Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Simulation und Graphik

Diploma Thesis

The UCT Algorithm Applied to Games with Imperfect Information

Author:

Jan Schäfer

July 11, 2008
(revised version)

Tutors:

Dr. Michael Buro
University of Alberta
Department of Computing Science
Edmonton, Alberta
Canada

Dr. Knut Hartmann
Otto-von-Guericke-Universität Magdeburg
Institut für Simulation und Graphik
Magdeburg
Germany

Schäfer, Jan:
The UCT Algorithm Applied to Games with Imperfect Information
Diploma thesis
Otto-von-Guericke-Universität Magdeburg, 2007.

Acknowledgement

Many people have helped me to finish this thesis. Some of them I wish to thank here. First of all I would like to thank Michael Buro for his patience and his great tips during the implementation of the UCT player. I have learned very much from him again. Knut Hartmann deserves thanks for putting the thesis through at the university. Frank Herling and Patrick Mähler played against the UCT player in many games and showed me, that there is still a lot work to do. The source code for the DDSS player and for XSkat was given to me by the authors Sebastian Kupferschmid and Gunter Gerhardt. The ideas for the possibilities of analysing the data were discussed in a consultation with Evelyn Kuhnt. The spelling and grammar was checked by Friederike Girlich and Patrick Mähler. Last but not least I have to thank my family and my friends for their understandig and support during the last months.

Abstract

The UCT algorithm was already used to search the best strategy in games with perfect information. The algorithm is known to handle games with high branching factors in the search tree very well. For the game of Go a computer player was implemented that uses the UCT algorithm and was able to cope with the best computer Go players. For games with imperfect information, this algorithm is now used for the first time. For the card game of Skat an implementation of the UCT algorithm is available that uses the algorithm in all three phases of the Skat game. These phases are bidding, discarding and playing the tricks. To be able to use the algorithm all unknown information must be replaced by simulated information. After that, the UCT algorithm can be used as in a game with perfect information. All results of the simulations are saved in one search tree with information sets.

The playing strength of the UCT player is tested against other computer players and against human players. For this purpose a tournament simulation program for Skat was developed. With this simulation program it is possible to let AI players compete against other AI players in games with predefined card distributions. Every player is given then the same chances to make all games. In games against human players a normal Skat series is simulated.

The UCT player can keep up with other computer players. Although it might not have reached its highest playing strength. The best combination of parameters is still to be found. The number of simulations the player can compute in a reasonable time for each move decision is also too low. The presented implementation is not optimized for speed, yet. Higher numbers of simulations will result in a higher playing strength. The UCT player can also keep up with human players. It looses the Skat series only due to conservative bidding.

Contents

List of Figures **ix**

List of Tables **xii**

List of Abbreviations **xiii**

1 Introduction **1**

2 Fundamentals **3**

2.1 History of Skat 3

2.2 Rules of Skat 4

2.2.1 Dealing the Cards 4

2.2.2 Bidding 5

2.2.3 Discarding the Cards and Game Announcing 5

2.2.4 Playing the Tricks 5

2.2.5 Game Types 6

2.2.6 Game Value Calculation 7

2.2.7 Skat Series 8

2.3 Classification of Skat in Game Theory 8

2.4 Previous Works on the Game of Skat 8

2.4.1 Double Dummy Skat Solver 8

2.4.2 Monte Carlo Simulations with Imperfect Information 9

3 UCT Algorithm **11**

3.1 Multi-armed Bandit Problem 11
3.2 Monte Carlo Tree Search 12
3.3 UCT Algorithm Applied to Games with Perfect Information 13
3.4 UCT Algorithm Applied to Games with Imperfect information 14

4 Skat Tournament Arena **15**
4.1 Requirements of the Tournament Arena 15
4.1.1 Skat Series for Evaluation of Play Strength 16
4.1.2 Payoff for Calculation of Play Strength 16
4.2 Implementation of SkatTA 17
4.2.1 Representation of Skat Games, Skat Series and Cards 19
4.2.2 Generation of Card Distributions 20
4.2.3 Interface for AI Players 21
4.2.4 Inclusion of XSkat 22
4.2.5 Interface for XSkat Based AI Players 22
4.2.6 AI for Bidding and Discarding 22
4.2.7 AI for Trick Play 23
4.2.8 Graphical User Interface 24
4.2.9 Helper Classes 25
4.3 Data Output 27

5 Implementation of the UCT Algorithm for Skat **29**
5.1 UCT Player 29
5.1.1 Bidding 29
5.1.2 Discarding and Game Announcement 30
5.1.3 Comparison with Bidding of XSkat 31
5.1.4 Search Tree with Information Sets 31
5.1.5 Playing the Tricks 32
5.2 Comparison with Monte Carlo Simulation 33

6 Optimizations **35**

6.1 Selections Policies 35

6.1.1 Epsilon Greedy Policy 35

6.1.2 Monte Carlo Policy 36

6.2 Game Value Adjustment 37

6.2.1 Game Value in Ranges 37

6.2.2 Single Player Card Points 38

6.3 Combination of Selection Policy and Value Adjustment 38

6.3.1 Epsilon Greedy Policy 39

6.3.2 Monte Carlo policy 40

6.4 Optimized Card Simulation 40

6.5 Random Move on Equal Rating 41

6.6 Exploration Factor 42

6.7 Zobrist Hash Function 42

6.8 Adjusting the Aggressiveness During Bidding 43

6.9 Best Combination of Optimizations 44

7 Comparison with Other Players **45**

7.1 Comparison with DDSS 45

7.2 Comparison with Human Players 46

8 Summary and Future Work **49**

8.1 Summary 49

8.2 Ideas for Future Work 50

8.2.1 Best Combination of Parameters 50

8.2.2 Bidding and Discarding 50

8.2.3 Better Heuristic for Game Simulation 51

A Optimized Generation of Card Distribution **53**

B UCT Search Tree Example **55**

C Example for a Grand Game **57**

D Post Thesis Work **61**

Bibliography **63**

Theses **67**

List of Figures

3.1 UCB1 policy 12
3.2 ε-greedy policy 12
3.3 Monte Carlo tree search 13

4.1 Class diagram of the Skat Tournament Arena 18
4.2 Shuffle algorithm used in SkatTA 20
4.3 Dialog for setting all parameters for a new Skat series 25
4.4 Discarding phase for a human player 26
4.5 Playing the tricks 27

6.1 Step function for game value adjustment 38

A.1 Optimized generation of card distribution 54

B.1 Example for an UCT search tree 56

List of Tables

2.1 Values of the different cards 4
2.2 Card order in null games 6
2.3 Factors for suit and grand games 7

4.1 Player combinations for two player compare 17
4.2 Coding of the cards in SkatTA 19
4.3 Interface for an AI player 21

5.1 XSkat vs. UCT bidding 31
5.2 MC player vs. UCT player with MC policy 34

6.1 ε-greedy policy with $c = 4$ and $d = 0.5$ 36
6.2 Monte Carlo policy 36
6.3 Game value in ranges 37
6.4 Single player points divided by 120 39
6.5 Combination of ε-greedy policy and points in ranges 39
6.6 Combination of MC policy and single player points divided by 120 40
6.7 Optimized card simulation 41
6.8 Random move choosing on same rating 41
6.9 Exploration factor 8 42
6.10 Zobrist hash function without order of played cards 43
6.11 XSkat vs. UCT bidding 44

7.1 UCT with optimizations vs. DDSS 45

7.2 UCT player with optimizations vs. Human 1 47
7.3 UCT player with optimizations vs. Human 2 47
7.4 UCT player with optimizations vs. Human 3 47

C.1 Card distribution for the example game 57
C.2 XSkat vs. DDSS and UCT 58
C.3 DDSS vs. UCT and XSkat 58
C.4 UCT vs. XSkat and DDSS 59

D.1 UCT with optimizations vs. DDSS 61

List of Abbreviations

AI	Artificial Intelligence
CPU	Central Processing Unit
DDSS	Double Dummy Skat Solver
GIB	Goren In a Box or Ginsberg's Intelligent Bridgeplayer
GUI	Graphical User Interface
ISkO	International Skat Order
ISPA-World	International Skat Players Association
MC	Monte Carlo
SkatTA	Skat Tournament Arena
UCB	Upper Confidence Bounds
UCT	UCB applied to trees

Chapter 1

Introduction

The research in artificial intelligence (AI) is as old as the field of computer science itself. In the last years computer programs played very successful in games with perfect information. In these games every player has the same information about the game status because all information is visible to all players. Prominent representatives of this kind of games are Chess, Checkers, Othello (Reversi) and Go. Search algorithms like the min-max algorithm are used to find the optimal moves assuming that every player plays perfect. Some games with perfect information are now known as solved because no human player can play better than a perfectly playing artificial intelligence. The game of Checkers is currently the last example that cannot be won by a human player against the best AI. Recently it was shown in [SBB+07] that Checkers leads to a draw game if the players do not make mistakes.

Some games with perfect information are still hard to solve for AI players. Standard algorithms like the min-max algorithm fail in the game of Go because too many moves are allowed at every stage of the game. This leads to a high branching factor of the search tree. Therefore the optimal strategy can not be calculated in a satisfying time with the computers available today even with improved search technics like alpha-beta-pruning.[GW06]

There is another family of games that is also hard to solve with the search technics described above because some information about the game status is hidden. These games are called games with imperfect information. The card games Poker, Bridge and Skat are members of this family. One approach to solve these games are Monte Carlo (MC) simulations. The unknown information is set randomly and the best strategy for this simulated game can be searched. By doing many MC simulations a good estimation about the real status of the game can be approximated.

A new algorithm called Upper Confidence Bound applied to trees (UCT) was recently successfully used for the game of Go.[GW06] It is able to deal with the high branching

factor found in the search trees of a Go game. The goal of this diploma thesis is to show the applicability of this algorithm to games with imperfect information. Until now it was not adapted to these family of games yet. The applicability is shown by using the algorithm for the game of Skat.

In chapter 2 the fundamentals of the game Skat are explained and previous works are contemplated. The UCT algorithm is outlined in chapter 3. For the experiments presented in this thesis a tournament simulation program was developed. Its internal structure and features are described in chapter 4. The application of the UCT algorithm to the game of Skat can be found in chapter 5. In chapter 6 some improvements of the playing strength of the AI player that uses the UCT algorithm are probed. A comparison with other computer Skat players and with human players is given in chapter 7. The summary and an outlook to future work closes this thesis in chapter 8.

Chapter 2

Fundamentals

In this chapter, the rules of the game Skat are outlined. The classification of Skat in game theory is given and previous works are presented. This part is more or less a translation of the corresponding chapter of my bachelor thesis.

2.1 History of Skat

Skat was developed in the beginning of the 19th century in Altenburg, Germany. It has its roots in several older card games like Schafkopf, L'Hombre and Tarock. The name is derived from the Italian verb *scatare* wich means "to put away". After the dealing two cards are left over. These are called the Skat and gave the game its name. In the first years there were no strict rules available. The dealer of a game had to play the game as single player regardless it was possible to win the game or not. To eleminate this unfairness the bidding phase was introduced. From this time on an auction is held before each game to identify the player with the best cards.

The game was spread through the universities of Leipzig, Halle and Jena everywhere in Germany. The first rule books were published in 1884/85. The first congress about Skat was held up in 1886 in Altenburg. A binding order for the games did not exist a long time. Several attempts where made in 1899 and 1923. In 1927 a court was initiated in Altenburg. Until today it is responsible for clearing all questions about the rules of Skat. Finally in 1998 the International Skat Order (ISkO) was published which defines all official rules that are binding in Skat tournaments worldwide. Beside these rules many variants are still played in the pubs but are not supported by the Skat court.[Lin04]

Today, Skat is one of the most popular card games in Germany and it is also played worldwide by millions of players. European and world championchips are held every year alternately by the International Skat Players Association (ISPA-World).

2.2 Rules of Skat

In this section, only the basic rules of Skat are explained. They are necessary for the understanding of this thesis. The complete ISkO can be found on the webpage of the ISPA-World [IW03].

Skat is a card game played by three players. A card deck of 32 cards is used. It is divided into the four suits clubs (♣), spades (♠), hearts (♡) und diamonds (♢). The suit of clubs is the highest ranked and the suit of diamonds is the lowest ranked suit. Every suit has the following cards: seven (7), eight (8), nine (9), ten (10), jack (J), queen (Q), king (K) and ace (A). In this thesis, a card is described by the symbol for the suit and the abbreviation for the card value, e.g. ♣J stands for the jack of clubs. The values of the cards that are used for the calculation of the game result are listed in table 2.1. All card values sum up to 120 points.

Card	Value
Ace	11 points
Ten	10 points
King	4 points
Queen	3 points
Jack	2 points
Nine	0 points
Eight	0 points
Seven	0 points

Table 2.1: Values of the different cards

Although three players are playing the game of Skat, there are only two parties. One player always plays against the two others called the opposition players. The latter can not look into their cards but play together and try to force the single player lose the game.

2.2.1 Dealing the Cards

During the dealing every player receives ten cards. The two remaining cards are called the Skat and are put away for the time being. The dealing is done in a strict order. Every player receives three cards first. After that two cards are put away for the Skat. Then, the players get four and finally three cards until all cards are dealt. Every player has a special name according to the sitting position. The player sitting left from the dealer is called forehand. The next one on the left is middlehand. The third player is called hindhand.

2.2.2 Bidding

During the bidding the single player is determined. There is also a strict order in the bidding process. The middlehand starts announcing game values of possible games in ascending order to the forehand. The forehand player can always decide whether the bidding value can be held or if it is better to pass. The hindhand starts declaring new bids when one of the other players cannot raise or hold a bidding value anymore. This is done until no player can make another bid. The winner of the bidding becomes the single player who declares the game and plays against the two other players. Even if another player wins the bidding, the forehand player starts the play after the game announcement. If no bidding value is declared the game will be rejected and the cards will be shuffled and dealt again.

2.2.3 Discarding the Cards and Game Announcing

The winner of the bidding has the opportunity to look into the Skat. Up to two of the player's cards can be exchanged with cards from the Skat. The player is able to raise the value of the hand by discarding bad cards to the Skat and getting good cards from the Skat. After the discarding, two cards have to be in the Skat and ten cards on the player's hand. If the single player decides not to look into the Skat the game will be called a hand game.

After discarding the game type is announced by the single player. The value of the game must be at least as high as the last bidding value the player declared or hold. The forehand from the bidding phase starts the play when the announcement is done.

