POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Monte Carlo Tree Search algorithms applied to the card game Scopone

Relatore: Prof. Pier Luca Lanzi

Tesi di Laurea di:
Stefano Di Palma, Matr. 797319

Anno Accademico 2013-2014

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Codes

Acknowledgements

I wish to thank my supervisor Prof. Pier Luca Lanzi for reading and providing feedback on my thesis, for the interesting discussions on artificial intelligence, and for allowing me to do this work.

Last but not least, I would like to thank my family and my friends for their continuous support and motivation during the whole studies. A special thank to my parents that allow me to study what I like.

Abstract

The aim of this thesis is to design a competitive *Artificial Intelligence* AI algorithm for *Scopone*, a famous Italian-card game. In particular, we focused on the *Monte Carlo Tree Search* (MCTS) algorithms. However, MCTS requires full knowledge of the game state, therefore we also used an extension of the algorithm called *Information Set Monte Carlo Tree Search* (ISMCTS), that can work with partial knowledge of the game state. We developed different improvements of MCTS and ISMCTS, and we evaluated their playing strength against three rule-based AI encoding rules taken from well-known strategy books of Scopone. Our results showed that the deck team has an advantage over the hand team and the MCTS and ISMCTS algorithms proved to be stronger than the rule-based AI. Finally, we also developed an application to let human players interact with our AI.

Sommario

Lo scopo di questa tesi è quello di progettare un algoritmo d'*Intelligenza Artificiale* (IA) competitiva per Scopone, un famoso gioco di carte Italiano. In particolare, ci siamo concentrati sugli algoritmi di *Ricerca ad Albero Monte Carlo* (o *Monte Carlo Tree Search*, MCTS). Tuttavia, MCTS richiede la piena conoscenza dello stato di gioco, quindi abbiamo anche usato un'estensione dell'algoritmo chiamata *Information Set Monte Carlo Tree Search* (ISMCTS), che è pensata per funzionare usando una conoscenza parziale dello stato di gioco. Inoltre, abbiamo sviluppato diverse varianti degli algoritmi MCTS e ISMCTS, valutando il loro livello di gioco contro tre IA a regole, che codificano la conoscenza tratta da famosi libri di strategia di Scopone. I nostri risultati hanno dimostrato che la coppia di mazzo ha un vantaggio rispetto la coppia di mano e gli algoritmi MCTS e ISMCTS sono risultati più forti delle IA a regole. In fine, abbiamo sviluppato un'applicazione che permette ai giocatori umani di misurarsi con le nostre IA.

Estratto

Questa tesi si inquadra nell'ambito dell'*Intelligenza Artificiale* (IA) nei giochi di carte. Questo campo è nato negli anni '50 ed i primi algoritmi di IA, sviluppati per giochi da tavolo a due giocatori (come Dama e Scacchi), erano in grado di giocare solo mosse finali di una partita o competere con principianti. Negli anni successivi, grazie alla progettazione di tecniche più avanzate, i programmi erano in grado di competere contro professionisti. In alcuni casi, è stato possibile risolvere un gioco, ovvero prevedere il risultato di una partita in cui tutti i giocatori giocano in maniera perfetta.

Lo scopo di questa tesi è di progettare un algoritmo di intelligenza artificiale competitivo per *Scopone*, un famoso gioco di carte Italiano che richiede elevate capacità intellettuali per poter essere giocato. Per questo motivo, è spesso chiamato *Scopone Scientifico*, perché le regole gli conferiscono dignità di Scienza.

In particolare, il nostro obiettivo è quello di valutare il livello di gioco che può essere raggiunto con l'algoritmo di *Ricerca ad Albero Monte Carlo* (o *Monte Carlo Tree Search*, MCTS) applicato a Scopone. L'algoritmo MCTS è stato introdotto nel 2006 da Rémi Coulom et al. [13]. Poco dopo, Kocsis e Szepesvári hanno formalizzato questo approccio nell'algoritmo *Upper Confidence Bounds for Trees* (UCT), che oggi è l'algoritmo più utilizzato della famiglia MCTS. In contrasto con i classici algoritmi di IA (come Minimax), che esplorano completamente l'albero di ricerca, MCTS costruire un albero in modo incrementale e asimmetrico, guidato da molte simulazioni casuali delle svolgimento della partita. In questo modo, MCTS esplora solo le aree più promettenti dell'albero. Inoltre, l'esplorazione può essere interrotta in qualsiasi momento restituendo il miglior risultato finora ottenuto, questo rende MCTS molto efficiente in termini di tempo e memoria. Kocsis e Szepesvári sono stati anche in grado di dimostrare che, con un numero sufficiente di iterazioni dell'algoritmo, MCTS converge allo stesso risultato di Minimax. Tuttavia, Scopone, come molti giochi di carte, ha la caratteristica che i giocatori non conoscono le carte in possesso degli altri, quindi lo stato di gioco

dal punto di vista di un giocatore nasconde alcune informazioni. Invece, MCTS richiede piena conoscenza dello stato di gioco, quindi abbiamo utilizzato un'estensione dell'algoritmo proposto da Cowling et al. [14] nel 2012 chiamato *Information Set Monte Carlo Tree Search* (ISMCTS). ISMCTS è pensato per funzionare usando una conoscenza parziale dello stato di gioco, infatti basa la propria decisione su un albero di ricerca, dove ciascun nodo è un *information set*, che rappresenta tutti gli stati di gioco compatibili con le informazioni disponibili al giocatore radice. In ogni caso, abbiamo mantenuto anche l'algoritmo MCTS come giocatore sleale, cioè che può vedere le carte degli altri giocatori, e l'abbiamo usato come punto di riferimento per il miglior modo di giocare.

Per valutare il livello di gioco raggiunto dagli algoritmi MCTS e ISMCTS, abbiamo sviluppato tre AI a regole che codificano la conoscenza tratta da famosi libri di strategia di Scopone. La strategia *Greedy* cattura semplicemente le carte più importanti sul tavolo o gioca la carta meno importante che si ha in mano. La strategia *Chitarrella-Saracino* (CS) include le regole tratte dai libri di strategia di Chitarrella [11] e Saracino [24] sullo spariglio, il mulinello, il gioco di carte doppie e triple, e il gioco dei sette. La strategia *Cicuti-Guardamagna* (CG) è un'estensione della IA CS, e codifica le regole proposte da Cicuti e Guardamagna [12] sul gioco dei sette. Gli esperimenti hanno dimostrato che CS è la miglior strategia, mostrando che le regole supplementari per il gioco dei sette di CG non aumentano il livello di gioco.

Successivamente, abbiamo testato diverse varianti degli algoritmi MCTS e ISMCTS al fine di selezionare la migliore configurazione per Scopone. Abbiamo sperimentato quattro metodi per le ricompense: *Normal Score* (NS), utilizza il punteggio di ogni squadra; *Scores Difference* (SD), utilizza la differenza tra il punteggio delle squadre; *Win or Loss* (WL), utilizza 1 per una vittoria, -1 per una sconfitta, e 0 per un pareggio; e *Positive Win or Loss* (PWL), usa 1 per una vittoria, 0 per una sconfitta, e 0.5 per un pareggio. Il miglior metodo per le ricompense si è rivelato essere SD, con una costante UCT pari a 2.

Poi, abbiamo testato quattro strategie di simulazione: *Random Simulation* (RS), sceglie una mossa casuale; *Greedy Simulation* (GS), use la mossa scelta dalla strategia Greedy; *Epsilon-Greedy Simulation* (EGS), con probabilità ϵ sceglie una mossa casuale, altrimenti seleziona la mossa scelta dalla strategia Greedy; e *Card Random Simulation* (CRS), gioca una carta a caso, ma la strategia Greedy decide che carte catturare in caso ci siano più possibilità di presa. La miglior strategia di simulazione si è rivelata essere EGS con $\epsilon = 0.3$ per MCTS, e GS per ISMCTS. Probabilmente, ISMCTS è in grado di sfruttare tutte le conoscenze di gioco della strategia Greedy, perché entrambi non conoscono le carte degli altri giocatori. Considerando che, MCTS può anche vedere le carte degli avversari, contare completamente sulla strategia Greedy potrebbe indirizzare la ricerca in aree poco promettenti dell'albero.

Abbiamo anche sperimentato quattro gestori di mosse per ridurre in numero di mosse disponibili in ciascun nodo: *All Moves Handler* (AMH), genera tutte le mosse disponibili; *One Move Handler* (OMH), genera una mossa per ogni carta nella mano del giocatore e usa la strategia Greedy per scegliere le carte da catturare; *One Card Handler* (OCH), come OMH ma nella mossa è memorizzata solo la carta giocata; e *Greedy Opponents Handler* (GOH), quando non è il turno del giocatore radice, viene generata solo la mossa scelta dalla strategia Greedy. Gli esperimenti hanno dimostrato che AMH, OMH, e OCH sono equivalenti, ma AMH ha il vantaggio di sfruttare tutte le mosse disponibili. Mentre, GOH è significativamente più forte contro la strategia Greedy, ma con la strategia CS si è rivelata più debole. Questo è un caso di overfitting che si verifica soprattutto con MCTS. Per queste ragioni, abbiamo selezionato AMH come il miglior gestore di mosse.

Come ultima variante dell'algoritmo ISMCTS, abbiamo testato due determinizzatori: determinizzatore *Random*, sceglie uno stato nel information set radice in modo casuale; determinizzatore *Cards Guessing System* (CGS), limita il campione agli stati in cui ogni giocatore possiede le carte predette dal sistema di predizione delle carte. I due metodi si sono rivelati equivalenti, ma crediamo che CGS consenta a ISMCTS di evitare mosse che portano in modo semplice gli avversari a fare una scopa, dal momento che ISMCTS con CGS può prevedere le carte che gli avversari potrebbero possedere. Per questo motivo, abbiamo scelto CGS come il migliore determinizzatore.

Infine, abbiamo eseguito un torneo tra la strategia casuale, CS, MCTS, e ISMCTS. I risultati hanno confermato che la coppia di mazzo ha un vantaggio rispetto alla coppia di mano e il vantaggio aumenta con l'abilità del giocatore. Ovviamente, MCTS ha raggiunto le migliori prestazioni, perché è un giocatore sleale, mentre la strategia casuale è chiaramente la peggiore. La cosa importante è che ISMCTS ha dimostrato di essere più forte della strategia CS. Questo conferma che l'algoritmo ISMCTS è molto efficacie e merita lo svolgimento di ulteriori ricerche al riguardo.

Come conseguenza di questa tesi, abbiamo sviluppato due versioni del gioco: una pensata appositamente per testare le varie intelligenze artificiali, e un'applicazione, sviluppata con il motore di gioco Unity, che abbiamo in programma di rilasciare per Android e iOS. Con l'applicazione utente, si può giocare a Scopone contro le IA che abbiamo sviluppato e compilare un sondaggio sul livello di gioco percepito e il comportamento umano delle intelligenze artificiali.

Chapter 1

Introduction

This thesis focuses on the application of *Artificial Intelligence* (AI) in card games. This field was born in the '50s and the first AI algorithms, developed for two-players-board games (like Checkers and Chess), were able to play only final moves of the game or they could only play at the level of beginners. In the following years, due to the design of more advanced techniques, the programs could compete against human-expert players. In some cases, it has been possible to solve a game, i.e. predict the result of a game played from a certain state in which all the players did the optimal moves.

The aim of this thesis is to design a competitive AI algorithm for *Scopone*, a famous Italian card game that requires high intellectual skills in order to be played. For this reason, it is often called *Scopone Scientifico* (Scientific Scopone), because the rules give it the dignity of Science.

In particular, our goal is to evaluate the playing strength that can be achieved with the *Monte Carlo Tree Search* (MCTS) algorithm applied to Scopone. MCTS has been introduced in 2006 by Rémi Coulom et al. [13]. Shortly after, Kocsis and Szepesvári formalized this approach into the *Upper Confidence Bounds for Trees* (UCT) algorithm, which nowadays is the most used algorithm of the MCTS family. In contrast with the classical AI algorithms (like Minimax), that completely explore the search tree, MCTS build up a tree in an incremental and asymmetric manner guided by many random simulated games. In this way it can explore only the most promising areas of the tree. Moreover, the exploration can be stopped at any time returning the current best result, this make MCTS very efficient in terms of time and memory. Kocsis and Szepesvári were also able to prove that, with enough iterations of the algorithm, MCTS converges to the same result of Minimax. However, Scopone, like many card games, has the characteristic that the players do not know the cards held by the other players, therefore the game state perceived by one player includes some hidden information. In contrast, MCTS requires full knowledge of the game state, therefore we also used an extension of the algorithm proposed by Cowling et al. [14] in 2012 called *Information Set Monte Carlo Tree Search* (ISMCTS). ISMCTS can work with partial knowledge of the game state, in fact it bases its decisions on a tree search where each node is an *information set* representing all the game states compatible with the information available to the ISMCTS player.

In order to select the best possible MCTS and ISMCTS algorithms, we designed and experimented with different variations of these algorithms. We also developed three rule-based AI, one representing a beginner and two that encode the rules taken from well-known strategy books written by expert player of Scopone. The final results allow us to evaluate the playing strength of MCTS and ISMCTS against the rule-based AI, showing that the algorithms are competitive.

To perform these experiments, we designed two versions of the game: one strictly focused on testing the artificial players, and one application, made with the game engine Unity, to let human players interact with our AI.

1.1 Original Contributions

This thesis contains the following original contributions:

- The development of three rule-based AI for Scopone.
- The design of different variants of the MCTS algorithm applied to Scopone.
- The experimental comparison between the developed AI algorithm.
- The development of a research framework on MCTS and Scopone.
- The development of an application made in Unity in order to let the users play Scopone.

1.2 Thesis Outline

The thesis is structured as follows:

- In Chapter 1, we introduce the goals of this work, showing the original contributions and the thesis structure.

- In Chapter 2, we present several applications of AI in board and card games and we introduce the MCTS and ISMCTS algorithms.
- In Chapter 3, we describe the rules of Scopone.
- In Chapter 4, we show the rule-based AI we developed for Scopone.
- In Chapter 5, we present the various versions of MCTS and ISMCTS we developed for Scopone.
- In Chapter 6, we show and discuss the results obtained from the experiments we did between the AI we designed.
- In Chapter 7, we evaluate the work done for this thesis and we propose future research on this topic.
- In Appendix A, we briefly describe the two applications we developed.

Chapter 2

Artificial Intelligence in Board and Card Games

In this chapter we overview the most interesting applications of *Artificial Intelligence* (AI) in board and card games related to this work. Then we introduce the *Monte Carlo Tree Search* (MCTS) algorithm and we compare it with the well-known AI algorithm, *Minimax*, showing advantages and disadvantages of the two methods. Finally, we discuss the *Information Set Monte Carlo Tree Search* (ISMCTS) algorithm, an extension of MCTS that can deal with imperfect information games.

2.1 Artificial Intelligence in Board Games

Artificial Intelligence aims to develop an opponent able to simulate a rational behavior, that is, do things that require intelligence when done by humans. Board games are particularly suited for this purpose because they are difficult to solve without some form of intelligence, but are easy to model. Usually, a board configuration corresponds to a state of the game, while a legal move is modeled with an action that changes the state of the game. Therefore, the game can be modeled with a set of possible states and a set of legal actions for each state.

2.1.1 Checkers

The first applications of artificial intelligence to board games were presented in the '50s, when Christopher Strachey [1] designed the first program for the game of *Checkers*. Strachey wrote the program for *Ferranti Mark I* that could play a complete game of Checkers at a reasonable speed using evaluation of board positions. Later Arthur Samuel developed an algorithm to play Checkers that was able to compete against amateur players [33]. The algorithm used by Samuel was called *Minimax with alpha-beta pruning* (Section 2.2), which then became one of the fundamental algorithm of AI. Samuel tried to improve his program by introducing a method that he called *rote learning* [32]. This technique allowed the program to memorize every position it had already seen and the reward it had received. He also tried another way of learning, he trained his artificial intelligence by let it play thousands of games against itself [23]. At the end of the '80s Jonathan Schaeffer et al. began to work on *Chinook*, a program for Checkers developed for personal computers. It was based on alpha-beta pruning and used a precomputed database with more than 400 billion positions with at most 8 pieces in play. Chinook became world champion in '94 [27]. In 2007, Schaeffer et al. [26] were able to solve the game of Checkers (in the classical board 8 x 8) by proving that the game played without errors leads to a draw.

2.1.2 Chess

Chess is more widespread than Checkers but also much more complex. The first artificial intelligence to play this game was presented, in the '50s, by Dietrich Prinz [2]. Prinz's algorithm was able to find the best action to perform when it was only two moves away from checkmate [1], known as the mate-in-two problem; unfortunately the program was not able to play a full game due to the low computational power of the used machine, Ferranti Mark I. In 1962, Alan Kotok et al. designed Kotok-McCarthy, which was the first computer program to play Chess convincingly. It used Minimax with alpha-beta pruning and a single move took five to twenty minutes. In 1974, *Kaissa*, a program developed by Georgy Adelson-Velsky et al., became the first world computer chess champion. Kaissa was the first program to use bitboard (a special data structure), contained an opening book (set of initial moves known to be good) with 10000 moves, used a novel algorithm for move pruning, and could search during the opponent's move. The first computer which was able to defeat a human player was *Deep Thought* in 1989 [34]. The machine, created by the computer scientist of the IBM Feng-hsiung Hsu, defeated the Master of Chess David Levy, in a challenge issued by the latter. Later, Hsu entered in the *Deep Blue* project, a computer designed by the IBM to play Chess only. The strength of Deep Blue was due to its high computational power, indeed it was a massively parallel computer with 480 processors. The algorithm to play Chess was written in C and was able to compute 100 million of positions per second. Its evaluation functions were composed by parameters determined by the system itself, analyzing thousands of champions' games. The program's knowledge of Chess has been improved by the grandmaster Joel Benjamin. The opening library was provided by grandmasters Miguel Illescas, John Fedorowicz, and Nick de Firmian [38]. In 1996 Deep Blue became the first machine to win a chess game against the reigning world champion Garry Kasparov under regular time controls. However, Kasparov won three and drew two of the following five games, beating Deep Blue. In 1997 Deep Blue was heavily upgraded and it defeated Kasparov, becoming the first computer system to defeat a reigning world champion in a match under standard chess tournament time controls. The performance of Chess software are continuously improving. In 2009 the software *Pocket Fritz 4*, installed on a smart phone, won a category 6 tournament, being able to evaluate about 20000 positions per second [35].

2.1.3 Go

Another widely studied board game in the field of artificial intelligence is *Go*, the most popular board game in Asia. The board of Go is a square of 19 cells and basically a player can put a stone wherever he wants, therefore it has a very high branching factor in search trees (361 in the first ply), hence it is not possible to use the traditional methods such as Minimax. The first Go program was created in the '60s, when D. Lefkovitz [9] developed an algorithm based on pattern matching. Later Zobrist [9] wrote the first program able to defeat an amateur human player. The Zobrist's program was mainly based on the computation of a potential function that approximated the influence of stones. In the '70s Bruce Wilcox [9] designed the first Go program able to play better than an absolute beginner. His algorithm used abstract representations of the board and reasoned about groups. To do this he developed the theory of *sector lines*, dividing the board into zones. The next breakthroughs were at the end of the '90s, when both abstract data structures and patterns where used [23]. These techniques obtained decent results, being able to compete against player at higher level than beginners, however the best results was found with the Monte Carlo Tree Search algorithm, that we discuss in Section 2.4.

2.1.4 Other Games

Thanks to the good results with Checkers, Chess, and Go, the interest of artificial intelligence was extended to other games, one of them is *Backgammon*. The main difficulty in creating a good artificial intelligence for this game is the chance event related to the dice roll. This uncertainty makes the use of the common tree search algorithm impossible. In 1992 Gerry Tesauro [23] combining the learning method of Samuel with neural networks techniques was able to design an accurate evaluator of positions. Thanks to hundreds of millions of training games, his program *TD-Gammon* is still considered one of the most strong player in the world.

In the '90s, programs able to play *Othello* were introduced. The main applications of artificial intelligence for this game was based on Minimax with alpha-beta pruning. In 1997 the program *Logistello*, created by Micheal Buro defeated the world champion Takaeshi Murakami. Nowadays Othello is solved for the versions with board dimensions 4 x 4 and 6 x 6. In the 8 x 8 version (the standard one), although it has not been proven mathematically, the computational analysis shows a likely draw. Instead, for the 10 x 10 version or grater ones, it does not exist any estimation, except of a strong likelihood of victory for the first player [36].

With the spread of personal computers, the research field of artificial intelligence has also been extended to modern board games. Since 1983 Brian Sheppard [29] started working on a program that can play the board game *Scrabble*. His program, called *Maven*, is still considered the best artificial intelligence to play Scrabble [40] and competes at the World Championships. Maven combines a selective move generator, the simulation of plausible game scenarios, and the B* search algorithm.

The artificial intelligence algorithms have been applied to many other games. Using these algorithms it was possible to solve games like *Tic Tac Toe*, *Connect Four*, *Pentamino*, *Gomoku*, *Nim*, etc [42].

2.2 Minimax

Minimax is a tree search algorithm that computes the move that minimize the maximum possible loss (or alternatively, maximize the minimum gain). The original version assumes a two-player zero-sum game but it has also been extended to more complex games. The algorithm starts from an initial state and builds up a complete tree of the game states, then it computes the best decision doing a recursive calculus which assumes that the first player tries to maximize his rewards, while the second tries to minimize the rewards of the former [23]. The recursive call terminates when reaches a terminal state, which returns a reward that is backpropagated in the tree according to the Minimax policy. The Minimax pseudo-code is given in Algorithm 1. Minimax turns out to be computationally expensive, mostly in games where the state space is huge, since the tree must be completely expanded and visited. *Alpha-beta pruning* is a technique that can be used to reduce the

Algorithm 1 Minimax pseudo-code

```
function MINIMAX($node, maximizingPlayer$)
    if $node$ is terminal then
        return the reward of $node$
    end if
    if $maximizingPlayer$ then
        $bestValue \leftarrow -\infty$
        for all child of $node$ do
            $val \leftarrow$ MINIMAX($child$,False)
            $bestValue \leftarrow$ MAX($bestValue, val$)
        end for
        return $bestValue$
    else
        $bestValue \leftarrow +\infty$
        for all child of $node$ do
            $val \leftarrow$ MINIMAX($child$,True)
            $bestValue \leftarrow$ MIN($bestValue, val$)
        end for
        return $bestValue$
    end if
end function
```

number of visited nodes, by stopping a move evaluation when it finds at least one possibility that proves the move to be worse than a previously examined one.

2.3 Artificial Intelligence in Card Games

Card games are challenging for artificial intelligence, because they are the classic example of imperfect information games and also involve multi-player interactions. The uncertainty on the hand of the opponent makes the branching factor of the traditional tree search method very high, therefore the conventional methods are often inadequate.

