 2017.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[vRTV18] Jan N. van Rijn, Frank W. Takes, and Jonathan K. Vis. Computing and predicting winning hands in the trick-taking game of klaverjas. In Martin Atzmueller and Wouter Duivesteijn, editors, *Artificial Intelligence - 30th Benelux Conference, BNAIC 2018, 's-Hertogenbosch, The Netherlands, November 8-9, 2018, Revised Selected Papers*, volume 1021 of *Communications in Computer and Information Science*, pages 106–120. Springer, 2018.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Mach. Learn.*, 8:279–292, 1992.

[ZJBP07] Martin Zinkevich, Michael Johanson, Michael H. Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In John C. Platt, Daphne Koller, Yoram Singer, and Sam T. Roweis, editors, *Advances in Neural Information Processing Systems 20, Proceedings of the Twenty-First Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 3-6, 2007*, pages 1729–1736. Curran Associates, Inc., 2007.

Appendix

Link to the code base: `https://github.com/Bart-Aaldering/Thesis`

Usage of ChatGPT

In the writing of this thesis, ChatGPT was used in the following ways.

- ChatGPT was used to create a better understanding of certain topics by asking it to explain them. If I did not find the answers useful or believable I looked for the information I needed somewhere else. I used the information I learned to write about the topics.
- I also used ChatGPT to find flaws in the text I wrote or ideas I had. For example flaws in, grammar, the accuracy of facts, and completeness of a topic or flow. I asked it what it thought of sentences or paragraphs I had written, and used the feedback it gave to correct things where I saw fit.
- I used ChatGPT to rewrite drafts of paragraphs I made in Dutch to English. These formed the first draft of a paragraph. In many iterations, these paragraphs were then adjusted to fit in with the changing text around them and convey the meaning I intended using my own words. This way of using ChatGPT was only used on a few occasions at the start of the writing process.

Although ChatGPT proved useful in some areas, it was not used in the vast majority of the writing process.