Universiteit Leiden

Opleiding Informatica

Using probabilities to enhance Monte Carlo search in the Dutch card game Klaverjas

Name:	Cedric Hoogenboom
Date:	17–01–2017
1st Supervisor:	Walter Kosters
2nd supervisor:	Hendrik Jan Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Using probabilities to enhance Monte Carlo search in the Dutch card game Klaverjas

Cedric Hoogenboom

Abstract

KLAVERJAS is one of the most popular card games in the Netherlands. It is a trick-taking card game, resembling the French game *Belote* and the German *Jass*. It is played with four players, of which the opposite players are team mates. In a trick, the highest (trump) card is the winner. The goal for a team is to obtain more points in a game than the other team. A full match usually consists of 16 games.

In this thesis a Monte Carlo strategy will be proposed for KLAVERJAS, and several methods of improving the strategy will be elaborated. The first improvement focuses on the playouts for the Monte Carlo player. Pure Monte Carlo normally does these playouts randomly, here a *semi-random* player is created to rule out some moves that would be considered “dumb”. More improvements focus on the distribution of the cards in the playouts. The pure Monte Carlo player redistributes these cards randomly, but in a way no player gets a card he cannot have (otherwise he would have cheated according to game rules). As an improvement these cards can be distributed based on a probability distribution.

Two of these probability distributions will be presented: one for the probability a player has to receive a certain suit and one for the probability a player has to receive a certain trump card. These probabilities are supposed to make use of more knowledge gained throughout the progress of a game.

The different strategies proposed in this thesis were tested against each other. From this experiment can be concluded that the Monte Carlo player with probability distribution performed best of the strategies described in this thesis.

Contents

1 Introduction

KLAVERJAS is a traditional Dutch card game, played mostly in the Netherlands. The name dates back to around 1800 and 1895 [1] and is composed of the Dutch words "*Klaver*" and "*Jas*". "*Klaver*" translates to "Clubs", and this is the trump suit each first game in a full match. "*Jas*" is another word for the Jack of trumps, which is the highest card in the game.

The game is played throughout the Netherlands, although the rules may differ among regions. It is played often in competitions, many clubs facilitate these competitions. The "Nederlandse Klaverjas Uni"[2] has 321 clubs registered, and facilitates Dutch Championships in KLAVERJAS.

For this thesis we research the effectiveness of a Monte Carlo strategy for KLAVERJAS, and explore ways to enhance this technique. A computer program was created in which the game can be played by and against human players and against different strategies explained in this thesis. These strategies include a random player, which simply does a random (legal) move. This player was also enhanced by ruling out some moves that are generally are not considered as smart moves, this will be called the semi-random player. These two players will both be used by different types of Monte Carlo players.

The idea of a Monte Carlo strategy in Computer Science is for each possible move to play a number of simulations, and choose the move that had the highest average score. For KLAVERAS we created Monte Carlo players based on the "pure" random player, and on the semi-random player.

We also try to utilize as much knowledge gained from the progress of the game as possible, much like a skilled human player does. Throughout the game it may become clear that a player has many, or few, cards of a certain suit. We provide a way in which the Monte Carlo player can take this type of information into account.

The game is explained in Section 2, and in Section 3 the implementation of the game is described. In Section 4 we propose three strategies for the computer players to determine whether they should play or pass on a hand of cards. The pure Monte Carlo player for the game itself is described in Section 5, and in Section 6 the different methods of enhancing the pure Monte Carlo player are elaborated. Finally, in Section 7 the different strategies are tested against each other and against human players, and in Section 8 we arrive at a conclusion.

This thesis is a bachelor thesis written for the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University. The supervisors for this thesis are Walter Kosters and Hendrik Jan Hoogeboom.

1.1 Related work

The main strategy applied in this thesis is Monte Carlo search. Monte Carlo (or the extended Monte Carlo Tree Search) has been applied to many games and problems in Computer Science. KLAVERJAS is, unlike games like Chess or Go, an imperfect information game. Not all knowledge is known, since players do not know the other players' cards. More information on Monte Carlo Tree Search for imperfect information games can be found in [3].

A three-player trick-taking card game resembling KLAVERJAS to which a Monte Carlo player also has been adapted is the German game SKAT. This recursive Monte Carlo algorithm is described in [4].

Although not much research has been done on KLAVERJAS, for the resembling trick-taking card game BRIDGE there has even been a World Computer-Bridge Championship since 1997 [5]. The winner of the 2016 world championship was the program *Wbridge5*, the results can be seen in [6].

Ever since the IBM chess-computer DEEP BLUE defeated the world champion Kasparov in 1997 [7], the world champions of many games have been beaten by computers. For the game Go this has always remained a large challenge, due to its large search space. Until recently all the computer Go players were considered very weak, and none had been able to beat a professional player. This changed in 2016, when Google's ALPHAGO beat Lee Sedol, the top Go player in the world over the past decade [8]. ALPHAGO relied heavily on computer power and two neural networks, but it also utilizes Monte Carlo Tree Search for determining its moves.

2 The Game

In this chapter we will explain the basic rules of KLAVERJAS. The game is played in many different variants, which will be displayed in this chapter. We will mainly focus on the *Rotterdam* variant with classical bidding system.

2.1 Explanation

The Dutch game KLAVERJAS is a trick-taking card game played by $P = 4$ players. Each player has a hand consisting of 8 cards, totalling up to 32 cards in the game. Only the cards 7 and higher from the standard deck are used to play (also called the *Piquet*-deck [9]). A game is finished when all players have played all of their 8 cards. The players sitting opposite from each other are team mates, and the goal is for each game to gather more points than the opponent.

Each game begins by determining which suit will be trump; this will be explained in Section 2.2. After the trump suit is determined the elder hand (the player sitting next to the dealer, clockwise) begins by playing a card. Every next player has to follow suit. If a player cannot follow suit he[1] must play a trump card, if he also cannot play a trump card any card is possible. If a trump card was already played this trick each next trump must be higher than the one previously played, but if a player does not have a higher trump he must play a lower one. When he also cannot play a lower trump card, any card is possible. The player who played the highest card wins the trick and takes the four cards played, and trump cards always beat non-trump cards. The order and points of the cards are shown in Table 2.1. The winner of this trick begins the next trick by playing a card, and this continues until all cards are played. The last trick is worth 10 additional points. Then the points are counted and written down, and the next hand can begin. Normally 16 games are played in a full match of KLAVERJAS. The full rules of the game can be found at [10], and a full for learning KLAVERJAS can be found at [11].

[1]Throughout the thesis the word *he* is used to indicate *he/she*

Non-trump	Points	Trump	Points
Ace	11	Jack (“jas”)	20
10	10	9 (“nel”)	14
King	4	Ace	11
Queen	3	10	10
Jack	2	King	4
9	0	Queen	3
8	0	8	0
7	0	7	0

Table 1: Order and points of the cards

2.2 Determining the trump suit

Besides the 32 playing cards we have the other 20 cards, consisting of the cards 6 and lower of the standard deck. These cards are also shuffled at the beginning of the game, and will be used to determine the trump suit. Before each game a card will be shown of this deck, representing a potential trump suit. The elder hand will start by saying if he will play or pass on this potential trump suit. If he plays the game begins and the elder hand can play a card. If he passes the player sitting next to him will have to play or pass. If all players have passed on a suit a new one will be shown from the deck. If again all players pass, the elder hand will have to play on the next suit in the deck, regardless of whether or not it has been shown already.

When a player plays on a suit it is his objective to gather more points than his opponent. If the playing team has fewer points than the opponent all of the points will go to the opponent. In this case he *fails*[2] and the playing team gets 0 points.

2.3 Roem

Another important aspect of KLAVERJAS is the concept of *“roem”*[3]. *Roem* occurs for example when three cards are played in consecutive order in a trick, which gives additional points to the winner of that trick. There are five different cases of *roem*. The *roem* is also written down, next to the normal points, and is added to the points of a team at the end of the game.

Each trick, four cards are played by the four different players. If three of these are in the same suit in consecutive order, the winner of the trick gets 20 points *roem*, see Figure 1. The consecutive order of these cards is different from the rank of cards (or trumps) in the *Klaverjas*-game, namely *7, 8, 9, 10, J, Q, K, A*.

When all four cards in a trick are same-suited cards and in consecutive order even more points are awarded to the team that wins the trick: not just 20, but 50 additional *roem*-points are added.

When the King and Queen of the trump suit are both played in a trick we speak of *Marriage*[4], and 20 additional points are awarded to the winner of the trick. This can also be in combination with a 3- or 4-card *roem*. This way up to

[2]The original Dutch term for this is *nat gaan*.
[3]The Dutch word *roem* roughly translates to *fame*.
[4]The original Dutch term is *Stuk*.

70 *roem*-points can be awarded when for example Jack, King, Queen and Ace of the trump suit are played in a trick.

When in a trick all 4 cards are of the same type but different suit, 100 extra points are awarded to the trick winner. This type of *roem* almost never occurs, usually of the time a player can prevent it by playing a different card.