2.3.1 Bridge

A game that provided many ideas for research in this field is *Bridge* [37]. The game requires two players divided in two teams. In the early '80s Throop et al. [6] developed the first version of *Bridge Baron*, an artificial intelligence for the game of Bridge which used a planning technique called *hierarchical task network* (HTN) [30], based on the decomposition of tasks. HTN decomposes recursively the tasks creating smaller and smaller ones, until it reaches primitive tasks that can solve directly. Despite the excellent performance in the Baron Barclay World Bridge Computer Challenge, Bridge Baron was not able to compete with expert human players [30]. In 1999, Ginsberg et al. [23] developed a program called GIB (Ginsberg's Intelligent Bridge-player), which won the world championship of computer bridge in 2000. The Ginsberg's algorithm, instead of choosing the move for each possible configuration of cards in the game, based its evaluation on a random sample of 100 different cards ordering. GIB also used a technique called *explanation-based generalization* that computes and stores rules of excellent moves in different standard situations of the game [23]. Despite the excellent levels achieved by programs such as GIB, the research on artificial intelligence in the game of Bridge has not yet been able to reach the expert level [6].

2.3.2 Poker

Another game extensively studied in this context is *Poker*. In addition to imperfect information and non-cooperative multi-player, the central element in this game is the psychological factor of the *bluff*. In the early 2000s Darse Billings et al. [7] created the program *Poki*, which was able to play with reasonable accuracy at Poker, in its *Texas Hold'em* variant. The strategy of Poki was divided into three steps: (i) calculate the effective strength of the hand; (ii) use this information combined with an opponent's behavior modeling to create a probability distribution for the three possible actions (fold, call, raise); (iii) generate a random number to choose the action. The behavior modeling of each opponent was done by a feed-forward neural network, trained with data collected in online games between human players, improving accuracy by means of cross-validation with data collected in previous games of each player.

2.3.3 Mahjong

In 2009, Wan Jing Loh developed an artificial intelligence to play *Mahjong*. The program evaluated the strength of the hand, seeing it as a constraint satisfaction problem, and indicated the possibility of victory for each possible move. The method builds up a histogram which is then used to choose a move [19].

2.4 Monte Carlo Tree Search

The traditional artificial intelligence algorithms for games are very powerful but require high computational power and memory for problem with a huge state space or high branching factor. Methodologies to decrease the branching factor have been proposed, but they often rely on an evaluation function of the state in order to prune some branches of the tree. Unfortunately such function may not be easy to find and requires domain-knowledge experts. A possible algorithm to overcome these issues is the *Monte Carlo method*. This technique can be used to approximate the game-theoretic value of a move by averaging the reward obtained by playing that move in a random sample of games. Adopting the notation used by Gelly and Silver [17], the value of the move can be computed as

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} I_i(s,a)\, z_i$$

where $N(s,a)$ is the number of times action a has been selected from state s, $N(s)$ is the number of times a game has been played out through state s, z_i is the result od the ith simulation played out from s, and $I_i(s,a)$ is 1 if action a was selected from state s on the ith playout from state s or 0 otherwise. If the actions of a state are uniformly sampled, the method is called *Flat Monte Carlo* and this achieved good results in the games Bridge and Scrabble, proposed by Ginsberg [18] and Sheppard [29] respectively.

However Flat Monte Carlo fails over some domains, because it does not allow for an opponent model. Moreover, it has no game-theoretic guarantees, i.e even if the iterative process is executed for an infinite number of times, the move selected in the end may not be optimal.

In 2006 Rémi Coulom et al. combined the traditional tree search with the Monte Carlo method and provided a new approach to move planning in computer Go, now known as *Monte Carlo Tree Search* (MCTS) [13]. Shortly after, Kocsis and Szepesvári formalized this approach into the *Upper Confidence Bounds for Trees* (UCT) algorithm, which nowadays is the most used algorithm of the MCTS family. The idea is to exploit the advantages of the two approaches and build up a tree in an incremental and asymmetric manner by doing many random simulated games. For each iterations of the algorithm, a *tree policy* is used to find the most urgent node of the current tree, it seeks to balance the exploration, look at areas which are not yet sufficiently visited, and the exploitation, look at areas which can returns a high reward. Once the node has been selected, it is expanded by taking an available move and a child node is added to it. A simulation is then run from the child node and the result is backpropagated in the tree. The moves during the simulation step are done according to a *default policy*, the simplest way is to use a uniform random sampling of the moves available at each intermediate state. The algorithm terminates when a limit of iterations, time or memory is reached, for this reason MCTS is an *anytime* algorithm, i.e. it can be stopped at any moment in time returning the current best move. A Great benefit of MCTS is that the intermediate states do not need to be evaluated, as for Minimax with alpha-beta pruning, therefore it does not require a great amount of domain knowledge, usually only the game's rules are enough.

2.4.1 State of the art

Monte Carlo Tree Search attracted the interest of researchers due to the results obtained with *Go* [4], for which the traditional methods are not able to provide a competitive computer player against humans. This is due to the fact that Go is a game with a high branching factor, a deep tree, and there are also no reliable heuristics for nonterminal game positions (states in which the game can still continue, i.e. is not terminated) [10]. Thanks to its characteristics, MCTS achieved results that classical algorithms have never reached.

Hex is a board game invented in the '40s, is played on a rhombus board with hexagonal grid with dimension between 11 x 11 and 19 x 19. Unlike Go, Hex has a robust evaluation function for the intermediate states, which is why is possible to create good artificial intelligence using alpha-beta pruning techniques [3]. Starting in 2007, Arneson et al. [3] developed a program based on Monte Carlo Tree Search, able to play the board game Hex. The program, called *MoHex*, won the silver and the gold medal at Computer Olympiads in 2008 and 2009 respectively, showing that it is able to compete with the artificial intelligence based on alpha-beta pruning.

MTCS by itself is not able to deal with *imperfect information* game, therefore it requires the integration with other techniques. An example where MCTS is used in this type of games is a program for *Texas Hold'em Poker*, developed by M. Ponsen et al. in 2010 [22]. They integrated the MCTS algorithm with a Bayesian classifier, which is used to model the behavior of the opponents. The Bayesian classifier is able to predict both the cards and the actions of the other players. Ponsen's program was stronger than rule-based artificial intelligence, but weaker than the program Poki.

In 2011, Nijssen and Winands [21] used MCTS in the artificial intelligence of the board game *Scotland Yard*. In this game the players have to reach with their pawns a player who is hiding on a graph-based map. The escaping player shows his position at fixed intervals, the only information that the other players can access is the type of location (called *station*) where they can find the hiding player. In this case, MCTS was integrated with *Location Categorization*, a technique which provides a good prediction on the position of the hiding player. Nijssen and Winands showed that their program was stronger than the artificial intelligence of the game Scotland Yard for Nintendo DS, considered to be one of the strongest player.

In 2012, P. Cowling et al. [15] used the MCTS algorithm on a simplified variant of the game *Magic: The Gathering*. Like most of the card games, Magic has a strong component of uncertainty due to the wide assortment of cards in the deck. In their program, MCTS is integrated with *determinization* methods (Section 2.5.1). With this technique, during the construction of the tree, hidden or imperfect information is considered to be known by all players.

Cowling et al. [31] also developed, in early 2013, an artificial intelligence for *Spades*, a four players card game. Cowling et al. used *Information Set Monte Carlo Tree Search* (Section 2.5.2), a modified version of MCTS in which the nodes of the tree represents information sets (Section 2.5). The program demonstrated excellent performance in terms of computing time. It was written to be executed on an Android phone and to find an optimal solution with only 2500 iterations in a quarter of a second.

2.4.2 The Algorithm

The MCTS algorithm relies on two fundamental concepts:

- The expected reward of an action can be estimated doing many random simulations.
- These rewards can be used to adjust the search toward a best-first strategy.

The algorithm iteratively builds a partial game tree where the expansion is guided by the results of previous explorations of that tree. Each node of the tree represents a possible state of the domain and directed links to child nodes represent actions leading to subsequent states. Every node also contains statistics describing at least a reward value and the number of visits. The tree is used to estimate the rewards of the actions and usually they become more accurate as the tree grows. The iterative process ends when a certain computational budget has been reached, it can be a time, memory or iteration constraint. With this approach MCTS is able to expand only the most promising areas of the tree avoiding to waste most of the computational budget in less interesting moves. At whatever point the search is halted, the current best performing root action is returned.

The basic algorithm can be divided in four steps per iteration, as shown in Figure 2.1:

- *Selection*: Starting form the root node n_0, it recursively selects the most urgent node according to some utility function until a node n_n is reached that either represents a terminal state or is not fully expanded (a node representing a state in which there are possible actions that are not outgoing arcs from this node because they have not been expanded yet). Note that can be selected also a node that is not a leaf of the tree because it has not been fully expanded.
- *Expansion*: If the state s_n of the node n_n does not represent a terminal state, then one or more child nodes are added to n_n to expand the tree. Each child node n_l represents the state s_l reached from applying an available action to state s_n.
- *Simulation* (or *Rollout* or *Playout*): A simulation is run from the new nodes n_l according to the default policy to produce an outcome (or reward) Δ.

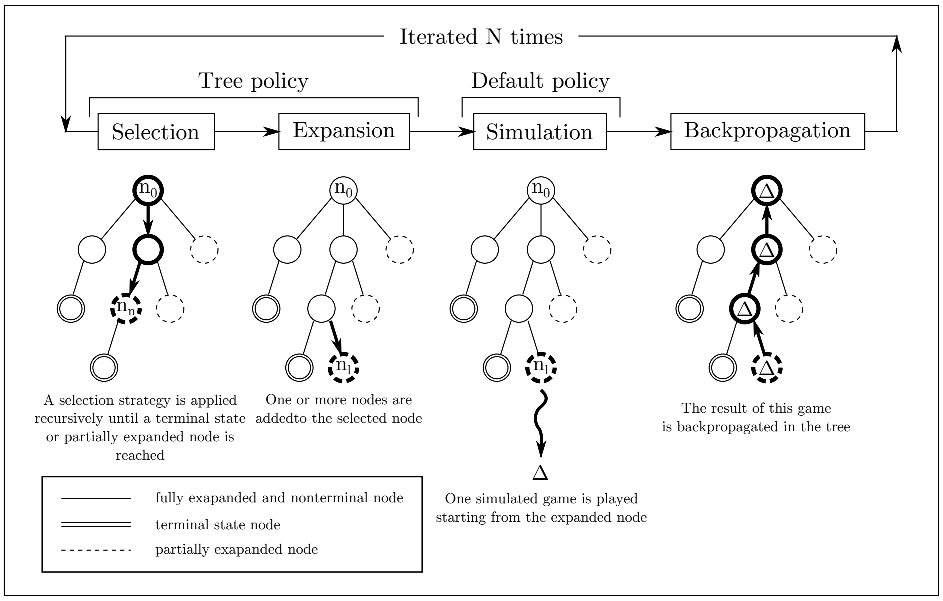

Iterated N times
Tree policy
Default policy
Selection
Expansion
Simulation
Backpropagation
n0
nn
n1
Δ
A selection strategy is applied recursively until a terminal state or partially expanded node is reached
One or more nodes are addedto the selected node
One simulated game is played starting from the expanded node
The result of this game is backpropagated in the tree
fully exapanded and nonterminal node
terminal state node
partially exapanded node

Figure 2.1: Steps of the Monte Carlo Tree Search algorithm.

- *Backpropagation*: Δ is backpropagated to the previous selected nodes to updates their statistics; usually each node's visits count is incremented and its average rewards updated according to Δ.

These can also be grouped into two distinct policies:

- *Tree policy*: Select or create a leaf node from the nodes already contained in the search tree (Selection and Expansion).
- *Default policy*: Play out the domain from a given nonterminal state to produce a value estimate (Simulation).

These steps are summarized in Algorithm 2. In this algorithm $s(n)$ and $a(n)$ are the state and the incoming action of the node n. The result of the overall search $a(\textsc{BestChild}(n_0))$ is the action that leads to the best child of the root node n_0, where the exact definition of "best" is defined by the implementation. Four criteria for selecting the winning action have been described in [25]:

- *max child*: select the root child with the highest reward.
- *robust child*: select the most visited root child.
- *max-robust child*: select the root child with both the highest visits count and the highest reward; if none exists, then continue searching until an acceptable visits count is achieved.

Algorithm 2 Monte Carlo Tree Search

```
1: function MCTS($s_0$)
2:     create root node $n_0$ with state $s_0$
3:     while within computational budget do
4:         $n_l \leftarrow$ TreePolicy($n_0$)
5:         $\Delta \leftarrow$ DefaultPolicy($s(n_l)$)
6:         Backpropagate($n_l, \Delta$)
7:     end while
8:     return $a$(BestChild($n_0$))
9: end function
```

- *secure child*: select the child which maximizes a lower confidence bound.

Note that since the MCTS algorithm does not force a specific policy, but leaves the choice of the implementation to the user, it is more correct to say that MCTS is a family of algorithms.

2.4.3 Upper Confidence Bounds for Trees

Since MCTS algorithm leaves the choice of the tree and default policy to the user, in this section we present the *Upper Confidence Bounds for Trees* (UCT) algorithm which has been proposed by Kocsis and Szepesvári in 2006 and is the most popular MCTS algorithm.

MCTS uses the tree policy to select the most urgent node and recursively expand the most promising parts of the tree, therefore the tree policy plays a crucial role in the performance of the algorithm. Kocsis and Szepesvári proposed the use of the *Upper Confidence Bound* (UCB1) policy which has been proposed to solve the *multi-armed bandit problem.* In this problem one needs to choose among different actions in order to maximize the cumulative reward by consistently taking the optimal action. This is not an easy task because the underlying reward distributions are unknown, hence the rewards must be estimated according to past observations. One possible approach to solve this issue is to use the UCB1 policy, which takes the action that maximizes the UCB1 value defined as

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

where $\bar{X}_j \in [0, 1]$ is the average reward from action j, n_j is the number of times action j was played, and n is the total number of plays. This formula faces the *exploitation-exploration dilemma*: the first addendum considers the current best action; the second term favors the selection of less explored actions.

The UCT algorithm takes the same idea of the UCB1 policy applying it to the selection step, it treats the choice of the child node to select as a multi-armed bandit problem. Therefore, at each step, it selects the child node n' that maximizes the UCT value defined as

$$UCT(n') = \frac{Q(n')}{N(n')} + 2C_p\sqrt{\frac{\ln N(n)}{N(n')}}$$

where $N(n)$ is the number of times the current node n (the parent of n') has been visited, $N(n')$ is the number of times the child node n' is visited, $Q(n')$ is the total reward of all playouts that passed through node n', and $C_p > 0$ is a constant. The term $\bar{X}_j$ in the UCB1 formula is replaced by ${}^{Q(n')}/_{N(n')}$ which is the actual average reward obtained from all the playouts. When $N(n')$ is zero, i.e. the child has not been visited yet, the UCT value goes to infinity, hence the child is going to be selected by the UCT policy. This is why in the selection step we use the UCT formula only when all the children of a node have been visited at least once. As in the UCB1 formula, there is the balance between the first (exploitation) and second (exploration) term. The contribution of the exploration term decreases as each node n' is visited, because it is at the denominator. On the other hand, the exploration term increases when another child of the parent node n is visited. In this way the exploration term ensures that even low-reward children are guaranteed to be selected given sufficient time. The constant in the exploration term C_p can be chosen to adjust the level of exploration performed. Kocsis and Szepesvári showed that $C_p = {}^{1}/_{\sqrt{2}}$ is optimal for rewards $\Delta \in [0, 1]$, this leads to the same exploration term of the UCB1 formula. If the rewards are not in this range, C_p may be determined from empirical evaluation. Kocsis and Szepesvári also proved that the probability of selecting a suboptimal action at the root of the tree converges to zero at a polynomial rate as the number of iterations grows to infinity. This means that, given enough time and memory, UCT converges to the Minimax tree and is thus optimal. Algorithm 3 shows the UCT algorithm in pseudocode. Each node n contains four pieces of data: the associated state $s(n)$, the incoming action $a(n)$, the total simulation reward $Q(n)$, and the visits count $N(n)$. $A(s)$ is the set of possible actions in state s and $f(s, a)$ is the state transition function, i.e. it returns the state s' reached by applying action a to state s. When a node is created, its values $Q(n)$ and $N(n)$ are set to zero. Note that in the UCT formula used in line 31 of the algorithm, the constant $c = 1$ means that we are using $C_p = 1/\sqrt{2}$.

2.4.4 Benefits

MCTS offers three main advantages compared with traditional tree search techniques:

- *Aheuristic*: it does not require any strategic or tactical knowledge about the given game, it is sufficient to know only its legal moves and end conditions. This lack of need for domain-specific knowledge makes it applicable to any domain that can be modeled using a tree, hence the same MCTS implementation can be reused for a number of games with minimum modifications. This is the main characteristic that allowed MCTS to succeed in computer Go programs, because its huge branching factor and tree depth make it difficult to find suitable heuristics for the game. However, in its basic version, MCTS can have low performance and some domain-specific knowledge can be included in order to significantly improve the speed of the algorithm.

- *Anytime*: at the end of every iteration of the MCTS algorithm the whole tree is updated with the last calculated rewards and visits counts through the backpropagation step. This allows the algorithm to stop and return the current best root action at any moment in time. Allowing the algorithm for extra iterations often improves the result.

- *Asymmetric*: the tree policy allows spending more computational resources on the most promising areas of the tree, allowing an asymmetric growth of the tree. This makes the tree adapts to the topology of the search space and therefore makes MCTS suitable for games with high branching factor such as Go.

2.4.5 Drawbacks

Besides the great advantages of MCTS, there are also few drawbacks to take into consideration:

- *Playing Strength*: the MCTS algorithm may fail to find effective moves for even games of medium complexity within a reasonable amount of time. This is mostly due to the sheer size of the combinatorial move space and the fact that key nodes may not be visited enough times to give reliable estimates. Basically MCTS might simply ignore a deeper tactical moves combination because it does not have enough resources

Algorithm 3 UCT algorithm

```
function UCT($s_0$)
    create root node $n_0$ with state $s_0$
    while within computational budget do
        $n_l \leftarrow$ TreePolicy($n_0$)
        $\Delta \leftarrow$ DefaultPolicy($s(n_l)$)
        Backpropagate($n_l, \Delta$)
    end while
    return $a$(BestChild($n_0$))
end function

function TreePolicy($n$)
    while $s(n)$ is nonterminal do
        if $n$ is not fully expanded then
            return Expand($n$)
        else
            $n \leftarrow$ BestUctChild($n, c$)
        end if
    end while
    return $n$
end function

function Expand($n$)
    $a \leftarrow$ choose untried actions from $A(s(n))$
    add a new child $n'$ to $n$
    $s(n') \leftarrow f(s(n), a)$
    $a(n') \leftarrow a$
    return $n'$
end function

function BestUctChild($n, c$)
    return $\arg\max_{n' \in \text{children of } n} \frac{Q(n')}{N(n')} + c\sqrt{\frac{2 \ln N(n)}{N(n')}}$
end function
```

```
function DefaultPolicy($s$)
    while $s$ is nonterminal do
        $a \leftarrow$ choose uniformly at random from $A(s)$
        $s \leftarrow f(s, a)$
    end while
    return reward for state $s$
end function

function Backpropagate($n, \Delta$)
    while $n$ is not null do
        $N(n) \leftarrow N(n) + 1$
        $Q(n) \leftarrow Q(n) + \Delta$
        $n \leftarrow$ parent of $n$
    end while
end function
```

to explore a move near the root, which initially seems to be weaker in respect to the others.

- *Speed*: MCTS requires many iterations to converge to a good solution, for many applications that are difficult to optimize this can be an issue. Luckily, there exists a lot of improvements over the basic algorithm that can significantly improve the performance.

2.5 Monte Carlo Tree Search in Games with Imperfect Information

As we have shown, MCTS has been applied successfully in *deterministic game with perfect information*, i.e. games in which each player perfectly knows the current state of the game and there are no *chance events* (e.g. draw a card from a deck, dice rolling) during the game. However, there are a lot of games in which there is not one or both of the two components: these type of games are called *stochastic* (chance events) *game with imperfect information* (partial observability of states).

Stochastic games with imperfect information provide an interesting challenge for AI research because, in such a way, the algorithms have to consider all the possible states that are compatible with the observed information or all the possible outcomes that can result from a chance event. In a tree search approach, this results in a huge state space and high branching factor.

Moreover, opponent modeling is more important as the opponent's policy generally depends on their hidden information, hence guessing the former allows the latter to be inferred. Furthermore, players may be able to infer some opponent's hidden information from the actions they make, and then may be able to mislead their opponents into making incorrect inferences. All of these situations lead to an increased complexity in decision making and opponent modeling compared to games with perfect information.

In this thesis, we mainly focus in game with imperfect information because Scopone does not have chance events, by the way we show some methods that can deal also with that. Imperfect information games can have three type of uncertainty [14]:

- *Information set*: is the set of all possible states in which a game can be, given the player's observed information. For instance, in a card game the player knows its cards, hence the information set contains all states which correspond to all possible permutations of opponent cards. By definition, the player knows in which information set he is, but not which state within that information set.
- *Partially observable move*: is a move performed by a player, but some information about that move are hidden from the opponents.
- *Simultaneous move*: is a move performed by a player without knowing which move the opponents have done simultaneously. The effect of the move is resolved when all the players have performed their move. The well-known game of *Rock-Paper-Scissors* is an example of this.

In Scopone there are not partially observable and simultaneous moves, therefore we mainly discuss information sets.

2.5.1 Determinization technique

One approach to designing AI for games with stochasticity and/or imperfect information is *determinization*. A determinization is a conversion of a stochastic game with imperfect information to a deterministic game with perfect information, in which the hidden information and the outcomes of all future chance events are fixed and known. In other words, a determinization is a sampled state in the current information set of the game. For example, the determinization of a card game is an instance of the current game in which the player knows the opponents' hand and the deck is completely visible. The decision for the original game is made by sampling several determinizations from the current game state, analyzing each one using AI techniques for deterministic games of perfect information, and combining these decisions. This method is also called *Perfect Information Monte Carlo Sampling.*

Determinization has been applied successfully to games such as Bridge [18], Magic: The Gathering [15], and Klondike Solitaire [8]. However, Russell and Norvig [23] describe it as "averaging over clairvoyance". They point out that determinization will never make a move that causes an opponent to reveal some hidden information or avoids revealing some of the player's hidden information to an opponent. Moreover, Frank and Basin [16] identify two key problems with determinization:

- *Strategy fusion*: In imperfect information game we must find a strategy that chooses the same move regardless the actual state within the current information set, because we cannot distinguish between these states. However, we broke this constraint by applying a deterministic AI algorithm to many determinizations, because in different determinizations we can make a different decision in two states within the same information set.