2.2.4 Playing the Tricks

Every game is divided into ten tricks. Players can play one card during each trick. Forehand plays the first card of the trick. After this, middlehand and finally hindhand play their cards. The trick is ended and the trick winner gets all cards and becomes the forehand of the next trick. The trick is won by the highest card played in the trick.

There are special cards called trump cards in the game of Skat. The four jacks and a suit that is announced by the single player belong to the trump cards. These cards have a higher rating than all other cards in a trick. The jacks rank above the cards of the trump suit.

The first card of a trick is the lead card. It defines which cards are allowed to be played by the middlehand and hindhand in this trick. If the second and third player in a trick have cards of the same suit in their hands, they have to follow that suit and will be allowed to play cards of this suit only. If they cannot follow the suit they can play

each card on their hand. If one player plays a trump card instead of a card with the suit of the lead card, this card will be trumped. If a player does not follow the suit by mistake, the game will be ended immediately and lost for the party of the failing player.

2.2.5 Game Types

There are three different game types in the game of Skat: suit, grand and null game. The first two differ only in the number of trump cards. In suit games, the jacks and all cards of a suit announced by the single player are trump cards. In grand games only the jacks are trump. Despite their low value of two points for the calculation of the game value the jacks have an outstanding position in the game. They rank above all other cards. In suit and grand games the aim of the single player is to get more than 60 points. As soon as 61 points are reached, the game is won by the single player and cannot be lost anymore even if the player plays wrong in a later trick.

The null game is the opposite game type to the suit and grand games. There are no trump cards and the jacks are put in line with the other cards. Also, the ten is of lower value for the trick winner calculation. The changed order is listed in table 2.2. The single player in a null game tries not to get a single trick. If one trick is won by the single player, the game will be ended immediately regardless if the cards in the trick result in points or not.

Card	Order
Ace	highest card of a suit
King	
Queen	
Jack	
Ten	
Nine	
Eight	
Seven	lowest card of a suit

Table 2.2: Card order in null games

All three game types can be played with or without looking into the Skat. If the single player decides not to look into the Skat, the game will be called a hand game. The single player can also decide to reveal the cards to the other players. This game variant is called ouvert game. All cards of the single player's hand must be displayed before the first trick. The opposition players play with hidden cards. The null and ouvert variants can be combined too. Every variant results in a higher game value than in normal games.

2.2.6 Game Value Calculation

In suit and grand games the number of the so called multipliers is calculated. This means the number of trump cards in a row without gaps starting from the ♣J. If the single player has the ♣J, all following trump cards will be counted until the first gap. The game is calculated "with multipliers". Otherwise, the player plays "without multipliers" and all missing cards until the first trump card are counted. If a player holds for example the ♣J, ♠J and the ♢J on the hand, the game will be calculated "with two". In another example with only the ♢J on the player's hand the game is played "without three". At the end the number of multipliers is incremented by one and the games are labeled "with two play three" or "without three play four".

For suit and grand games, further winning levels are added to the count of the multipliers. If the single player gains more than 90 points, the opposition players will be played "schneider". Did the single player make 120 points, the others have played "schwarz". In both cases, another winning level is added.

The sum of multipliers and winning levels is multiplied by the factor of the suit or grand game that was played. Table 2.3 lists all factors. A ♠ game "without one play two" is worth 22 points. A grand game "with four play five" and "schneider" for the opponent players is granted 144 points.

Game Type	Factor
♢	9
♡	10
♠	11
♣	12
Grand	24

Table 2.3: Factors for suit and grand games

Null games have four different game values. A simple null game is worth 23 points, the value for null hand is 35 points, a null ouvert game makes 46 points and, finally, the null hand ouvert game is rewarded with 59 points.

The game values are also used during the bidding. Every player decides depending on the multipliers on the hand and the game types which games are possible to win. The value of the highest rated game is used as upper limit for their bidding.

When a game is lost, the game value is multiplied by -2. If the single player also plays "schneider", that means only 30 points or less were made, then the negative game value will be multiplied by 2 again. The same will be carried out if the single player played "schwarz".

2.2.7 Skat Series

A fixed number of games is played in a Skat series. It should be a multiple of the number of players sitting at the table. At the end of each game, the game value is listed for the single player regardless the game was won or not. The player sitting left from the dealer becomes the new dealer in the next game and deals the cards. The other sitting positions are also changed in this order. After all games are completed, the game values are summed up and the player with the highest sum wins the series of games.

2.3 Classification of Skat in Game Theory

The game of Skat is a two person null sum game with imperfect information. Although three players are playing, there are only two parties. The single player plays against the two others called opponent players. Skat is a null sum game because the benefit of one party is as high as the loss of the other party. At the beginning of the game the players only know their own cards. Therefore Skat belongs to the family of games with imperfect information. During the game all cards are revealed step by step. Some players might gain perfect information about the cards before the last trick by watching the play of the other players and by combining this with the knowledge about the suit enforcement rule during the trick playing. From this point, the game is a perfect information game for this player. The others are still playing with imperfect information.

2.4 Previous Works on the Game of Skat

2.4.1 Double Dummy Skat Solver

The first work about an AI for Skat was done by Kupferschmid.[Kup03] The Double Dummy Skat Solver (DDSS) implements the ideas proposed by Levy in [Lev89] and presented for the game of Bridge by Ginsberg in [Gin99]. During game simulations an assumption about the hidden information is made with Monte Carlo (MC) simulations and the game with imperfect information is transferred hereby into the field of perfect information games. Now, the game can be solved with the standard search technics like the min-max search. This algorithm finds the optimal move in this simulated game assuming a perfect play of all players.

Kupferschmid implemented an optimized min-max search for the DDSS. This alpha-beta search including extensive pruning was able to solve a complete Skat game

in a very short time[1]. Unfortunately the playing strength of the resulting AI player was not investigated in great detail in the diploma thesis.

The approach was already critised by FRANK et al. in [FB98] for the game of Bridge, because the transfer of games with imperfect information into the perfect information space can lead to a mix of strategies which may result in the optimal strategy in the end.

2.4.2 Monte Carlo Simulations with Imperfect Information

In my bachelor thesis a new approach was investigated that uses MC simulations without losing the imperfect information.[Sch05] It was also implemented for the game of Skat. After the simulation of a possible world and playing the next card in this world, a heuristic was used to play the game until the end without revealing the actual moves. Only the result of the game was returned for evaluation of the move being made before playing the game to the end. With this information the AI was able to decide wich move is the best at the current stage of a game. The comparison with the DDSS showed the same playing strength for both players, although the MC player with imperfect information did not have the complete knowledge in the simulated games like the DDSS. The heuristic that was used to play the games to the end is also known to be not very strong. The combination of both was the key for the playing strength of this MC player.

[1] A complete evaluation of a search tree was possible in 0.06 seconds.

Chapter 3

UCT Algorithm

The UCT algorithm was developed for games for which the standard approach of min-max tree search is not feasible. It has its origins in solving the multi-armed bandit problem. In this chapter, the problem and the algorithm are described and previous works about games with perfect information are presented. At the end, the application of the UCT algorithm to games with imperfect information is outlined.

3.1 Multi-armed Bandit Problem

The search for new actions in an unknown environment versus doing the empirical best action is a dilemma between exploration and exploitation. The balance is searched between exploring new moves to find more profitable actions while doing the best move most of the time for aiming at the highest reward.

One simple example for this dilemma is called the multi-armed bandit problem. A fixed number of gambling machines (K) with different but unknown rewards are available to play. Successive play of each machine generates random variables ($X_{i,n}$, for $1 \leq i \leq K$ and $n \geq 1$ where i is the index of the machine and n is the number of times the machine was played) with an independent but unknown distribution for each machine.[ACBF02] The problem for a gambler is to choose the next machine to play. Choosing the machine with the highest mean reward discovered so far may not result in the highest overall reward because a machine with higher rewards might not have been discovered yet. But playing a new machine with unknown reward can generate a regret.

Auer et al. proposed in [ACBF02] the use of some policies that achieve logarithmic regret. They are simple to implement and have a complexity of $O(\log n)$. The first one is the UCB1 policy which was originally published by Agrawal.[Agr95] It is outlined in figure 3.1. The reward of the machine must lie in the interval $[0; 1]$. Before using this policy, every machine must be played at least one time to get information about the

reward distribution. This restriction can be replaced by setting the initial reward to a predefined value.

Initialization: Play each machine once
Loop: Play machine that maximizes

$$\overline{x}_i + C * \sqrt{\frac{\ln n}{n_i}}$$

where $\overline{x}_i$ is the mean reward obtained from machine i so far, n_i is the number of times machine i has been played and n is the overall number of plays done. The constant C must be adjusted for different problems.

Figure 3.1: UCB1 policy

Another simple policy proposed by Auer et al. is the ε-greedy policy. It switches randomly from exploration to exploitation and back. It is outlined in figure 3.2. The reward of the machine must lie in the interval $[0; 1]$ again. The parameters c and d must be adjusted for different problems.

Parameters: $c > 0$ and $0 < d < 1$
Loop: Define

$$\varepsilon_n = \min\{1, \frac{cK}{d^2 n}\}$$

- Let i_n be the machine with the highest current average reward.
- With probability $1 - \varepsilon_n$ play i_n and with probability ε_n play a random arm.

Figure 3.2: ε-greedy policy

3.2 Monte Carlo Tree Search

The MC tree search is a best-first search algorithm guided by the results of previous MC simulations. It runs in a loop with four steps as long as there is time left. These four steps are called selection, expansion, simulation and backpropagation and are visualized in figure 3.3. During selection, the tree is traversed from the root node to one leaf after

a selection strategy. The second step expanses the tree by one new leaf node. From the timepoint of teh game that is represented in the newly added leaf, the game is simulated until the end with self-play of a heuristic or random play. The result of the game is propagated back to the root by adding the game value to each node selected during the selection phase. If there is time left, all four steps will be run again.[CWUvdH07]

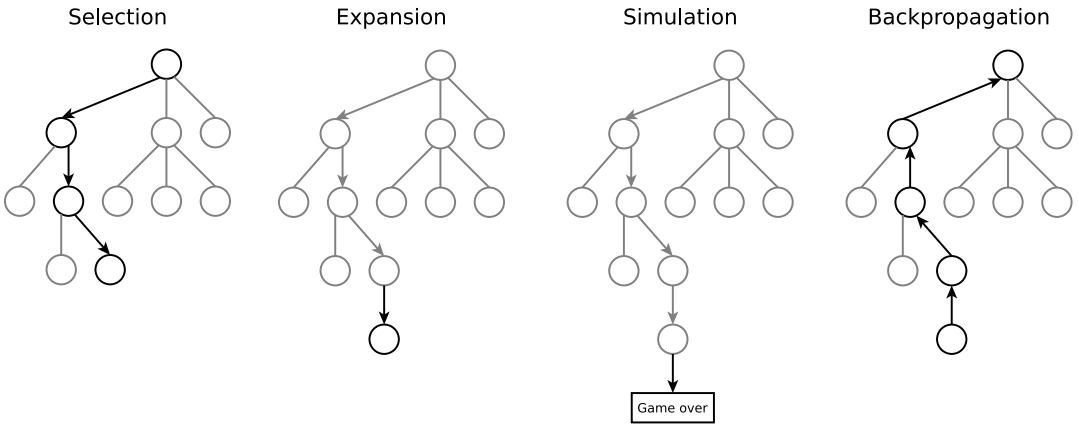

Figure 3.3: Monte Carlo tree search

3.3 UCT Algorithm Applied to Games with Perfect Information

The UCT algorithm is a MC tree search where the actions are sampled according to UCB1 policy described above. During the selection phase, the move with the best evaluation is chosen until a leaf node is reached. It must be guaranteed that every possible move from a node is investigated at least one time before the search is continued in a lower level of the tree. The sequence of moves from the root node to the leaf node is saved for the backpropagation step. After finishing the game, the game value is propagated back through the sequence of moves found during the selection phase. The number of visits is also incremented in each node. This value is very important for the UCB1 policy to decide whether to do exploration or exploitation. The approach was proposed by Kocsis et al. in [KS06].

The UCT algorithm was successfully applied to games with perfect information. It was used in the Go playing program MoGo. After the selection of a move sequence and the creation of a new leaf, the game is ended by random play to determine the game value. After propagating the game result back to the root node the simulation is runned again. MoGo is now known as one of the strongest Go programs. MoGo won several

tournaments against other AI players and has led the ranking lists on a Go server playing on a 9x9 and 13x13 board at the time of the publication.[GW06]

3.4 UCT Algorithm Applied to Games with Imperfect information

Until now, the UCT algorithm has not been applied to any game with imperfect information, yet. To use the algorithm, the unknown information must be replaced by simulated values. MC simulations are widely used in games with imperfect information if the true information is not available. The unknown positions are set randomly. Better results can be achieved when the random generation is biased with information that is already known. In my bachelor thesis, the generation of card distributions was biased with the information about the suits a player can no longer have. These suits are then not set for this player anymore. This approach leads to a higher playing strength because the simulations reflect the possible card distributions of the real world better than a truly random generation.[Sch05]

After the simulation of a possible world, the search tree can be traversed from the root node to the leaves. It is important to visit every possible move in a node first, before descending to a lower level in the tree. After the creation of a new leaf by making a move in a node, the game is ended by self-play. This can be done by playing random moves or letting heuristics play. The moves of this end game are not revealed. The game result is determined and propagated back as described above. These steps are carried out as long as there is time left or a maximum number of simulations is reached.

For every new iteration, a new world is generated by MC simulations and all worlds are stored in the same search tree. The search for the optimal move is done across all simulated worlds simultaneously and not only in a single world like the DDSS does. The branching factor of the search tree can be very high. Every possible move can be made in every possible world. If there are many potential worlds it is very likely that an unseen world is generated and all allowed moves have to be investigated in this world first before the algorithm can descend to a lower level of the search tree. The UCT algorithm will, therefore, not descend very deep.

Chapter 4

Skat Tournament Arena

For comparing the playing strength of different AI Skat players, a Skat tournament simulation program called Skat Tournament Arena (SkatTA) was implemented. The predecessor of this program was used in my bachelor thesis for comparing computer Skat players. Parts of the feature description here are therefore only a translation of the corresponding parts of that thesis. There are also a lot of new features that have been newly created for the diploma thesis. The main improvements are the introduction of a bidding and discarding interface for AI players, the two player compare option and the implementation of a graphical user interface (GUI) that allows games between AI players and humans.