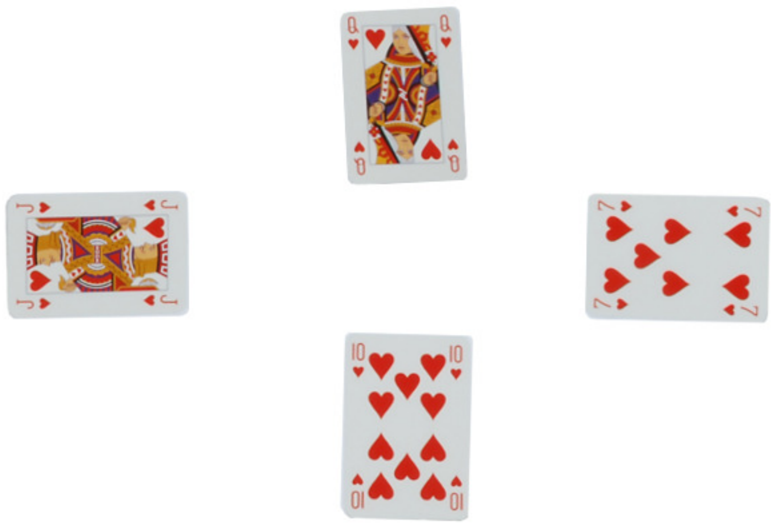


Figure 1: Example of 20 points of *roem*.

2.4 Variants

KLAVERJAS is a popular game in the Netherlands, but even there the rules differ in some regions. In this thesis we mainly focus on the *Rotterdam* rules, but in the program the *Amsterdam* rules are also supported. The *Rotterdam* rules are mainly played in the provinces Zuid-Holland, Zeeland, Noord-Brabant, West-Vlaanderen and Oost-Vlaanderen, the *Amsterdam* rules dominate in the rest of the Netherlands and Belgium. In this section we will cover some variants of the game played throughout the country.

2.4.1 Amsterdam

With the *Rotterdam* rules, players must always play a higher trump card when possible. When the game is played with *Amsterdam* rules, this is not necessary when the trick is to be be won by a player's team mate. For example, when a player cannot follow suit he has to play a trump card. When his team mate also cannot follow suit, according to the *Rotterdam* rules he has to play a higher trump. According to the *Amsterdam* rules, this is not necessary and this player can play any card. Due to this rather small change in the game rule tactics might change for the players.

2.4.2 Variants in determining the trump suit

The rules for determining the trump suit explained in Section 2.2 are known as classical bidding system. There are some minor variants on this system, and some major ones.

The minor variants focus on when suits are turned from the stack of remaining cards (6 and lower from the deck), and all players passed on both suits. According to the classical rules, the elder hand must now play on the next suit to be turned from the stack. There is a variant in which all cards need to be dealt again. This is mostly used in competitive games. Another variant lets the elder hand pick a suit after all players have passed.

In the *Free choice* variant the elder hand can choose a trump suit, or pass. When a player passes, the next player can pick a suit or pass. When all players have passed, the elder hand must pick a suit.

There is also a variant called *Utrecht*. In this variant players cannot play or pass, but every game the elder hand has to pick a suit. This way all players play an equal amount of games, but players are sometimes forced to play on hands they would normally pass on every suit.

The variant that differs the most from the classical rules is the *Leiden* bidding system. Here the bidding system of BRIDGE has been adapted to the KLAVERJAS rules. Because this changes the rules much more drastically than the other variants, we will not go into detail regarding this variant, although it is becoming increasingly popular among KLAVERJAS players.

2.5 Common tactics

Because the game of KLAVERJAS is played in two teams who cannot communicate about their cards, a large amount of common tactics are known among players. Players should know and follow these rules so team mates can rely on each other to make sure they do not unnecessarily waste points. In this section some of these tactics are presented. One of our goals is to see if the program can teach itself (some of) these tactics.

2.5.1 Tactics in playing

When a player plays[5] a game, he is basically saying he has the best hand. If he and his team mate do not get more than half of the points, they fail and get 0 points. Therefore it is important for players to play together and know what they can expect from each other in order to win the game.

When a player plays, most of the time he wants to get all of the trump cards of his opponents out of the game. Often he will have the highest trump, the Jack of trumps, and most of the time he will play this card to get as many trumps out of the game as possible. Because this is the highest card in the game he will always win this trick, and he can lead the next trick. Then he can choose to continue playing trump cards to get more of his opponents' out of the game or play other suits.

If a player has the elder hand and his team mate has chosen to play in the phase of determining trump suit, his team mate will most likely have a good hand and it is wise to have him lead a card as soon as possible. The team mate

[5]Throughout the thesis the player that *plays* should be interpreted in the sense of Section 2.2.

can then choose to get trump cards out of the game as mentioned earlier. The player could assume that his team mate probably has the Jack of trumps, so he can play a trump card to let his team mate win this trick by playing his Jack. This way it can be smart for the player to play a card that is sensitive for *roem*, like the Queen of trumps. If his team mate has the Jack of trumps, the Queen and Jack will be played and if the opponent must play a 10 or King the playing team will have extra points for *roem*, or the opponent may be tempted to play his 9 to avoid this.

When a player plays the game and he does not have the highest card, he probably also will want to get the Jack of trumps out of the game as quick as possible. This could prevent the opponent from winning points by playing a high trump that the player could otherwise have won, for example by an Ace. In this case the player can try to get the Jack out of the game by playing a card high enough that the opponent will have to play his Jack to follow the rule of always playing a higher trump than previously played in the trick. Players have to balance playing a high enough card so that the Jack will be played and not giving too many points to the opponent, given they will most likely win the trick.

2.5.2 Signalling

Team mates are not allowed to talk about their cards, but it can be very useful to know for example what suit your team mate has an Ace of. There are mechanisms players use to signal their mate which suits they want them to play or not to play using the cards they play. Here some of these mechanisms are explained. Most of these can only be used by a player if his team mate is about to win the trick.

When a player's team mate is about to win the trick and he has to play a card, but cannot follow suit, he normally will play a card that adds points (for example playing a 10 of another suit) to the trick. Instead of adding points he could also throw a low card (7, 8 or 9), with which he signals his mate that he has the highest card of that suit. His team mate can then play the suit his team mate indicated, who can then win that trick as well. A player can also indicate which suit he does not want to be played. This can be done in the same way as earlier, but instead of a low card a high card (Jack, Queen, King) of a suit signals a team mate not to play that suit.

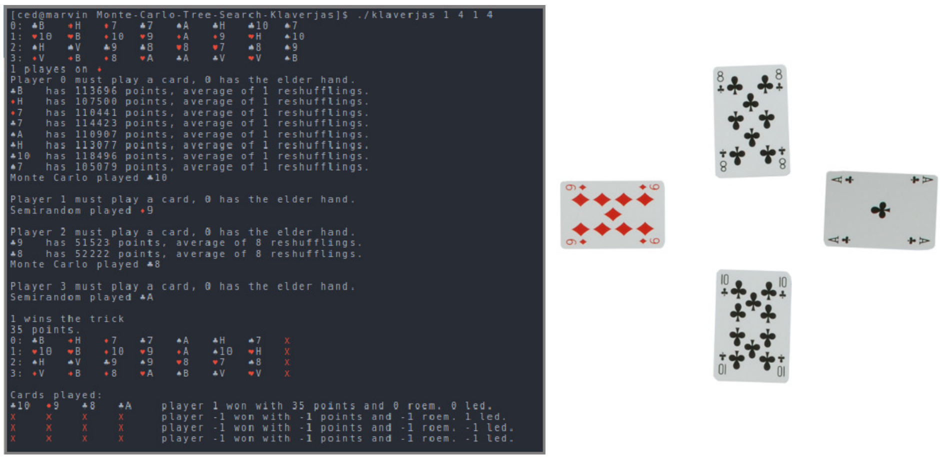
Another way of signalling a team mate which suit to play is by using opposite suit signals. The idea of this type is similar, it is also used when the trick is about to be won by a player's team mate. The difference with the type of signalling mentioned earlier is that instead of playing a suit that the player wants to be played later, the player plays the opposite suit, of the same colour. For example, when a player wants Diamonds to be played, he plays a low card of Hearts. This way players can signal but keep cards of that suit.

The last type of signalling we will cover is the lead signal. This is a way the player determining trump and playing the first card can use to signal his team mate he has a weak trump hand. If he has the Jack of trumps but little else, he comes out with a low card of another suit. If he does not have the Jack of trumps, he leads with an Ace of another suit. His mate then knows whether or not it is safe to play trump cards.

Teams should agree in advance what types of signalling they will use throughout the match.