- *Nonlocality*: It occurs when an opponent has some knowledge of the actual game state and plays toward parts of the game tree that are most favorable in the determinizations he expects. Hence, some portions of the tree may never be reached under particular determinization. This leads the deterministic AI algorithm to erroneously consider rewards that are meaningless in that portion of the tree.

Starting from the work of Frank and Basin, Long et al. [20] proposed three parameters that can be used to determine when a game tree is suited to apply determinization successfully:

- *Leaf Correlation*: gives the probability that all the sibling terminal nodes have the same reward value. Low leaf correlation means that a player can always affect his reward also later in the tree.

- *Bias*: measures the probability that the game will favor one of the players.

- *Disambiguation factor*: determines how quickly the hidden information is revealed during the game, in other words, how quickly the states within player's information set decrease with regard to the depth of the tree.

The study found that determinization performs poorly in games where the leaf correlation is low or disambiguation is either very high or very low. The effect of bias was small in the examples considered and largely dependent on the leaf correlation value. Cowling et al. [14] showed another weakness of the algorithm: it must share the computational budget between the sampled perfect information games. The trees often have many nodes in common, but the determinization approach does not exploit it.

2.5.2 Information Set Monte Carlo Tree Search

In 2012 Cowling et al. [14] proposed the *Information Set Monte Carlo Tree Search* (ISMCTS) family of algorithms. They presented three type of ISMCTS algorithms:

- *Single-Observer ISMCTS* (SO-ISMCTS): It assumes a game with no partially observable moves and it resolves the issue of strategy fusion arising from the fact that a deterministic solver may make different decisions in each of the states within an information set.
- *SO-ISMCTS With Partially Observable Moves* (SO-ISMCTS+POM): It is an extension of SO-ISMCTS that can deal with partially observable moves. It resolves the issue of strategy fusion arising from the fact that a deterministic solver will assume that a partially observable move can be observed and that it can make a different decision depending on the actual move made by the opponent. However, it uses a weak opponent modeling since it is assumed that the opponent chooses randomly between actions that are indistinguishable to the root player.
- *Multiple-Observer ISMCTS* (MO-ISMCTS): It resolves the issue introduced with SO-ISMCTS+POM by maintaining a separate tree for each player and during each iteration the trees are searched simultaneously.

These algorithms can also deal with chance events and simultaneous moves. Handling of chance events can be done introducing an *environment player* that acts in correspondence of chance nodes. Chance nodes have as outgoing arcs all the possible outcomes of the corresponding chance event. The environment player gets always a reward of zero ensuring that all the outcomes are selected uniformly by the UCT formula. Simultaneous moves can be modeled by having players choose their actions sequentially, while hiding their choices from the other players, until finally an environment player reveals the chosen actions and resolves their effects. This approach obviously requires the handling of partially observable moves. From now on, we refer to SO-ISMCTS as ISMCTS because partially observable moves are not present in Scopone, therefore are out of the scope of this work.

The Algorithm

The idea of ISMCTS is to construct a tree in which nodes represents information sets rather than states. Each node represents an information set from the root player's point of view and arcs correspond to moves played by the corresponding player. The outgoing arcs from an opponent's node have to represent the union of all moves available in every state within that information set, because the player cannot know the moves that are really available to the opponent. However, the selection step has to be changed because the probability distribution of the moves is not uniformly distributed: a move may be available only in a less significant part of the states within the information set. This can cause a move to be considered as optimal, when it is actually almost never available. To address this issue, at the beginning of each iteration, a determinization is sampled from the root information set and the following steps of the iteration is restricted to regions that are consistent with that determinization. In this way, the probability of a move being available for selection on a given iteration is precisely the probability of sampling a determinization in which that action is available. The set of moves available at an opponent's node can differ between visits to that node, hence we have to use the *subset-armed bandit problem* for the selection step. This is handled with a simple modification in the UCT formula (Section 2.4.3):

$$ISUCT(n') = \frac{Q(n')}{N(n')} + 2C_p\sqrt{\frac{\ln N'(n')}{N(n')}}$$

where $N'(n')$ is the number of times the current node n (the parent of n') has been visited and node n' was available for selection, $N(n')$ is the number of times the child node n' was visited, $Q(n')$ is the total reward of all playouts that passed through node n', and $C_p > 0$ is a constant. Without this modification, rare moves are over-explored: whenever they are available for selection the term $\frac{\ln N(n)}{N(n')}$, where $N(n)$ is the number of times the current node n (the parent of n') has been visited, is very high, resulting in a disproportionately large UCT value. The ISMCTS pseudo-code is given in Algorithm 4. This pseudo-code uses the following notation:

- $c(n)$ = children of node n.
- $a(n)$ = incoming move at node n.
- $N(n)$ = visits count for node n.
- $N'(n)$ = availability count for node n.
- $Q(n)$ = total reward for node n.
- $A(d)$ = set of possible moves in determinization d.
- $f(d, m)$ = state transition function, i.e. it returns the determinization d' reached by applying move m to determinization d.
- $c(n, d) = \{n' \in c(n) : a(n') \in A(d)\}$, the children of n compatible with determinization d.
- $u(n, d) = \{m \in A(d) : \nexists n' \in c(n, d) \text{ with } a(n') = m\}$, the moves from d for which n does not have children in the current tree.

When a node is created, its values $Q(n)$ and $N(n)$ are set to 0, and $N'(n)$ is set to 1.

Benefits and Drawbacks

ISMCTS inherits the same benefits of the MCTS algorithm, in addition it can deal with imperfect information games and resolves the issue of strategy fusion. However, the nonlocality problem is still present, but can be resolved with techniques such as opponent modeling and calculation of belief distributions. ISMCTS has also the advantage of focusing all the computational budget on a single tree, whereas a determination technique would split it among several trees. This usually allows for a deeper search, however, for some games, the branching factor can drastically increase due to the high number of available moves at each information set, hence the performance may decrease.

2.6 Summary

In this chapter we overviewed applications of artificial intelligence in board and card games. We discussed artificial intelligence for Checkers, Chess, Go, and other board games. Then, we showed the well-known AI algorithm, Minimax, that can deal with deterministic game with perfect information. Next, we presented AI programs able to play the card games Bridge, Poker,

Algorithm 4 ISMCTS algorithm

```
 1: function ISMCTS($IS_0$)
 2:     create root node $n_0$ with information set $IS_0$
 3:     while within computational budget do
 4:         $d_0 \leftarrow$ randomly choose a determinization $\in IS_0$
 5:         $(n_l, d_l) \leftarrow$ TreePolicy($n_0, d_0$)
 6:         $\Delta \leftarrow$ DefaultPolicy($d_l$)
 7:         Backpropagate($n_l, \Delta$)
 8:     end while
 9:     return $a$(BestChild($n_0$))
10: end function
11:
12: function TreePolicy($n, d$)
13:     while $d$ is nonterminal do
14:         if $u(n, d) \neq 0$ then
15:             return Expand($n, d$)
16:         else
17:             for all $n' \in c(n, d)$ do
18:                 $N'(n') \leftarrow N'(n') + 1$
19:             end for
20:             $n \leftarrow$ BestIsuctChild($n, d, c$)
21:             $d \leftarrow f(d, a(n))$
22:         end if
23:     end while
24:     return $n$
25: end function
26:
27: function Expand($n, d$)
28:     $m \leftarrow$ randomly choose a move $\in u(n, d)$
29:     add a new child $n'$ to $n$
30:     $d \leftarrow f(d, m)$
31:     $a(n') \leftarrow m$
32:     return $(n', d)$
33: end function
34:
35: function BestIsuctChild($n, d, c$)
36:     return $\arg\max_{n' \in c(n,d)} \frac{Q(n')}{N(n')} + c\sqrt{\frac{2 \ln N'(n')}{N(n')}}$
37: end function
```

```
function DefaultPolicy($d$)
    while $d$ is nonterminal do
        $a \leftarrow$ choose uniformly at random from $A(d)$
        $d \leftarrow f(d, a)$
    end while
    return reward for state $d$
end function

function Backpropagate($n, \Delta$)
    while $n$ is not null do
        $N(n) \leftarrow N(n) + 1$
        $Q(n) \leftarrow Q(n) + \Delta$
        $n \leftarrow$ parent of $n$
    end while
end function
```

and Mahjong, showing how they handle the imperfect information of these games. Then, we introduced the Monte Carlo Tree Search algorithm, showing the state of the art, the basic algorithm, the Upper Confidence Bounds for Trees implementation, and benefits/drawback with respect to Minimax. Finally, we discussed the use of MCTS in imperfect information games, showing the determinization technique and the Information Set Monte Carlo Tree Search algorithm.

Chapter 3

Scopone

In this chapter we present the card game *Scopone* starting from a brief history of the game, the cards used, the rules, and some variants of the original game.

3.1 History

Scopone is a very ancient card game played in Italy. It is a four players extension of another Italian two players game named *Scopa* [41], in fact Scopone means “a big Scopa”. The origin of the game are unknown, although many authors consider that the first book about Scopone has been written by Chitarrella [11] in 1750. In his book, Chitarrella defines 44 rules, that include the rules of the game and basic strategies. Some of these strategies, however, have been criticized by many authors, for instance Saracino [24] and Cicuti-Guardamagna [12] commented on the rules about the handle of 7s. Chitarrella also wrote that the game was very spread in Italy. Since Scopone is very difficult to play and, at that time, most of the people were illiterate, Saracino argued that probably Scopone has been played for centuries before Chitarrella. Initially, it might have been played only by aristocrats who might have been formidable players. Later, the game might have spread among less cultured classes and handed down orally from generation to generation until Chitarrella formally wrote it down. Unfortunately, nobody found a proof that Chitarrella wrote these rules in 1750, the oldest book of Chitarrella ever found is dated 1937. Therefore, the first book about Scopone might be the one written by Capecelatro in 1855, “Del giuoco dello Scopone”. Capecelatro wrote that the game was known by 3-4 generations, therefore it might have been born in the eighteenth century. He also wrote that the game was invented by four notables of a little town, who got bored by playing Scopa and found a way to play it in four. Later, the book of Capecelatro has been followed by several other publications that introduced more advanced strategies. In 1969, Unione Italiana Gioco Scopone (UIGC) and Federazione Svizzera Gioco Scopone (FSGS) wrote the official rules of Scopone used still now in every tournament. Nowadays, Federazione Italiana Gioco Scopone [28] (FIGS), located in Naples, is the reference for all Scopone enthusiasts. On their website there are the full list of publications and a lot of other information about Scopone.

3.2 Cards

Scopone is played with the traditional Italian deck composed by 40 cards, including 10 ranks of each of the four suits, *coins* (Denari in Italian), *swords* (Spade), *cups* (Coppe), and *batons* (Bastoni). Each suit contains, in increasing order, an ace (or one), numbers two through seven, and three face cards. The three face cards are: *knave* (Fante or eight); *horse* (Cavallo or nine) or *mistress* (Donna or nine); and *king* (Re or ten). This deck is not internationally known like the 52 cards French deck, probably due to the fact that each region of Italy has its own cards style. For this reason the FIGS adopted an official deck (Figure 3.1) that merges the Italian and French suits in order to be understood easily by everyone. The correspondences are: coins with diamonds ($\diamondsuit$), swords with spades ($\spadesuit$), cups with hearts ($\heartsuit$), and batons with clubs ($\clubsuit$).

3.3 Game Rules

Scopone is played by four players divided into two teams. The players are disposed in the typical North-East-South-West positions and teammates are in front of each other. At the right of the dealer there is the *eldest hand* (in Italian "primo di mano"), followed by the *dealer's teammate* and the *third hand*. The team of the dealer is called *deck team*, whereas the other is called *hand team*. The game is composed by many rounds and ends when a team reaches a score of 11, 16, 21 or 31. If both the teams reach the final score, then the game continues until one of the team scores more than the other one.

3.3.1 Dealer Selection

At the beginning of the game, the dealer is randomly chosen through some procedure, because being the last player to play has some advantages. For

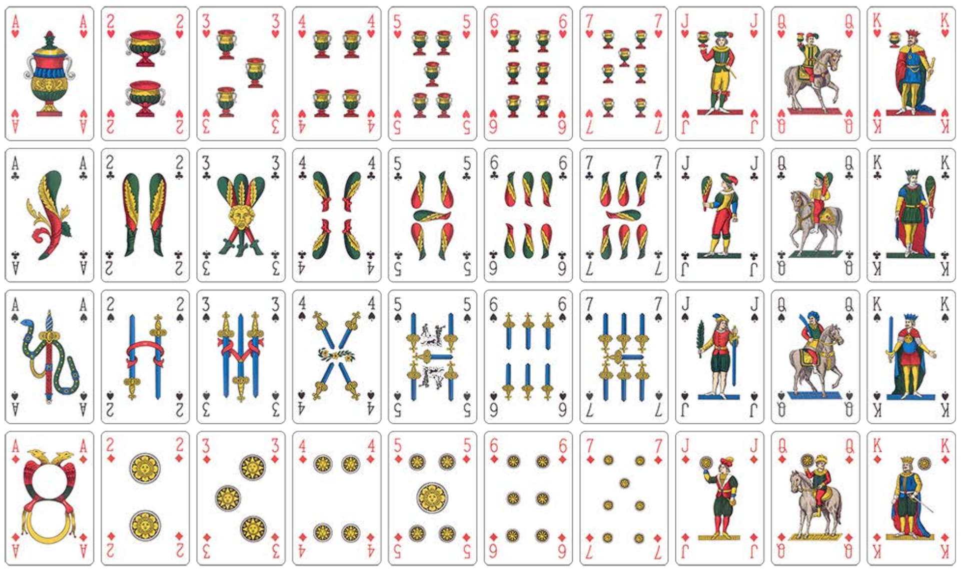


Figure 3.1: Official deck of Federazione Italiana Gioco Scopone.

example, the deck is shuffled and the player who draws the highest card deals; if there is a tie, the involved players continue to draw a card until one wins. Another method consists of dealing a card to each player until one gets an ace. In the following rounds the eldest hand of the previous round become the new dealer.

3.3.2 Cards Distribution

At the beginning of each round, the dealer shuffles the deck and offers it to the third hand for cutting. Then, it deals the cards counterclockwise three by three, starting with the eldest hand, for a total of nine cards for each player. The dealer must not reveal the cards, if this happens it must repeat the process. During the first distributions, the dealer also leaves twice on the table a pair of face-up cards, for a total of four cards. If, at the end of the distribution, on the table there are three kings, the dealer must repeat the process because three kings do not allow doing scopa for the whole round. At the end of the distribution the table looks like in Figure 3.2.

3.3.3 Gameplay

A round is composed by 36 turns. The eldest hand plays first, then it is the turn of the dealer teammate and so on counterclockwise. At each turn the player must play a card from his hand. The chosen card can either be placed on the table or capture one or more cards. A capture is made by matching

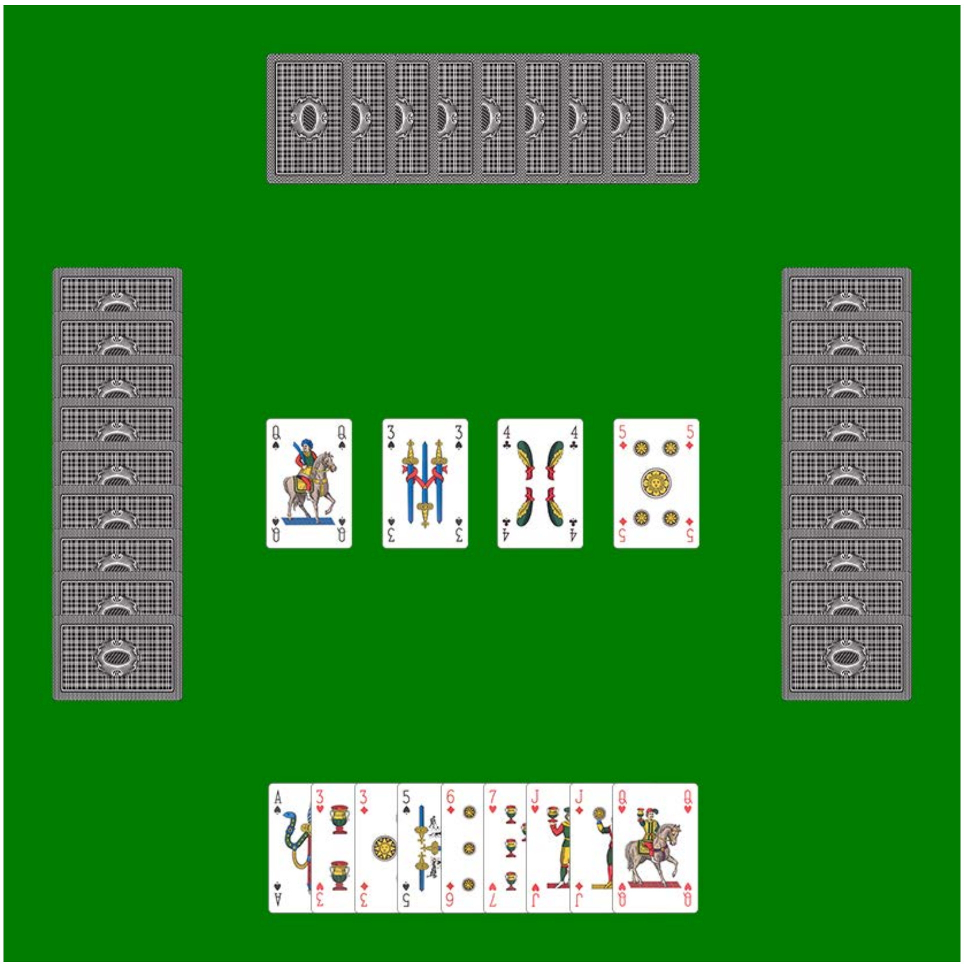


Figure 3.2: Beginning positions of a round from the point of view of one player. Each player holds nine cards and there are four cards on the table.

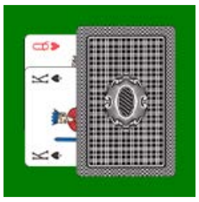


Figure 3.3: The player's pile when it did two scopa.

a card in the player's hand to a card of the same value on the table, or if that is not possible, by matching a card in the player's hand to the sum of the values of two or more cards on the table. In both cases, both the card from the player's hand and the captured card(s) are removed and placed face down in a pile in front of the player. Usually, the teammates share the same pile that is placed in front of one of them. These cards are now out of play until scores are calculated at the end of the round. For example, in Figure 3.2, the player may choose to place on the table the $A\spadesuit$ or capture the $3\spadesuit$ with $3\diamondsuit$ or capture the $3\spadesuit$ and $5\diamondsuit$ with the $J\diamondsuit$. Note that it is not legal to place on the table a card that has the ability to capture. For instance, in Figure 3.2, the player cannot place on the table the $Q\heartsuit$, because it can capture the $Q\spadesuit$. In case the played card may capture either a single or multiple cards, the player is forced to capture only the single card. For example, in Figure 3.2, the $Q\heartsuit$ cannot capture the $4\clubsuit$ and $5\diamondsuit$ because is forced to capture the $Q\spadesuit$. When the played card may capture multiple cards with different combinations, the player is allowed to choose the combination it prefers. If by capturing, all cards were removed from the table, then this is called a *scopa*, and the card from the player's hand is placed face up under the player's pile (in Figure 3.3 the player did two scopa), in order to take it into account when the scores are calculated. However, at the last turn, it is not allowed to do a scopa. This move is called scopa (in Italian means *broom*) because it looks like if all the cards in the table are swept by the played card. If, at the end of the round, there are still cards on the table, then these cards are placed in the pile of the last player who did a capturing move.

3.3.4 Scoring

When the round ends, the round's scores are calculated and added to the games's scores. There are five ways to award points:

- *Scopa*: a point for each scopa is awarded.
- *Cards*: the team who captured the largest number of cards gets one

Table 3.1: Cards' values for the calculation of primiera. Each column shows the correspondence between card's rank and used value.

Card's Rank	7	6	A	5	4	3	2	J	Q	K
Value	21	18	16	15	14	13	12	10	10	10

point.

- *Coins*: the team who captured the largest number of cards in the suit of coins gets one point.
- *Settebello*: the team who capture the seven of coins gets one point.
- *Primiera*: the team who obtain the highest prime gets one point. The prime for each team is determined by selecting the team's best card in each of the four suits, and summing those four cards' point values. Table 3.1 shows the cards' values used in this calculation. If a team does not have an entire suit, the point is awarded to the opponents, even if they have a minor sum.

The four points awarded with cards, coins, settebello, and primiera are called *deck's points*. Note that, for both cards, coins, and primiera, in case of a tie, everybody gets no point. Table 3.2 shows an example of a round's scores calculation. The hand team's primiera is calculated on the cards 7$\diamondsuit$ 7$\spadesuit$ 5$\heartsuit$ 7$\clubsuit$ $(21 + 21 + 15 + 21 = 78)$, whereas the deck team's primiera is calculated on the cards 6$\diamondsuit$ 6$\spadesuit$ 7$\heartsuit$ $A\clubsuit$ $(18 + 18 + 21 + 16 = 73)$.

3.4 Variants

There are many variants of Scopone that make the game more exciting and difficult. Each region of Italy has its own favorite variant and the variants can be also combined, for example in Milan area is common to play the variant that combines *Scopone a 10*, *Napola*, and *Scopa d'Assi*.

3.4.1 Scopone a 10

In this variant, the players are dealt ten cards each so that none is left on the table. This means that, initially, the hand team may let the opponents do a series of scopa and this makes the game more exiting. *Scopone Scientifico* (Scientific Scopone) is often said to refer to *Scopone a 10*, but FIGS states that both Scopone and *Scopone a 10* can be called *Scopone Scientifico*, because it means that the rules give it the dignity of Science, that both these variants have.

Table 3.2: Example of scores calculation. The first part shows the cards in the pile of each team at the end of a round. The second part shows the points achieved by each team and the final scores of the round.

Team's pile										
Hand team	5♠	5♡	4♢	*A*♠	3♣	3♢	3♡	7♣	7♠	*Q*♢
	Q♠	*Q*♡	*Q*♣	7♢	5♢	2♡	4♠	4♡	*K*♣	6♣
	J♣	4♣								
Deck team	*J*♢	*J*♡	*K*♢	*K*♡	6♢	6♠	*J*♠	2♠	6♡	5♣
	3♠	2♣	*A*♡	*A*♢	*K*♠	*A*♣	2♢	7♡		

Round's scores		
	Hand team	**Deck team**
Scopa	1	3
Cards	22	18
Coins	5	5
Settebello	1	0
Primiera	78	73
Scores	4	3

3.4.2 Napola

The team that capture the ace, two, and three of coins achieves the *Napola* and is awarded additional points equal to the highest consecutive coin they obtain, e.g. if a team captures the ace, two, three, four, five, and eight of coins, that team is awarded 5 additional points. This variant makes the game more enjoyable because the players who try to achieve the *Napola* can obtain many points.