4.1 Requirements of the Tournament Arena

For comparing the playing strength of different types of Skat players, it is necessary to let them play in a controlled environment. The following features are essential for a tournament arena:

- Playing all phases of a Skat game (bidding, discarding and trick playing)
- Defined interface for AI Skat players
- Free combination of different players in a series of Skat games and direct comparison of two player types
- Repetition of games with a different combination of player types by playing of predefined card distributions
- Collecting data for statistical analyses of the playing strength
- GUI for playing against human players

4.1.1 Skat Series for Evaluation of Play Strength

The playing of Skat series according to the rules of the ISkO is possible within SkatTA. For comparing the playing strength of different types of AI player, this option is not optimal because the chances to play the game as single player are not distributed equally over the sitting positions for one card distribution. In a normal Skat series, the sitting positions of the players change after each game and the cards are shuffled. The player have no chance of playing a card distribution on each sitting position. Normally, only 50 to 100 games are played per Skat series. The success of a player depends not only on the playing strength in this environment. The distribution of the cards is also important. A player who receives a lot of bad hands can not bid very high and, therefore, plays fewer games. The number of games is also too low for statistical analyses.

SkatTA has multiple features to eliminate this bias. The easiest way is to play more games. In this thesis 1,000 games are played in every experiment. Repeating all games with the same card distribution but with different sitting positions for the players, is another possibility. This is done because the dealt cards and the player positions forehand, middlehand and hindhand are directly linked in Skat. Every player should have the chance to play every card distribution on every sitting position against the other player types. The number of game repetitions depends on the number of different player types playing in the series. If only one player type is playing on each position, no game repeats will be necessary. For three different player types, six games are needed to let every player play on each position against every combination of the other players.

A two player comparison mode was newly implemented for this thesis. It is used to compare two player types directly without an interfering third player. The player combinations of such games are listed in table 4.1. For this purpose two instances of every player type are created. The results are combined for each player type at the end.

The bidding, discarding and game announcement can be done by a heuristic derived from the open source Skat program XSkat.[Ger04] It can be used for AI players that do not have a bidding and discarding module implemented. It is also used for comparing the playing strength during trick playing without looking at the bidding and discarding. The heuristic returns the same decisions for bidding, discarding and game announcement for all three players for a given card distribution every time. With this approach the card distribution has no effect on the outcome of the experiments, because every player plays the same games.

4.1.2 Payoff for Calculation of Play Strength

In a normal Skat series the game value is scored for the single player regardless if the game was won or not. After a fixed number of games that should be a multiple of the

Game no.	Player combination
1	AAB
2	ABA
3	BAA
4	BBA
5	BAB
6	ABB

Table 4.1: Player combinations for two player compare

number of all players on the table, all points are counted together and the player with the most points wins the series.

Official Skat tournaments have a different scoring mode for games. Usually only about 18 to 48 games are played per series. The normal scoring mode would be unfair for this low number of games because a player could get an advantage with a single high valued grand game. Therefore, the Seeger-Fabian-System was introduced for performance evaluation.[IW03] The single player will get additional 50 points added to the game value, if the game is won. If the game is lost, it will be charged with additional 50 minus points. The opposition players get a reward of 40 points for a game that the single player loses. With this system, the good play as opposition player is also taken into account. SkatTA supports both payoff calculations.

4.2 Implementation of SkatTA

The tournament simulation arena was implemented with C++. It has an object oriented design which has multiple advantages. SkatTA can call the player functions through a defined interface. New player types can be included easily by implementing this interface. The access to the data of an AI player is only possible through methods defined by the interface. With this, the AI player can use different codings for its internal data than the coding used in SkatTA. By implementing the interface it must only be assured that the results are returned according the coding used in SkatTA. The AI player gets information about changes in the game status through methods, too. These data can be checked for plausibility first and then the internal data can be adjusted accordingly. The second reason for choosing C++ as programming language was the reuse of the heuristics of the open source Skat program XSkat which is implemented in C. A change to another language family would have been too complex.

The classes in SkatTA are divided into three groups. On one side, there are the classes for the AI players. On the other side are control and helper classes. The control

classes manage the Skat series by card shuffling, dealing and giving the information about changes in the game status to the AI players. They also call the defined methods of the AI player interface to ask the AI players for their next move. The helper classes accumulate often used methods for input and output, some Skat rules and option handling. The class diagram in figure 4.1 shows the connections between the classes.

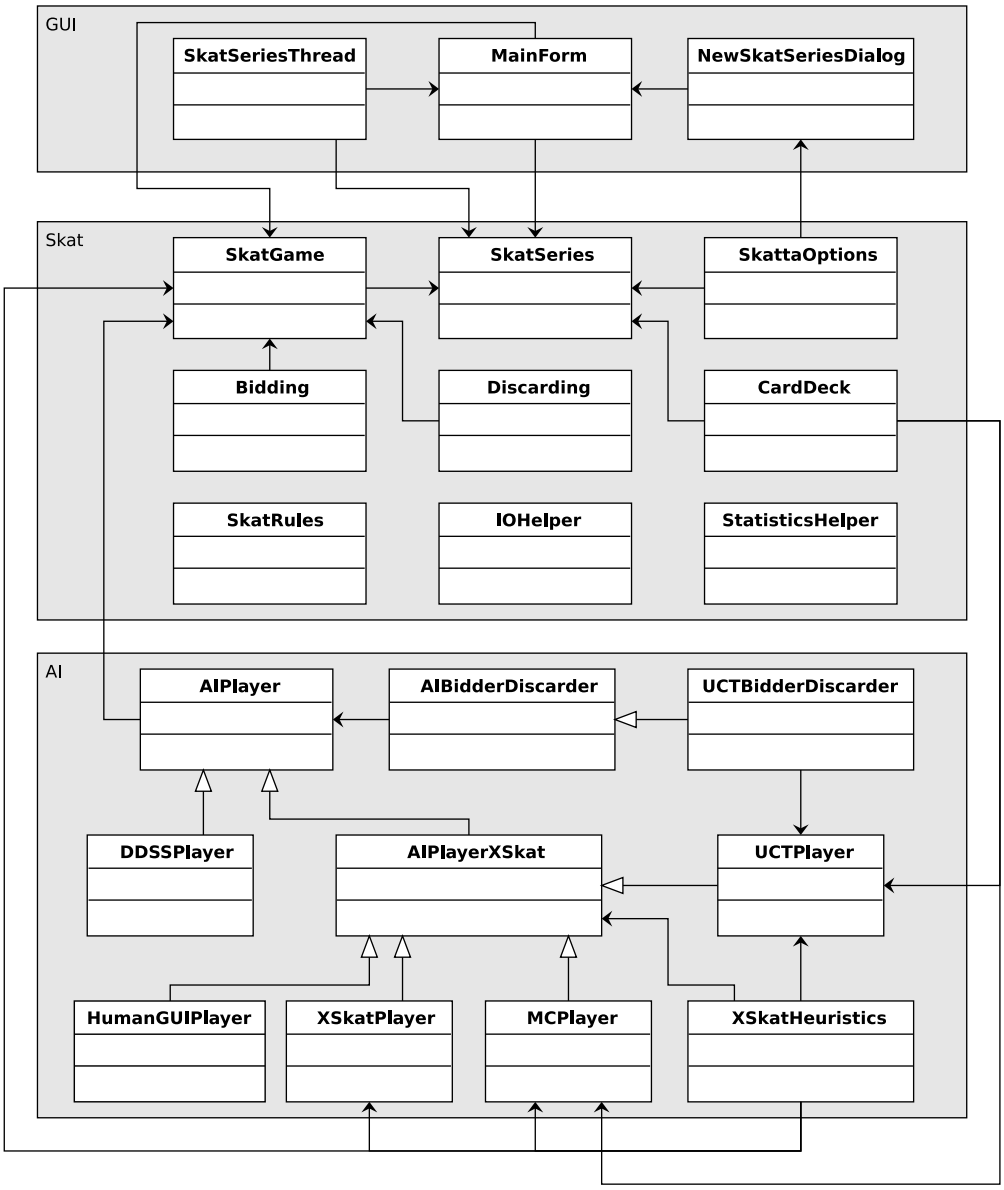

Figure 4.1: Class diagram of the Skat Tournament Arena

4.2.1 Representation of Skat Games, Skat Series and Cards

A Skat game is represented by instances of the class `SkatGame`. All data of a game like player types, game type, tricks and the game result are saved here. The coding of these data was taken over from XSkat. The Skat game object controls all phases of a game from bidding over discarding to the playing of the tricks and calculation of the game result. The bidding, discarding and game announcement can be done by a single heuristic derived from XSkat. This mode is used for experiments on the playing strength during trick playing. When the bidding and discarding is investigated all AI player bid and discard for their own. All results of the Skat game can be accessed by calling getter methods of the Skat game object.

The class `SkatSeries` holds all Skat game objects played in one series and controls them. Before the start of a new game, a card distribution is randomly generated or a predefined distribution is read from a text file and stored in an instance of the class `CardDeck`. After this step, the player objects will be instantiated. References to these four objects are then passed on to the Skat game object. If only trick playing is investigated, the heuristics derived from XSkat do the bidding, discarding and game announcement. The Skat game objects asks every player object for the next card to play during the trick playing. To prevent false play it checks whether the player can actually have the card or not. At the end of a trick, the trick winner is calculated and an information about the trick result is sent to all players. At the end of a game, the game results are polled by the Skat series object and saved in data files for later statistical analyses.

The cards in SkatTA are coded by integer values. The coding was adopted from XSkat. All cards of a suit are sorted down from the ace to the seven. The suits are sorted in the order ♢, ♡, ♠ and ♣. The coding is illustrated in table 4.2. The suit code can be easily calculated by an integer division by 8. The value of a card is obtained by a modulo operation with basis 8.

♢A	♢10	♢K	♢Q	♢J	♢9	♢8	♢7	♡A	...	♡7	...	♣7
0	1	2	3	4	5	6	7	8	...	15	...	31

Table 4.2: Coding of the cards in SkatTA

The cards of a Skat game are stored in an instance of the class `CardDeck`. The assignment of the cards to the players is also stored here. Unknown card positions are coded with -1. The class `CardDeck` not only saves card distributions it also generates possible card distributions for unknown card positions. A detailed description is given in the next section.

The correct shuffling of the cards is very important for generating unbiased card distributions. The shuffle algorithm from Fisher and Yates is used in SkatTA.[FY38] It was originally published in [Dur64] for the use as computer algorithm but became more famous after Knuth described it.[Knu98] Today, it is also known as Knuth shuffle. Figure 4.2 shows the algorithm in pseudo code. It has a complexity of $O(n)$ and is known as unbiased shuffle algorithm when implemented in the right way and when a good random number generator is used.

```
void shuffleCards(int * cardArray) {

    int random;

    for(int i = 31; i >= 0; i--) {

        rand = getRandomIntegerInRange(0, i);

        swapCards(cardArray[i], cardArray[rand]);
    }
}
```

Figure 4.2: Shuffle algorithm used in SkatTA

4.2.2 Generation of Card Distributions

After dealing the cards, the player only knows the ten cards on the player's hand. If looking into the Skat after winning the bidding, two new cards will be revealed. The cards of the other players are disclosed, too, during the trick playing. For calculating a play strategy it is, therefore, important to generate only distributions of the unknown cards that are possible and reflect the current game status.

The simplest algorithm is the random distribution. All unknown cards are determined and assigned randomly to the unknown card positions. This algorithm can be used at the beginning of the game when no information about the unknown cards is available.

During the bidding and the trick playing, the other player reveal some information about their cards. When a player bids until 23 and passes at 24, it is fairly certain that this player wanted to make a null game. A similar example is when a player bids until 24 and passes at 27. This player has a club game on the hand with or without the ♣J. In the trick playing phase, the players have to follow the suit of the lead card of every trick. If a player cannot follow the suit, it is certain that this player will not have the suit anymore. Otherwise, the player would have played "schwarz" and the game would be lost.

This information can be used to bias the generation of the card distributions. Only the assured information is used in SkatTA. The players save information about all players independently. Every player saves which suits the other players could still be holding on the hand. This information is then used during the generation of the card distribution to assure that a player with unknown cards only receives cards from suits that can still be on the player's hand. In appendix A the algorithm is described in detail.

4.2.3 Interface for AI Players

All AI player classes are inherited from the class `AIPlayer`. This class defines the methods that must be implemented by the AI player and that are used by SkatTA to communicate with the player objects. Some methods must be overwritten by the AI player classes if a different coding of the game status is used internally. All methods defined by the class `AIPlayer` are listed in table 4.3.

Game phase	Method	Description
Bidding	`newGame()`	Sets the AI player back to the initial status.
	`takeCards()`	Deals the cards to the AI player.
	`startBidding()`	Informs the AI player about the start of the bidding phase.
	`declareBid()`	Asks the AI player whether to declare the next bid level or not.
	`holdBid()`	Asks the AI player whether to hold a bid or not.
Discarding	`lookIntoSkat()`	Asks the AI player whether to look into the Skat or not.
	`takeSkatCards()`	Shows the Skat cards to the AI player.
	`discardCards()`	Asks the AI player to discard cards.
	`announceGame()`	Asks the AI player to announce a game.
Playing	`startGame()`	Informs the AI player about its sitting position, the game type, the single player and the first forehand player.
	`nextMove()`	Informs the AI player about a move done by another AI player.
	`trickCompleted()`	Informs the AI player about the trick winner at the end of a trick.
	`makeNextMove()`	Asks the AI player to play a card.

Table 4.3: Interface for an AI player

The methods where the AI player is asked to make a move, are not implemented in the class `AIPlayer`. They have to be implemented by the AI players themselves. Here,

the AI players calculate the next moves in the different game phases. All other methods are used to pass information to the AI player objects. They are fully implemented in the class `AIPlayer` and should only be overwritten if the AI player has to do additional tasks when the information arrives. The player's knowledge about the game status is also stored in the AI player objects. Information about the game type, all tricks and the cards revealed by the other players so far is saved here.

4.2.4 Inclusion of XSkat

XSkat is an open source Skat program. The source code can be used and modified freely.[Ger04] For SkatTA the version 4.0 of XSkat was used to derive a class of Skat heuristics. These heuristics are rule based. For every situation in a Skat game expert rules are hardcoded. In some situations random decisions are made.

All parts of the XSkat-GUI and the network play functionality were removed from the source files first. Then, everything was capsulated in a class called `XSkatHeuristics`. Instances of this class are capable to play a complete Skat game for three players. More interesting is the usage of special methods for bidding, discarding or the search for playable cards. This is easily done by setting up the current game status in the XSkat object and calling the desired methods for the next moves.