3 Implementation

For the implementation of Klaverjas a C++ program was created in which the game can be played. Using the program the game can be played by human players or one of the different algorithms. In this chapter some of the implementation choices and datastructures are elaborated. A screenshot of the program with the corresponding cards is shown in Figure 2. The source code of the program can be found on GitHub [12].



```
[ced@marvin Monte-Carlo-Tree-Search-Klaverjas]$ ./klaverjas 1 4 1 4
0: ♣B   ♦H   ♦7   ♣7   ♠A   ♣H   ♣10  ♠7
1: ♥10  ♥B   ♦10  ♥9   ♦A   ♦9   ♥H   ♠10
2: ♠H   ♠V   ♣9   ♣8   ♥8   ♥7   ♠8   ♠9
3: ♦V   ♦B   ♦8   ♥A   ♣A   ♣V   ♥V   ♠B
1 playes on ♦
Player 0 must play a card, 0 has the elder hand.
♣B   has 113696 points, average of 1 reshufflings.
♦H   has 107500 points, average of 1 reshufflings.
♦7   has 110441 points, average of 1 reshufflings.
♣7   has 114423 points, average of 1 reshufflings.
♠A   has 110907 points, average of 1 reshufflings.
♣H   has 113077 points, average of 1 reshufflings.
♣10  has 118496 points, average of 1 reshufflings.
♠7   has 105079 points, average of 1 reshufflings.
Monte Carlo played ♣10

Player 1 must play a card, 0 has the elder hand.
Semirandom played ♦9

Player 2 must play a card, 0 has the elder hand.
♣9   has 51523 points, average of 8 reshufflings.
♣8   has 52222 points, average of 8 reshufflings.
Monte Carlo played ♣8

Player 3 must play a card, 0 has the elder hand.
Semirandom played ♣A

1 wins the trick
35 points.
0: ♣B   ♦H   ♦7   ♣7   ♠A   ♣H   ♠7   X
1: ♥10  ♥B   ♦10  ♥9   ♦A   ♠10  ♥H   X
2: ♠H   ♠V   ♣9   ♠9   ♥8   ♥7   ♠8   X
3: ♦V   ♦B   ♦8   ♥A   ♠B   ♣V   ♥V   X

Cards played:
♣10  ♦9   ♣8   ♣A     player 1 won with 35 points and 0 roem. 0 led.
X    X    X    X      player -1 won with -1 points and -1 roem. 1 led.
X    X    X    X      player -1 won with -1 points and -1 roem. -1 led.
X    X    X    X      player -1 won with -1 points and -1 roem. -1 led.
```

Figure 2: Example of a game state in the computer program, with the corresponding cards played as illustration.

3.1 The cards

The cards in the game are represented by integers in base 8. The suits themselves are also integers, in a way that a card's suit can be deducted by simply taking the first digit. For example, if the card's number is 23, the suit is 2 (Clubs) and the card is 3 (King). The order of the numbers of the cards are represented in Table 2 and the suits is shown in Table 3.

The order of the cards is the same as the order of trump cards. This is done so that the program can easily check if players played a higher trump than previously played.

The suits have no specific order, other than that red and black suits alternate each other for visibility.

The cards and the suits are represented in a simple `enum`-structure, which is where they get their numbers. For printing cards and numbers the C++ operator `<<` was overloaded to be able to print the `enum`-structures for cards and suits. This allows us to use `cout` to print cards and have them look like the corresponding symbols. For the Hearts and Diamonds suits, an escape sequence was used to make them show red if the terminal supports colours.

Number	Card
0	7
1	8
2	Q
3	K
4	10
5	A
6	9
7	J

Table 2: Order and number of cards in the implementation.

Number	Suit
0	♠
1	♡
2	♣
3	♢

Table 3: Order and number of suits in the implementation.

3.2 Cards played

Cards played by the players are kept in a two-dimensional array. With the standard KLAVERJAS amounts of players and cards (4 players, 8 cards per player), the structure of the array is as follows.

The first 8 rows each represent a played trick. The last (9th) row is reserved for the final scores per team, who played this game, and which suit is trump. The last (7th) value in this row is a boolean, which is 1 if the playing team were forced to play (because all the players had passed on two suits) and 0 if they played the game on this suit by choice. In each of the first 8 rows the first 4 values are cards played by each player that trick. The 5th value is for which player leads that trick and played a card first, the 6th value is which player won the trick. The 7th value is the number of points scored in this trick and the last value is the amount of *roem* in the trick. These last two values are both to be awarded to the winner of the trick.

A example of a played game is displayed in Table 4. In this representation the wind directions represent the cards played by the different players. The first row of the table is just there for clarification and not present in the actual array. The dotted lines indicate the difference between the cards played and the other variables.

3.3 Checking for roem

As explained in Section 2.3, an important aspect of the game is the concept of *roem*. The program checks for cards in consecutive order every trick as follows.

To get all the extra points the function begins by checking if all 4 cards have the same value. As explained in Section 2.3 this is a very rare situation but if it occurs is worth 100 extra points for the winner. Next the function puts the cards in a different order. Normally the cards are in order of trump cards, but for *roem* they must be in the order described in Section 2.3. The procedure for checking

South	West	North	East	Leads	Won	Points	*Roem*
♢10	♣10	♢K	♢J	0	1	26	0
♠Q	♠J	♠A	♠K	1	2	20	50
♡A	♡9	♡Q	♡7	2	0	14	0
♣J	♣Q	♣K	♣7	0	0	27	40
♢8	♠8	♣8	♢9	0	2	0	0
♡10	♠7	♡J	♡8	2	0	12	0
♢A	♠9	♣9	♢Q	0	2	28	0
♢7	♠10	♡K	♣A	2	3	35	0
191	61	191	61	0	♣	0	

Table 4: Example of the two-dimensional array containing the history of a game.

roem is basically putting the cards in that order, using InsertionSort to sort them, and then checking whether they are in consecutive order. If three of them are in order 20 points are added, if four are in order 50 points are added. After this check has completed, the algorithm also checks if *Marriage* occurs, and if so 20 additional points are added and the total amount of *roem* is returned. Because the cards were already in *roem*-order and sorted this check is trivial, it just checks if the Queen of trumps is in the first three cards, and if so it checks if the next card is the King of trumps.

3.4 Reading files

In order to perform experiments and compare different algorithms in certain situations, we have to be able to play the game with a predefined configuration of the cards. In order to define the configuration of cards the functionality to read files with predefined situations has been built. To this end a file format was defined, typically with the `.kvj` extension. These files must contain the distribution of the cards and the types of the players (e.g., Monte Carlo, Random or Human players), and can optionally contain the cards played until a certain point so that the game can continue from that point. The structure of these files is explained in this section.

The first line of the file contains the types of players. This can for example be 2 0 2 0 for a team of Monte Carlo players against a team of Human players. The next four lines are used for the distribution of the cards This is done in a way that the second line of the file represents the 8 cards for player 0 separated by spaces, the third line represents the cards for player 1, and this continues onto the fifth line. The sixth line can either contain the trump suit, who plays and who leads, or just a -1 in order for the trump suit and who plays to be defined at the beginning of the game. If the game is to be started at the beginning, this would be the end of the file. In case the game should continue at a certain point, line 7 contains the current trick the game is in. The next lines are used to define tricks, by denoting what cards players played in that trick. This is done by first indicating the cards players played, also separated by spaces. The amount of lines following line 7 must be equal to the value on line 7. An example of a `.kvj` file with one trick played is as follows:

```
5 0 1 1
23  12  34 37  0  6  33  36
25  35  14 2  7  16  3  4
11  30  17  20  32  10  27  31
22  1  24 21  15  5  13  26
3 0 0
1
37 35 32 21
```

In order to create these files a Python script was written, which takes parts of the output of the program as input to create the file. This way users will not have to manually input and remember the numbers for cards defined in Section 3.1.

4 Playing or passing

In the phase of determining which suit will be trump, players can either play or pass on a certain suit. If they play and not get more than half of all the points in the game, they get 0 points and the opponent will get all of their *roem* points. Because a team might get 0 points if they misjudge their cards, determining if they play at the beginning of a game is a very important aspect of KLAVERJAS. In this section some of the different tactics the program can use in order to determine to play or pass are explained.

4.1 Using points

In the game of BRIDGE it is common to use points to give some value to a hand of cards, and using this value to determine how a player should bid. A common method for BRIDGE is called *High Card Point* (HCP), in which the four highest cards are given points. In BRIDGE the four highest cards are Ace, King, Queen and Jack, and in HCP they are given respectively 4, 3, 2 and 1 points. This method was first published by Milton Work in 1929 [13]. This method is adapted to the game of KLAVERJAS, and used as tactic for playing or passing.

BRIDGE is played with a full deck of cards, and KLAVERJAS with just the cards 7 and higher. Therefore we decided not to use four cards to give value to a hand, but just the highest three. This way the Ace gets 3 points, the 10 gets 2 points and the King gets 1 point. Because BRIDGE is played with all of the cards, the chances are high that a suit can be played more than once without someone playing a trump card. In KLAVERJAS these chances are much lower, because when a suit is played twice without a player playing a trump card, this suit must have been divided equally among the players. Therefore the chances are also lower that a player can use a 10 to win a trick, thus we decided that 10 cards only get points when the player has another card of that suit. This way the player does not have to play his 10 if he knows the trick will be won by the opponent's Ace or trump card. The HCP method in BRIDGE does not take trump cards into account. In KLAVERJAS a hand cannot be valued decently if trump cards are not taken into account, because they have such a big impact on the game. If a player has a good hand but not many trump cards, there is a big risk that his high cards are lost to the opponent by their trump cards. Therefore we decided to give an extra point to each trump card. Because the Jack and 9 of trumps are the highest trump cards and essential to winning a game, these cards are given respectively 5 and 4 additional extra points instead of the single extra point for trump cards.