3.4.3 Scopa d'Assi or Asso piglia tutto

In this variant, playing an ace captures all cards currently on the table but does not count as a scopa. When there is already an ace on the table (this may happens or not, depending on the other chosen variants), the player, who play an ace, can only capture the ace on the table. This is called "burning an ace" because it is wasting the power of the played ace. This variant change completely the strategy of the players, since a player can capture many cards that might never be captured without the ace.

3.4.4 Sbarazzino

This variant works exactly like *Scopa d'Assi*, but the ace count as a scopa. This bias the game toward fortune, because the players get points by having a lucky card distribution.

3.4.5 Rebello

The team who capture the king of coins gets one extra point, exactly like the settebello.

3.4.6 Scopa a 15

In *Scopa a 15* a capturing move can be done only if the played card and the captured cards sum up to 15. For example, suppose that the table is A 3 4 5 J, then by playing a 6, one can capture either A J $(6 + 1 + 8 = 15)$ or 4 5 $(6 + 4 + 5 = 15)$.

3.4.7 Scopa a perdere

In this variant the players aim to get the lowest possible score, basically the players have to invert their strategy.

3.5 Summary

In this chapter we presented the card game Scopone. We showed that the origin and creator of Scopone is unknown, and that the oldest book about Scopone has been written by Capecelatro in 1855 and not by Chitarrella in 1750. Then, we presented the 40 Italian cards used in FIGS tournaments, the game rules of Scopone, how the cards are distributed, the legal moves, and the scoring system. Finally, we illustrated some game variants that make it more enjoyable. In the next chapter, we see the three types of rules-based artificial intelligence we have designed for Scopone, that are based on rules taken from strategy books.

Chapter 4

Rule-Based Artificial Intelligence for Scopone

In this chapter, we present the three types of rule-based artificial intelligence that we have developed for Scopone: *Greedy*, *Chitarrella-Saracino*, and *Cicuti-Guardamagna*. The first one encodes simple rules in such a way it represents, more or less, the strategy of a beginner. The second one encodes a union of the rules written by Chitarrella and Saracino in [24], therefore it implements what we may consider an expert player. The last one encodes the rules written by Cicuti and Guardamagna in [12], they introduced several rules about the handle of 7s that can be added to the rules of Chitarrella and Saracino. We have developed these artificial intelligence for the purpose of compare them with the *Monte Carlo Tree Search* (MCTS) algorithms we have designed. In this way, we can validate the playing strength of MCTS against a beginner and expert players.

4.1 Greedy Strategy

The *Greedy* strategy represents the playing strength of a beginner that plays elementary using the basic rules of the game. Therefore, it's strategy is: if it is possible to capture some cards, then the most important ones must be captured; otherwise the least important one must be placed on the table. To define the meaning of a card to be important, we used a modified version of the cards' values for the primiera (Table 4.1), in which the value of the 7s is changed from 21 to 29, 10 points are added to cards in the suit of coins, and other 100 points are added to the 7$\diamondsuit$. Using almost the same values for the primiera ensures that the *Artificial Intelligence* (AI) tries to achieve the highest possible primiera score. The greatest value is assigned to the

Table 4.1: Cards' values used by the Greedy strategy to determine the importance of a card. Each column of the first part shows the correspondence between a card in the suit of coins and the used value. Each column of the second part shows the correspondence between a card's rank in one of the other suits and the used value.

Coins suit										
Card	7♢	6♢	A♢	5♢	4♢	3♢	2♢	J♢	Q♢	K♢
Value	139	28	26	25	24	23	22	20	20	20

Other suits										
Card's Rank	7	6	A	5	4	3	2	J	Q	K
Value	29	18	16	15	14	13	12	10	10	10

7♢, that alone it is one point for the settebello. The additional 10 points for the cards in the suit of coins ensures that the program try to capture the maximum number of coins card and therefore try to accomplish the coins point. The value 29 for the 7s is needed to let them more important than any card in the suit of coins, because they are important both for the primiera and the achievement of the settebello. The accomplishment of the cards point is ensured by the strategy itself, because if it can do a capturing move, then it will do it. Moreover, in case there are two possible moves capturing the greatest-value card, the AI will choose the one that maximize the sum of all the captured cards' values. For example, suppose that the table is 6♢ A♣ 4♢ 3♠ 6♡ and the player has the 7♢ in its hand. Obviously, besides the other cards it has, it is going to do a capturing move with the 7♢ because it is the greatest-value card, anyway it can do three possible capturing moves:

- Capture 6♢ and A♣, for a total value of 183 $(28 + 16 + 139)$.
- Capture 4♢ and 3♠, for a total value of 176 $(24 + 13 + 139)$.
- Capture 6♡ and A♣, for a total value of 173 $(18 + 16 + 139)$.

In this case, the AI will capture 6♢ and A♣ because it sums up to the greatest value.

We did not take into consideration the scopa points. This appears to be a very important aspect, since by doing many scopa one can easily overcome the deck's points of the opponents. There are two problems to take into account: (i) avoid allowing the opponents to do a scopa; (ii) force or not the scopa when it is possible. To solve the first one, the program applies the above strategy considering the set of moves that do not leave on the table a combination of cards that can be captured by a card that has *unknown* position. A card has *unknown* position if it is in the other players' hand, but the AI does not know who holds it. If such a set is empty, then it applies the above strategy considering all the moves. For the second problem, we decided not to force the scopa, because in some circumstances it may be better to skip it in favor of the capture of some important cards. Nevertheless, avoiding a scopa can leave on the table a combination of cards allowing the opponents to do a scopa. This is always avoided, hence, in most of the cases, it will be forced to do the scopa.

The Greedy AI is summarized in Algorithm 5. This pseudo-code uses the following notation:

- A game state has two properties: $legalMoves$, the set of legal moves from that state; $unKnownCards$, the set of cards that have unknown position from the point of view of the player about to act, i.e. the union of the cards held by the other players.
- A card is represented by its $rank$ and $suit$.
- A move has two properties: $playedCard$, the card played by the player making the move; $cards$, the set of cards involved in the move, i.e. both the played card and captured cards.
- $cardValue$ is a dictionary that relates each card with its value given in Table 4.1.

4.2 Chitarrella-Saracino Strategy

The *Chitarrella-Saracino* (CS) strategy aims to behave like an expert player of Scopone. In order to achieve this result, we took the rules of Chitarrella and the most important playing strategies from the book of Saracino [24], which have been summarized in [5].

4.2.1 The Spariglio

One of the most important rules of Scopone involves the *spariglio* (decoupling in English), that consists in playing a card that matches to the sum of the values of two or more cards on the table. For example, if we capture 3 and 2 with 5, then we did the spariglio $3 + 2 = 5$. We know that each card rank appears four times in the deck, because the number of suits. Therefore, at the beginning of a round, all the cards are *coupled*, in the sense that, by doing moves not involving the spariglio, each card is taken by its copy

Algorithm 5 Greedy strategy pseudo-code

```
function GreedyStrategy($state$)
    $moves \leftarrow state.legalMoves$
    $nonScopaMoves \leftarrow \{m \in moves : \neg \text{BringToScopa}(state, m)\}$
    if $nonScopaMoves \neq \emptyset$ then
        return BestMove($nonScopaMoves$)
    end if
    return BestMove($moves$)
end function

function BringToScopa($state, move$)
    $remainingSum \leftarrow \sum_{c \in \{\text{cards on the table after } move\}} c.rank$
    return $\exists c \in state.unKnownCards : c.rank = remainingSum$
end function

function BestMove($moves$)
    $capturingMoves \leftarrow \{m \in moves : m \text{ is a capturing move}\}$
    if $capturingMoves \neq \emptyset$ then
        $cardMaxValue \leftarrow \max_{m \in capturingMoves}$
                                  $\max_{c \in m.cards} cardValue[c]$
        $maximumMoves \leftarrow \{m \in capturingMoves :$
                         $\max_{c \in m.cards} cardValue[c] = cardMaxValue\}$
        return $\arg\max_{m \in maximumMoves} \sum_{c \in m.cards} cardValue[c]$
    end if
    return $\arg\min_{m \in moves} cardValue[m.playedCard]$
end function
```

until there are no more cards on the table at the end of the round. This advantages the deck team, because they always play after the opponents. For example, let us assume that all the players have the same cards, then, without doing some spariglio, all the cards will be captured by the deck team. When a player does a spariglio move, the involved cards that were *coupled*, or *even*, will become *decoupled*. Conversely, the involved cards that were *decoupled*, or *odd*, will return *coupled*. For example, if we have that 3 and 5 are coupled and 2 is decoupled, then by doing the spariglio $3+2=5$, 3 and 5 will become decoupled and 2 will return coupled.

CS1 This is the fundamental rule of Scopone: The dealer and his teammate seek to keep the number of cards of the same rank even, while the opponents try to decouple them. Keeping the cards even is needed to capture good cards during the game; hence, if the opponents decouple them, the deck team have to recouple them. For the same reason the opponents of the dealer have to decouple them as much as they can.

Because of this rule, each player has to remember all the odd cards and change its strategy accordingly. This is called "fare il quarantotto" ("do the forty-eight" in English). Chitarrella did not know the meaning of this word and Saracino tried to explain it by saying that 4-8 is the only spariglio of two cards that is *irreducible*, i.e it is not possible to do a spariglio that reduces the number of decoupled cards to one. Cicuti and Guardamagna seem to have found the final explanation: forty-eight is the sum of the cards involved in the spariglio where all cards but 7 are decoupled ($1+2+3+4+5+6+8+9+10 = 48$). In fact, it is easy to remember the number of 7s played during the game, because they are the most important cards, but only expert players can remember all the odd cards. Everyone has its own method to remember the decoupled cards. Saracino proposed to remember them in increasing order, whereas Cicuti and Guardamagna introduced a method using the fact that the number of odd-rank cards, in the decoupled ones, must be even.

CS2 Who is interested in maintaining the decoupling has to play the last decoupled card of the highest rank: in that case the recoupling is impossible.

For example, let us assume that someone did the spariglio $2+3=5$. The deck team will try to recouple these cards by playing 2 or 3 to favor the capture of 5. The hand team can avoid it by playing the last decoupled card of the highest rank, 5. In this way, it has to be captured by another 5 and the last one remains decoupled.

CS3 Whereas the hand team has to play the last decoupled card of the highest rank, the dealer's teammate must never do it, leaving to the dealer the choice of playing it when it is convenient.

For instance, when the dealer has no 7s and wants to cause a higher spariglio.

CS4 When the dealer holds the last two cards, having an even and an odd card, it has to play the odd one.

In this way, the dealer is almost sure to do the last capturing move with the even card that it has kept and take all the cards remaining on the table. This last card is called "tallone". This rule applies also when one has to choose between an odd 7 and an even card.

CS5 When the third hand holds the last two cards, having an even card and a decoupled card of higher rank, it has to play the even one.

For example, let the spariglio be 3-5-8-10 and the table be 5 K, both the last ones. The third hand holds 4 and the last J. It knows that the cards still in play are one 3 and three 4. If the dealer holds two 4, it will capture the 4 played by the third hand keeping the other 4 as tallone, but the third hand will capture 3 and 5 with J. Instead, if the dealer holds 3 and 4, it will play 3 because of CS4 and the third hand will capture 3 and 5 with J. Conversely, if the third hand plays J, then the dealer can simply play 3 by maintaining 4 as tallone.

4.2.2 The Mulinello

The *mulinello* (eddy in English) is a play combination in which two players continuously capture and the other two play without capturing any cards. It often happens at the beginning of the round, when the eldest hand can do a good capture on the four cards on the table. For example, let us assume that the table is 1♢ 3♡ 3♣ 6♠. The eldest hand captures 1♢ 3♡ 6♠ with K♢, challenging the fourth 3 because it holds the other one. The dealer's teammate, to avoid a scopa, plays Q♣ that the third hand captures with Q♡. The dealer follows the teammate and plays Q♠ that is captured by the eldest hand with Q♢, and so on, especially if the deck team does not have double or triple cards.

CS6 If it is possible to do the mulinello either on a low-rank or face card, then the hand team must always prefer to do it on the low-rank card.

It can also be applied to the deck team, because the mulinello on the low-rank card can bring to do, at some point, one or two scopa. But if, for practical reasons, the dealer does not want to create some spariglio, then it will be better to do the mulinello on the face card.

CS7 One must never play a *fourth card*, i.e the last card of a quartet of which three cards have already been captured, when there is the risk that the opponents do the mulinello on that fourth card.

For example, let us assume that there is on the table a K and it is known that the kings are in the opponents hand. Having the fourth 2, one must not play it, because the opponent will capture K causing the mulinello on 2. Another situation where one must avoid to play a fourth card is when there are three cards on the table that cannot be involved in a spariglio. In fact, by playing the fourth card, the opponent at its right will capture one of the three cards, its teammate the second one, and the opponent at its left the third one. In this way, a mulinello, in favor of the opponents, has been created on the fourth card that has been erroneously played.

CS8 At the beginning of a round there are on the table two cards of the same rank. It is advisable to capture none of these cards, even if one is in the suit of coins, unless it is possible to do the mulinello on another card.

In this case the mulinello is easily predictable. For example, let us assume that at the beginning of a round there are two Q on the table, another one is held by the eldest hand, and the last one by one of the opponents. If the eldest hand captures one of the queens, then the last Q is held by an opponent who can easily create the mulinello on that card. Of course, the fourth Q could be in the hand of the teammate, in this case they can create the mulinello against the opponents, but from the point of view of the eldest hand the fourth Q is more likely to be in the opponents hand.

4.2.3 Double and Triple Cards

Since in Scopone the players are not allowed to speak, there are some common rules and ways of playing that make your teammate aware of the cards you hold. Of course, also the opponents can guess the cards which one holds by its moves. Therefore, there is a trade off between what you want your teammate to known and what you want your opponents to know. Sometimes, it is also useful resorting to the *bluff* in order to confuse your opponents and capture some important card.

CS9 In general, when it is possible to capture some cards, one has to do it.

CS10 If it is not possible, one has to play a *double card*, i.e. a card of which one also holds the other card of the same rank.

CS11 If the opponent captures it and the teammate has the other card of the same rank, it must play it in order to create the mulinello on that card. This card is called "franca", because the last one is held by the player who first played it.

CS12 In general, the teammate has not to capture the double card of its partner, but it has to wait that one of the opponents captures it, creating the mulinello.

CS13 It is useless and detrimental playing the card of which the teammate has not replayed, because if it is held by the opponents, they can create the mulinello on that card.

The rule of playing double cards does not exclude the 7s, because it is better to risk one them instead of playing them at the end of the round without any hope.

CS14 Having double and triple cards, one has to play the triple ones first.

Having triple cards is never an advantage, especially when you have more than one triple, because the scope of action of the player is reduced. It is also a disadvantage to hold all single cards, because it is not possible to apply the rule CS10. The rule CS14 applies also to 7s. Beginners often hesitate to play one of the three 7 they hold when they have not the settebello, but, obviously, who holds it will never play it. It often happens that they remain with the three 7s as the last three cards and they are forced to play one of them even if it is possible to form a spariglio with a low-rank card like $7 + 2 = 9$. For this reason they often lose all the 7s.

CS15 It is better to play a double card rather than a *third card*, i.e. the third card of a quartet of which two cards have already been captured, when there is the risk of a series of scopa.

For example, the eldest hand, after a scopa of the dealer, has played a double 2. The dealer's teammate did a scopa. The third hand did not have the 2 to replay to its teammate, hence it played an ace, but the dealer did another scopa. The eldest hand does not have the ace of the teammate but it has the third 2 and a couple of 5. It will play 5 instead of 2 hoping to find the correspondence with its teammate, because in case the fourth 2 was in the hand of the opponent, then the series of scope might continue till the end.

CS16 The eldest hand, having to choose between a low-rank-double card and higher one, it must always play the lower one, provided that the cards on the table allow it.

CS17 When the hand team has done no spariglio, it is intended to bring its cards to the deck team, because the latter plays after it. To exit from the grip that nails them and try to score some points or tie them, the third hand has to play single cards.

In fact, if the dealer does not capture the card, considering it double, the eldest hand will be able to capture a card that otherwise it would have never captured. Moreover, if the fourth card is held by the dealer's teammate, the deck team, that will hesitate to play that card believing to bring it to the third hand, holds a franca card, but it will know it only at the end of the round.

CS18 When one captures a couple of cards and one of the players plays immediately a card of the same rank, it means that the latter holds also the fourth card.

For example, let the table be $Q\heartsuit$ $J\spadesuit$ $3\diamondsuit$. One of the players capture the 3, another one plays the third 3; this means that it holds also the fourth 3. If it played the third 3 and it did not hold the fourth one, it would make a serious mistake, because, besides the fact that it would mislead its teammate, it would allow the opponents who hold it to do the mulinello on that card.

CS19 The play of a face card on a low-rank card is not an indication for the teammate of having also the copy of that face card. It could have played it to avoid a scopa.

4.2.4 The Play of Sevens

In Scopone, 7s are the most important cards since they count for both the points of primiera and settebello. Therefore, the game is mostly played around these cards, hence they deserve special rules.

CS20 The dealer's teammate must always capture the 7 played either by the dealer or by the opponents.

Capturing the one played by the dealer, the third 7 will be captured by the dealer, because it holds the fourth one since it played it first. It has to capture the 7 played by the eldest hand because, if the other one is held by the dealer, then they will capture all the 7s; otherwise, if the other one is held by the third hand, the dealer's teammate has captured the one reserved to it. It has to capture the 7 played by the third hand because it does not have to allow the third hand to capture its own 7.

CS21 When the dealer's teammate has the chance to capture a 7 on the table or do a spariglio involving it, e.g. $7 + 1 = 8$, it must always do the spariglio if it holds only one 7.

In fact, the eldest hand does not hold any 7 because it did not capture it, hence the other two 7s are held by the dealer and the third hand. In case the two 7s are held by the third hand, the latter, at same point, will be forced to play one of them and the dealer's teammate will capture it, whereas the last one will be captured by the dealer with the tallone. If the two 7s are divided between the dealer and the third hand, then the former, at its second-last turn, will play the decoupled 7 because of CS4; next, the dealer's teammate will capture it and the last 7 will be captured by the dealer with the tallone.

CS22 If the dealer's teammate does not hold any 7 and has the chance to capture a 7 on the table with a spariglio, e.g. $7+1 = 8$, it must always do it even if it decouples three cards.

Here there are four situations:

- The other three 7s are held by one player. If they are held by the eldest or the third hand, then the hand team will capture two of them but the last one will be captured by the dealer with the tallone. If the three 7s are held by the dealer, then the deck team will easily capture all the four 7s.
- The other three players hold one 7 each. The dealer, at its second-last turn, will play the decoupled 7, because of CS4, that will be captured by the eldest hand; next, the dealer will capture the last one with the tallone. The dealer could also play a card that allows the recoupling of the 7s, e.g. by playing an even 6 on an odd 1, in this case the deck team will capture three 7s.
- Two 7s are held by the dealer and one by an opponent. The dealer will be forced to play one of the two 7s that will be captured by the opponents; by the way, it will capture the last one with the tallone.

- Two 7s are held by one of the opponents and one by the dealer. The dealer will capture the 7 played by the opponent and then it will capture the last one with the tallone. In this case, the deck team will capture all the four 7s.

The spariglio done by the dealer's teammate worths at least two 7s for its team.

CS23 The dealer's teammate, that holds two 7s, must capture the 7 on the table and must not capture it with a spariglio, e.g. $7 + 1 = 8$.

By doing the spariglio, it favors the game of the opponents, because it remains with two decoupled 7s.

CS24 The dealer's teammate has to avoid to capture a card, that summed to a double card it holds, may bring to the spariglio of 7s.

For example, let us assume that the table is 3 5 10 and the dealer's teammate holds 3, 5, and two 4. It will prefer to capture the 5 instead of the 3. In this way the 3 will be captured by the opponents and it will safely play the 4, controlling the play of the 3s.

CS25 When the dealer's teammate does a spariglio with 7, e.g. $4+3 = 7$, being able to avoid it, it is clear that it holds another 7.

CS26 Each player, that holds two 7s or the settebello, must capture the 7 played by the other players.

CS27 When no 7 has been played yet, who does not hold any 7 must not play cards that may bring to the spariglio of 7s.

If the player at its right holds two 7s, then it will capture one 7 with the spariglio and two 7s with a capturing move. Whereas, it would let them be captured by the teammate of whom has caused the spariglio. Moreover, who has caused the spariglio has misled its teammate, who will believe that it holds a 7.

CS28 The dealer, that holds two 7s, must play them as late as possible. However, if it has the chance to capture one of them with a spariglio, it must always do it.

It must play them as late as possible because it will capture the other two 7s, if they are held by either the eldest or the third hand. It must always capture the 7 with a spariglio because:

- If the other two 7s are held by one player, then it will capture one with its 7 and the last one with the tallone.
- If the opponents hold one 7 each, then it will give its 7 to the eldest hand at the second-last turn, but it will capture the last one with the tallone. In this case, it cannot capture more than two 7s.
- If the eldest hand holds no 7, its 7 will be captured by the dealer's teammate and the last one with the tallone.

CS29 A player, that holds three 7s, must never capture 7 with a spariglio, but it must let the opponents do the spariglio with the fourth 7, especially when it does not hold any 6.

If the three 7s are held by the dealer or its teammate and the other 7 has been decoupled ($1 + 6 = 7$, $2 + 5 = 7$, $3 + 4 = 7$), then they will capture two with a normal move and the other with the tallone or with a spariglio. If the three 7s are held by the eldest or the third hand, then they will capture, for sure, two of them with a normal move and the other one may be captured with a spariglio.

4.2.5 The Algorithm

The rules of Chitarrella and Saracino are not easily translatable to a computer program, because it often happens that several rules can be applied in a given situation; in this case the choice of which one to apply is left to the user experience. In a rule-based AI, we would like to have that only one rule for each situation can be applied, but this would require distinguishing from many situations and it is often infeasible. Alternatively, it is possible to assign a priority to each rule and choose the one with the highest priority. Unfortunately, Chitarrella and Saracino did not assign priorities to their rule explicitly, hence we tried to assign the priorities that better approximate the advices given by the two authors.