4.2.5 Interface for XSkat Based AI Players

Some of the implemented AI players use an instance of the class `XSkatHeuristics` for the calculation of their strategies. For this it is essential that the current game status is also reflected in the data of the XSkat object. The class `AIPlayerXSkat` extends the class `AIPlayer` by holding an instance of the XSkat heuristic class and adjusting the data in the XSkat object according to the current game status changes.

4.2.6 AI for Bidding and Discarding

The AI for bidding and discarding is implemented in a seperate class. This gives the chance of exchanging these classes between the players and combining different approaches for bidding, discarding and playing. For this thesis, two bidding and discarding classes were implemented:

- The `RNDBidderDiscarder` is a random bidder and discarder. It was also used for testing the functionality of SkatTA.

- The application of the UCT algorithm to the phase of bidding and discarding resulted in the `UCTBidderDiscarder`. It is described in detail in the sections 5.1.1 and 5.1.2.

Before the bidding starts, the method `newGame()` of every player object is called by the Skat game object. The players are in an initial status afterwards. The cards are dealt by the method `takeCards()` and the information about the start of the bidding is given by the method `startBidding()`.

The methods `declareBid()` and `holdBid()` of the AI players are called by the Skat game object during the bidding. The call is handed over from the player object to the bidding and discarding object. It decides whether to do the bid or not. The result is returned to the player object that returns it to the Skat game object. The discarding works in the same way. The Skat game object calls the method `lookIntoSkat()` from the winner of the bidding. Here the player decides whether to look into the Skat or not. If so, the cards of the Skat are passed to the player by the method `takeSkatCards()` and the discarding is started by the call of the method `discardCards()`. Finally, the method `announceGame()` is used to ask the player for the game announcement.

4.2.7 AI for Trick Play

In SkatTA there are currently 6 different AI players implemented:

- The `XSkatPlayer` plays only according to the heuristics of XSkat.
- In the class `MCPlayer` the approach of using MC simulations with imperfect information is implemented. This AI player is the result of the work done for my bachelor thesis. It uses some instances of the XSkat heuristics for the calculation of the next moves.
- The `DDSSPlayer` is the implementation of the DDSS from [Kup03]. It is directly derived from the class `AIPlayer` because it does not need the XSkat heuristics.
- For testing purposes the `RNDPlayer` was implemented. It plays strictly random and was used to test the functionality of SkatTA. It can also be used for comparison with other AI players which should play better than a random player.
- The `HumanGUIPlayer` is not an AI player. This allows for games between AI players and one human player over the GUI described below.
- The work for this diploma thesis resulted in the `UCTPlayer`. The player uses the UCT search for finding the next move. A detailed description of its implementation is given in section 5.1.5.

The Skat game object calls the method `makeNextMove()` to ask the player objects which card they want to play. The information about the played card is handed over to all players by calling the method `nextMove()`. At the end of the trick, the trick winner is calculated by the Skat game object and this information is returned to all players by the method `trickComplete()`.

4.2.8 Graphical User Interface

For letting the AI players play against humans it is essential to have a GUI. The GUI for SkatTA was implemented with the use of the Qt library from Trolltech.[Tro07] All phases of a Skat game can be played.

To start a Skat series all parameters for the series and the Skat players must be set. This is done with a dialog box shown in figure 4.3. The number of games can be set. It is also possible to select a textfile with predefined card distributions. The bidding can still be done by the heuristic of XSkat. But more interesting is the bidding done by the human and AI players themselves. This can be activated by selecting the bidding check box. The first forehand player can be defined as well as the frequency of the central processing unit (CPU)[1]. The parameters for the Skat players can be set in three different tab pages below. For Player 0 and Player 1 only AI players can be selected. Player 2 can be a human player, too. After selecting the player type only the valid parameters for this player type are activated.

If only AI players are playing, the Skat series will run to the end without stops. In case of a human player playing, the GUI waits for the input of the human player and stops for a moment at the end of a trick. This gives the human player the chance to follow the game. For this purpose the classes for bidding, discarding, the Skat game and the Skat series had to be implemented as threads. They are inherited from the class `QThread` from the Qt library. Now, it is easy to stop the execution of a method and to wait for an event or input by the human player. When the human player has to make a move, the threads stop. After the human player decided for a move, the thread is woken up and the move is processed. One example is the discarding by the human player. The method `discardCards()` is called by the Skat game object and the GUI shows the Skat cards as pictured in the screenshot in figure 4.4. The bidding thread stops and the human player can exchange some of the cards from the player's hand with cards from the Skat. When the discarding is done the button labeled "Done" is pressed. Now, the thread is woken up and the discarded cards are determined.

[1]This option is used for better compare of time limited experiments run on different machines. The reference machine is an AMD Athlon XP with 3,000 MHz. On faster machines, the time for simulations set in the options is reduced to achieve the same number of simulations as on the reference machine. The time is extended on slower machines.

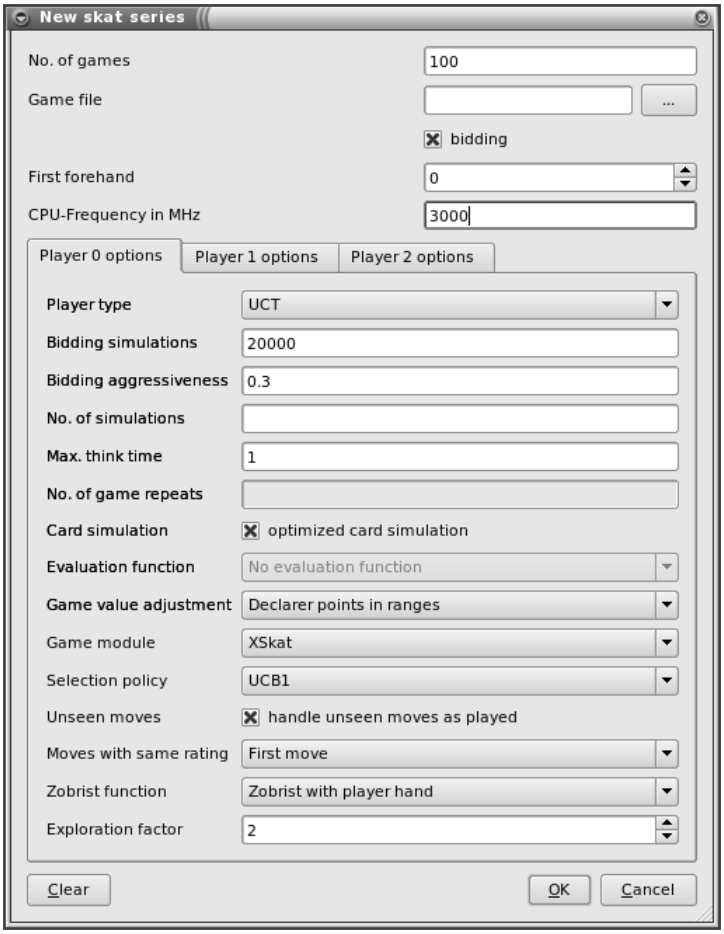

Figure 4.3: Dialog for setting all parameters for a new Skat series

After the game announcement, the tricks are played. The active player is highlighted in a light yellow background. The current forehand player is highlighted in red. If a human player is playing in the series, SkatTA waits for one second after each move of an AI player object and two seconds after each trick. At the end of the game, the results are calculated and displayed in the table in the middle of the GUI. New games are started until the maximal number of games is reached or all card distributions from a textfile are played.

4.2.9 Helper Classes

There are several helper classes called `SkattaOptions`, `StatisticsHelper`, `Random` and `IOHelper`. They provide public methods for all classes included in SkatTA. Some do not have to be instantiated as objects. Their functions are then implemented as static methods.

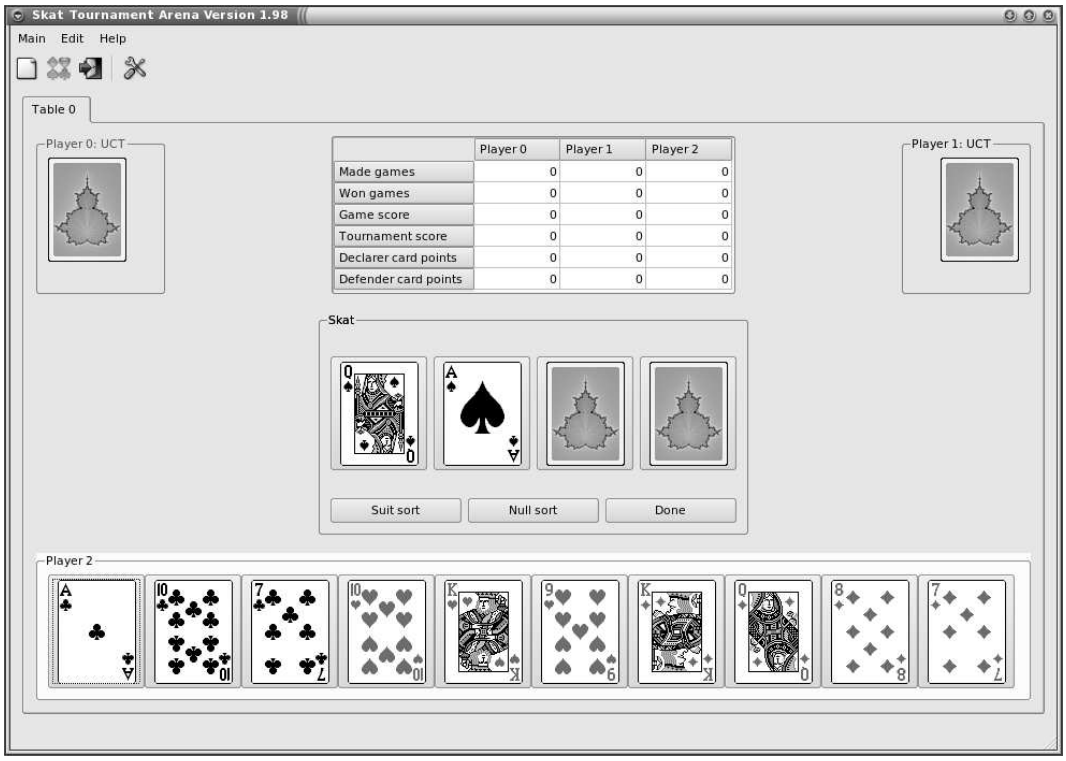

Figure 4.4: Discarding phase for a human player

The current options are stored in an instance of the class `SkattaOptions`. It saves the options given as command line parameters at the start of the SkatTA or the options set in the dialog box described above. When a Skat series starts, the object with the options is passed over to the Skat series object and acts as a container for all options. It holds information about the number of games to play, the file name of a text file with predefined card distributions and all player options. It is easily extendable with new options. This can be done without changing the interface of the Skat series object.

The class `StatisticsHelper` computes statistical values like the mean or median value of data in an array. These statistical values are used during simulations for the selection of the best card. All methods are implemented as static methods

The `Random` class provides methods for generating random values in the interval $[0; 1]$ or for selecting a random index of a array or vector. This class has not to be instantiated as a object, too. All methods are static.

The methods of `IOHelper` are used for the input and output of cards. The cards can be printed out in human readable form and can also be put in in the same way. The methods of this class perform the coding into the internal coding of the cards. This class has also only static methods.

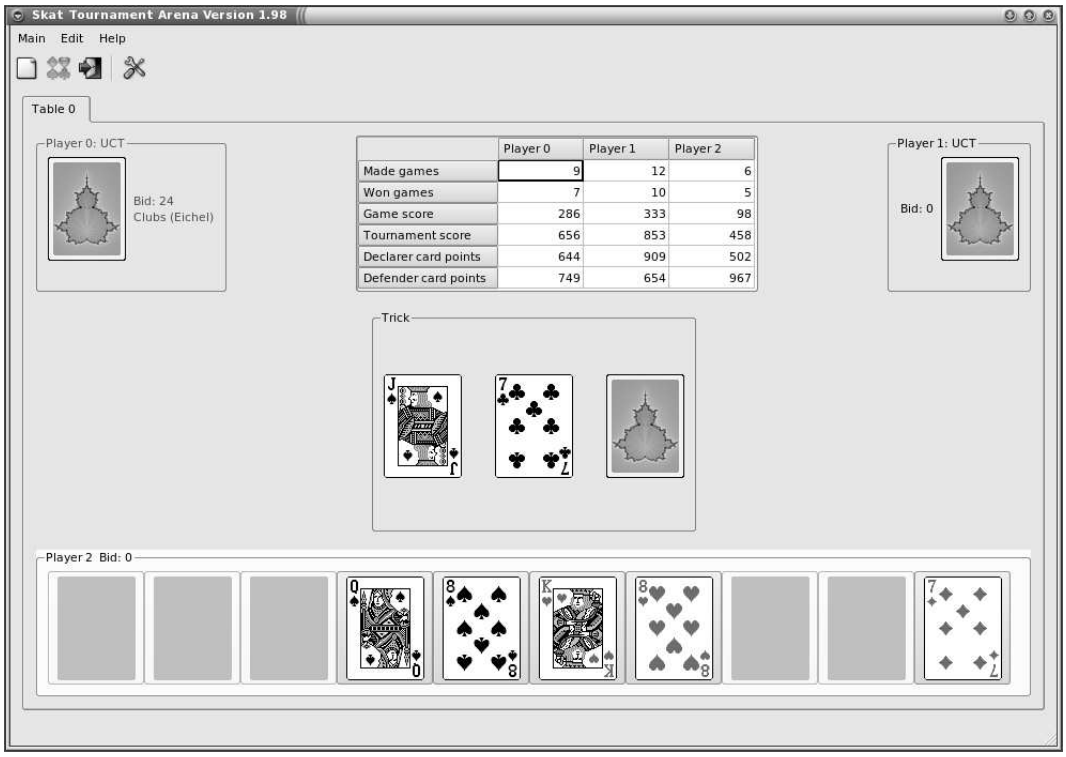

Figure 4.5: Playing the tricks

4.3 Data Output

While the games are played, the results of all games are written to a data file for later statistical analyses. This file contains the game number, the game type, the player signatures[2], a flag whether the player played the game as single or opponent player, the points achieved by the players as single or opponent player, the result of the game (won/lost) and, finally, the game value in normal and tournament calculation.

These data can be analyzed in many ways with the help of the statistical package R. First, a descriptive analysis is done. The rate of won games is calculated as percentage of games won out of all games done by the player. For the other values the mean value is computed. The difference between two players is investigated by statistical tests. As preparation for the tests the results of all repeated games under one card distribution are aggregated. The number of won games per card distribution are summed up. The resulting variable has the four values 0, 1, 2 and 3 because every player in a two player compare plays three games per card distribution. The frequency of these values is counted over all card distributions and then tested for difference with McNemar's χ-squared test. Since the result of one player depends on the result of the other players and vice versa

[2]A signature contains the player type and all parameters that define the player type.

this test is the appropriate test to analyze these data. For the other values the mean within one card distribution is computed first. Then, the distribution of these means is tested by a two sided paired t-test. A significant level of 5% is chosen.