The total amount of points for this system in the game is 39. One fourth of this is (rounded) 10, which is what is used as boundary for playing or passing. This can be tweaked up a bit, making the player somewhat more restrained. 10 points could be the Jack and 9 of trumps with one additional King or trump card, which can be considered a minimum hand with which half of the points can be scored.

This strategy for playing or passing is reasonably good, but far from perfect.

4.2 Monte Carlo

As with the game itself, a Monte Carlo strategy was also implemented for determining whether to play or pass. This way the algorithm can also determine whether it plays or passes without too much domain knowledge.

As with the implementation of the game the playing or passing algorithm is based on reshuffling the cards and playing random playouts. It simply divides the rest of the card randomly over the other players, sets the suit for which the player has to decide to play or pass as the trump suit and plays the rest of the game with all four players playing randomly. If the final score for this hand and trump suit is more than the score of the opponent, the algorithm decides to play.

4.3 Rule-based strategy

Finally a strategy was defined to mimic the way a human player would decide to play or pass. It looks at how many trumps the player has, if he has the highest or the second highest trump and how many aces the player has. In the following cases the algorithm will decide to play:

- If the player has the two highest trump cards, the Jack and the 9, the algorithm will always play. There might be some cases in which this may not be a smart move, but in almost all cases this can be considered a good hand.

- If the player has only the Jack, not the 9, and more than two other trump cards the algorithm plays. If he has the Jack, not the 9 and less than or equal to two other trumps the algorithm only plays if the player has at leas two Aces from the other suits.

- If the player has the 9 of trumps as only trump card the algorithm will never play. This is not considered a good hand, because if the opponent has the Jack the chances are high that the opponent will win from the 9 and the player will not have any trump cards left. If however the player has the 9 of trumps and more than three other trump cards the algorithm will play. If the player has two or less trump cards, the algorithm will also only play if the player has two or more Aces.

5 Pure Monte Carlo

The Monte Carlo technique is a technique widely used in Computer Science, specifically in Artificial Intelligence. The key idea is to find a reasonably good move by using random games as playout. The main advantages of this technique are that very little domain knowledge is needed and that the program can always run within a reasonable amount of time.

5.1 Explanation

The idea of the Monte Carlo technique is to use random playouts as a basis for determining what move to play in a game. In this section we will explain the basis of this technique, and then move to the implementation of KLAVERJAS.

The basis of an algorithm for playing a game is making decisions. At any time there can be n different possibilities, each resulting in a different outcome. There are algorithms which explore all (or many) of the possibilities and the results, building up the complete (or parts of) the game tree. Examples are MINIMAX which builds the complete game tree, or ALPHA-BETA PRUNING which skips redundant parts of the tree.

The pure Monte Carlo technique does not build this game tree. Instead, for all of the n possible moves, it plays m random games which all return an end result. The move with the best average result is then played. So for each move, a copy of the game is made and the move is played in this copy. Then the game continues in this copy with random moves for all players. This is repeated m times.

Because Monte Carlo is based on random playouts, moves that are generally very bad are also considered. If m is high enough, the idea is that these bad moves will not get good scores, and therefore will generally not be played. But, because it is based on randomness and chance, bad moves could be played. It therefore will often not find the best move, but generally will find a reasonably good one.

5.2 Reshuffling the cards

For Monte Carlo to be effective in a card game like KLAVERJAS, every random playout needs to be played with a different distribution of the cards. This way (if enough random playouts are performed) most of the possible situations will have been tried, and the move that performed best on average will be performed. To this end a method was devised that reshuffles the cards left in the deck. The cards left are all of the cards without the cards of the current player and the cards already played. It also takes into account that some players may not have a certain suit, because they could not follow that suit earlier in the game. By reshuffling the cards each random playout, hopefully most of the possible situations (depending on the size of m) will be explored and played out.

Throughout the game, players eventually may not be able to follow a suit played. In this case they must play a trump card, and if they do not have any trump cards left they may play any other card. This gives knowledge about a players hand to the other players, since they now know that he does not have any cards of that suit left. While reshuffling the cards, we have to take this information into account to make the Monte Carlo technique perform well. This puts a constraint on the distributions of the cards we create. To this end, before reshuffling the cards we create a matrix consisting of which player may not receive a certain suit. An example of such a matrix can be seen in Table 5.

	Player 0	Player 1	Player 2	Player 3
♠	`false`	`false`	`true`	`false`
♡	`false`	`true`	`true`	`false`
♣	`false`	`false`	`false`	`false`
♢	`true`	`true`	`false`	`true`

Table 5: Example of a matrix indicating which player may not receive which suit.

Besides knowledge gained from when players cannot follow suit, we also gain knowledge when a player cannot play a higher trump card than previously played that trick. The rules dictate that when a player plays a trump card, he must play a higher trump card than all of the trump cards played earlier that trick. If he does not have a higher trump card than the highest played, he must play a lower trump card. If he also does not have a lower one, he may play any card. If he does not play a trump card, this is already stored in the matrix mentioned earlier. But if he *does* follow suit, but does not play a higher trump card, this is not stored in that matrix. However, in every redistribution he may not receive a higher trump card. Therefor an additional matrix was introduced, indicating which players may not receive certain trump cards. An example of such a matrix can be found in Table 6. In this example you can see that Player 0 or 1 played a 10, and Players 2 and 3 played a lower trump card than that 10. Information about players that do not have any trump cards at all is not stored in this matrix, since this can already be found in the matrix for suits. It can be possible that in the example of Table 6 Player 1 did not follow trump at all, when player 0 led a card.

	7	8	Q	K	10	A	9	J
Player 0	`false`	`false`	`false`	`false`	`false`	`false`	`false`	`false`
Player 1	`false`	`false`	`false`	`false`	`false`	`false`	`false`	`false`
Player 2	`false`	`false`	`false`	`false`	`false`	`true`	`true`	`true`
Player 3	`false`	`false`	`false`	`false`	`false`	`true`	`true`	`true`

Table 6: Example of a matrix indicating which player may not receive a certain trump card.

Our method of redistributing the cards mostly relies on simply redistributing the cards randomly, and checking if it fits this matrix. If it does, a distribution is found. If it does not, the cards are once again randomly distributed until a suitable distribution is found. As an improvement to this method one optimisation was implemented, which applies when according to the matrix for suits only one player may receive a certain suit. In this case first all cards of that suit are given to that player, and then the rest of the cards are randomly divided.

5.3 Implementation in Klaverjas

The first step in creating a Monte Carlo player for KLAVERJAS was implementing the game, and creating a random player for it. The game was implemented by creating a C++ program in which the game can be played. With this basis created, multiple types of players could be created.

5.3.1 Random player

The random player simply generates the possible (legal) moves, and picks a random one from these. While this in itself is not very complicated, the function generating the possible and legal moves is somewhat more interesting.

This function takes as input, among others, the cards played already this trick, the current player's cards and who leads, and generates an array containing the possible moves and an integer determining the amount of possibilities. As briefly explained in Section 2.1, there are some rules determining which cards are possible and which are not. The order of possible cards (if the current player does plays the first card this trick) is as follows:

- Follow suit. If a player has the suit the elder hand led with, other players always have to follow this suit if they can.
- Higher trump card. If a player cannot follow suit, he must play a trump card. When a trump card is already played this trick, he must play a higher trump than the one played before. In the *Amsterdam* rules there is an exception to this rule when the highest trump played is from a player's team mate, but in the *Rotterdam* rules a player always has to play a higher trump. When a player leads with a trump card, all other players must also not only follow the trump suit but play a higher trump card if possible.
- Lower trump card. If the player cannot follow suit and does not have a higher trump than played earlier this trick, he must play a lower trump. In the *Amsterdam* rules there is also an exception to this rule if a player's team mate is about to win the trick.
- Any card. If a player cannot follow suit and does not have any trump cards, he may play any card.

The algorithm simply uses the cards previously played this trick to determine in which situation mentioned above the player is, and adds all possible cards to the array.

5.3.2 Monte Carlo player

Just like the random player, the Monte Carlo player also begins by listing all possible legal moves. But here, instead of just picking a random move the next steps are executed for each of the n possible moves:

A copy of the board[6] is made. A copy is also made of the current player's cards, and the rest of the cards are reshuffled (as mentioned in Section 5.2) among the other players. After the cards are dealt the move can be played in

[6]The board consists of the previously played cards each trick, who won which trick and how many points and *roem* were in a trick.

the copy, and a random game can be played by calling the function also used to start this game, but with all players being random players. The result of the game for the current player is added to a variable, and after m random games with this move the algorithm continues with the next possible move. The move which has the highest total points after m random games is returned and played. In case of a tie the first move the algorithm encountered is played.