The Algorithm 6 shows the high-level pseudo-code of the CS strategy. A rule can be applied in two ways: (i) by immediately returning the move chosen by the rule; (ii) by removing the moves that do not follow the rule, in this way the remaining moves can be processed by the following rules. The rules involving the tallone have the maximum priority because are necessary for capturing several cards at the end of a round. Then follow the rules about the play of sevens, that are the most important card. Next, the program takes into consideration the moves that bring to the mulinello, i.e. it considers only the moves that do not leave on the table a combination of cards that allow the opponents to capture some cards. This is done by verifying that all the cards combinations sum up to a card's rank of which the quartet has already been captured or the last one is held by the teammate. If such moves exist, then the AI will play the best one of them, i.e. the move chosen by using the same policy of the Greedy strategy (BestMove function in Algorithm 5). The next step of the algorithm looks at moves that do not allow the opponents to do the mulinello. The program considers only the moves that do not bring to a state in which the opponent can do a move that bring to the mulinello. If such moves exist, then the next steps only consider them as possible moves. The remained moves are checked for scopa, i.e the AI considers the moves that do not allow the opponent to do a scopa. This is done in the same way of the Greedy strategy, by considering the set of moves that do not leave on the table a combination of cards that can be captured by a card that has unknown position. If such moves exist, then the program only considers those moves in the following steps. At this point, the AI prioritizes the capturing moves over the others. If such moves exist, the program selects the best move accordingly to the fundamental rule of Scopone (CS1), i.e. the move that minimizes or maximizes the number of decoupled cards after the move, depending on the player about to act. The program can easily know the decoupled cards by counting the cards previously played, hence it does not use any of the methods proposed in Section 4.2.1. The selected move is then validated by three rules; in the case it is not compatible with one of them, it is removed from the available moves and the program returns to the point in which it has to prioritize the capturing moves. Whereas, if it does not exist any capturing move, the AI try to apply some other rules and finally it returns the best remaining move.

Some rules, the exploit of the mulinello, and the scopa prevention need somehow the guessing of the cards the other players hold. For this reason, we designed a *card guessing system* that keeps track of the cards a player might hold, looking at the moves it did during the round. Each time a player do a non-capturing move, it is likely to hold also the copy of that card, because of rule CS10, therefore the program adds it to the list of that player guessed cards. In this way the other part of the program can benefit of this important information. When the player plays also the other card of the same rank, the first one is removed from the list of that player guessed cards, because we cannot assume that it holds also the third one. Also rules CS18, CS19, and CS25 are used for this purpose.

Algorithm 6 Chitarrella-Saracino strategy pseudo-code

```
function CSStrategy($state$)
    $moves \leftarrow state.legalMoves$
    if Rule CS5 then
        return move $\in moves$ chosen by this rule
    end if
    if Rule CS4 then
        $moves \leftarrow$ filtered $moves$ according to this rule
    end if
    if Rule CS27 then
        $moves \leftarrow$ filtered $moves$ according to this rule
    end if
    if Rule CS29 then
        $moves \leftarrow$ filtered $moves$ according to this rule
    end if
    if Rule CS28 then
        if it must capture then
            return move $\in moves$ chosen by this rule
        else
            $moves \leftarrow$ filtered $moves$ according to this rule
        end if
    end if
    if Rule CS26 then
        return move $\in moves$ chosen by this rule
    end if
    if Rule CS21 then
        return move $\in moves$ chosen by this rule
    end if
    if Rule CS22 then
        return move $\in moves$ chosen this rule
    end if
    if Rule CS20 then
        return move $\in moves$ chosen this rule
    end if
    if Rule CS23 then
        return move $\in moves$ chosen this rule
    end if
    $mulinelloMoves \leftarrow$ subset of $moves$ that bring to the mulinello
    if $mulinelloMoves \neq \emptyset$ then
        if Rule CS6 then
            return move $\in moves$ chosen this rule
        end if
        return BestMove($mulinelloMoves$)
    end if
    $nonOppMulMoves \leftarrow$ subset of $moves$ that do not bring to the
                                    mulinello of the opponents
    if $nonOppMulMoves \neq \emptyset$ then                ▷ also enforces Rule CS7
        return PreventingScopaMove($nonOppMulMoves$)
    end if
    return PreventingScopaMove($moves$)
end function

function PreventingScopaMove($moves$)
    $nonScopaMoves \leftarrow$ subset of $moves$ that do not bring to scopa
    if $nonScopaMoves \neq \emptyset$ then
        return DetermineMove($nonScopaMoves$)
    end if
    return DetermineMove($moves$)
end function

function DetermineMove($moves$)
    $capturingMoves \leftarrow$ subset of $moves$ that capture some cards
    if $capturingMoves \neq \emptyset$ then                ▷ enforces Rule CS9
        $move \leftarrow$ ChooseCapturingMove($capturingMoves$)
        if $move$ is not compatible with Rule CS24 then
            $move \leftarrow$ TryAnotherMove($moves, move$)
        end if
        if $move$ is not compatible with Rule CS8 then
            $move \leftarrow$ TryAnotherMove($moves, move$)
        end if
        if $move$ is not compatible with Rule CS12 then
            $move \leftarrow$ TryAnotherMove($moves, move$)
        end if
        return $move$
    end if
    if Rule CS11 then
        $moves \leftarrow$ filtered $moves$ according to this rule
    end if
    if Rule CS13 then
        $moves$ ← filtered $moves$ according to this rule
    end if
    if Rule CS17 then
        $moves$ ← filtered $moves$ according to this rule
    end if
    if the player holds some non-single cards then
        if Rule CS16 then
            return move ∈ $moves$ chosen by this rule
        end if
        return move ∈ $moves$ chosen by Rules CS10, CS14, CS15
    end if
    if Rule CS2 then
        return move ∈ $moves$ chosen by this rule
    end if
    if Rule CS3 then
        $moves$ ← filtered $moves$ according to this rule
    end if
    return BESTMOVE($moves$)
end function

function CHOOSECAPTURINGMOVE($moves$)          ▷ enforces Rule CS1
    if player ∈ hand team then
        return move ∈ $moves$ that maximizes the number of
                decoupled cards
    end if
    return move ∈ $moves$ that minimizes the number of
                decoupled cards
end function

function TRYANOTHERMOVE($moves, move$)
    $otherMoves$ ← $moves \setminus \{move\}$
    if $otherMoves \neq \emptyset$ then
        $move$ ← DETERMINEMOVE($otherMoves$)
    end if
    return $move$
end function
```

4.3 Cicuti-Guardamagna Strategy

The book of Cicuti and Guardamagna [12] refines the techniques proposed by Saracino, especially the rules about the play of sevens. The most important of these rules have been summarized in [5]. With the *Cicuti-Guardamagna* (CG) strategy we aim to encode these additional rules to allow the AI for better ways of playing.

4.3.1 The Rules

CG1 If the dealer's teammate holds two 7s, then it must always capture the 7, also with a spariglio.

CG2 The player, who holds three 7s with the settebello, must immediately play one of them.

CG3 The player, who holds two 7s without the settebello, must play the 7 as late as possible.

CG4 The opponent of the dealer, who holds two 7s with the settebello, must play the 7 when the state of the game allows its teammate to redouble the combination of 7.

In this way, if the opponent at its right does not capture the 7, the teammate, that does not hold any 7, can favor the capture of the settebello by *redoubling* the combination of 7, i.e. it can play a card that sums up to 7 with another card on the table.

CG5 The player, who holds the last two 7s with the settebello, if it belongs to the hand team, it must play the settebello; if it belongs to the deck team, it must play the other 7.

In fact, it often happens that the second-last 7 is in danger because of the play of a low-rank card that allows for the spariglio with a face card. If this happens, the settebello is likely to be captured with the tallone by the dealer. Therefore, the loss of the second-last 7 is not a big problem for the deck team, but is a serious issue for the hand team, because they risk of losing also the settebello. Hence, for the hand team, it is convenient to play the settebello; in this way even if the teammate cannot capture the low-rank card played by the opponent, it can at least capture the settebello with a face card.

CG6 The dealer's teammate, the third hand, and the dealer, who hold three 7s without the settebello, must play the 7 on a low-rank card that allows for the spariglio of 7.

In fact, if one does the spariglio, then the low-rank-decoupled card can favor the capture of the second 7 with another spariglio and therefore allow the capture of the settebello.

CG7 When there are still three 7s in game and the dealer holds the settebello, the dealer's teammate, that does not hold any 7, must remains with an even card that sums up to 7 with another decoupled card.

CG8 When the third hand plays 7 and the eldest hand has the possibility to redouble the combination of 7, the dealer must capture the 7 of the third hand.

CG9 When the third hand plays 7, the dealer, who does not hold any 7, must not redouble the combination of 7, but it must try to prevent the eldest hand from redoubling the combination of 7.

CG10 When the dealer's teammate and the third hand hold the last two 7s, the dealer can favor the capture of a 7 by its teammate with a spariglio, by remaining with an even card that sums up to 7 with another decoupled card.

CG11 When the decoupled cards are 3, 4, 7; or 2, 5, 7; or 1, 6, 7; and the third hand holds the last 7, the dealer must remain respectively, in the first case with a 3 or a 4; in the second case with a 2 or a 5; and in the third case with an ace or a 6. This rules also holds for the dealer's teammate, when the last 7 is held by the eldest hand.

CG12 When there are still three 7s in game, the eldest hand, who does not hold any 7, must create a block for the transit of 7, e.g. with a J if the ace is decoupled, with a Q if the 2 is decoupled, with a K if the 3 is decoupled. When it does not have this possibility, it must remain with an even card that sums up to 7 with another decoupled card.

CG13 The eldest hand must not capture the 7 played by the dealer unless it gains the primiera point.

4.3.2 The Algorithm

With the rules of Cicuti and Guardamagna there is still the problem discussed in Section 4.2.5 about the priority of each rule. Therefore, once more, we tried to assign the priorities that better fit the advices of the authors.

The high-level pseudo-code of the CG strategy is shown in Algorithm 7. Like in the CS strategy, each rule can be applied in two ways: (i) returning the move chosen by the rule; (ii) removing the moves that do not follow the rule. Since the rules of the CG strategy are an extension to the rules of the CS strategy, the program try to apply the CG rules first; if no rule can be applied, then the AI returns the move chosen by the CS strategy.

4.4 Summary

In this chapter, we showed the rule-based Artificial Intelligence (AI) we developed for Scopone. We designed three AI that aim to represent the playing strength of a beginner, an expert, and another expert with refined techniques. The Greedy strategy represents the beginner and basically it captures the most important card currently available on the table; otherwise it plays on the table the less important card it holds. The Chitarrella-Saracino (CS) strategy represents an expert player and it encodes the rules taken from the books of Chitarrella and Saracino. The rules are related to four important aspects of the game: the spariglio, the mulinello, the play of double and triple cards, and the play of sevens. The Cicuti-Guardamagna (CG) strategy is an extension of the CS AI, it encodes the rules proposed by Cicuti and Guardamagna in their book. Basically, they introduced more refined techniques for the play of sevens.

Algorithm 7 Cicuti-Guardamagna strategy pseudo-code

```
function CGStrategy($state$)
    $moves \leftarrow state.legalMoves$
    if Rule CG9 then
        $moves$ ← filtered $moves$ according to this rule
    end if
    if Rule CG8 then
        return move ∈ $moves$ chosen by this rule
    end if
    if Rule CG2 then
        $move$ ← move ∈ $moves$ chosen by this rule
        if $move$ does not bring to scopa or all $moves$ bring to scopa then
            return $move$
        end if
    end if
    if Rule CG6 then
        $move$ ← move ∈ $moves$ chosen by this rule
        if $move$ does not bring to scopa or all $moves$ bring to scopa then
            return $move$
        end if
    end if
    if Rule CG1 then
        return move ∈ $moves$ chosen by this rule
    end if
    if Rule CG5 then
        $move$ ← move ∈ $moves$ chosen by this rule
        if $move$ does not bring to scopa or all $moves$ bring to scopa then
            return $move$
        end if
    end if
    if Rule CG4 then
        $move$ ← move ∈ $moves$ chosen by this rule
        if $move$ does not bring to scopa or all $moves$ bring to scopa then
            return $move$
        end if
    end if
    if Rule CG3 then
        $moves$ ← filtered $moves$ according to this rule
    end if
    **if** Rule CG7 **then**
        *moves* $\leftarrow$ filtered *moves* according to this rule
    **end if**
    **if** Rule CG10 **then**
        *moves* $\leftarrow$ filtered *moves* according to this rule
    **end if**
    **if** Rule CG11 **then**
        *moves* $\leftarrow$ filtered *moves* according to this rule
    **end if**
    **if** Rule CG12 **then**
        *moves* $\leftarrow$ filtered *moves* according to this rule
    **end if**
    **if** Rule CG13 **then**
        *moves* $\leftarrow$ filtered *moves* according to this rule
    **end if**
    **return** CSStrategy(*state*)
**end function**
```

Chapter 5

Monte Carlo Tree Search algorithms for Scopone

In this chapter we show the *Monte Carlo Tree Search* (MCTS) algorithms we designed for Scopone. Besides the basic MCTS algorithm, we investigated some changes that can be made to improve the playing strength of the algorithm. In particular, we used different reward methods for the back-propagation step, some variants in the simulation strategies, and few ways to reduce the number of available moves in each state. We also used a different determinization procedure in order to reduce the number of states within an information set in the *Information Set Monte Carlo Tree Search* (ISMCTS) algorithm.

5.1 Monte Carlo Tree Search Basic Algorithm

In the basic version of the MCTS algorithm, each node represents a single state of the game and memorizes five pieces of information: the incoming move, the visits count, the total rewards, the parent node, and the child nodes. Each state memorizes the list of cards on the table, in the players hands, and the cards captured by each team. It also memorizes the cards that caused a scopa, the index of the last player to move, and the last player who did a capturing move. Each state also records all the moves previously done by each player along with the state of the table before each move, these informations are needed especially by the rule-based artificial intelligence. Every move is composed by the played card and the list of captured cards, that can be also empty. Each card is represented by its rank and suit. Every state can return the list of available moves and, in case of a terminal state, the score of each team. It also offers a method to apply a move and jump in another state of the game.

The selection step of MCTS is done with the *Upper Confidence Bounds for Trees* (UCT) formula of Algorithm 3 with a constant $c = 1$. In the expansion step, the move to expand the node is chosen randomly from the moves available in the corresponding state. For the simulation step the game is played randomly until a terminal state. The backpropagated reward is a vector containing the score of each team. During the selection step, the UCT formula considers only the team's score of the player acting in that node. At the end of the algorithm, the incoming move of the most visited root node is selected.

The MCTS algorithm described above could not be used with Scopone because it is a game with imperfect information. Therefore, we decided to use it as *cheating player*, i.e. a player that has access to the hidden information of the game, in this case the cards in the players' hand. Using a cheating player is not a valid approach to *Artificial Intelligence* (AI) for games with imperfect information, but it provides a useful benchmark for other algorithms since it is an approach which is expected to work better than approaches that do not cheat.

The game tree resulting from an exhaustive search, starting from an initial state of the game, has $(9!)^4 = 1.73 \times 10^{22}$ leaf nodes. It can be calculated by considering that, at the beginning of a round, each player holds 9 cards and, at the end of the fourth turn, we can be in any of the 9^4 possible states. At the eighth turn, we can be in any of the 8^4 possible states under any of the previous 9^4 states, for a total of $(9 \cdot 8)^4$. Continuing in this way until each player holds no card, we get the result. The previous result can be used to calculate the total number of nodes in the full-expanded tree. We know that, at the end of every four turns, we have $(\prod_{t=x}^{9} t)^4$ leaf nodes, where x is the number of cards that each player holds during its last turn. Moreover, in these four turns we have a total of $(x + x^2 + x^3 + x^4)$ nodes for each leaf of the previous four turns. Therefore, the total number of nodes is

$$1 + \sum_{x=1}^{9} \Big(\prod_{t=x+1}^{9} t \Big)^4 (x + x^2 + x^3 + x^4) = 1.03 \times 10^{23}$$

(the +1 is for the root node), hence the leaf nodes are the 16.8% of the total nodes. Note that when $x = 9$, then t should vary from 10 to 9 and this produces the empty product, hence 1 is returned. The depth of the tree is 36, i.e. the total number of cards in the players' hand, therefore the Effective Branching Factor (EBF) [23] is 4.33, calculated enforcing the constraints:

$$treeNodes = \frac{EBF^{treeDepth+1} - 1}{EBF - 1}, \; EBF > 0.$$

Table 5.1: Complexity values of the Monte Carlo Tree Search algorithm applied to Scopone.

State space	**Tree nodes**	**Leaf nodes**	**Tree depth**	**EBF**
2.02×10^{47}	1.03×10^{23}	1.73×10^{22}	36	4.33

Note that these values are calculated by considering that a move consists in playing a card from a player's hand. This is not completely true, since sometime it is possible to do different capturing moves by playing the same card and this results in a higher branching factor. Therefore, these values have to be considered as a lower bound of the actual values. An idea of the dimension of the state space can be calculated starting from the number of possible initial states and multiply it by the total number of nodes in the full-expanded tree. For sure, every nodes in the tree represents a different state, but we can have that the same state can be represented in different game tree, hence the value that we will obtain is only an upper bound of the actual dimension of the state space. Although the total number of deck shuffles is $40! = 8.16 \times 10^{47}$, many of them result in the same initial state because the cards' order in the players' hand and on the table does not matter. Therefore, the number of possible initial states can be calculate as

$$\binom{40}{9}\binom{31}{9}\binom{22}{9}\binom{13}{9}\binom{4}{4} = 1.96 \times 10^{24},$$

hence the dimension of the state space is $1.96 \times 10^{24} \cdot 1.03 \times 10^{23} = 2.02 \times 10^{47}$. Table 5.1 summarizes the complexity of the basic MCTS algorithm applied to Scopone. According to [39], Scopone with MCTS has a game tree complexity and EBF comparable with *Connect Four*, and a state space similar to *Chess*.

5.2 Information Set Monte Carlo Tree Search Basic Algorithm

In the basic version of the *Information Set Monte Carlo Tree Search* (ISMCTS) algorithm, each node represents an information set from the root player's point of view. The only additional information required to each node, with respect to MCTS, is the availability count.

The ISMCTS algorithm applies the same steps of MCTS, but it uses the ISUCT formula of Algorithm 4 with a constant $c = 1$. It also has to create a determinization of the root state at each iteration, this is done by randomize the cards held by the other players.

Table 5.2: Complexity values of the Information Set Monte Carlo Tree Search algorithm applied to Scopone.

ISS	**Tree nodes**	**Leaf nodes**	**Tree depth**	**EBF**
2.23×10^{58}	1.14×10^{34}	3.95×10^{33}	36	8.8

In contrast with the MCTS algorithm, ISMCTS is suited for games with imperfect information, therefore it can be used with Scopone without any form of cheating. However, this increases the branching factor of the tree search, resulting in a higher complexity of the algorithm.

This time, the leaf nodes in the full-expanded game tree are $9!\,27! = 3.95 \times 10^{33}$. In fact, at the first branch, the root player knows its cards, therefore there are 9 possible branches. At the second branch, however, the next player could play any of the 27 cards that have unknown position from the point of view of the root player (remember that the outgoing arcs from an opponent's node have to represent the union of all moves available in every state within that information set). Analogously, in the third and fourth branch the available moves are respectively 26 and 25. At the fifth branch, it is again the turn of the root player and it can do 8 moves. The next branches are of 24, 23, 22 moves, and so on until the end. To calculate the total number of nodes in the tree, we can proceed as before by summing the total branches at each turn. Therefore, we obtain a game tree of

$$1 + \sum_{i=1}^{9} \sum_{j=1}^{4} \prod_{x=i}^{9} x \prod_{y=3(i-1)+j}^{27} y = 1.14 \times 10^{34}$$

nodes. This time the leaf nodes are the 34.65% of the total nodes. The depth of the tree is still 36, but the EBF is increased to 8.8. Note that these values are a lower bound of the actual ones, for the reason explained in the previous section. Moreover, since the nodes correspond to information sets, additional capturing moves may be available, because they can be done in some particular states within that information set. Of course, the state space of the game is not going to change, but we can have an idea of the dimension of the *Information-Set Space* (ISS) by multiplying the number of initial states with the total number of nodes in the ISMCTS tree, thus we obtain $1.96 \times 10^{24} \cdot 1.14 \times 10^{34} = 2.23 \times 10^{58}$ information sets. Table 5.2 summarizes the complexity of the basic ISMCTS algorithm applied to Scopone. According to [39], Scopone with ISMCTS has a game tree complexity comparable with *Congkak*, an EBF like *Domineering*(8×8), and a state space similar to *Hex*(11×11).

5.3 Reward Methods

In the MCTS algorithm the role of the reward is very important, since it is used in the UCT formula to orient the search in the most promising areas of the tree. In the basic algorithm we used the score of each team as reward (we called it *Normal Scores* (NS)), but, in order to chose the best possibility, we create three other types of reward:

- *Scores Difference* (SD): for each team, we used the difference between the score of that team and the opponents. For example, if the final scores of a game were $(4, 1)$, the reward for the first team would be 3 and -3 for the other one.
- *Win or Loss* (WL): it is returned a reward of 1 to the winning team, -1 for the other one, and 0 in case of a tie.
- *Positive Win or Loss* (PWL): The winning team gets a reward of 1, the losing team gets 0, and in case of a tie both teams get 0.5.

We presume that NS and SD, rewards that do not only distinguish from win, loss or tie, will behave in a more human-like way, because they will try to achieve the maximum score even if they are going to lose the game. PWL is similar to WL, but we want to exploit the fact that, for rewards between $[0, 1]$, we know the optimal UCT constant.

5.4 Simulation strategies

The simulation strategy is another important aspect of MCTS. It is responsible to play a game from a given state until the end, in order to get an approximation of the result the game will always have if played from that state. In the basic version of the MCTS algorithm we used a *Random Simulation* (RS) strategy, this is very efficient, but it does not always give good results. Previous studies [15] suggest that using heuristics in the simulation step can increase the performance of the algorithm. Therefore, we designed other three simulation strategies:

- *Greedy Simulation* (GS): It simply uses the Greedy strategy to play the game until the end.
- *Epsilon-Greedy Simulation* (EGS): At each turn, it plays at random with probability ϵ, otherwise it plays the move chosen by the Greedy strategy.

- *Card Random Simulation* (CRS): It plays a card at random, but the Greedy strategy decides which capturing move to do in case there are more than one.

We decided to use EGS because, relying on a non-random simulation strategy, can be too restrictive, the results from a given state will be always equal. Therefore, we think that adding some random factor can be beneficial, by the way ϵ will be determined experimentally.

5.5 Reducing the number of player moves

The basic MCTS algorithm uses all the legal moves of a state in order to expand the corresponding node (we called this approach *All Moves Handler* (AMH)). This exploits all the possibilities of the game, but makes the tree search very expensive. In order to decrease the dimension of the tree search, it is possible to apply some pruning strategies to filter out some moves. Therefore, we created three moves handlers:

- *One Move Handler* (OMH): It generates one move for each card in the player's hand. If it is possible to do different capturing moves with a card, it will choose the most important one, exactly in the same way the Greedy strategy would do.
- *One Card Handler* (OCH): It generates one move for each card in the player's hand, moreover the move memorizes only the played card. The eventual captured cards are determined by the Greedy strategy only when the move has to be done.
- *Greedy Opponents Handler* (GOH): When it is not the turn of the root player, it is generated only the move chosen by the Greedy strategy.