All card distributions are saved for reuse in text files, too. There are different card distribution files. Every card distribution is saved in a file called `all_games.txt`. This file can be reused in later experiments for testing the same card distributions on players with different parameters. An extra card distribution file is generated for every game type (`suit_games.txt`, `grand_games.txt` and `null_games.txt`). If a suit or grand game is lost but was very close to a win[3] the card distribution of this game will be saved in a file called `close_games.txt`. These close games can be used during optimization of the AI player to see whether an optimization can lead to more won games or not. All lost games are saved by the Skat series object in the file `lost_games.txt`.

For the experiments presented in this thesis, the same set of 1,000 card distributions was used every time. It was already used for the experiments in my bachelor thesis. Every card distribution in this set is a playable Skat game but there is no guaranty for the win of this game. When XSkat is used for bidding, discarding and game announcement and no repetition of games is used, the games are nearly distributed equally over all players. Null games are not very present in this set with only 6 games. This can be explained by the bidding strategies of XSkat which bid very rarely for a null game. The other game types are split into 255 grand games and 734 suit games with 135 ♢ games, 175 ♡ games, 204 ♠ games and 220 ♣ games. When the bidding and discarding is carried out by the AI players themselves, the distribution of the game types varies from experiment to experiment.

[3]These are suit or grand games with 55 to 60 points for the single player.

Chapter 5

Implementation of the UCT Algorithm for Skat

In this chapter the implementation of the UCT algorithm for the game of Skat is described. There are different usages of the algorithm for the three phases bidding, discarding and trick playing.

5.1 UCT Player

The UCT player is inherited from the class `AIPlayerXSkat` because the heuristics from XSkat are used to simulate the games instead of random play. During the bidding and discarding phase, the player simulates different decisions under simulated card distributions. During trick playing, the player performs a MC tree search with information sets and the UCB1 policy is used as evaluation function. For each move decision a limited number of simulations is done. The number of simulations is limited either by a fixed number or by a limited computing time. In time limited simulations the computing time is determined after each simulation and a new simulation is started if there is time left. After all simulations are done the move with the highest mean game value is chosen for play out.

5.1.1 Bidding

The UCT player has more than one chance to carry out simulations during the bidding. A tree search is not needed for the bidding phase because only one decision has to be simulated. Everytime the player is asked do declare or hold a bid, game simulations are done. In every simulation a possible card distribution is generated, the cards in the Skat are exchanged with cards from the player's hand randomly, a game is announced

and then played to the end by the heuristics derived from XSkat. Which game type is chosen is decided by the UCB1 policy. It assures that every game type was played at least once. After this, the best game type is chosen while still investigating other games types from time to time. During the simulations only these game types are tested which are possible under the current bid level. This are game types with the same or a higher game value than the current bidding value are allowed.

To calculate the maximal bidding value the multipliers must be calculated after the ISkO rules. The multipliers are then multiplied with the factor for the game type. The resulting value is the maximal bidding value for a suit or grand game. Null games have special bidding values. To prevent the player from bidding too high it is neccessary to adjust the value for the multipliers. A hand with four jacks and a hand without four jacks have the same value for the multipliers but the latter might have fewer trump cards wich lead more often to lost games. By adjusting the value for the multipliers it is assured that the UCT player stops bidding when the number of won games is too low and, therefore, the chance of winning is low. The complete formula is

$$\left\lceil m * \frac{w}{n} \right\rceil * f$$

where m is the value for the multipliers, n is the number of games made, w is the number of games won and f is the factor for the suit or grand game from table 2.3. Null games are also considered. If half of the simulated null games are won a normal null game will be possible. If 90% of the games are won a null hand game will be suspected.

5.1.2 Discarding and Game Announcement

Before a player object can discard cards it has to decide whether to look into the Skat or to do a hand game. Therefore hand games are simulated for all possible game types. The game type for a simulated game is chosen by the UCB1 policy again. The UCT player looks into the Skat when less than 95% of the games were won during the simulations in the bidding phase.

For the decision which cards to discard, every possible discarding is tested against possible card distributions. There are 96 different possibilties to discard two cards including the situation where the cards of the Skat are not taken. There are six different game types to announce: null, grand and the four suit games. To prune the number of discardings only game types are investigated that are possible to play under the current biding value. Game types with lower game values will not be included in the simulations.

For discarding the UCT algorithm works fine again. For every simulated card distribution a combination of a pair of cards to discard and a game type to play is picked from the available discardings and game types. The discarding and game announcement

is done and the heuristics play the game to the end. The game result is saved for the pair of cards and the game type. UCT assures again that every possible combination is investigated at least once. After that the best combination is further investigated in new card distributions most of the time.

5.1.3 Comparison with Bidding of XSkat

For comparing the bidding with the UCT algorithm to the bidding of XSkat a Skat series with three equal players was set up. The first play of the 1,000 standardized games was run with XSkat bidding. After this, the set of games was run again under bidding according the UCB1 policy. The results are listed in table 5.1. The numbers in parantheses are the percentage or the mean value of the measured value under all games made.

Measured value	XSkat bidding	UCT bidding
Made games	1,000	337
Won games	852 (85.2%)	318 (94.36%)
Game score	32,709 (32.71)	14,804 (43.93)
Tournament score	79,749 (79.75)	31,274 (92.80)
Declarer card points	77,561 (77.56)	28,037 (83.20)
Defender card points	84,878 (42.44)	24,650 (73.15)

Table 5.1: XSkat vs. UCT bidding

Unfortunately, this approach leads to weak results for the overall playing strength. The player using this bidding strategy often passes the bidding and, therefore, makes fewer games than it could. In the end, the player has won nearly all of the games it made but the number of games is too low to win the whole Skat series. It is not only important to win all the games in Skat. The number of games made is also important. To make many games and lose some of them is better than making few games and win them all. By looking at the results of the simulations two groups of card distributions can be seen. In card distributions where it is very clear which game type is the best, this approach works perfectly. But there are a lot of card distributions where more than one game type is possible. In these cases the UCT algorithm cannot decide which is the best game type. The results are getting better if more simulations are done.

5.1.4 Search Tree with Information Sets

The search tree is composed of nodes with game status data. The nodes contain data about all tricks played so far, the current player that has to make the next move and all

cards playable by the current player. At the beginning of the selection phase a possible card distribution is generated. During the selection the tree is traversed from the root node to the leaves under this card distribution. At every node, the best move so far is chosen according to the UCB1 policy. When a leaf node is reached a card from the list of cards that have not been investigated yet is played. The game is then played until the end with the help of the heuristics of XSkat. The game result is returned from the heuristics and propagated back to the root node. The game value is added to every node in the sequence which was built up during the selection phase. The number of visits of every node is incremented too. This number is used to determine nodes that have rarely been investigated. They will occasionally be selected even if they have a bad reward.

In the search tree for the UCT player the nodes are grouped in so called information sets. This means that nodes containing the same data are stored only once and are used and updated every time in the search tree where the same game status is reached. This idea was proposed by Blair et al. in [BMvL96] for pruning in games with imperfect information. The problem while selecting new nodes in the selection phase is to determine whether the the game status has been seen before or not. This is done by calculating a hash value from all game status data and storing the nodes under this hash value. For this, a hash function must be found that returns distinctive hash values for every possible game situation. The Zobrist hash does exactly what is needed here. It assigns big random numbers to all possible moves during a game. Then, all random numbers for the moves that have been seen so far are combinded with the `XOR` function.[Zob70] After that, the player position for the current player is added in the same way. Finally, all possible cards playable by the current player are used in the hash calculation. If the node has been seen before, the already stored data will be used. Otherwise, a new node is created and stored under the calculated hash value. The calculation of the Zobrist hash works correctly only if no random number is used twice. This is assured by predefined random numbers that are tested for uniqueness.

Regarding the tricks 960 different random numbers are needed for the hash calculation. This number results from the ten tricks, the three players who play one card per trick and the 32 different cards that can be played by each player in every trick. There are 90 random numbers used for calculation of the player position. This is due to the fact that every player can be forehand, middlehand or hindhand in every trick of the game. Finally, 96 random values are needed for calculation of the player cards. This is due to the 32 different cards a player could be able to play at a given stage in the game.

5.1.5 Playing the Tricks

For the decision which card to play, the UCT player simulates possible games. This is done by simulating a possible card simulation, descending the search tree to a leaf,

playing out one card and letting the XSkat heuristics end the game. The game result is propagated back through all moves seen in the selection phase. During the first simulations it is assured that every possible card is played at least once in each simulated card distribution. After this, only the best evaluated card is chosen for the next simulations. The policy used for evaluation take care of a good balance between exploration and exploitation. So, not only the best card so far is chosen. Sometimes, rarely investigated cards are chosen, too, to test whether they might be better in a deeper investigation of the search tree or not. After all simulations are done, the card with the best mean reward is chosen for play out.

5.2 Comparison with Monte Carlo Simulation

The comparison of the UCT algorithm with MC simulations was carried out with the AI player implemented for my bachelor thesis. It does MC simulations to fill the gaps of unknown cards in the card distribution. After this, every possible card is played out in a separate game. The games are finished by the heuristics derived from XSkat. The game results are saved. After all simulations the card with the best results throughout all games is chosen.[Sch05]

To compare this approach with the UCT algorithm a special selection policy called MC policy was implemented. During the simulations the UCT player does not descend the search tree. The MC policy chooses the best card according the UCB1 policy from all possible cards only once. After this, the game is finished again by the XSkat heuristic. The result is returned and saved for the next simulation. Instead generating a card distribution and testing it with every card $s * p$ card distributions are generated where s is the maximal number of simulations for the current experiment and p is the number of playable cards.

In all experiments on comparison the strength of two AI players a set of 1,000 predefined card distributions was played in two player mode. This mode is described in detail in Section 4.1.1. The results are shown in Table 5.2. Five experiments with different limitations of the number of simulations are done. The experiments S_{100}, S_{500}, S_{1000} and S_{2000} have a limited number of simulations from 100 to 2000 simulations per move decision. The experiment T_1 is a time limited experiment with an 1 second time limit. Other time limited experiments could not be done due to the lack of time at the end of the editing time of the thesis. The listed results are the percentage or the mean value of the measured value under all games made. The UCT player is able to play as good as the MC player. There is no statistical difference in the values for won games, game score and tournament score. But the UCT player gains this result with making fewer card points during a game. This can be seen in the values for declarer and defender card

points. These differences are highly significant with p-values < 0.001. The decrease of the values in experiments with higher number of simulations can be explained by the fact that with more simulations the defender also gain a higher playing strength. Therefore it is harder vor the declarer to make points and to win the game in the end.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
MC player	Won games	86.3%	83.6%	84.2%	83.3%	83.5%
	Game score	33.78	31.10	31.01	30.62	31.08
	Tournament score	27.41	26.04	26.03	25.70	26.01
	Declarer card points	80.46	78.96	78.88	78.57	78.70
	Defender card points	42.80	43.74	43.82	44.06	44.30
UCT player with MC policy	Won games	84.7%	84.0%	84.1%	84.4%	83.8%
	Game score	33.03	31.64	31.97	32.41	31.74
	Tournament score	26.26	26.05	26.18	26.49	26.09
	Declarer card points	76.26	75.47	75.36	75.19	74.89
	Defender card points	40.48	41.83	41.95	42.18	42.11

Table 5.2: MC player vs. UCT player with MC policy

Chapter 6

Optimizations

The performance of the UCT algorithm can be optimized in several ways. In this chapter another selection policy and better game value adjustments are investigated. An optimized simulation of card distributions is tested, too. The last optimizations are random move selection for equally rated moves, the change of the factor that controls the balance between exploration and exploitation and another hashing function for a game status.

6.1 Selections Policies

The selections policies in this thesis use the game results of simulated games played so far for finding the best move at a current stage in the search tree. The game value of a Skat game depends on the game type and in suit and grand games on the number of multipliers and winning levels. The game value can reach from some hundred minus points to some hundred plus points. The selection policies only accept values in the interval of $[0; 1]$. Therefore, the game values have to be adjusted to this interval. The easiest adjustment is setting the game value to 0 for a lost game and setting it to 1 for a won game.

6.1.1 Epsilon Greedy Policy

The ε-greedy policy is adjusted by two parameters c and d where $c > 0$ and $0 < d < 1$. The best combination of both was found in several experiments by letting c go from 1, 2, 4 and 8 and letting d take the values of 0.1, 0.25, 0.5, 0.75 and 0.9. The best combination was $c = 4$ and $d = 0.5$. The results are shown in table 6.1. With more simulations the differences between the UCB1 policy and the ε-greedy policy become smaller. In the experiment T_1 all p-values lie above 0.5. Therefore, a difference can

not be seen statistically. The UCB1 policy and the ε-greedy policy are equivalent when enough simulations are performed.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	87.4%	84.7%	83.2%	81.6%	81.0%
	Game score	32.58	30.05	27.69	26.55	26.28
	Tournament score	26.81	25.79	25.12	24.37	24.26
	Declarer card points	76.16	75.05	74.47	74.03	73.89
	Defender card points	43.82	44.61	45.46	46.10	46.22
UCT Player with epsilon greedy policy	Won games	86.6%	83.8%	81.4%	81.2%	80.7%
	Game score	31.86	29.26	27.12	26.27	25.91
	Tournament score	26.27	25.21	24.12	24.09	23.91
	Declarer card points	76.29	75.50	74.63	73.95	73.90
	Defender card points	43.74	44.84	45.43	45.92	45.99

Table 6.1: ε-greedy policy with $c = 4$ and $d = 0.5$

6.1.2 Monte Carlo Policy

The MC policy is not really following the principle of descending the tree. Applying this policy, the best card is chosen only in the first level of the search tree. The decision is made after the evaluation according to the UCB1 policy. The card is played out and the game is simulated to the end. This is compareable to the behavior of the MC player created during my bachelor thesis.[Sch05]

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	82.9%	80.7%	80.5%	80.4%	79.7%
	Game score	26.90	25.42	25.58	24.52	24.98
	Tournament score	23.58	23.32	23.57	23.60	23.64
	Declarer card points	73.80	73.12	73.06	72.72	73.26
	Defender card points	43.68	45.62	46.11	46.87	47.09
UCT Player with MC policy	Won games	87.1%	83.0%	81.4%	79.7%	79.6%
	Game score	32.40	28.80	26.07	24.20	24.32
	Tournament score	27.52	25.67	24.44	23.32	23.41
	Declarer card points	76.99	74.60	73.93	73.19	72.65
	Defender card points	45.53	46.66	46.89	47.22	47.00

Table 6.2: Monte Carlo policy

The results between a simple UCT player and an UCT player that uses the UCB1 policy for choosing moves but does not descend the tree are shown in table 6.2. Only the means of the declarer points in the experiment T_1 differ statistically significant ($p = 0.02$).