5.4 Number of playouts

The number of playouts performed per possible move can be an important factor in a Monte Carlo strategy. If too few playouts are performed, the algorithm will most likely not do the best move possible. The outcome of the playouts could depend too much on moves by the opponent which would never happen in a real game. If too many playouts are performed, the algorithm will become very slow and unusable in most situations.

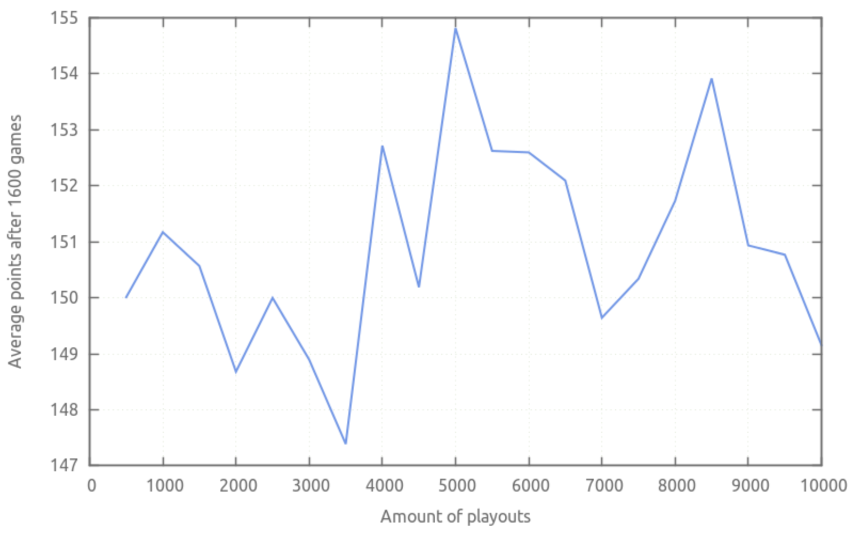


Figure 3: Results of different numbers of playouts for the Monte Carlo player.

Throughout the thesis, when not specified, the Monte Carlo player uses 1000 random playouts per possibility. In the program this can be adjusted and specified per player, to allow users to try different combinations of playouts per player and strategy. In Figure 3 the amount of playouts is displayed against the number of points gathered by the Monte Carlo player. For this experiment to be as pure as possible, the results are for a team of pure Monte Carlo players against a team of semi-random players (this player is explained in Section 6.1. The Monte Carlo players used in this experiment also use this semi-random player for the playouts). In this graph one can see that the results are reasonably stable.[7] It is also as expected that higher amounts of playouts (with most noteworthy 5000 and 8500) also have higher results. But as displayed in Figure 4, these

[7]This figure may seem unstable, which corresponds to the fact that the vertical axis begin at 147. In reality, these values are very close to each other.

differences in the results are not significant enough to justify the large increase in calculation time.

In this graph one can clearly see the increase in calculation time for higher amounts of playouts. The calculation time is taken for four games at a time, as these were performed in parallel. There appears to be a linear relation between the increase in playouts and the increase in calculation time, which is as expected.

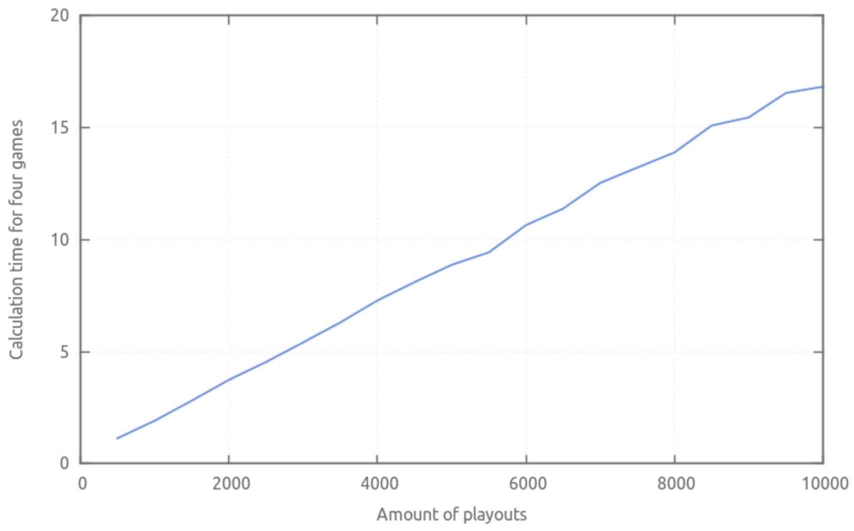


Figure 4: Amount of time taken per simulation of four games by the Monte Carlo player.

When comparing the results of Figure 3 and Figure 4 we arrived at the conclusion that using 1000 playouts appears to be a suitable choice. The results for this amount of playouts are somewhat less than the results of higher amounts of playouts, but the calculation time is significantly lower. This allows for a quicker play, but not at the expense of much lower results.

6 Enhancements on Monte Carlo

As mentioned in Section 6.1, the Monte Carlo player performed reasonably well, but there was much room for improvements. Most importantly, we think that the algorithm did not use all the knowledge of the progress of the game so far it could use. For example: when a player plays a game, he states that his cards are good enough for him and his team mate to win more than half of the points in the game. This means that a player could assume that if his opponent plays, he will probably have good cards. Another example is that when a trick is about to be won by a player's opponent, and he plays a card that adds *roem* points to this trick while the opponent still wins it, he will not likely have other cards of that suit.

6.1 Semi-random player

When the Monte Carlo player was first implemented, it played reasonably well but there was room for improvement. What was most peculiar about its playing style was that it would play cards very sensitive for *roem*. This can be a very good tactic if you or your team mate wins the trick, because you get extra points next to the points for the cards. But if your opponent wins the trick, it can be a very bad choice, because then you are giving them extra points which could help them win, or worse: make you fail the game. Therefore human KLAVERJAS players generally do not play this sensitive for *roem* unless they are almost certain the trick will not go to the opponent.

The reason the Monte Carlo player played this way lies in the fact that it is based on random playouts, and is based on the average score of many of these games. A game without *roem* has 162 points, and these points are divided among the two teams at the end of a game. The Monte Carlo player takes only the points his team has after the game into account, and takes the average best move from this. When a move is sensitive to *roem*, sometimes these extra points will go to his team and the rest of the time to his opponent. There are many situations where in a game a player has two possible cards to play. With one he may win the trick, but with the other the opponent does, and might also get extra points for *roem*. In these situations a smart player will always play the move in which he wins, but random players do this only a certain amount of the time. The Monte Carlo player determines his move based on these random playouts, which may have been based on the opponent not playing a "smart" move every normal player would play.

This problem was eventually solved by modifying the random player, so that the Monte Carlo player would base its decisions on a less "dumb" player. The result was a semi-random player, which has the following improvements over the random player:

- If the current player or his team mate is not about to win the trick, it tries to avoid *roem*. If the current player is the third or last player to play a card, he can know for sure when he or his team mate will not win the trick. If he is the last player he can also be sure there will be *roem*, and if he is the third player in many cases he can also be sure. In these situations we first check the amount of *roem* without the player's card, and then after. If it is more *roem* afterwards, we try to avoid that card.

- The next improvement applies when the current player or his team mate is not about to win the trick, the current player cannot follow suit and does not have any trump cards. The current player may play any card he has in his hand, but it would be very unwise to throw away an Ace or 10.[8] These cards can be used to either win another trick or to generate more points for another trick won by his team mate. This improvement prevents Aces and 10's to be thrown away if the trick is won by the other team.

  It can happen that the trick will be won by the player's team mate if he can play the last card. This can only happen if the current player is the second to play a card, and even then it would be unwise to throw away an Ace or a 10.

[8]This improvement always prevents Aces to be thrown away, but if 10's are also protected from being thrown away, can be disabled using a parameter.

- If the trick *is* about to be won by the player's team mate, and a card can be played that gives the team extra *roem* points, this card should always be played. This applies only when the current player is the last player to play a card, so we know for sure that the trick cannot be won by the opponent.

These improvements are achieved by splitting them into two groups; cards that are "smart" and should be played, and cards that are "dumb" and should not be played.

The "smart" cards are listed in the third bullet point of the list above. They are added to an array and if this array is not empty a random card from it will be returned.

If the array of "smart" cards is empty, the algorithm looks at the "dumb" cards. These cards are listed in the first two points of the list above. They are also added to an array and if this array is not empty and if the array is not the same as the array of all possible moves, these cards are removed from the array of possible moves. Then a random move is returned from the array of possible moves.

6.2 Roem points for the opponent

A lot can be said about a player's cards based on the cards he plays. The algorithm described in Section 5 takes into account that certain players cannot receive certain cards in a redistribution. This is based on certainty, since those players cannot possibly have those cards while following the game rules. This occurs for example when a player cannot follow suit and plays a trump card; in a redistribution he cannot receive the suit he could not follow earlier. In this section we try to use knowledge that is not entirely certain. The knowledge we use here comes from the following situation: a player is the last player in a trick to play a card, and his opponents are about to win the trick. If the player plays a card that awards additional *roem* points to his opponent, he will most likely did not have any choice. If he had the choice to play another card which either lets hem win the trick or does not give extra points to the opponent, he would most likely have played that card.