At a first look, OMH and OCH would seem to have the same behavior. In fact, it is true for MCTS, they both ensure that the game tree has a maximum complexity equal to the values illustrated in Table 5.1. However, for ISMCTS, only OCH can ensure it for the values illustrated in Table 5.2. This is because using OMH does not guarantee that the chosen captured cards will be available in all the states within the information set, therefore other capturing moves might be added. GOH has been introduced to consistently decrease the complexity of the game tree. In fact, with MCTS, it reduces the leaf nodes to $9! = 3.63 \times 10^5$, the total nodes to

$$1 + 4\sum_{i=1}^{9}\prod_{x=i}^{9} x = 3.95 \times 10^6,$$

and the EBF to 1.48. For ISMCTS, the reduction cannot be calculated. We have to remember that each node represents an information set, therefore in each opponent's node there are available moves as many states are within that information set, because we have a Greedy move for each state. Therefore, we can have two extreme situations: (i) in every state within the information set, the Greedy strategy chooses the same move, in this case we have only one outgoing arc from the node; (ii) the Greedy strategies chooses different move in each state, in the worst situation, we have outgoing arcs as the number of unknown cards in that information set. For this reason, in some situation it might reduce the complexity of the tree search, in others it might not.

5.6 Determinization with the Cards Guessing System

In order to reduce the complexity of the ISMCTS tree, we can also try to decrease the number of states within an information set. In this way the number of available moves at each node could be reduced, since we can exclude some moves that were only available in the eliminated states. The idea is to remove the states that are less likely to happen, in this way, besides the possible moves' reduction, the search will be focused only in the most likely states. For these reasons, we integrated, in the determinization step of ISMCTS, the cards guessing system that we designed for the Chitarrella-Saracino strategy. Therefore, at each iteration of the ISMCTS algorithm it is generated a determinization in which each player holds the cards guessed by the cards guessing system.

5.7 Summary

In this chapter we showed how the basic MCTS and ISMCTS algorithm have been implemented, and the complexity of these search methods on Scopone. Next, we illustrated some methods that can be used to reduce the complexity of the algorithms and increase the playing strength of AI. We proposed three different reward methods: Scores Difference, Win or Loss, and Positive Win or Loss. Three simulation strategies: Greedy Simulation, Epsilon-Greedy Simulation, and Card Random Simulation. Three moves handlers: One Move Handler, One Card Handler, and Greedy Opponents Handler. Finally, we integrated the cards guessing systems in the determinization step of ISMCTS.

Chapter 6

Experiments

In this chapter, we discuss the experiments we performed to evaluate the playing strength of the various strategies we developed. For this purpose, we first determined the best *Artificial Intelligence* AI for each of the three categories: rule-based, *Monte Carlo Tree Search* (MCTS), and *Information Set Monte Carlo Tree Search* (ISMCTS). Then, we did a tournament to rank them.

6.1 Experimental Setup

In order to provide consistency between experiments, and reduce the variance of our results, we randomly generated a sample of 1,000 initial game states and we used it in all our experiments. Moreover, for MCTS and ISMCTS we played each game 10 times to further reduce the variance, due to the stochastic nature of the algorithms. For each of the experiments we assigned an AI to the hand team and another one to the deck team, then we compared the winning rate for each team and the percentage of ties. We also introduced a random playing strategy in order to have a performance benchmark.

6.2 Random Playing

In this first experiment, we used the random strategy for both the hand team and deck team. With this experiment we want to find out how much the game is biased towards a specific team. Table 6.1 shows the winning rates of the hand team and the deck team, and the percentage of ties, when both the teams play randomly. The results in Table 6.1 confirm that the deck team has a slight advantage over the hand team. In fact, the deck team

Table 6.1: Winning rates of the hand team and the deck team, and the percentage of ties, when both the teams play randomly.

Hand Team	Deck Team	Ties
41.69%	45.71%	12.6%

wins the 45.71% of the games, against the 41.69% of the opponents. This was mentioned also in the strategy books we previously presented, but it has never been estimated quantitatively.

6.3 Rule-Based Artificial Intelligence

In the second experiment, we compared the different rule-based AI. Initially, we determined how the most important aspects of the game strategy influence the winning rate. For this purpose, we designed two special versions of the *Chitarrella-Saracino* (CS) strategy. A *CS Without Scopa Prevention* (CSWSP) that encodes all the rules of the CS strategy but it does not include the function PreventingScopaMove of Algorithm 6, that gives priority to the moves that do not leave on the table a combination of cards allowing the opponents to do a scopa. A *CS Without Play of Sevens* (CSWP7): that encodes all the rules of the CS strategy except the ones related to the play of sevens (Section 4.2.4). Then, we performed a tournament between the basic CS, CSWSP, and CSWP7 by playing 1,000 games in each match. Table 6.2 shows the percentage of wins, losses, and ties in each match of the tournament. While, Table 6.3 shows the final scoreboard of the tournament, calculated by counting the wins, losses, and ties each AI did during the tournament. From the results it is clear that CSWSP is the worst strategy losing the 54.65% of the games and winning only the 34.63% of the times. This is not surprising, since CSWSP does not try to avoid a move that will bring to an opponent's scopa, and this is a crucial aspect of Scopone, since each scopa worth one point and there is no limit on the number of scopa one can do. The results show also that CS and CSWP7 are almost equivalent, they both win about the 48% of the matches. This is unexpected, since CSWP7 has no strategy for the play of sevens, but sevens are the most important cards of the game and it is quite natural to think that they deserve some special strategies. However, this result may be due to the fact that the importance of the sevens is already taken into account by the BestMove function of Algorithm 5. In fact, either this strategy selects the capturing move involving the most important cards currently available, or it plays the least important one on the table. Therefore, additional rules

Table 6.2: Tournament between the different versions of the Chitarrella-Saracino strategy. In each section, the artificial intelligence used by the hand team are listed at the left, while the ones used by the deck team are listed at the bottom. The first section shows the percentage of wins of the hand team, the second one shows the percentage of losses of the hand team, and the third one shows the percentage of ties.

Winning rates of the hand team			
CSWSP	38.1%	24.1%	25.4%
CSWP7	52%	38.2%	38.6%
CS	54%	36.8%	38.7%
Hand Team/Deck Team	**CSWSP**	**CSWP7**	**CS**

Losing rates of the hand team			
CSWSP	52.5%	66.1%	65.2%
CSWP7	35.1%	47.9%	47.3%
CS	32.6%	47.5%	46.6%
Hand Team/Deck Team	**CSWSP**	**CSWP7**	**CS**

Tying rates			
CSWSP	9.4%	9.8%	9.4%
CSWP7	12.9%	13.9%	14.1%
CS	13.4%	15.7%	14.7%
Hand Team/Deck Team	**CSWSP**	**CSWP7**	**CS**

Table 6.3: Scoreboard of the Chitarrella-Saracino strategies tournament. It shows the percentage of wins, losses, and ties each artificial intelligence did during the tournament.

AI	**Wins**	**Losses**	**Ties**
CSWP7	48.38%	38.23%	13.38%
CS	48.1%	38.23%	13.67%
CSWSP	34.63%	54.65%	10.72%

for sevens might not be needed. We also notice that, if we consider the matches between the same AI (the diagonal values of the tables), we find out that the winning rate of the hand team is decreased from 41.69% to 38%, comparing the result obtained with the random strategy. This may be explained by considering that the CS strategy has more knowledge of the game and can better exploit the advantage of being part of the deck team. Moreover, one can notice that CSWSP, when it plays against itself as deck team, has a winning rate that is greater than the one of the other AI in the same situation: 52.5% versus 47.9% and 46.6%. This suggests that having a player, as hand team who does not use a scopa prevention, increases the advantages of the deck team, because it is the first player to move and it is likely to give more scopa opportunities.

Finally, in order to select the best AI, we performed a tournament between all the rule-based AI that we developed: *Greedy*, *Chitarrella-Saracino* (CS), and *Cicuti-Guardamagna* (CG). The Greedy strategy should behave like a beginner, while CS and CG represent expert players, with CG encoding additional rules for the play of sevens. Table 6.4 shows the results of this tournament, in which 1,000 games have been played in each match, and Table 6.5 shows the final scoreboard. Unsurprisingly, the Greedy strategy is the worst of the AI, but it turns out to be stronger than we expected. In fact, it wins the 39.83% of the games and there is only a difference of about 4% of wins from the two other AI. Probably, the scopa prevention and the playing of the best move considering only the importance of the cards are sufficient to obtain a good strategy for Scopone. However, we believe that these strategies are not sufficient to give to a human player the feeling of a good way of playing. The results also show that the CS and CG strategies have almost the same playing strength, they win about the 44% of the games. Moreover, CS turns out to be slightly better than CG. This is unexpected, because we know that the CG strategy is supposed to be an improvement over the CS strategy, for the reason that it adds advanced rules for the play of sevens. Once more, we found out that special rules for the play of sevens might not be needed, since the CS strategy somehow already handles them. However, in our opinion, the CG strategy should behave more similarly to an expert human player. Moreover, one can notice that CG, when it plays against itself as deck team, has a winning rate which is grater than the CS one in the same situation: 49.1% versus 46.6%. This tells us that the additional rules are more beneficial to the deck team, that can exploit them to win a greater number of games. Another thing to notice is that Greedy versus Greedy has a tying rate which is greater than the one of other matches: 17% versus about 14%. This may be explained by the fact

Table 6.4: Tournament between the rule-based artificial intelligence. In each section, the artificial intelligence used by the hand team are listed at the left, while the ones used by the deck team are listed at the bottom. The first section shows the percentage of wins of the hand team, the second one shows the percentage of losses of the hand team, and the third one shows the percentage of ties.

Winning rates of the hand team			
Greedy	39.3%	36.5%	37.7%
CS	46.7%	38.7%	37.5%
CG	43.7%	37.9%	37.3%
Hand Team/Deck Team	**Greedy**	**CS**	**CG**

Losing rates of the hand team			
Greedy	43.4%	48.8%	48.2%
CS	40.1%	46.6%	47.6%
CG	42%	48.2%	49.1%
Hand Team/Deck Team	**Greedy**	**CS**	**CG**

Tying rates			
Greedy	17.3%	14.7%	14.1%
CS	13.2%	14.7%	14.9%
CG	14.3%	13.9%	13.6%
Hand Team/Deck Team	**Greedy**	**CS**	**CG**

Table 6.5: Scoreboard of the rule-based artificial intelligence tournament. It shows the percentage of wins, losses, and ties each artificial intelligence did during the tournament.

AI	**Wins**	**Losses**	**Ties**
CS	44.42%	41.23%	14.35%
CG	43.97%	41.97%	14.07%
Greedy	39.83%	45.02%	15.15%

that the Greedy strategy seeks to capture as much and as best as possible, and this leads to more frequent ties since the points are equally distributed. Thus, the best rule-based AI is the CS strategy, therefore it will be used in the final tournament against the MCTS methods.

6.4 Monte Carlo Tree Search

In the third set of experiments, we selected the best MCTS algorithm for Scopone. In order to do that, we compared all the different variations of MCTS that we discussed in Chapter 5. Remember that MCTS is a cheating player, i.e. it knows the cards in the other players' hand. We used it as a benchmark for the best possible strategy. In all the experiments, the MCTS algorithm played 1,000 games versus the Greedy strategy, both as hand team and deck team, and each game was played 10 times.

6.4.1 Reward Methods

The first experiment concerns the reward method, that guides the search in each node. We tested the four different rewards methods that we designed: *Normal Score* (NS), *Scores Difference* (SD), *Win or Loss* (WL), and *Positive Win or Loss* (PWL). For each one of them, MCTS played 1,000 games 10 times against the Greedy strategy. Figure 6.1 plots the winning rate as a function of the number of iterations when MCTS plays as the hand team, while Figure 6.2 shows the results when MCTS plays as the deck team. From the plots, WL appears to be the best strategy, because at the maximum iterations is the best for both the hand team and the deck team. Moreover, for the deck team, it is the best for all the iterations. However, we can see that, for the hand team, SD is the best for the iterations before 700. Another thing to notice is that, NS and SD have almost the same trend, even if SD is better. The same thing happens between PWL and WL. This is not surprising, because PWL and WL only consider win, lose or tie, whereas the rewards returned by NS and SD take into account the scores of the game. However, the negative rewards WL and SD are always better than their positive counterpart PWL and NS. For these reasons, we chose to maintain both WL and SD for the next set of experiments. The results also show that MCTS already outperforms the winning rate achieved by CS versus the Greedy strategy. In fact, the current best MCTS wins the 80.52% of the games as hand team and the 90.04% as deck team. Whereas, CS wins only the 46.7% and 48.8% of times respectively. This result was expected, since MCTS is a cheating player, who knows the cards of all the

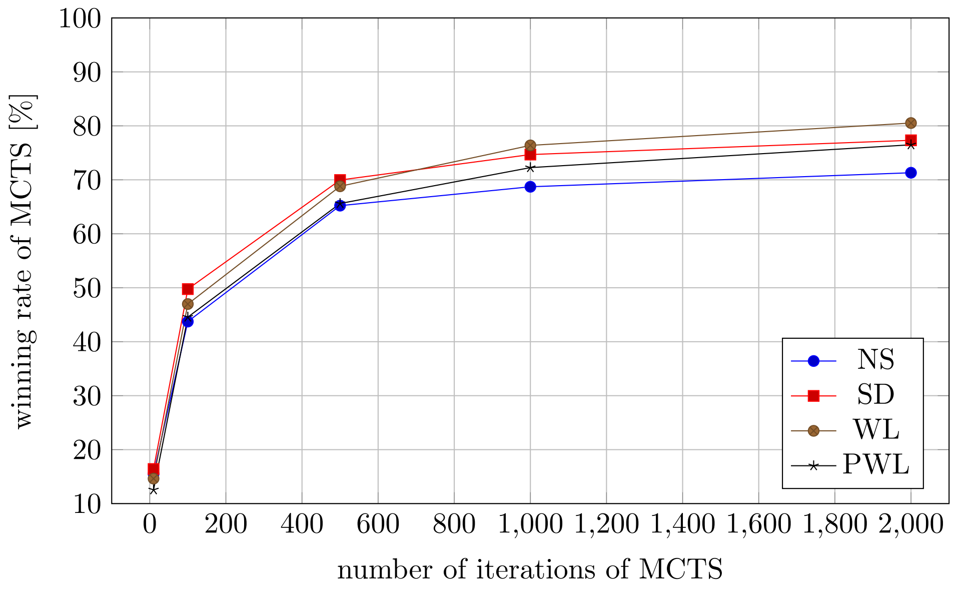


Figure 6.1: Reward methods comparison on the Monte Carlo Tree Search winning rate as a function of the number of iterations when it plays against the Greedy strategy as the hand team.

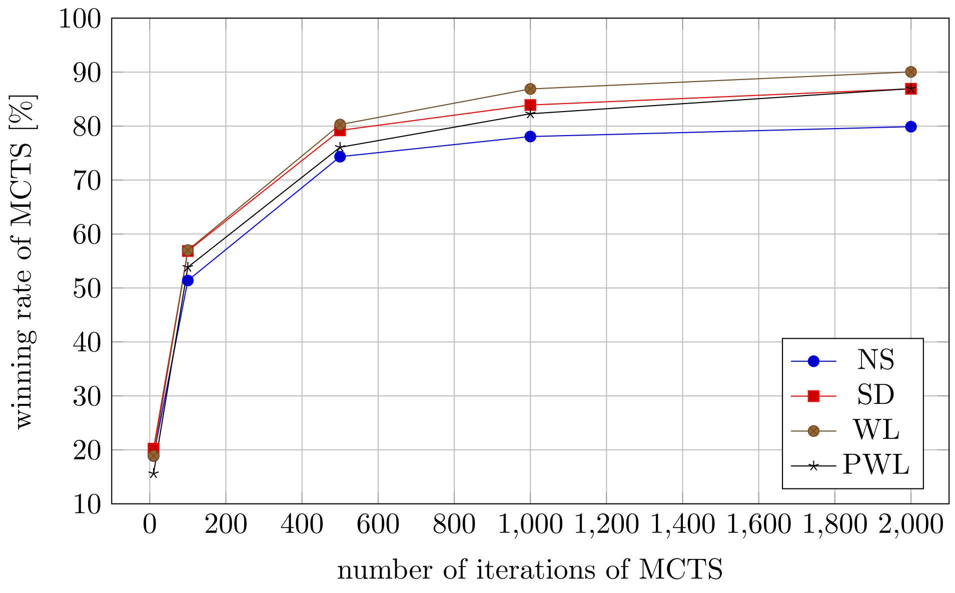


Figure 6.2: Reward methods comparison on the Monte Carlo Tree Search winning rate as a function of the number of iterations when it plays against the Greedy strategy as the deck team.

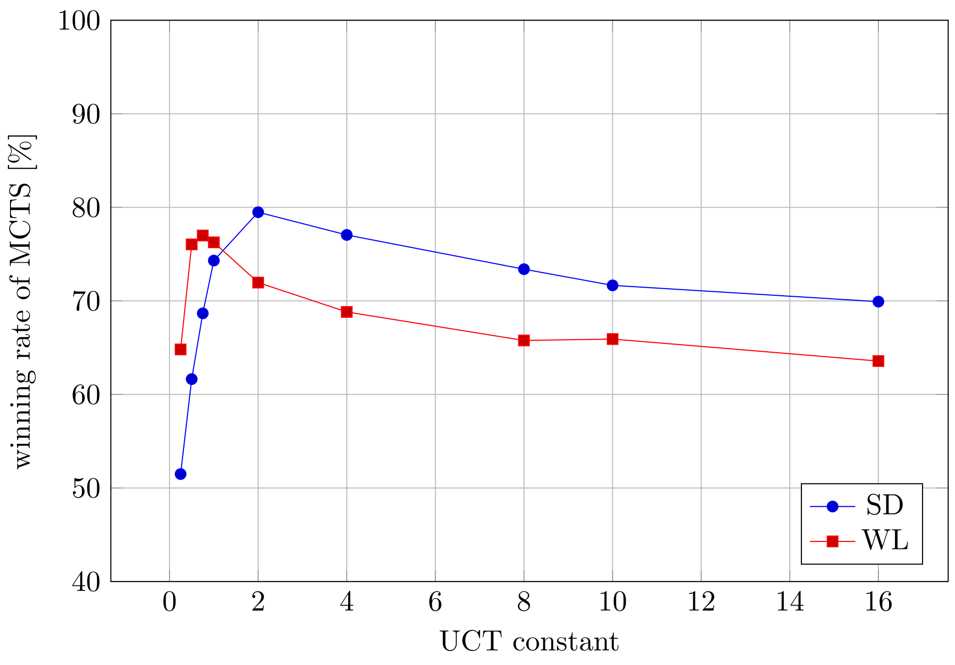


Figure 6.3: Scores Difference and Win or Loss reward methods comparison on the Monte Carlo Tree Search winning rate as a function of the Upper Confidence Bounds for Trees constant when it plays against the Greedy strategy as the hand team.

other players.

6.4.2 Upper Confidence Bounds for Trees Constant

The previous experiments showed that WL and SD are the best rewards methods. However, the *Upper Confidence Bounds for Trees* (UCT) formula uses also an exploration constant in the selection step of the algorithm. The objective of the next experiments is to determine the best UCT constant. For this purpose, we fixed the number of iteration of the MCTS algorithm to 1,000, the point in which it begins to stabilize itself (Figure 6.1 and Figure 6.2), and, in each experiments, MCTS is playing 1,000 games 10 times against the Greedy strategy. Figure 6.3 plots the winning rate as a function of the UCT constant when MCTS plays as the hand team, while Figure 6.4 shows the results when MCTS plays as the deck team. The results show that the best UCT constant for WL is 0.75, whereas for SD it is 2. Moreover, from the plots, we can see that best values of the UCT constant for WL are condensed near 0.75; whereas, for SD, they are spread around 2. This can be explained by the fact that the rewards of WL range from -1 to 1, while the rewards of SD have not a limited range, since there is no

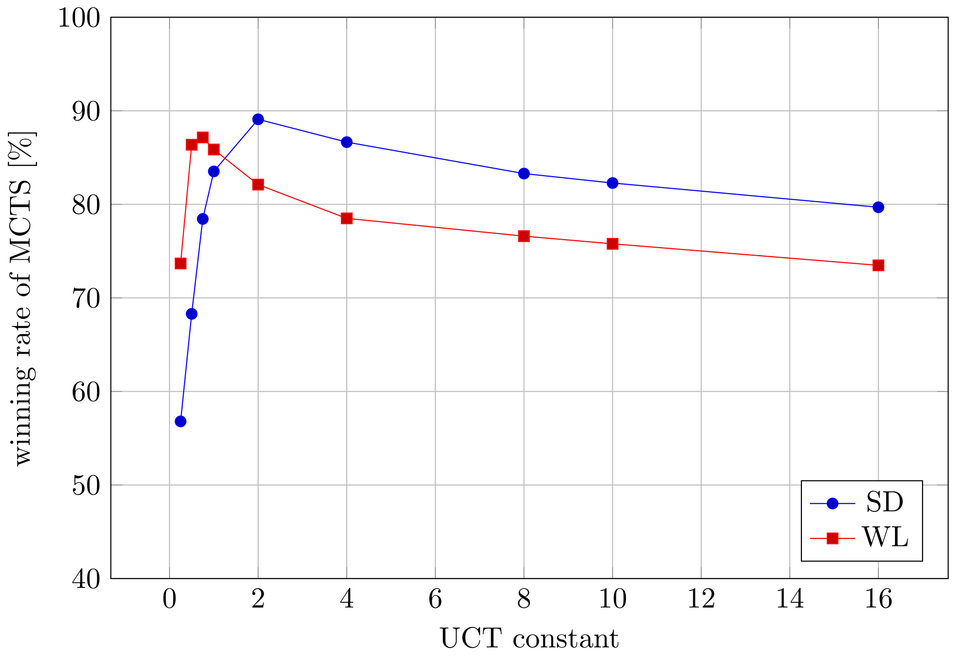


Figure 6.4: Scores Difference and Win or Loss reward methods comparison on the Monte Carlo Tree Search winning rate as a function of the Upper Confidence Bounds for Trees constant when it plays against the Greedy strategy as the deck team.

limit on the points one can do in a round. Therefore, in order to balance the exploration term of the UCT formula, the greater the rewards are, the greater the best constant has to be. SD turns out to be the best choice, because it outperforms WL of about the 2% of wins. Therefore, we fixed this setting for the next experiments. Between the hand team and the deck team, besides the fact that the latter reaches higher winning rates, there is no significant difference.