6.2 Game Value Adjustment

The selection policies described in the former section use the outcome of the simulated games for finding the best moves. The simplest adjustment is assigning 0 to a lost game and 1 to a won game. Now, the outcome of all game types is compareable with the policies. In suit and grand game the player can gain higher game values if the opponent players are playing "schneider" or "schwarz". The game value adjusted to the interval $[0; 1]$ can also include this information instead of the binary won/lost information.

6.2.1 Game Value in Ranges

The points a player gains in a suit or grand game as single player are directly linked to the winning levels won, "schneider" and "schwarz". The single player can make a maximum of 120 points in a game. In suit and grand games this is the same as the winning level "schwarz". The winning level "schneider" is reached with 119 to 91 points. The range from 90 to 61 points lead to a normal won game. The same winning levels can be found for lost games. In the end there are the following winning levels: -8, -4, -2, 1, 2, and 3. In figure 6.1 the adjustment of the winning levels to the interval of $[0; 1]$ is illustrated as step function. Null games have still the binary representation of 0 for a lost and 1 for a won game because as soon as the single player gets a trick the game is lost regardless of the points in this trick.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	85.6%	83.5%	83.6%	82.7%	81.4%
	Game score	30.64	28.62	28.09	26.85	25.69
	Tournament score	24.99	24.20	24.37	23.57	23.35
	Declarer card points	75.13	74.53	74.04	73.62	73.37
	Defender card points	41.32	42.84	43.40	43.70	44.42
UCT Player with game value in ranges	Won games	90.7%	89.0%	87.2%	87.2%	85.4%
	Game score	39.27	37.65	35.89	35.71	33.01
	Tournament score	30.06	29.39	28.34	28.61	27.34
	Declarer card points	80.37	78.57	78.07	77.62	76.78
	Defender card points	43.18	44.06	44.48	45.06	45.43

Table 6.3: Game value in ranges

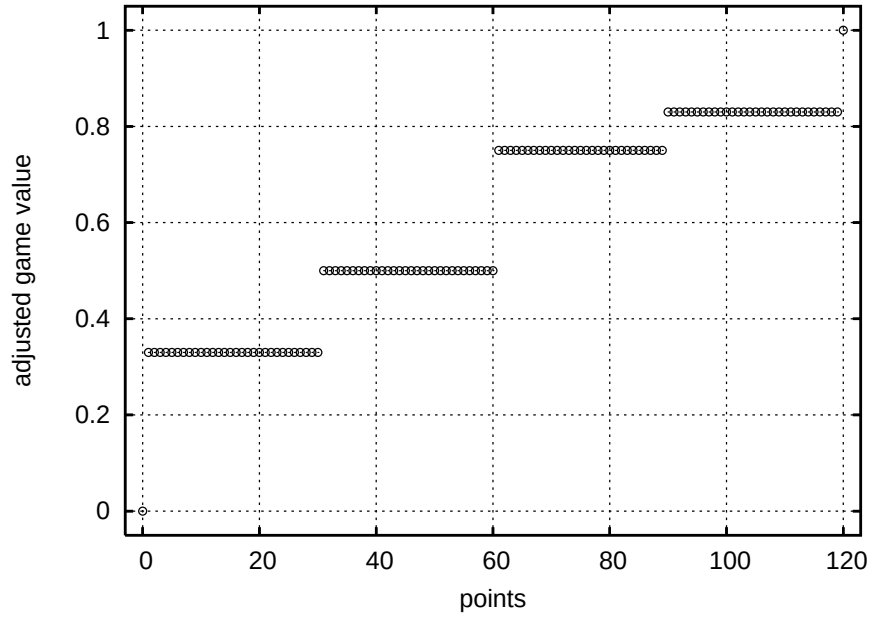

Figure 6.1: Step function for game value adjustment

The results in table 6.3 show a difference between the two players which highly significant. Every measured value differs with a p-value < 0.001.

6.2.2 Single Player Card Points

The game value adjustment in ranges has a disadvantage. The points of a single player at the beginning of a step are evaluated with the same value as the points at the end of a step. In this subsection the single player points are directly adjusted to the interval of $[0; 1]$ by dividing them by 120. The resulting function has no leaps. Higher points of the single player are evaluated with a higher value.

The results in table 6.4 show again an improvement in the playing strength in comparison to the UCT player without game value adjustment. The differences are again highly significant. It is interesting that the player with the game value adjusted to ranges gains less card points as single player and still plays with the same strength.

6.3 Combination of Selection Policy and Value Adjustment

The selection policies are using the game values of the simulated games to decide which move is the best at a current stage. For the policies investigated in this thesis the game

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	86.1%	83.1%	84.0%	82.8%	81.2%
	Game score	30.43	27.70	28.82	27.33	25.94
	Tournament score	24.83	23.72	24.59	23.89	23.32
	Declarer card points	74.97	73.78	73.62	73.22	72.76
	Defender card points	38.68	40.70	41.37	42.27	42.83
UCT Player with single player points divided by 120	Won games	91.6%	89.0%	87.6%	86.7%	85.7%
	Game score	40.61	37.50	35.80	34.60	32.59
	Tournament score	30.70	29.49	28.39	28.05	27.32
	Declarer card points	84.23	82.07	81.09	79.85	79.18
	Defender card points	42.12	43.45	43.92	44.66	45.23

Table 6.4: Single player points divided by 120

value is adjusted to the range of $[0, 1]$. Until now, it is not known which selection policy is best combined with which game value adjustment.

6.3.1 Epsilon Greedy Policy

First, the ε-greedy policy was combined with the game value adjustment in ranges and with the division of the single player points by 120. The former gained better results than the latter. But it can still not compete with the combination of the UCB1 policy and the division of the single player points by 120 as shown in the previous section. The results are presented in table 6.5 anyway.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT player	Won games	85.9%	83.0%	82.3%	82.3%	81.7%
	Game score	30.42	28.47	27.12	27.24	26.95
	Tournament score	25.06	23.94	23.57	23.66	23.83
	Declarer card points	75.30	74.44	74.03	73.98	73.87
	Defender card points	41.26	42.70	43.29	43.73	44.48
UCT player with ε-greedy policy and points in ranges	Won games	90.3%	89.1%	87.7%	87.7%	85.2%
	Game score	39.20	37.48	36.14	35.73	33.35
	Tournament score	29.90	29.51	28.86	28.66	27.40
	Declarer card points	80.42	78.70	78.21	77.53	76.41
	Defender card points	43.02	44.16	44.47	44.77	45.24

Table 6.5: Combination of ε-greedy policy and points in ranges

6.3.2 Monte Carlo policy

The MC policy was also tested in combination with the adjustment of the game value in ranges and with the division of the single player points by 120. For this policy the latter gained better results. Therefore, only the results of the second combination is presented in table 6.6. The results of the experiment T_1 are even higher than in the combination of the UCB1 rule and the division of the single player points by 120.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT player	Won games	83.8%	82.3%	81.2%	81.2%	81.2%
	Game score	28.22	26.46	25.53	25.76	25.31
	Tournament score	23.39	23.11	22.71	23.12	22.82
	Declarer card points	73.03	72.92	72.38	72.51	72.55
	Defender card points	38.63	40.59	41.54	41.97	42.23
UCT with MC policy and single player points divided by 120	Won games	92.5%	89.1%	87.9%	86.2%	87.2%
	Game score	42.07	37.70	35.74	33.90	34.82
	Tournament score	31.79	29.70	28.99	27.93	28.45
	Declarer card points	84.95	82.36	81.15	80.59	80.27
	Defender card points	43.38	44.12	44.93	44.93	44.95

Table 6.6: Combination of MC policy and single player points divided by 120

6.4 Optimized Card Simulation

The simulation of card simulations that are not possible anymore can lead to weaker game results at the end. The simple distribution of all unknown cards over the other players does not always reflect the information the player has about the card distribution. The play of the other players can reveal some information about their cards during the game. If a player can not follow a suit or trump and plays a card from another suit it is known to all players on the table that this player does not have the suit or trump anymore. Otherwise, the player would have played "schwarz". This information can be used for the card simulations. As already investigated in my bachelor thesis, the information about suits the player does not have anymore is handed over to the simulation algorithm. Now, unknown cards from these suits are never given to this player again. [Sch05] The algorithm is presented in detail in appendix A.1.

The results are listed in table 6.7. There can be seen no statistically significant difference between the players, although the Skat series was won by the player with optimized card simulation with 77,156 to 74,332 points in experiment T_1. The optimized

card simulations might be one parameter that works better in combination with other optimizations like the game value adjustment.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	86.2%	82.5%	82.1%	81.0%	79.9%
	Game score	31.41	27.29	26.99	25.83	24.78
	Tournament score	26.21	24.34	24.44	23.99	23.48
	Declarer card points	75.76	74.34	74.12	73.81	73.07
	Defender card points	44.28	45.45	45.95	46.51	46.65
UCT Player with optimized card simulation	Won games	86.0%	84.2%	82.3%	81.0%	80.5%
	Game score	30.82	29.71	27.41	25.98	25.72
	Tournament score	26.02	25.77	24.65	24.09	24.04
	Declarer card points	75.68	74.76	74.08	73.34	73.60
	Defender card points	44.28	45.45	45.84	46.34	46.68

Table 6.7: Optimized card simulation

6.5 Random Move on Equal Rating

There is a chance that more than one move is rated best after all simulations are done. Only choosing the first move found with this evaluation would lead to a bias that is caused by the sorting of the cards. Choosing a random candidate from all best rated moves can eliminate this bias.

The results in table 6.8 show a slight advantage for the player with random move choosing. Only the game score differs statistically significant. ($p = 0.04$)

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	87.2%	84.0%	82.4%	81.3%	79.4%
	Game score	32.31	29.14	27.51	26.27	23.65
	Tournament score	26.67	25.29	24.42	23.99	23.06
	Declarer card points	76.19	74.75	74.18	73.88	73.07
	Defender card points	43.47	44.80	45.30	45.56	46.64
UCT Player with random choosing policy	Won games	87.2%	84.1%	83.3%	82.4%	80.4%
	Game score	32.77	29.32	28.96	28.12	25.49
	Tournament score	26.65	25.40	25.44	25.06	23.96
	Declarer card points	76.82	75.45	74.99	74.73	73.58
	Defender card points	43.52	44.99	45.53	45.82	46.71

Table 6.8: Random move choosing on same rating

6.6 Exploration Factor

Different games can have a different optimal balance between exploration and exploitation. The best exploration factor for Skat must not implicitly be the best factor for other card games. Therefore, different factors are investigated in this section to find the best for Skat. The factor was set to 0, 1, 4, 8, 12 and 16 and then compared to the factor of 2 from the original paper.[ACBF02]

The results in table 6.9 are listed for the best factor of 8. Higher factors than 8 gained weaker results again. It is interesting that with the same amount of declarer and defender card points the player with exploration factor 8 can gain better results. The experiment T_1 is the only one where a difference in the measured values can be seen. This can be explained by the low number of simulations in the other experiments. The UCT algorithm needs more simulations to gain good results.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	86.7%	84.0%	82.9%	81.1%	79.1%
	Game score	31.81	29.05	28.04	26.13	23.63
	Tournament score	26.43	25.27	25.02	24.12	22.84
	Declarer card points	76.08	74.83	74.31	73.70	73.10
	Defender card points	43.99	45.37	45.89	46.42	46.94
UCT Player with exploration factor 8	Won games	86.5%	84.2%	82.2%	81.2%	81.0%
	Game score	31.19	29.51	27.25	26.20	25.36
	Tournament score	26.12	25.46	24.42	24.14	24.17
	Declarer card points	75.89	74.62	74.02	73.54	73.12
	Defender card points	44.04	45.18	45.78	46.34	46.83

Table 6.9: Exploration factor 8

6.7 Zobrist Hash Function

The standard implementation of the hash function for game states takes all moves into account that have been played by all players so far in the order they appeared. Additionally, the players position in the current trick and all cards on the players hand are used to calculate the hash value.

The information about the cards that have been played so far might be enough to differentiate the game status. Therefore, another approach for the zobrist hash of a game status is investigated here. The random numbers for the cards that have been played already are only used without their time of appearance for the calculation of the hash

value. With this approach more nodes should be summarized to an information set. Therefore a deeper investigation of the search tree might be possible.

Table 6.10 lists the results for the experiments on the hash functions. A better performance for the new hash function can only be seen in the experiment S_{100}. In games with a higher number of simulations the use of the standard zobrist hashing gained slightly better results. This difference is not statistically significant. Because of the lack of time, it could not be tested which hash functions is better in games with a even higher number of simulations.

Player	Measured value	S_{100}	S_{500}	S_{1000}	S_{2000}	T_1
UCT Player	Won games	85.6%	83.3%	81.8%	82.0%	81.0%
	Game score	30.95	28.44	26.22	26.41	25.65
	Tournament score	25.72	24.95	24.05	24.47	24.04
	Declarer card points	75.94	74.45	73.68	73.55	73.34
	Defender card points	43.91	45.44	45.94	46.54	46.71
UCT Player with zobrist hash ignoring the order of the played cards	Won games	86.9%	83.5%	82.6%	81.1%	80.4%
	Game score	32.37	28.59	28.01	25.88	25.81
	Tournament score	26.88	25.18	25.00	23.83	23.88
	Declarer card points	76.05	74.46	74.25	73.44	73.33
	Defender card points	44.10	45.64	46.12	46.47	46.61

Table 6.10: Zobrist hash function without order of played cards

6.8 Adjusting the Aggressiveness During Bidding

The performance of the bidding can also be improved by adding a factor for aggressiveness. This means that a player will bid longer even though the results of the simulations during the bidding tell the player to stop the bidding. For this purpose the evaluation function from section 5.1.1 was extended by a factor for aggressiveness. With adding this factor to the ratio of won and made games a higher multiplier for the calculation of the maximal bidding value is gained. The extended formula is

$$\left\lceil m * \min\left(\frac{w}{n} + a, 1\right)\right\rceil * f$$

where m is the number of multipliers, w is the number of won games, n is the number of all games, a is the factor for aggressiveness and f is the factor for the suit or grand game from table 2.3. For null games the ratio of won games to made games is also adjusted by the aggressiveness factor.