An expansion to the pure Monte Carlo player was created to utilize this knowledge while still relying on the method of Section 5. Simply saying that a player does not have a certain suit in the matrix defined in Section 5.2 would not work, because a player *can* play a card awarding extra points to the opponent. If that were to happen and the player does have that suit, the algorithm might not be able to find a redistribution at all. Therefore this player creates two of these matrices. The first is the same as earlier, containing the suits a player cannot receive. The second one also contains this, but additionally contains the suits a player *probably* does not have, because he played a card awarding additional points to the opponent. When the algorithm runs the random games, some percentage of the time he will use the "probable" matrix, and the rest of the "certain" matrix. It can happen that the algorithm cannot find a redistribution using the "probable" matrix. To this end a maximum is defined for the number of attempts, if this is exceeded the algorithm will simply try again using the "certain" matrix.

Apart from not being able to find a suitable redistribution of cards, it can also happen that a player had two cards of that suit, both of which awarded additional points to the opponent. If the algorithm only relied on the estimate that a player does not have a suit while actually the player had a choice between two “bad” cards, it would often make the wrong assumption.

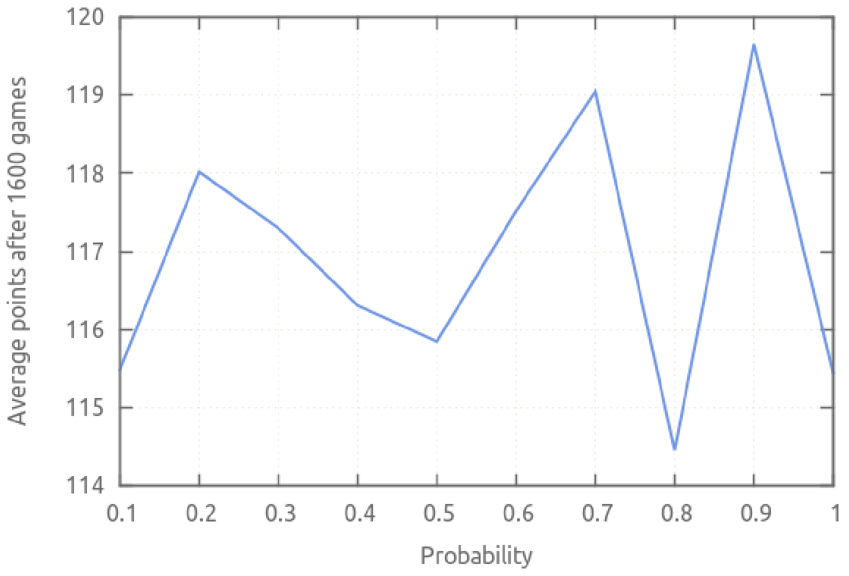


Figure 5: Average results against the pure Monte Carlo player per percentage used.

The percentage of the time the algorithm uses the “probable” matrix is defined to be 0.9. This number is chosen by experiments, the results of which can be seen in Figure 5. These are the results of 1600 games[9] against a team of pure Monte Carlo players (with semi-random playouts) on the vertical axis, and the percentage of times the “probable” matrix is used on the horizontal axis.[10] What first strikes about the results is that for probabilities higher than 0.5 the results get higher as well (apart from 0.8), and it is certainly noteworthy that 0.9 gets the highest results. This is most likely because at these percentages the algorithm uses the “probable” matrix far more often than the “certain” matrix. If it uses the “probable” matrix, it assumes that a player who played a card that resulted in *roem* for the opponent does not have any cards of that suit left. In these cases the algorithm might have that assumption right, in which case the results will most likely be higher. If it does not get the assumption right, it might often not be able to find a suitable redistribution. If that is the case the algorithm tries to find a redistribution until the maximum of 1000 shufflings has been reached, and then switches to the “certain” matrix.

[9]The number 1600 is based on 100 × 16 *full matches*, see also Section 2.1.

[10]This figure may seem unstable, which corresponds to the fact that the vertical axis begin at 114. In reality these results are very close to each other.

6.3 Punishment for roem

In Section 6.2 it is explained that when a player plays a card that gives additional *roem* points to the opponent, he most likely will not have had the choice to play a card that did not give away *roem* points. During the testing of our algorithm, we notices that our implementation in some cases did play the card giving away *roem*, while having another option that prevented this.

An example of this is displayed in Figure 6, where the last player (on the east side) has the choice to play the 8 or the King of Hearts. The trick is about to be won by the player on the north, and neither 8 or the King can change this. However, if he plays the 8, his opponent will receive an additional 50 *roem* points, which they will not receive when he plays the King. For that reason no human player would play the 8 in this case, but our implementation initially did.

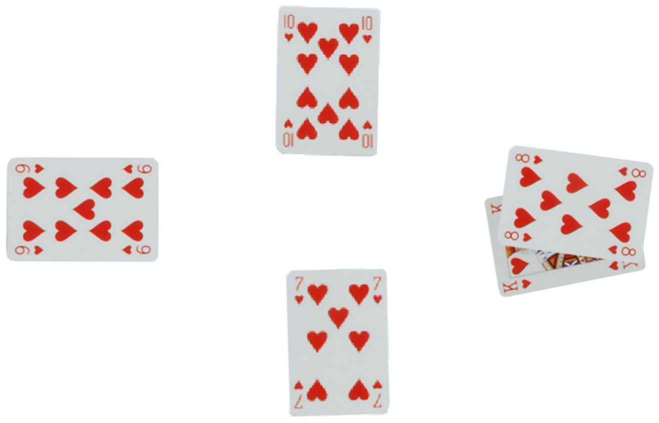


Figure 6: Example where the last player has a choice to give *roem* points to the opponent.

The reason for this is that each playout the Monte Carlo player only looks at his own points at the end of the game. In the example above, the Monte Carlo player will not notice the *roem* points, since they do not change the amount of points his team will receive at the end of the game. Normal points for the opponent will be noticed by the Monte Carlo player, because each point the opponent receive, his team cannot, andhe total amount of points each game is always 162 (excluding *roem*). A King is worth four points and an 8 is not worth any points, so when the Monte Carlo player plays the King he only notices that his opponent would receive more points that trick. However, when he plays the 8 his opponents will receive four points less but 50 *roem* points more.

To prevent this from happening a system of punishment was implemented in each of the Monte Carlo players. The idea is that for each move that gives *roem* points to the opponent, that amount of points will be subtracted from the score of the playout. So if in the example above the Monte Carlo player would have 72 points after a playout, 50 points will be subtracted and the result will be 22 points for the playout.

Another solution for this problem we tried was by not looking at the points the player's team had at the end of the playout, but by looking at the percentage of the total amount of points of both teams. While this also prevented the situation explained above, it turned out this solution performed less well than the method of penalty points.

6.4 Probability distribution for suits

The next improvement on the pure Monte Carlo player involves a bigger change to the player defined in Section 5. The pure Monte Carlo player redistributes the cards by simply distributing the cards randomly and checking whether the redistribution fits with the matrix. The method described in this section distributes the cards differently: by creating a probability distribution.

This probability distribution is an approximation of the distribution of the remaining cards by looking at probabilities for individual suits. An example for such a distribution is shown in Table 7. Each cell represents the probability a player has to receive a certain suit. In the example of Table 7 one can see that there are no Clubs and Diamonds left in the game (or the current player has all of them himself). Player 2 played this game, and the trump suit is Hearts. Player 1 cannot receive any Hearts, because he has not followed this suit earlier in the game.

This probability distribution currently relies on knowledge gained from two assumptions: the player that plays has good cards and if a player plays a card that gives the opponent additional *roem* points, there is a bigger chance this card was his last card of that suit.

	♠	♡	♣	♢
Player 0	0.17	0.4	0	0
Player 1	0.42	0	0	0
Player 2	0.42	0.6	0	0
Cards left	2	4	0	0

Table 7: Example of the probability distribution for suits.

The implementation of the first assumptions is done in two parts, both relying on a certain multiplier for the probabilities of that player. The assumptions is that if a player plays the game, he has good cards. Good cards in KLAVERJAS mostly mean many trump cards, with one or two of the highest trump cards, and several aces. We will only focus on the trump cards in this section. We can use this knowledge only if a player, other than the current player, plays the game and if he played by choice. If the current player plays we have no knowledge of the other players. If a player plays but not by choice (if all players have passed on two suits), we also have no knowledge about his cards. It can be said that if he passed earlier on that suit we know that his hand most likely will *not* be a good hand. We do not use this knowledge however, since every player passed on this suit it would eventually have the same effect as if no knowledge was used.

If the current player plays or another player plays but not by choice, all probabilities are evenly divided. The probability per player become simply $1/\ell$, where l is the amount of players that may receive this suit. When each player may still receive a certain suit, their probability for this suit will become $\frac{1}{3}$. With the probability evenly divided, the results are the same as with the method described in Section 5.2. The algorithm determines per card to which player this card will be distributed, based on the probability in the probability distribution.