6.4.3 Simulation strategies

MCTS requires the estimation of the state's value of a leaf node. The simulation strategy is responsible to play a game from a given state until the end and obtain an approximation of the state's value. Previous studies [15] suggest that using heuristics in the simulation step can increase the performance of the algorithm. Therefore, we performed another set of experiments to test the four simulation strategy we designed: *Random Simulation* (RS), *Greedy Simulation* (GS), *Epsilon-Greedy Simulation* (EGS), and *Card Random Simulation* (CRS). We used the same setup of the previous experiments, where MCTS played 1,000 games 10 times against the Greedy strategy. Figure 6.5

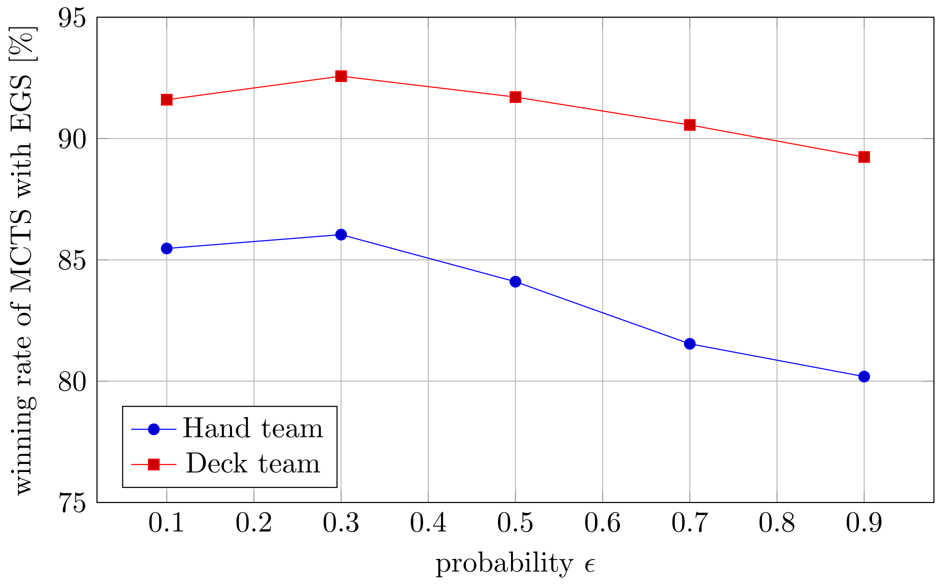


Figure 6.5: Winning rate of Monte Carlo Tree Search as a function of the ϵ value used by the Epsilon-Greedy Simulation strategy when it plays against the Greedy strategy both as hand team and deck team.

plots the winning rate of MCTS with EGS as a function of the probability ϵ of playing at random at each turn. Figure 6.6 shows the comparison between the best EGS probability and the other simulation strategies, with MCTS playing both as hand team and deck team. From the results, it is clear that the best value of ϵ is 0.3 and the EGS strategy turns out to be the best choice. In fact, it outperforms RS of about 7% as hand team, and 3% as deck team. Therefore, the inclusion of the Greedy strategy in the simulation step gives some advantages, because it allows MCTS to exploit its encoded knowledge of the game. However, relying completely on it is not the best choice, probably because MCTS, since it is a cheating player, can also see the hidden cards of the opponents and following too much a strategy that cannot see them is not beneficial. In fact, GS turns out to be even worse than RS. CRS also includes some knowledge of the game, but probably it is not sufficient to give some advantages, in fact it is almost equivalent to RS. For these reasons, we fixed the EGS strategy with $\epsilon = 0.3$ for the next experiments.

6.4.4 Reducing the number of player moves

Finally, we tested the four moves handlers we designed to reduce the number of moves available in each node to the MCTS player: *All Moves Handler*

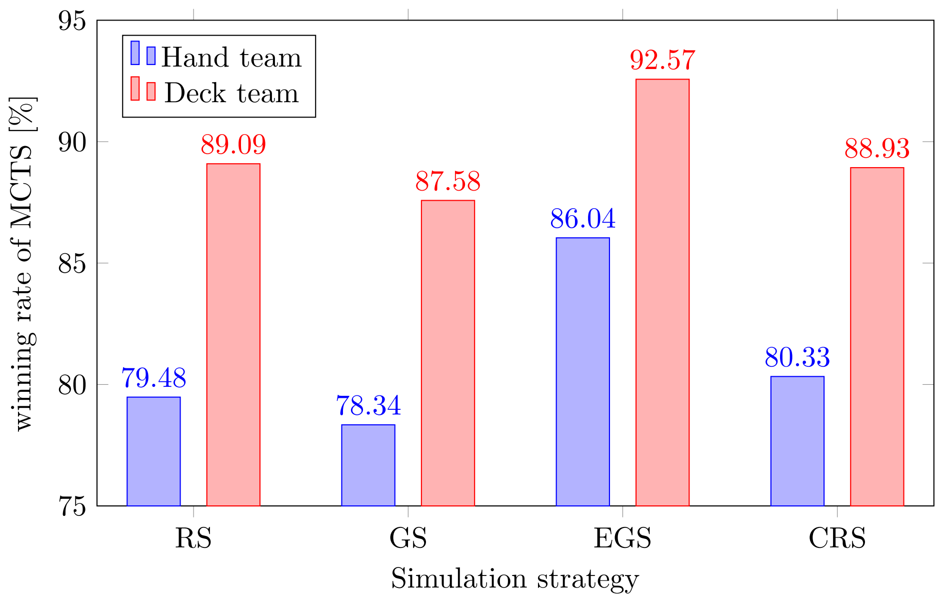


Figure 6.6: Simulation strategies comparison on the Monte Carlo Tree Search winning rate when it plays against the Greedy strategy both as hand team and deck team.

(AMH), *One Move Handler* (OMH), *One Card Handler* (OCH), and *Greedy Opponents Handler* (GOH). For each strategy, MCTS played 1,000 games 10 times against the Greedy strategy. Figure 6.7 shows the comparison between the moves handlers, with MCTS playing both as hand team and deck team. From the results, it is easy to see that AMH, OMH, and OCH are equivalent. While, GOH significantly outperforms the other methods. This is not surprising because GOH allows MCTS to predict the moves of the opponents and choose the best move accordingly. However, this might be a case of overfitting, because MCTS is playing against the Greedy strategy that is also used by GOH to predict the opponents' moves, therefore the prediction is always correct. Moreover, the advantage of the deck team over the hand team is significantly decreased when MCTS uses GOH, this may be another clue of the fact that we are overfitting. For these reasons, we also tested AMH and GOH with MCTS playing against the CS strategy. We chose to maintain only AMH because OMH and OCH did not provide any advantage, neither in terms of winning rate nor in terms of speed. Figure 6.8 shows the comparison between AMH and GOH, with MCTS playing both as hand team and deck team. The results show that GOH is significantly weaker than AMH, this is the proof that MCTS with GOH overfits the Greedy strategy. Therefore, AMH turns out to be the best moves handler and we fixed it for the final tournament.

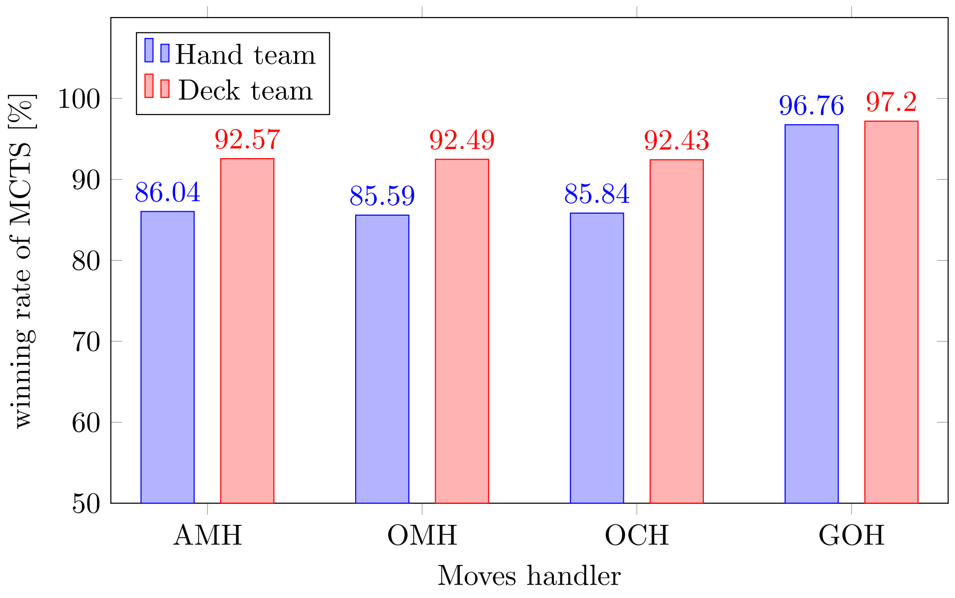


Figure 6.7: Moves handlers comparison on the Monte Carlo Tree Search winning rate when it plays against the Greedy strategy both as hand team and deck team.

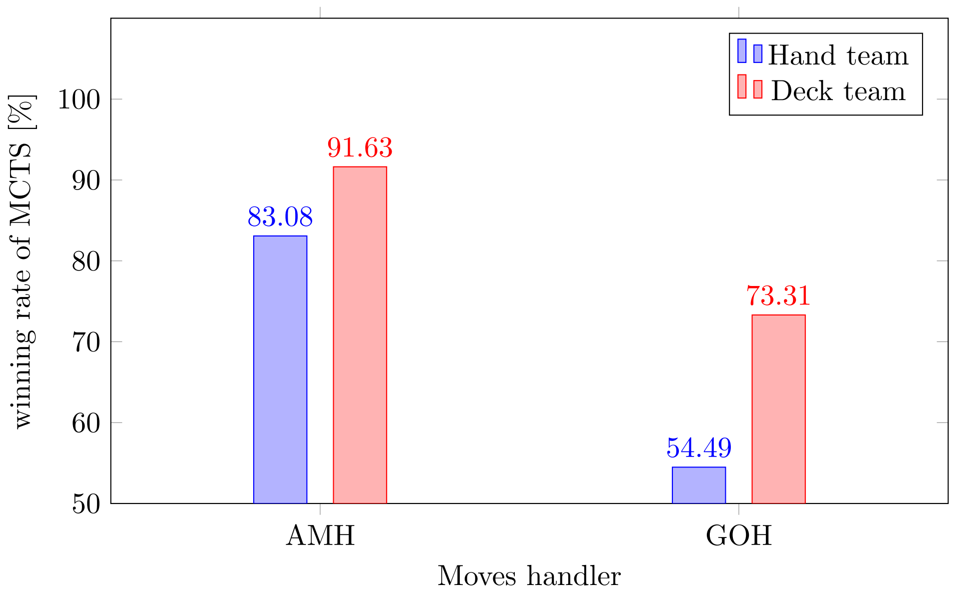


Figure 6.8: Moves handlers comparison on the Monte Carlo Tree Search winning rate when it plays against the Chitarrella-Saracino strategy both as hand team and deck team.

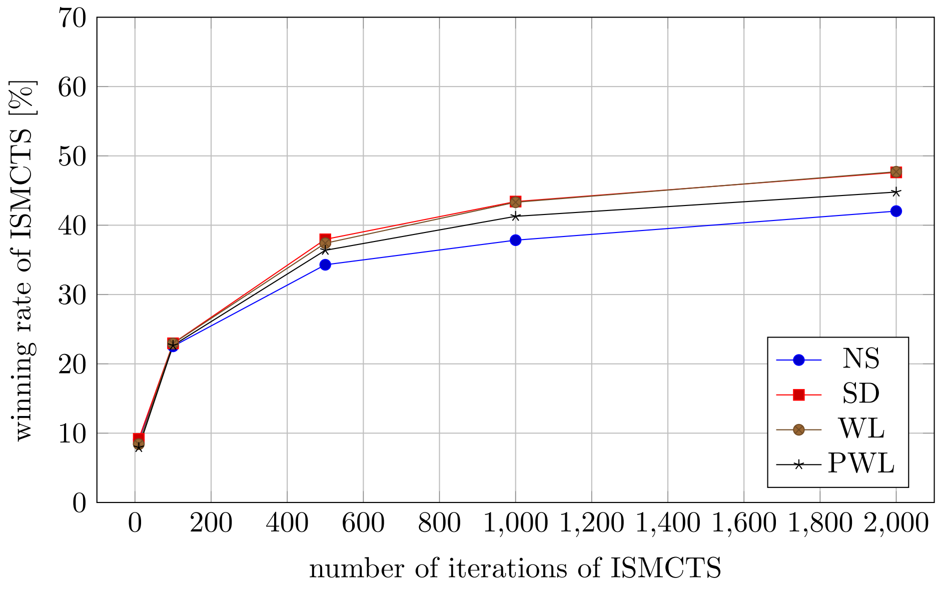


Figure 6.9: Reward methods comparison on the Information Set Monte Carlo Tree Search winning rate as a function of the number of iterations when it plays against the Greedy strategy as the hand team.

6.5 Information Set Monte Carlo Tree Search

We repeated the same experiments using ISMCTS. In this case, ISMCTS is not cheating, it is not aware of which cards the other players hold. It bases its decisions on a tree search, where each node is an information set representing all the game states compatible with the information available to the ISMCTS player.

6.5.1 Reward Methods

In the case of ISMCTS, the comparison of the reward methods produced the following results: Figure 6.9 plots the winning rate as a function of the number of iterations when ISMCTS plays as the hand team, while Figure 6.10 shows the results when ISMCTS plays as the deck team. The trends of the plots are similar to the MCTS ones, but they stabilize with a higher number of iterations, in fact they seem to grow also after the 2,000 iterations. Also this time, WL and SD reach the higher winning rate, moreover they are equivalent for the hand team. Because of this consideration and the ones we discussed for MCTS, we chose to use them also in the next experiment. The results also show that ISMCTS already outperforms the winning rate achieved by CS versus the Greedy strategy. In fact, the current best ISM-

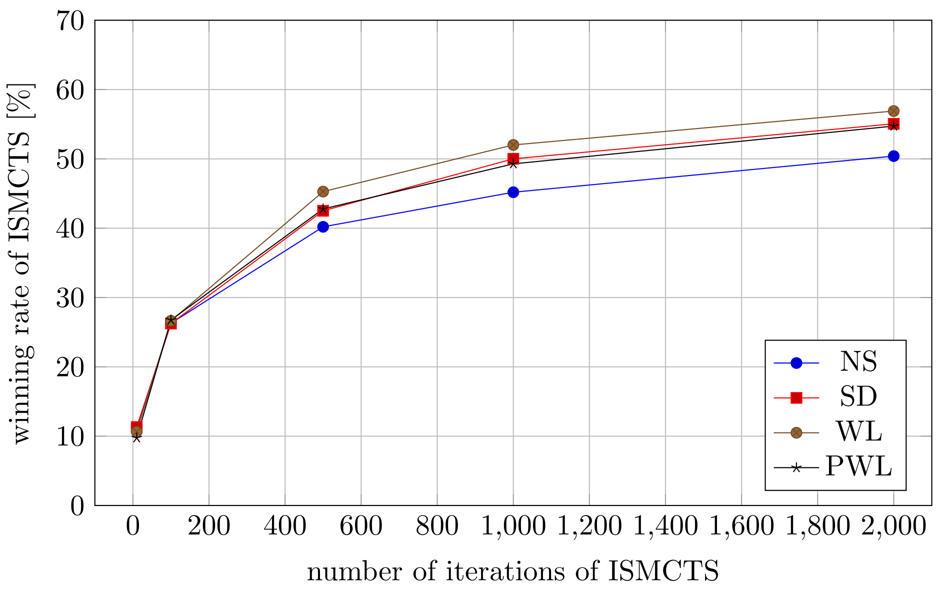


Figure 6.10: Reward methods comparison on the Information Set Monte Carlo Tree Search winning rate as a function of the number of iterations when it plays against the Greedy strategy as the deck team.

CTS wins the 47.71% of the games as hand team and the 56.9% as deck team. Whereas, CS wins the 46.7% and 48.8% of times respectively. This also tells us that ISMCTS plays better as deck team, probably because it better exploits the advantages of this role.

6.5.2 Upper Confidence Bounds for Trees Constant

The *Information Set Upper Confidence Bounds for Trees* (ISUCT) formula uses also an exploration constant in the selection step of the ISMCTS algorithm. In order to determine the best ISUCT constant, we fixed the number of iteration of the ISMCTS algorithm to 1,000. Figure 6.11 plots the winning rate as a function of the ISUCT constant when ISMCTS plays as the hand team, while Figure 6.12 shows the results when ISMCTS plays as the deck team. The plots confirm the results obtained for MCTS: the best ISUCT constant for WL is 0.75, whereas for SD it is 2. The general trend remains the same of the MCTS algorithm and between the hand team and the deck team there are no significant differences. This is not surprising, because it depends on the rewards. Also this time, SD turns out to be the best choice, because it outperforms WL of about the 2% of wins. Therefore, we fixed this setting for the next experiments.

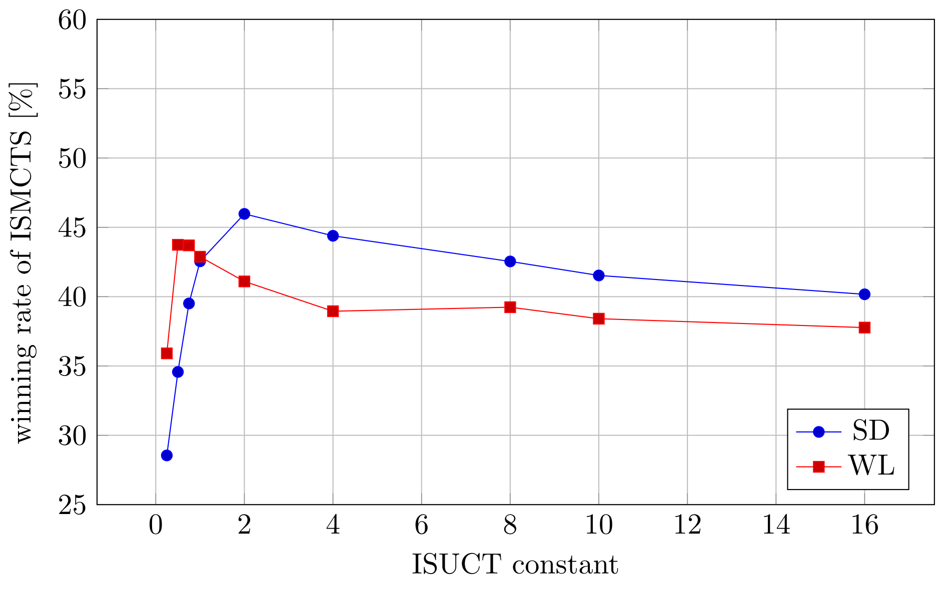


Figure 6.11: Scores Difference and Win or Loss reward methods comparison on the Information Set Monte Carlo Tree Search winning rate as a function of the Information Set Upper Confidence Bounds for Trees constant when it plays against the Greedy strategy as the hand team.

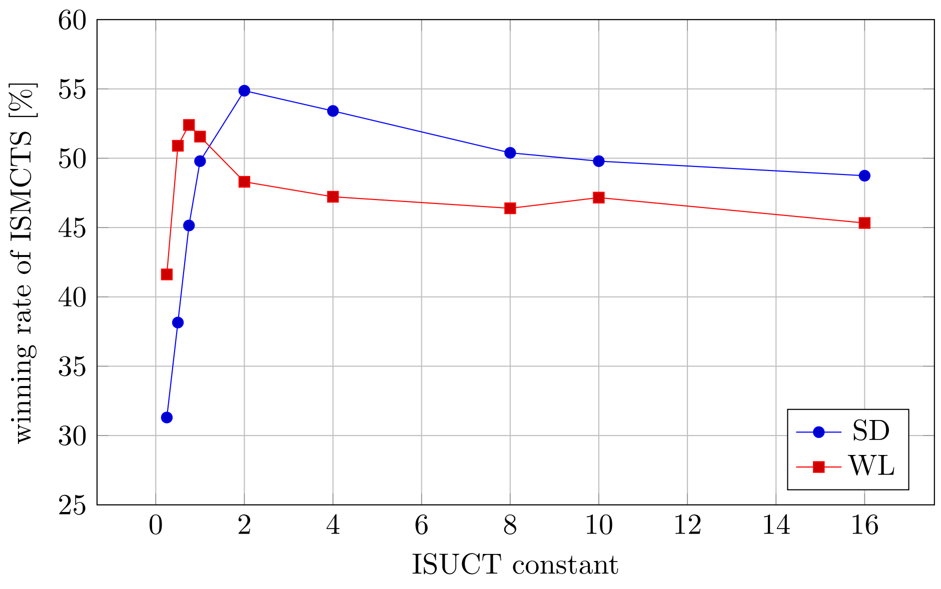


Figure 6.12: Scores Difference and Win or Loss reward methods comparison on the Information Set Monte Carlo Tree Search winning rate as a function of the Information Set Upper Confidence Bounds for Trees constant when it plays against the Greedy strategy as the deck team.

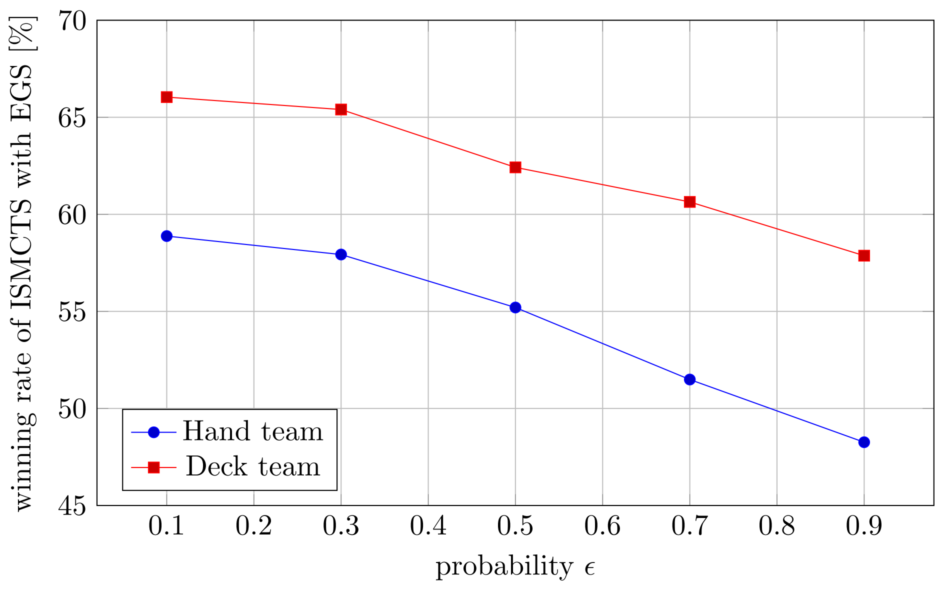


Figure 6.13: Winning rate of Information Set Monte Carlo Tree Search as a function of the ϵ value used by the Epsilon-Greedy Simulation strategy when it plays against the Greedy strategy both as hand team and deck team.