Table 6.11 shows the results. The players did 20,000 simulations per decision for a move. The aggressiveness factor was set to 0.3. The results under XSkat bidding are listed again for comparison. The bidding with aggressiveness factor is significantly better than without adjusting the evaluation function. The number of games made by the players has raised to 85% but is still lower than under XSkat bidding. The percentage of won games is lower than under normal bidding but higher than under XSkat bidding. The game score and the tournament score are also lower than under XSkat bidding. Therefore, the bidding with aggressiveness factor has to be more optimized to compete with XSkat bidding.

Measured value	XSkat bidding	UCT bidding	UCT bidding with aggr.
Made games	1,000	337	851
Won games	852 (85.2%)	318 (94.36%)	741 (87.07%)
Game score	32,709 (32.71)	14,804 (43.93)	25,188 (29.60)
Tournament score	79,749 (79.75)	31,274 (92.80)	65,538 (77.01)
Declarer card points	77,561 (77.56)	28,037 (83.20)	59,987 (70.49)
Defender card points	84,878 (42.44)	24,650 (73.15)	81,674 (47.99)

Table 6.11: XSkat vs. UCT bidding

The distribution of game types also changed under UCT bidding with aggressiveness in comparison to the XSkat bidding. The number of null games was significantly higher. The UCT player played 104 null games (12,2%) and 178 grand games (20.9%). The other games are distributed over the four suit game types. The ♢ and ♡ game type were played equally often in 97 times. The ♠ game type occured 155 times and finally the ♣ game type was played 220 times. Higher values suit games are prefered by the UCT bidding and discarding.

6.9 Best Combination of Optimizations

In the sections above the optimized parameters are only investigated for their own. Section 6.3 already showed that a combination of two or more optimizations can lead to even better results, although every optimization for their one gained no better results. Therefore, the best combination was searched by trial and error in many experiments. The best combination so far is a UCT player with exploration factor of 8, a game value adjustment with division of single player points by 120, an optimized simulation of card distribution, random move choosing under all equally evaluated moves and a Zobrist hash function with taking the order of appearance of the cards into account.

Chapter 7

Comparison with Other Players

This chapter presents the results of the comparison of the UCT player with other player types. At first, DDSS player was compared to the UCT player. In the second part the UCT player played against human players.

7.1 Comparison with DDSS

For the comparison with the DDSS the set of 1,000 standardized games was used again. Every card distribution was played by every player on every sitting position. Therefore, 3,000 games were made by each player type. For the bidding, discarding and game anouncement the heuristics of XSkat was used. The experiment was done in this way because DDSS does not have a working bidding and discarding module, yet. The UCT player was set up with the best parameter combination presented in the previous chapter. Both players had a time limit of one second. The limit was chosen for better comparison with the experiments against human players. The results are listed in table 7.1. The values in parantheses are again the percentage of won games under all games or the mean of the measured value.

Measured value	UCT	DDSS
Won games	2,514 (83.8%)	2,592 (86.4%)
Game score	94,750 (31.58)	93,850 (31.28)
Tournament score	23,0150 (25.57)	24,0570 (26.73)
Declarer card points	24,0192 (80.06)	22,1642 (73.88)
Defender card points	26,5416 (44.24)	25,0742 (41.79)

Table 7.1: UCT with optimizations vs. DDSS

The UCT player was able to gain the same game score ($p = 0.74$) under normal scoring with fewer won games ($p = 0.01$). Therefore the game score under tournament scoring is won by the DDSS player ($p = 0.05$). The number of card points gained by the UCT player as declarer and defender player is higher ($p < 0.0001$). The results might not show the real power of the UCT algorithm. In the experiment the UCT player was able to simulate about 4,000 games in one second. The Go playing program MoGo does about 400,000 simulations per move decision. This would not be possible to compute for the UCT player in a reasonable time. The implementation is not optimized for speed, yet. But it shows that the optimal number of simulations is still beyond the numbers seen in the experiments.

7.2 Comparison with Human Players

With the GUI developed for this thesis it is possible to let the AI players play against human players for the first time. All phases of a Skat game can be played by every player for their own. For every game a random card distribution was generated. In the Skat series two instances of the UCT player competed against one human player. The results for the two UCT players were summed and the mean was computed. The UCT player was tested by three different human players in a total of 538 games.

During bidding, the UCT player had a limit of 10,000 simulations of possible card distributions and the aggressiveness parameter was set to 0.2. For the decision which cards to discard and which game to play, 100,000 simulations were done. The UCT player was set up with a slightly different set of optimization parameters for the trick playing phase, because the best combination was not known at the start of the experiments. The optimized card simulation was used in combination with an exploration factor of 2, game value adjustment in ranges, random choosing of equally evaluated moves and the zobrist hash function without taking the order of the played cards into account. Unfortunately three of the parameters are different to the optimal parameter set. The UCT player had a time limit of 1 second for the simulations during the trick playing. With this time limit the game flow is not disturbed too much. The results are listed in the tables at the end of this section.

The results can not be compared directly to the experiments between AI players. The total number of games is too low to see statistical significant differences. The games were not repeated because the human players would have recognized it and would have played with perfect information in the repeated games. The distributions of the cards have still a strong effect on the outcome.

The UCT player played reasonable well against the humans during trick play. Sometimes the player suprises with unusual moves. If no higher winning level can be reached,

the UCT player will be discarding high valued cards during trick play in some situations. This did not affect the outcome of the games. A human player would not have done such moves. The bidding strength is still too weak to compete against a human player. Here it can be seen again, that it is important to make many games to win the Skat series in the end.

Measured value	UCT	Human 1
Made games	36.5	51
Won games	31 (84.9%)	37 (72.6%)
Game score	1,045 (28.63)	706 (13.84)
Tournament score	3,100 (50.00)	2,296 (18.52)
Declarer card points	2,716.5 (74.42)	3,163 (62.02)
Defender card points	4,250.5 (68.56)	3,327 (45.58)

Table 7.2: UCT player with optimizations vs. Human 1

Measured value	UCT	Human 2
Made games	93.5	140
Won games	73 (78.1%)	113 (80.7%)
Game score	1,792 (19.17)	3,196 (22.83)
Tournament score	6,317 (38.64)	9,136 (27.94)
Declarer card points	6,595 (70.53)	9,857 (70.41)
Defender card points	11,418 (69.83)	9,250 (49.47)

Table 7.3: UCT player with optimizations vs. Human 2

Measured value	UCT	Human 3
Made games	31.5	24
Won games	24 (76.2%)	22 (91.7%)
Game score	335.5 (10.65)	1,058 (44.08)
Tournament score	1,540.5 (35.41)	2,658 (30.55)
Declarer card points	2,174 (69.02)	2,004 (83.50)
Defender card points	2,422 (55.68)	3,212 (50.98)

Table 7.4: UCT player with optimizations vs. Human 3

Chapter 8

Summary and Future Work

The last chapter summarizes the results of the thesis and gives an outlook on ideas for future work. There are many ideas that come to mind. Therefore, the work on the game of Skat has just begun.

8.1 Summary

The applicability of the UCT algorithm for the game of Skat has been shown in this thesis. The playing strength of the UCT player is even higher than under MC simulations and can also compete with the search of the best strategy with a min-max search algorithm like the DDSS does. With better optimizations, it might be possible to clearly defeat the DDSS some day. Skat belongs to the family of games with imperfect information. Therefore, the UCT algorithm might be applicable for other games of this family, too. The uncertainty in information is no barrier for using this algorithm.

Skat has some characteristics that make the UCT algorithm better applicable than in other games with imperfect information. During the game the number of possible worlds decreases. The number of unknown cards is reduced from trick to trick. The number of possible worlds can be decreased further by using the information gained from the play of the other players. If a suit was not followed by a player, it is no longer possible that this player still owns cards of this suit. The number of possible moves decreases, too, during the game. The forehand player in a trick can play a maximum of 10 different cards in the first trick. This number is decreased by 1 from trick to trick. The middlehand and hindhand player are even more restricted by the suit enforcement rule. Often, only 2 or 3 cards are possible to play for the middlehand and hindhand. In the middle game, it is very likely that all moves have been investigated in all worlds for the first level of the search tree and that other levels can be added. During the last tricks, the complete tree can be computed.

8.2 Ideas for Future Work

The time for completing a diploma thesis is always too short for implementing all ideas that arise during the work. At the end three of them will be outlined as outlook for future work.

8.2.1 Best Combination of Parameters

The UCT algorithm is controlled by a lot of parameters. In chapter 6 many of them are investigated for their own. It is not clear by now whether they are really independent or not. Regarding the selection policy and the game value adjustment it was shown that the combination of both can lead to better results. To investigate all parameters in combination and to find the optimal parameters, evolutionary algorithms could be used. With this approach the parameters are changed in little steps every time a simulation of a Skat series is runned. If the results are better than in the last series the changes on the parameters will survive. Otherwise, they are changed again. In the long run the best combination of all parameters will be achieved. This can only be accomplished for Skat when the same card distributions are used in every Skat series during the optimization of the parameters.

The dynamic adjustment of the exploration/exploitation factor in the UCB1 policy during the game was suggested in [ACBF02]. The standard UCB1 policy is too exploratory for nodes that are rarely visited. A fine tuned policy may lead to better results for these nodes.

More simulations can help to gain better results, too. The UCT algorithm is only descending to lower levels in the search tree when all moves are investigated in all possible worlds. During the first tricks, the number of possible worlds is very high. Therefore, doing as much simulations as possible is an appropriate way to enhance the playing strength. The simulations could even be parallelized to make use of modern multiprocessor machines.

8.2.2 Bidding and Discarding

For the game of Skat it is essential to make as much games as possible and win most of them. A good bidding and discarding strategy is crucial to be able to play many games as single player. The balance between bidding and passing is still not optimal in the algorithm discussed in this thesis. The right amount of aggressiveness is still needed to be found.

The use of a heuristic for discarding should also be superior to the random discarding that was used during the simulations in the bidding phase. This could be done with simple rules like "do not discard trump cards".

8.2.3 Better Heuristic for Game Simulation

The heuristics of XSkat are known to be not very strong. Even an intermediate player can win against the XSkat player. One reason is the conservative bidding. XSkat bids only on hands if the games can be won with high certainty. A heuristic with better evaluation of the player's hand would make more games and win the Skat series even if it loses some of the games.

Another evaluation module has to be created to exchange the heuristics derived from XSkat. A rule based heuristic is very hard to do and it is known that this approach seldom leads to good results. Neural networks could be used to create evaluation modules for the different game phases without programming every rule. This was successfully done for the game of Backgammon. The neural networks of the program TD-Gammon were trained in self-play games and achieved a playing strength level of a human master.[Tes94]

A simple neural network is built up in three layers of neurons. These are called input, hidden and output layer. Every neuron in the input layer is connected to each neuron in the hidden layer. So are the neurons of the hidden and output layer. A certain value is attached to each connection between neurons. This value defines how strong the signal is transferred from one neuron to the other. The input signals for neural networks are coded in the interval $[0; 1]$. Therefore, the game status must be transferred to this coding. The number of neurons in the input layer depends directly on the number of values are needed to code the current game status in 0s and 1s. The optimal number of neurons in the hidden layer has to be found through try and error. The output layer also codes the result of the evaluation of the input patter in values from the interval $[0; 1]$. After setting up the input signal in the input layer, the neural network evaluates the pattern and determines the values of the neurons of the output layer. To train the network sets of training data are defined at the input layer and the output is compared to the desired result. If there is a difference, a backpropagation step is done and the values of the connections are adjusted to achieve the desired result.

For the game of Skat a lot of possible applications of neural networks come to mind. For the bidding phase, networks for every game type could be trained to evaluate the cards on the player's hand for these game type. For discarding the same is feasible. During trick play, different networks for the single and the opponent players might be necessary. Also, every game type needs its specially trained networks.

Neural networks also have another advantage. They can adjust the strategies to the play of a human player during a Skat series. This is more flexible than a strict rule set. The resulting AI player will always learn better strategies for situations that occur during the Skat series.

Appendix A

Optimized Generation of Card Distribution

In this appendix, the algorithm for the optimized generation of card distributions is outlined. The algorithm is given in pseudocode on the next page. The idea of the optimized generation of card distributions is to avoid distributions in which a player gets cards that cannot be on the player's hand anymore. These simulations will bias the simulations with worlds that are not possible under the game progress seen so far.

First, the opponent players with unknown card positions are search. Under these players the one with most restrictions is searched. A restriction is a suit a player cannot have anymore. The simulations of cards starts on the player with more restrictions. The other player is simulated afterwards.

For the simulations of the player cards all cards of a card deck are checked whether they are already have been seen or not. A card is alread known if it was played or on the hand of the current player. If the position of the card is still unknown it will be tested on which player's hand the card could be. A card cannot be on a player's hand, if this player has a restriction on the suit of the card. If the current player cannot have the card it will be treated as a known card. If the card cannot be on both player's hand it must be in the skat. In the case the other player cannot have the card but the current player can, the card is treated as obligatory card and must be put on the current player's hand. If there are still unknown card positions on the players hand, a random sample of cards will be dealed into the unknown positions. All these steps are done for the less restricted player afterwards.

```
getOptimizedCardDistribution(player) {

  constant allCards = {0, 1, 2, ... , 30, 31}
  knownCards = skatCards = obligatoryCards = unknownCards = {};
  opponentPlayers = getDifference(allPlayers, player);
  mostRestrictedPlayer = getMostRestrictedPlayer(opponentPlayers);

  forall (opp in opponentPlayers startingwith mostRestrictedPlayer) {
    forall (card in allCards) {

      if (cardWasAlreadyPlayed(card) or
          cardOnPlayersHand(player, card)) {
        knownCards.add(card);
      }
      else {

        icp1 = isCardPossible(opp);
        icp2 = isCardPossible(getDifference(opponentPlayers, opp));

        if (!icp1) {
          knownCards.add(card);
        }
        if (!icp2) {
          if (!icp1) {
            skatCards.add(card);
          }
          else {
            obligatoryCards.add(card);
    } } } }

    if (skatCards.count() > 0) {
      putIntoSkat(skatCards);
      knownCards.add(skatCards);
    }
    if (obligatoryCards.count() > 0) {
      dealCardsToPlayer(opp, obligatoryCards);
      knownCards.add(obligatoryCards);
    }
    unknownCards = getDifference(allCards, knownCards);
    unknownCards.shuffle();
    dealToPlayer(unknownCards, opp);
  }
}
```

Figure A.1: Optimized generation of card distribution

Appendix B

UCT Search Tree Example

For debugging purposes the UCT player is able to save the search tree into a text file. The graph language DOT is used to describe the nodes and their connections in the tree. With the tools from the Graphviz collection, graphics of the search tree can be generated from the text file in different graphic formats afterwards.[Gra07]

The example on the next page is only a part of a search tree that was created with simulations during trick play of a suit game with clubs as trump color. Each node labled with "MAX" represents the move of the simulating player. The other nodes are labled with "MIN". This information is used during backprogation to apply the end result of the game accordingly. The rest of the node label shows the player ID. The moves are labled with a letter for the suits (D $\widehat{=}$ ♢, H $\widehat{=}$ ♡, S $\widehat{=}$ ♠ and C $\widehat{=}$ ♣) and the code for the card value. The dashed lines mark the end of a trick.