It can happen that when all the cards have been distributed, one player has too and another has too few cards. If this is the case, the algorithm is repeated until a suitable distribution is found.

If another player than the current player plays by choice, we assume he has good cards. This is done by multiplying his probability of trump cards with a certain multiplier M_1, initiated by experiment on 1.2. These experiments are performed in combination with the multiplier defined in Section 6.5. The results can be found in Section 6.6. The probabilities for the trump suit are now calculated by first determining what the "evenly distributed" probability would have been, and multiplying this by M_1 as the probability for the playing player. Then the "rest" of the probability is evenly divided among the other players. This way the probability for the player that plays become $p_1 = M_1 \cdot \frac{1}{\ell}$, where M_1 is the multiplier and ℓ is the amount of players who may receive this suit. The probabilities for the other players become $p_2 = \frac{1-p_1}{\ell-1}$, where p_1 is the previously defined probability for the player that plays. In the example of Table 7 this can bee seen by looking at the trump suit, in this case Hearts. There are two players that may still receive Hearts, so the "evenly distributed" probability is $\frac{1}{2}$. Player 2 plays the game, his probabilities will be multiplied by M_1. Therefore his value in the probability distribution becomes $\frac{1}{2} \cdot 1.2 = 0.6$, and the value for Player 0 will become $\frac{1-0.6}{1} = 0.4$.

It can happen that (because of the multiplier) the expected amount of trump cards for the player that plays becomes larger than one. To this end a maximum probability of 0.98 was defined, and when the probability for a player becomes larger than one, the probability for this player becomes $0.98 \cdot k$. This also means that for the other players the probability will be 0.01 or 0.02, depending on how many players may still receive that suit.

	♠	♡	♣	♢
Player 0	0	0	0	0.33
Player 1	0	0.5	0	0.33
Player 2	0	0.5	0	0.33
Cards left	0	1	0	5

Table 8: Example of the probability distribution for suits. The two possible distributions are displayed in Table 9.

In order to maximize the results of this technique the method of Section 6.2 was also applied to this probability distribution. This applies when a player plays a card that results in additional *roem* points for his opponents. In this case we assume that this was most likely not by choice, and that the probability for him to have more of this suit are lower. Therefore, when calculating the probability distribution we check whether this situation has occurred, and if so the probability for that player will be multiplied by the percentage defined in Section 6.2. The "rest" of his probability, the amount by which it was reduced, is evenly divided among the players that may still receive this suit. This ensures that the total of all probabilities remains the total amount of cards of this suit left. In the example of Table 7 this has applied to Player 0, apparently he played a Spade that resulted in *roem* for the opponent. Therefore his probability is reduced to 0.17, and the probability for the other players become $\frac{1-0.17}{2} = 0.42$.

This probability distribution is an approximation of the actual possible distributions of the cards in the current game. For example, from the probability distribution in Table 8 two "actual" distributions of the cards can be deduced, presented in Table 9. There are six cards left in the game (two cards for each player), one of the Hearts suit and five of the Diamonds suit. It is clear that Player 0 can only receive Diamonds, since his possibilities for the other suits is 0. That leaves three Diamond cards left to distribute over two players who are both to receive two cards. This means one of them has two Diamond cards, the other has one. The player that has one receives the last Hearts card. The distribution in Table 8 is an approximation since it states that the probability for Diamonds for Player 0 is 0.33, while in reality this is 1. The algorithm will however never generate distributions wherein Player 0 does not have two Diamond cards.

	♠	♡	♣	♢
Player 0	0	0	0	2
Player 1	0	0	0	2
Player 2	0	1	0	1

	♠	♡	♣	♢
Player 0	0	0	0	2
Player 1	0	1	0	1
Player 2	0	0	0	2

Table 9: The two actual possible distributions from the probability distribution in Table 8.

Note that this algorithm might generate distributions for the remaining cards that do not correspond with the individual probabilities from the table. It is unsure if each "actual" possible distribution is visited as often as the possibilities point out. To examine this, a simulation could be developed.

6.5 Probability distrubution for trump cards

In KLAVERJAS it may be more important *which* trump cards a player has, rather than just *how many*. The two highest cards in the game, the Jack and 9 of trumps, are more important for a good hand than just having many trump cards. Not only do they beat all other cards, they are also worth more points (respectively 20 and 14 points) than all other cards. Therefore, if a player plays the game, he will also be more likely to have these cards than other players.

To implement this increased probability, a second probability distribution was introduced in addition to the existing distribution. This distribution contains for each player the probability to receive a certain trump card. An example of this matrix can be found in Table 10.

	7	8	Q	K	10	A	9	J
Player 0	0	0	0.30	0.30	0	0	0.26	0.26
Player 1	0	0	0.30	0.30	0	0	0.26	0.26
Player 2	0	0	0.40	0.40	0	0	0.48	0.48

Table 10: Example of the probability distribution for trump cards.

This probability distribution has a structure that resembles the matrix indicating which player may not receive a certain trump card defined in Section 5.2.

In this example there are only four trump cards left in the game, and Player 2 plays the game. The probability for his "regular" trump cards are multiplied by M_1, defined in Section 6.4. The Jack and 9 of trumps have, besides being multiplied by M_1, also been multiplied by an additional multiplier M_2. This second multiplier applies only to the Jack and 9 of trumps, giving the player that plays a further increased probability for these cards. M_2 has been set by experiment on the 1.2, these experiments can be found in the next section.

6.6 Determining the value of the multipliers

In order to find the best suitable values for the multipliers M_1 and M_2 experiments were performed on several combinations. The results for each combination are based on 1600 games by a team of players with this configuration against a team of pure Monte Carlo players, with semi-random playouts but without probabilities.

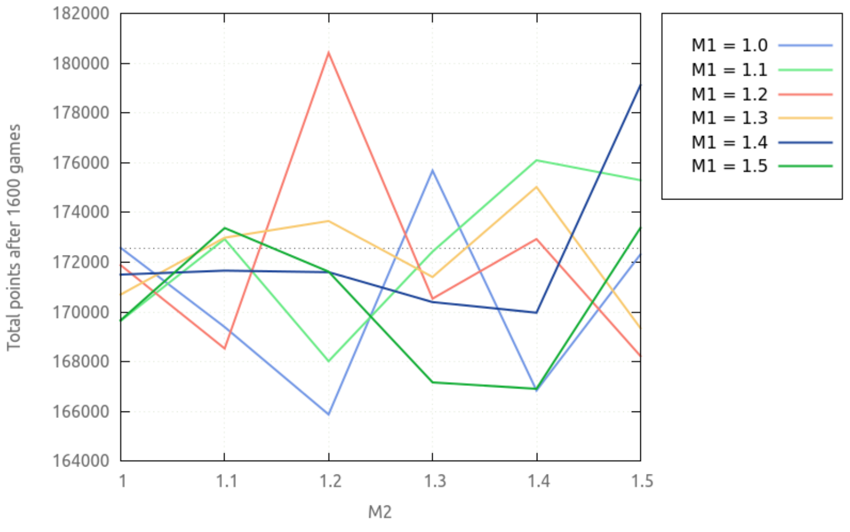


Figure 7: Comparison of results for different values of multipliers M_1 and M_2.

The results for these experiments are displayed in Figure 7. The first thing one notices about the graph is that according to the experiments, $M_1 = 1.2$ and $M_2 = 1.2$ (the red line) is the best configuration of all explored combination. Therefore, this is the configuration used in the main Monte Carlo player.

There is a dotted line plotted at the value of 172 578, this is the value for $M_1 = 1.0$ and $M_2 = 1.0$. With both M_1 and M_2 configured at 1.0, these are the results without the use of multipliers. Every configuration below the dotted line performs less than without the use of multipliers.

These results are against a team of Monte Carlo players with semi-random playouts, which performs well against a team of Monte Carlo players with probability distribution. The results can differ when tested a team of other players.

7 Comparing the players

In order to test the performance of the different types of computer players, two separate cases have to be tested. The first is how each player performs against each other player. The different strategies proposed in this thesis have been tested and "trained"[11] against one another, here these results will be shown.

The other case is against human players. It is more difficult to test and "train" the program against human players, because this would take much more time. While enhancing the Monte Carlo player, typical "human" techniques were taken into account. This is especially the case for the amount of trump cards for the player that plays the game.

7.1 Computer players against each other

To compare the performance of the different players created in this thesis, a competition was created as an experiment. In this section the results of that experiment are presented.

The results of this competition can be found in the schematic in Table 11. Horizontally the average amount of points a player scored in 1600 games against other players are shown. For example, the second cell of the first row (with the value of 87.91) represents that player 1 (the random player) scored an average of 87.91 points against player 2 (the semi-random player). The opposing cell, the first cell in the second row, represents the amount of points the semi-random player scored against the random player. These are the averages of the different sets of 1600 games, the results of the opponent are not shown in this table but are mostly the same as the opposing cells.