The search is started with a "MAX" node. All possible cards are played in all possible worlds before a new level in the search tree is created. "MAX" and "MIN" nodes can be mixed in one level, because the trick winner becomes the forehand player of the next trick. In the example the simulation has started for a forehand player. After playing ♡9 or ♣8 in different simulations the middlehand and hindhand player played a card. Some of the tricks are won by the "MIN" players but there are also two nodes where the "MAX" player has made the trick. Therefore, "MIN" and "MAX" nodes can be found in the fourth level of the tree. From this level the search is expanded mainly from the nodes where the "MAX" player has won the trick. This shows the selection strategy of the UCT algorithm that prefers higher rated branches during the selection phase. At the lowest level some moves coming from different branches finally end in the same node. These nodes represent the same game status. This shows the optimization of the Zobrist hash function when the order of the moves is not taken into account.

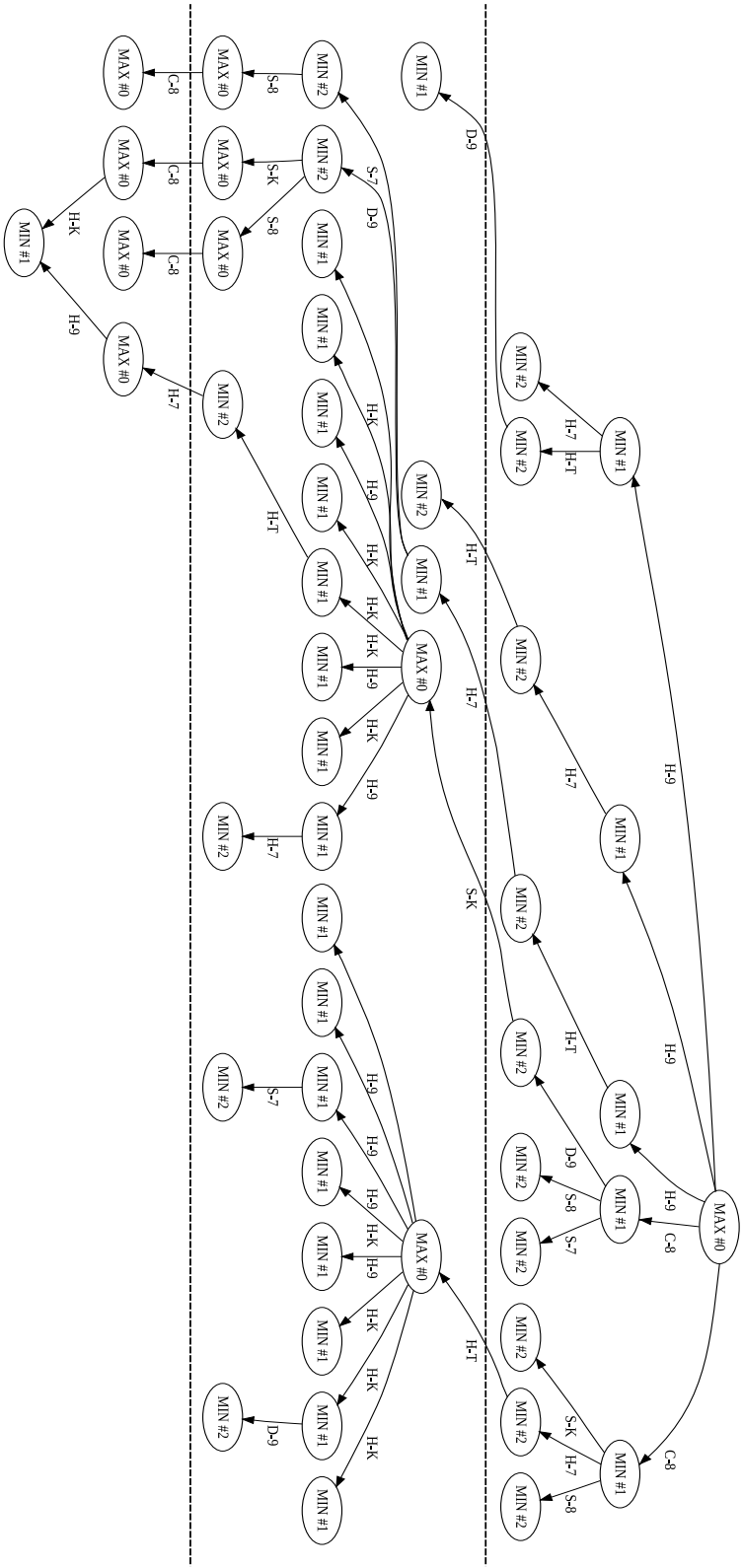

Figure B.1: Example for an UCT search tree

Appendix C

Example for a Grand Game

In this appendix a grand game is presented. The AI players are the UCT player with optimizations, the DDSS player and a XSkat player. The UCT player and DDSS player have a time limit of one second per move decision. Bidding and discarding was done with the heuristics of XSkat. Forehand won the bidding with a bid value of 18 and announced a grand game. The card distribution after bidding and discarding was the following:

Player	Cards			
Player 0		♡AKJ9	♠ATJ7	♣A8
Player 1	♢TJ7	♡8	♠K8	♣KQJ9
Player 2	♢AK8	♡TQ7	♠Q9	♣T7
Skat	♢Q9			

Table C.1: Card distribution for the example game

The move decisions of the AI players are listed in the following three tables. The sitting positions were changed in every game. The forehand player is listed in the first column. The middlehand player follows and the moves of the hindhand are shown in the last column. The game starts with the card of the forehand player. The underlined card is the trick winning card. The player which played that card is the forehand player of the next trick.

The XSkat player starts the game in table C.2 with the ♡J. After the second trick it becomes forehand, because the DDSS plays a jack again. The UCT player is always playing the lowest possible card in the first 7 tricks. In trick 8 the UCT player becomes forehand by playing the ♡T. The last tricks are all made by the UCT player. It is a comfortable win for the XSkat player with 70 to 50 points.

In the second game shown in table C.3 the DDSS player starts the game with the ♣A. After forcing the UCT player to play a jack in the second trick, the DDSS player

Trick	XSkat	DDSS	UCT
1	♡J	♣J	♡Q
2	♠J	♢J	♢8
3	♡A	♡8	♡7
4	♠A	♠8	♠9
5	♠T	♠K	♠Q
6	♠7	♣9	♣7
7	♣A	♣Q	♣T
8	♡9	♣K	♡T
9	♣8	♢T	♢A
10	♡K	♢7	♢K

Table C.2: XSkat vs. DDSS and UCT

Trick	DDSS	UCT	XSkat
1	♣A	♣9	♣7
2	♡J	♢J	♡7
3	♠A	♠8	♠9
4	♠T	♠K	♠Q
5	♡A	♡8	♡Q
6	♠7	♣Q	♢8
7	♠J	♣J	♢A
8	♣8	♢T	♢K
9	♡K	♣K	♣T
10	♡9	♢7	♡T

Table C.3: DDSS vs. UCT and XSkat

plays cards from suits that are represented often on its hand. In trick 6 the UCT player has the chance to win the trick be playing the ♣J. This could have led to a lost trick afterwards, because the ♠J is still on one hand of the other players. The UCT player waits for the DDSS player until it plays the ♠J. After that all tricks are won by the opponent players. The result of the game is a win for the DDSS player, but close to a lost game with 63 to 57 points.

The last game is played by the UCT player. It forces the XSkat player to play a jack in the first trick. The second trick is also won by XSkat, but the trick value has 0 card points. The XSkat player plays ♠8 in trick 3. It has no knowledge that ♢7 would have been the more appropriate card. The UCT player becomes forehand after trick 3 and stays in this position until the end of the game. The game result is 106 to 14 points for the UCT player. The oppenents have played “schneider” in this game.

Trick	UCT	XSkat	DDSS
1	♠J	♣J	♣T
2	♣8	♣9	♣7
3	♠A	♠8	♠9
4	♡A	♡8	♡7
5	♡J	♢J	♢8
6	♠T	♠K	♠Q
7	♠7	♣Q	♢K
8	♣A	♣K	♡T
9	♡K	♢7	♡Q
10	♡9	♢T	♢A

Table C.4: UCT vs. XSkat and DDSS

In this little example the different strategies of the AI players are illustrated very well. The UCT player often tries to remove higher cards on the opponent hands in the first tricks. After that it makes all tricks until the end. XSkat and DDSS rather try to make the most points in the first tricks. This strategy can be a winning strategy in other games. A further detailed investigation of the playing strategies was not possible due to the lack of time.

Appendix D

Post Thesis Work

After the completion of the thesis some bugfixes and a lot of optimizations were done in order to defeat the DDSS player. At the end the UCT player was able to calculate 15,000 to 20,000 simulations per second. Unfortunately this was still not enough to finally beat the DDSS player during the play of the 1,000 card distributions used in every experiment of this thesis. That's why new sets with 1,000 card distributions were generated for the last experiments and every distribution again was played six times to eliminate the bias from the dealing. The results between different card distribution sets varied from clearly defeating the DDSS player to playing on the same level. The results of the best round so far are presented in table D.1.

Measured value	UCT	DDSS
Made games	2,979	2,979
Won games	2,460 (82.6%)	2,420 (81.2%)
Game score	84,406 (28.33)	74,139 (24.89)
Tournament score	22,4016 (25.07)	21,0869 (23.60)
Declarer card points	23,3526 (78.39)	21,9343 (73.63)
Defender card points	26,7694 (44.93)	25,6488 (43.05)

Table D.1: UCT with optimizations vs. DDSS

In this experiment the UCT player used the UCB1 policy for selecting moves during the selection phase. If more than one move had the best rating a random move was selected from these moves. Unseen moves were treated as they have been played already. For these moves a game value of 0.5 was assumed. After finishing the simulated games the game value was adjusted in ranges to the interval of $[0; 1]$ before backpropagation. Finally optimized card simulation was used.

The weak heuristics of XSkat is the main cause that prevents the UCT player from playing stronger. The bidding often ends after two or three offers. The trick playing could also benefit from a stronger heuristic. Therefore the search for a replacement of the heuristics derived from XSkat will be the main part of future works. One focus will be the usage of neural networks and TD-learning.

The UCT player is currently playing on the International Skat Server (ISS) maintained by Michael Buro. Everyone is invited to play against the UCT player and other AI players as well. More informations can be found under http://buro.dnsalias.net/iss.

Bibliography

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47:235–256, 2002.

[Agr95] R. Agrawal. Sample mean based index policies with O(log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27:1045–1078, 1995.

[BMvL96] Jean R. S. Blair, David Mutchler, and Michael van Lent. Perfect Recall and Pruning in Games with Imperfect Information. *Computational Intelligence*, 12(4):131–154, 1996.

[CWUvdH07] G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. Progressive strategies for monte-carlo tree search. In *Information Sciences 2007 Proceedings of the 10th Joint Conference*, pages 655–661, 2007.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *Communications of the Association for Computing Machinery*, 7(7), 1964.

[FB98] Ian Frank and David Basin. Search in Games with Incomplete Information: A Case Study using Bridge Card Play. *Artificial Intelligence*, 100:87–123, 1998.

[FY38] R. A. Fisher and F. Yates. Example 12. *Statistical Tables*, 1938.

[Ger04] Gunter Gerhardt. `http://xskat.de/xskat-latest.html`, 2004.

[Gin99] M. L. Ginsberg. GIB: Steps Toward an Expert-Level Bridge-Playing Program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 584–589, 1999.

[Gra07] Graphviz. `http://www.graphviz.org/`, 2007.

[GW06] Sylvain Gelly and Yizao Wang. Exploration Exploitation in Go: UCT for Monte-Carlo Go. In *Twentieth Annual Conference on Neural Information Processing Systems (NIPS)*, 2006.

[IW03] ISPA-World. `http://www.skatcanada.ca/canada/forms/rules-2003.pdf`, 2003.

[Knu98] Donald E. Knuth. *The Art of Computer Programming Vol. 2.* Addison-Wesley, 2nd edition, 1998.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[Kup03] Sebastian Kupferschmid. Entwicklung eines Double Dummy Skat Solvers mit einer Anwendung für verdeckte Spiele. Diploma thesis, Albert-Ludwigs-Universität Freiburg, Fakultät für angewandte Wissenschaften, Institut für Informatik, 2003.

[Lev89] D. Levy. The Million Pound Bridge Program. In *Heuristic Programming in Artificial Intelligence*, pages 95–103. Ellis Horwood, 1989.

[Lin04] Peter Lincoln. *Skat lernen – Die besten Strategien.* Urania Verlag, 2004.

[SBB+07] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers Is Solved. *Science Express Research Articles*, 2007.

[Sch05] Jan Schäfer. Monte Carlo Simulation im Skat. Bachelor thesis, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Simulation und Graphik, 2005.

[Tes94] G. J. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[Tro07] Trolltech. `http://trolltech.com/products/qt/qt3`, 2007.

[Zob70] A.L. Zobrist. A new hashing method with application for game playing. Technical report, University of Wisconsin, Computer Science Department, 1970.

Statement of Authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise.

Groitzsch, July 11, 2008

Jan Schäfer

Theses

1. The UCT algorithm is applicable to games with imperfect information after replacing the unknown information with simulated values.
2. A higher number of simulations is needed for the UCT algorithm to get the same results as with other approaches in games with imperfect information.
3. The UCT algorithm can be used in all phases of a Skat game.
4. SkatTA is a flexible platform for the implementation, testing and comparing of new AI players to other AI players or to human players.
5. The interface for AI players defined in SkatTA supports all phases of a Skat game. An AI player that implements this interface is a complete Skat player that can play against human players in a normal Skat series.