	1	2	3	4	5
1		87.91	54.21	53.09	54.34
2	113.87		61.27	60.31	58.59
3	160.84	147.72		104.80	99.21
4	162.07	148.60	113.30		106.89
5	159.04	151.70	110.74	107.00	

[1] Random player
[2] Semi-random player
[3] MC player with random playouts
[4] MC player with semi-random playouts
[5] MC player with probability distribution

Table 11: Results of a competition of the different computer players.

It is interesting to note that the Monte Carlo player with probability distribution scored less points against the random player than the other Monte Carlo players. This is because the player with probability distribution makes assumptions about players' cards, for instance when a player plays a card that gives the

[11] Training in this instance is not automated training, but testing with most possible values. See Section 6.6.

opponent *roem* points. The random player does not takes this into account and often plays cards that give *roem* to the opponent. Therefore the Monte Carlo player with probability distribution sometimes makes wrong assumptions about their cards, and performs slightly worse than the "normal" Monte Carlo players.

	Player	Points	Average
1	Random player	1 117 645	69.85
2	Semi-random player	1 251 142	78.20
3	MC player with random playouts	1 978 588	123.66
4	MC player with semi random playouts	2 028 082	126.76
5	MC player with probability distribution	2 033 343	127.08

Table 12: Total sum of all points per computer player.

The total and average amount of points the players scored (including the games they were the "opponent" and games against themselves) can be found in Table 12. Here it can clearly be seen that the player with probability distribution performs best, shortly followed by the other Monte Carlo players.

7.2 Computer players against human players

In order to fully test the capabilities of the algorithm, a match was set up between two skilled human players and a team of Monte Carlo players with probability distribution.[12] Although not much can be said about one full match against a single team of human players, the match did give insight in how the algorithm functions.

For this experiment the implementation was extended with a competition-mode. Normally the program only plays a single game, but with this mode this is extended to a full match of 16 games. The program then also outputs the players' cards and information about what cards have been played and who has to play a card to a file for each player. Both of the human players had a tablet on which this file was shown, allowing them to view only the information they are allowed to see.

The results of the match can be seen in Table 13. As this table points out, the computer players beat the human players with more than double the points the humans had. Because this is such an insignificant amount of games, statistically nothing can be said about these results. In many cases, the computer players had more "luck" when being dealt the cards. To rule out luck many more games need to be played, which would take a very large amount of time.

When asked how the computer players performed, the human players pointed out that in most cases they appeared to play logical and natural. But as we have found as well, sometimes the algorithm makes moves that are sensitive for *roem* points, especially when being the first player to play a card. Often the first card the computer team played was a Jack or a Queen, this can result in many *roem* points for the opponent and human players often avoid this. Also the human players pointed out that the algorithm often tries to get trump cards out of the game, when it knows that only his team mate has trump cards left. This would win them the trick, but maybe lose a trick later in the game (or the last trick, which is worth extra points).

[12]Thanks go out to J. Spierenburg and B.V. Jongeneel for participating in this experiment.

Trick	AI players	Human players
1	51	131
2	14	198
3	150	12
4	121	41
5	181	51
6	144	38
7	147	35
8	154	48
9	162	0
10	202	0
11	64	178
12	142	90
13	165	17
14	182	0
15	110	72
16	162	0
Total	2151	911

Table 13: Score form for match of Monte Carlo players against Human Players.

8 Conclusion

In this thesis we examined the effectiveness of a Monte Carlo strategy on the game of KLAVERJAS, and different strategies of enhancing this strategy using domain knowledge and probabilities.

Given the results in Section 7.1, it can be concluded that of the different Monte Carlo players, the player with probability distribution performs best. This is most likely because this player combines all of the different strategies in enhancing the Monte Carlo search presented in this thesis.

The results in Section 7.2 point out that when given good cards, the algorithm with probability distribution is capable of winning against human players, and that in many cases the algorithm plays logically.

The Monte Carlo player with probability distributions for card suits and trump cards performs better against skilled players, since it is a combination of all strategies for improving the Monte Carlo player described in this thesis. It uses knowledge gained from players playing the game on a certain suit, assuming that they have many trump cards and a higher probability for the two highest trump cards. It also uses the knowledge explained in Section 6.2, assuming that when players give away *roem* points to the opponent, they did not have a choice. These strategies give the player a better estimate about the rest of the players' cards, and therefore produce a better player overall. This makes the Monte Carlo player with probability distribution a better player against skilled players, and performs worse against a non-logical player. An example of this is when a player plays the game, but does not have good cards. He most likely will not win, but the Monte Carlo player will also make the wrong assumptions about his cards.

The Monte Carlo technique in general is a reasonably good method for creating a computer player without much domain knowledge on the game. In this thesis the Monte Carlo player was enhanced by using domain knowledge, and from the results in Section 7.1 it can be concluded that the usage of probabilities for the distribution of cards can provide a better player.

Although a team of computer players described in this thesis can play KLAVERJAS reasonably well, a combined team of human and computer players might not work as good. This is mostly because human players often play according to the tactics described in Section 2.5, while the computer players often do not. They do not know the conventions in signalling, and therefore will not understand what a human player means by signalling with a certain card. Some tactics the computer player will indeed detect, since they are not just conventions. An example is when a player has to lead the first card in a game and his team mate plays the game. In those cases the Monte Carlo players mostly lead with a trump card sensitive for *roem*, as also described in Section 2.5. But with more comprehensive common tactics the Monte Carlo players will not understand what the human meant to signal. It currently seems not possible for the Monte Carlo player to learn itself to signal using cards. This might become possible by extending the semi-random player used for the playouts, since this is the basis the Monte Carlo player uses to determine its moves.

8.1 Further research

Initially, the idea behind the probability distribution was that the algorithm would eventually be able to learn the probabilities itself. Our final implementation based the probabilities on multipliers for knowledge gained. This basis of the knowledge is however still dependent on what we, human players, defined as technique. A future goal would be to develop a way to let the program teach itself how these probabilities develop throughout the game. A neural network could be trained to this end, using various input nodes (such as cards played, but also if a player played or passed on a suit) and a probability matrix such as in Section 6.4 or Section 6.5 as output. Another improvement to the probability matrix could be to not use probabilities per suit or per trump card, but for all cards. If this could also be combined with a neural network, the program might be able to develop a completely different technique than previously known by human players.

The pure Monte Carlo technique could be improved further by implementing Monte Carlo Tree Search, as described by Browne et al.[14]. By not visiting each possible move as often as the other, the results and speed of calculation could be improved.

The Monte Carlo players proposed in this thesis did not learn themselves to signal cards to give their team mates more information. The implementation could be extended to learn different tactics in signalling, either by learning itself or by implementing them manually. To this end the semi-random player could also be extended to allow the Monte Carlo player to learn these tactics.

References

[1] Klaverjas - Wikipedia
`https://en.wikipedia.org/wiki/Klaverjas`
[Accessed: 12–12–2016].

[2] Homepage Nederlandse Klaverjas Uni
`http://www.klaverjasunie.nl/`
[Accessed: 21–12–2016].

[3] Powley, E.J., Cowling, P.I. & Whitehouse, D.
Information Capture and Reuse Strategies in Monte Carlo Tree Search, with Applications to Games of Hidden Information.
Artificial Intelligence 217 (2014) 92–116.

[4] Furtak, T. & Buro, M.
Recursive Monte Carlo Search for Imperfect Information Games.
IEEE Conference on Computational Intelligence in Games (CIG) (2013), 8 pages.

[5] ACBL/WBF World Computer-Bridge Championship
`http://www.allevybridge.com/allevy/computerbridge/index.htm`
[Accessed: 11–01–2017].

[6] 20th Ourgame World Computer-Bridge Championship
`http://www.allevybridge.com/allevy/computerbridge/results2016.html`
[Accessed: 11–01–2017].

[7] Campbell, M., Hoane Jr., A.J. & Hsu, F.
Deep Blue.
Artificial Intelligence 134 (2002) 57–83.

[8] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel T. & Hassabis, D.
Mastering the game of Go with deep neural networks and tree search.
Nature 529 (2016) 484–489.

[9] Parlett, D.
The Penguin Book of Card Games.
Penguin (2008).

[10] Rules of Card Games: Klaverjas
`https://www.pagat.com/jass/klaverjassen.html`
[Accessed: 27–10–2016].

[11] Cursus Klaverjassen
`http://www.klaverjasunie.nl/main_course.htm`
[Accessed: 11–01–2017].

[12] GitHub - Monte Carlo Klaverjas
`https://github.com/choogenboom/Monte-Carlo-Klaverjas`
[Accessed: 16–01–2017].

[13] Kantar, N., Dimitrescu, D. & Lundqvist, M.
Bridge Classic and Modern Conventions, Vol I.
Example Product Manufacturer (2001).

[14] Browne, C., Powley, E.J., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Liebana, D.P., Samothrakis, S. & Colton, S.
A Survey of Monte Carlo Tree Search Methods.
IEEE Transactions on Computational Intelligence and AI in Games 4 (2012) 1–43.