6.5.3 Simulation strategies

As we did with MCTS, in these experiments we tested different simulation strategies for the ISMCTS algorithm. Figure 6.13 plots the winning rate of ISMCTS with EGS as a function of the probability ϵ of playing at random at each turn. Figure 6.14 shows the comparison between the best EGS probability and the other simulation strategies, with ISMCTS playing both as hand team and deck team. This time the results suggest us that the best probability for EGS is 0.1 and it has almost the same winning rate of GS. This is completely different of what happened with MCTS, where GS was even worse than RS. Probably, ISMCTS is able to exploit all the game knowledge of the Greedy strategy because they do not know the cards of the other players. Whereas, MCTS can also see the cards of the opponents and the Greedy strategy could direct the search in wrong areas of the tree. Therefore, both EGS and GS are in charge to be selected as the best solution. In fact, they both significantly outperform RS of about 13% as hand team, and 12% as deck team. However, we prefer to maintain only GS for the next experiments, for the reason that it is faster than EGS, since the latter requires the additional step of generating a random number. In this case, the little knowledge of CRS has some effect in the winning rate, but it is not comparable with the one reached by GS.

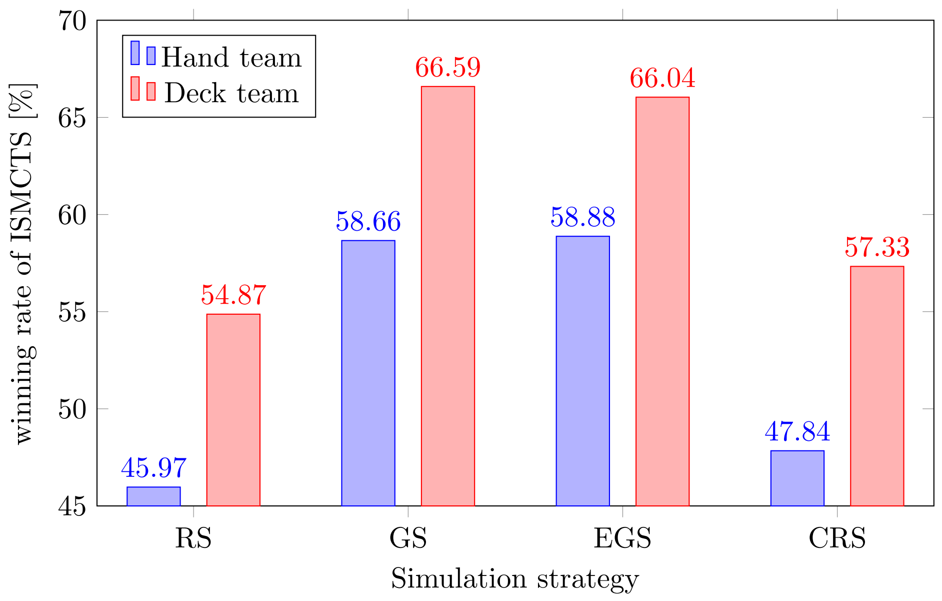


Figure 6.14: Simulation strategies comparison on the Information Set Monte Carlo Tree Search winning rate when it plays against the Greedy strategy both as hand team and deck team.

6.5.4 Reducing the number of player moves

In this set of experiments, we tested the four moves handlers that we used also for MCTS. Figure 6.15 shows the comparison between the moves handlers, with ISMCTS playing both as hand team and deck team. Once again, AMH, OMH, and OCH are equivalent and GOH appears to be the best handlers. But, as we did with MCTS, we tested AMH and GOH also with ISMCTS playing against the CS strategy, because GOH might overfit the Greedy strategy. Figure 6.16 shows the comparison between AMH and GOH, with ISMCTS playing both as hand team and deck team. This time, AMH and GOH are equivalent, probably this can be explained by the fact that GOH cannot successfully predict the moves of the opponents since it cannot see their cards. In fact, it predicts the move of the opponents in each determinization, but the set of all the predicted moves in all the determinizations may be equivalent to the one created by AMH. However, we believe that AMH is generally better than GOH because it does not restrict the search and allows ISMCTS to exploit all the moves, therefore we fixed it for the next experiments.

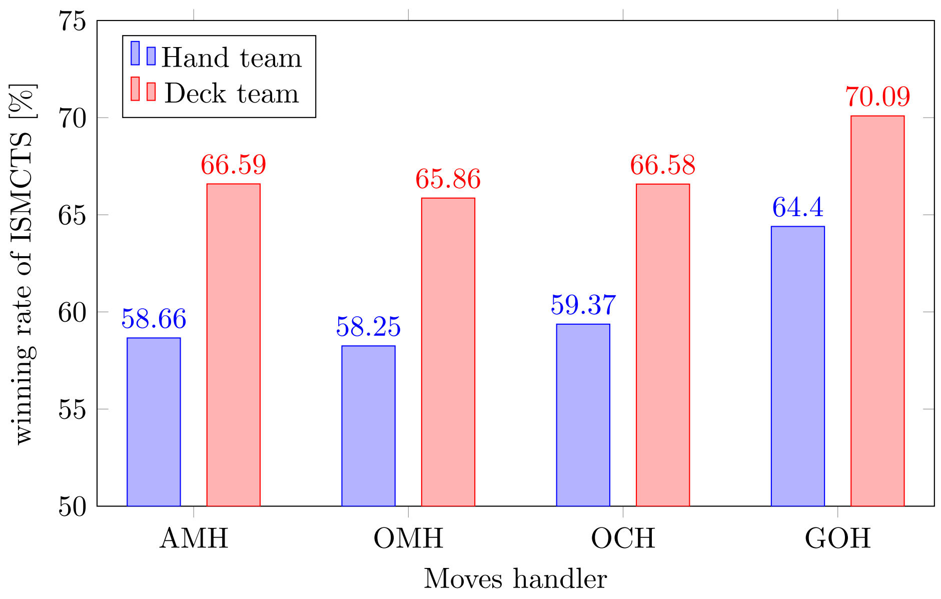


Figure 6.15: Moves handlers comparison on the Information Set Monte Carlo Tree Search winning rate when it plays against the Greedy strategy both as hand team and deck team.

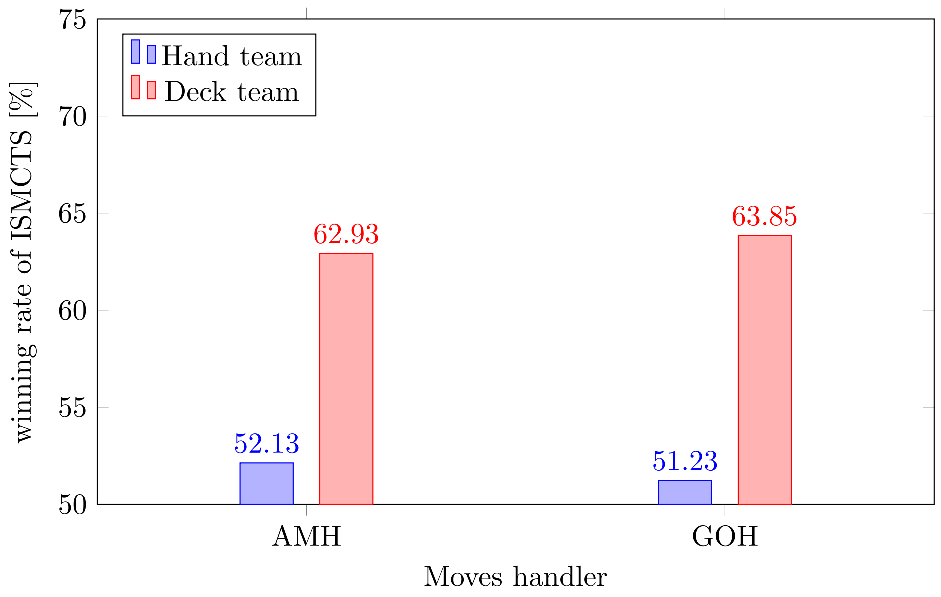


Figure 6.16: Moves handlers comparison on the Information Set Monte Carlo Tree Search winning rate when it plays against the Chitarrella-Saracino strategy both as hand team and deck team.

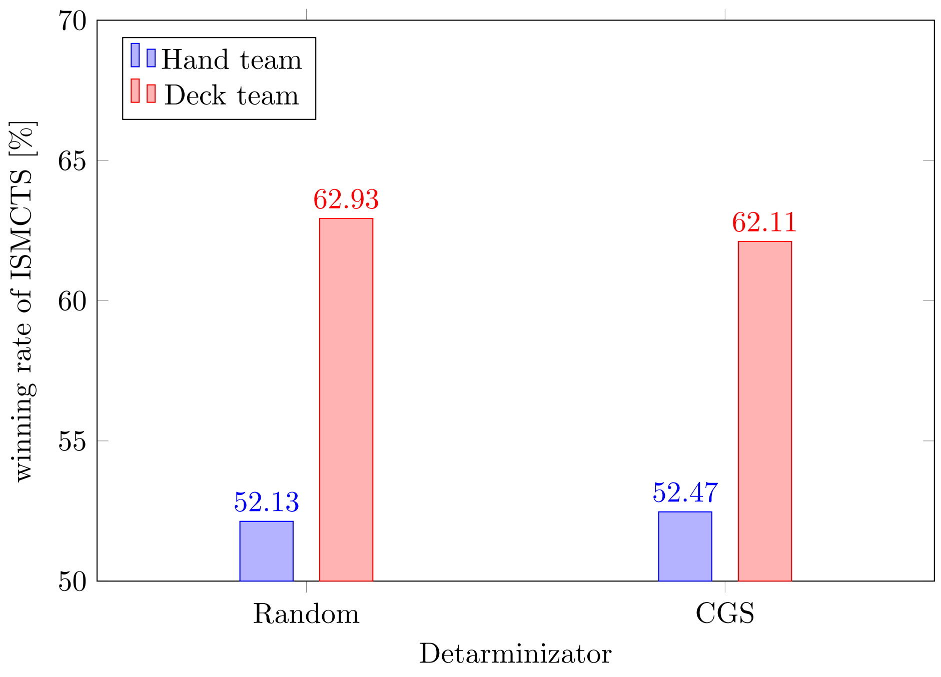


Figure 6.17: Determinizators comparison on the Information Set Monte Carlo Tree Search winning rate when it plays against the Chitarrella-Saracino strategy both as hand team and deck team.

6.5.5 Determinizators

In the last experiment, we tested the two methods that are used by ISMCTS to determinize the root information set at each iteration. The *Random* determinizator simply samples a state within the root information set, while the *Cards Guessing System* (CGS) determinizator restricts the sample to the states in which each player holds the cards guessed by the cards guessing system. Since the cards guessing system is designed to take into account the rules of the CS strategy, it does not make sense testing it against the Greedy strategy. For this reason, we tested both the determinizators only against CS. Figure 6.17 shows the comparison between the Random and CGS determinizators, with ISMCTS playing both as hand team and deck team. The results show that they are equivalent, but we believe that CGS allows ISMCTS to avoid moves that bring the opponents to an easy scopa, since with CGS it can predict the cards that the opponents might hold. For this reason, we fixed it for the final tournament.

Table 6.6: Final Tournament. In each section, the artificial intelligence used by the hand team are listed at the left, while the ones used by the deck team are listed at the bottom. The first section shows the percentage of wins of the hand team, the second one shows the percentage of losses of the hand team, and the third one shows the percentage of ties.

Winning rates of the hand team				
Random	41.69%	3.25%	0.08%	1.26%
CS	90.87%	38.7%	2.78%	24.6%
MCTS	98.72%	83.08%	30.29%	75.52%
ISMCTS	94.2%	52.47%	3.87%	33.67%
Hand Team/Deck Team	**Random**	**CS**	**MCTS**	**ISMCTS**

Losing rates of the hand team				
Random	45.71%	93.05%	99.56%	96.35%
CS	4.4%	46.6%	91.63%	62.11%
MCTS	0.31%	9.03%	51.94%	13.49%
ISMCTS	2.36%	34.72%	89.13%	52.63%
Hand Team/Deck Team	**Random**	**CS**	**MCTS**	**ISMCTS**

Tying rates				
Random	12.6%	3.7%	0.36%	2.39%
CS	4.73%	14.7%	5.59%	13.29%
MCTS	0.97%	7.89%	17.77%	10.99%
ISMCTS	3.44%	12.81%	7%	13.7%
Hand Team/Deck Team	**Random**	**CS**	**MCTS**	**ISMCTS**

6.6 Final Tournament

In the last experiment, we performed a tournament between the random strategy and the best AI resulting from the previous experiments: CS, the best rule-based AI; MCTS with the Scores Difference rewards, the UCT constant equals to 2, the Epsilon-Greedy Simulation strategy with $\epsilon = 0.3$, and the All Moves handler; ISMCTS with the Scores Difference rewards, the ISUCT constant equals to 2, the Greedy Simulation strategy, the All Moves handler, and the Cards Guessing System determinizator. In both the MCTS and ISMCTS algorithms were used 1,000 iterations per move. Table 6.6 shows the results of this tournament, in which 1,000 games have been played 10 times in each match, for a total of 80,000 matches, and Table 6.7 shows the final scoreboard. Unsurprisingly, playing at random is the worst strategy. In fact, it loses the 82.52% of times and wins only the 12.38% of games, the

Table 6.7: Scoreboard of the final tournament. It shows the percentage of wins, losses, and ties each artificial intelligence did during the final tournament.

AI	Wins	Losses	Ties
MCTS	77.48%	13.97%	8.54%
ISMCTS	51.1%	39.24%	9.67%
CS	42.54%	47.78%	9.68%
Random	12.38%	82.52%	5.1%

majority of them against itself. While, the best strategy is MCTS. In fact, it wins the 77.48% of times and loses only the 13.97% of games, the majority of them against itself. This was expected, since it is a cheating player and it can see the cards of the other players. The comparison between CS and ISMCTS is more interesting, since both of them have a partial knowledge of the game state. ISMCTS wins the comparison, because it wins the 51.1% of times and loses the 39.24% of games, against the 42.54% of wins and the 47.78% of losses of the CS strategy. This is an important result, since ISMCTS proved to be stronger than an AI encoding the rules written by expert players of Scopone. Moreover, Figures 6.9 and 6.10 suggest that the strength of ISMCTS can be improved by increasing the number of iterations of the algorithm, since at 1,000 iterations the convergence has not been reached yet. The results also confirm that the deck team has an advantage over the hand team. Moreover, the advantage increases with the ability of the player. In fact, if we consider the matches between the same AI (the diagonal values of the tables), we find out that the winning rate of the hand team decreases as the player's strength increases: 41.69% of the random strategy, 38.7% of CS, 33.67% of ISMCTS, and 30.29% of MCTS.

6.7 Summary

In this chapter, we discussed the experiments we performed to evaluate the playing strength of the Artificial Intelligence (AI) we designed. The first experiment showed that, by playing at random, the deck team has an advantage over the hand team, and this results has been confirmed also in the next experiments. Then, we did a tournament between the rule-based AI and the Chitarrella-Saracino (CS) strategy turns out to be the best one. In the next experiments, we selected the best optimizations for the Monte Carlo Tree Search (MCTS) and Information Set Monte Carlo Tree Search (ISMCTS) algorithms. The best MCTS configuration uses the

Scores Difference rewards, the Upper Confidence Bounds for Trees constant equals to 2, the Epsilon-Greedy Simulation strategy with $\epsilon = 0.3$, and the All Moves handler. While, the best ISMCTS configuration uses the Scores Difference rewards, the Information Set Upper Confidence Bounds for Trees constant equals to 2, the Greedy Simulation strategy, the All Moves handler, and the Cards Guessing System determinizator. Finally, we performed a tournament between the random strategy and the best AI resulting from the previous experiments. The results confirmed that the deck team has an advantage over the hand team and the advantage increases with the ability of the player. MCTS obviously achieved the best performance, since it is a cheating player, while the random strategy is clearly the worst one. The important thing is that ISMCTS proved to be stronger than the CS strategy.

Chapter 7

Conclusions

This thesis aimed at designing a competitive *Artificial Intelligence* (AI) algorithm for the card game of Scopone. In order to achieve this goal, we developed three rule-based AI encoding the rules taken from well-known strategy books of Scopone. The *Greedy* strategy simply captures the most important cards on the table or plays the least important one held by the player. The *Chitarrella-Saracino* (CS) strategy includes the rules taken from the strategy books of Chitarrella [11] and Saracino [24] about the spariglio, the mulinello, the play of double and triple cards, and the play of sevens. The *Cicuti-Guardamagna* (CG) strategy is an extension of the CS AI, and encodes the rules proposed by Cicuti and Guardamagna [12] about the play of sevens. The experiments showed that CS is the best one, showing that the additional rules for the play of sevens of CG do not increase the playing strength.

Next, we designed different improvements of the *Monte Carlo Tree Search* (MCTS) and *Information Set Monte Carlo Tree* Search (ISMCTS) algorithms in order to select the best configuration for Scopone. We experimented four reward methods: *Normal Score* (NS), it uses the score of each team; *Scores Difference* (SD), it uses the difference between the score of the teams; *Win or Loss* (WL), it uses 1 for a win, -1 for a loss, and 0 for a tie; and *Positive Win or Loss* (PWL), it uses 1 for a win, 0 for a loss, and 0.5 for a tie. The best rewards method turned out to be SD, with an *Upper Confidence Bounds for Trees* (UCT) constant equals to 2.

Then, we tested four simulation strategies: *Random Simulation* (RS), it chooses a move at random; *Greedy Simulation* (GS), it chooses the move selected by the Greedy strategy; *Epsilon-Greedy Simulation* (EGS), with probability ϵ it chooses a move at random, otherwise it selects the move chosen by the Greedy strategy; and *Card Random Simulation* (CRS), it plays a card at random, but the Greedy strategy decides which capturing move to do in case there are more than one. The best simulation strategy turned out to be EGS with $\epsilon = 0.3$ for MCTS, and GS for ISMCTS. Probably, ISMCTS is able to exploit all the game knowledge of the Greedy strategy because both of thems do not know the cards of the other players. Whereas, MCTS can also see the cards of the opponents and relying completely on the Greedy strategy could direct the search in wrong areas of the tree.

We also experimented with four moves handlers to reduce the number of moves available in each node: *All Moves Handler* (AMH), it generates all the legal moves of a state; *One Move Handler* (OMH), it generates one move for each card in the player's hand and uses the Greedy strategy to choose the cards to capture; *One Card Handler* (OCH), like OMH but the move memorizes only the played card; and *Greedy Opponents Handler* (GOH), when it is not the turn of the root player, it is generated only the move chosen by the Greedy strategy. The experiments showed that AMH, OMH, and OCH are equivalent, but AMH has the advantage to exploit all the available moves. While, GOH is significantly stronger against the Greedy strategy, but with the CS strategy turned out to be weaker. This is a case of overfitting and it happens especially with MCTS. For these reasons, we selected AMH as the best moves handler.

As last variation of the ISMCTS algorithm, we tested two determinizators: *Random* determinizator, it simply samples a state within the root information set; *Cards Guessing System* (CGS) determinizator, it restricts the sample to the states in which each player holds the cards guessed by the cards guessing system. The two methods turned out to be equivalent, but we believe that CGS allows ISMCTS to avoid moves that bring the opponents to an easy scopa, since with CGS it can predict the cards that the opponents might hold. For this reason, we selected it as the best determinizator.

Finally, we performed a tournament between the random strategy, CS, MCTS, and ISMCTS. The results confirmed that the deck team has an advantage over the hand team and the advantage increases with the ability of the player. MCTS obviously achieved the best performance, since it is a cheating player, while the random strategy is clearly the worst one. The important thing is that ISMCTS proved to be stronger than the CS strategy. This confirms that the ISMCTS algorithm is very effective and deserves further research on this topic.

As result of this thesis, we developed an application, using the game engine Unity, that we plan to release for Android and iOS. With the application, the user can play Scopone against the AI that we developed and compile a survey about the perceived playing strength and the human behavior of the AI.

7.1 Future research

There are several areas for potential future research. For instance, the CS and CG strategies do not include all the rules written in the corresponding books, but only the most important of them. Encoding additional rules might increase the playing strength of the AI and provide a better model of an expert player. Moreover, the priority of the rules can be adjusted to provide better results, for instance one might use a genetic algorithm to assign them.

An enhancement for the MCTS and ISMCTS algorithms might be to use the CS AI as simulation strategy. We showed that the Greedy strategy, used in the simulation step, increased the playing strength of the algorithm, therefore using a better AI like CS might increase the performance. Another improvements might be related to the moves handler. We showed that generating only the moves chosen by the Greedy strategy did not provide good results. However, using the CS strategy or trying to remove some moves that are worst for sure, might improve the playing strength of the algorithm. Furthermore, we did not experiment some well-known enhancements of the MCTS algorithm, like AMAF or RAVE [10], that have proved very successful for Go. Finally, during the experiments we performed, we have also collected useful data about the points that each AI achieved during a game. Future research might study this data in order to discover some interesting patterns in the points that each AI achieves.

Appendix A

The Applications

In this appendix we show the main features of the two versions of the application we developed for this thesis: one to do the experiments, and one to let the users play it.

A.1 The Experiments Framework

In order to do the experiments we need for this work, we developed a console application written in C# with the Microsoft Integrated Development Environment (IDE) Visual Studio 2013. We designed the framework in such a way that it is easy to use it also with other games. For this purpose, we defined two interfaces:

- *IGameState*: it represents the state of a game and includes methods to get the available moves, apply a move, check if it is a terminal state, get the scores fo the game, know the last player to move, clone the state, and get a simulation move.
- *IGameMove*: it represents a move of the game. It does not define any particular method because the moves of different games can be very different. We need only to test if a move is equal to another one, but this is already provided by the Object interface by overriding the Equals method.

Then, we implemented these interfaces with the classes SoponeGameState and ScoponeMove. We also want that the informations in the nodes of the tree are not fixed, such that one can implement its variant of the Monte Carlo Tree Search (MCTS) algorithm, and not just use the standard Upper Confidence Bounds for Trees (UCT) algorithm. For this reason, we defined other two interfaces:

- *ITreeNode*: it represents a node of the MCTS algorithm. It defines methods to add a child, get the parent node, get the incoming move and the player who did it, select the next node to visit, select one of the untried move, update the informations inside the node, and get the best move connecting its children. An ITreeNode creation requires in input the associated IGameState, IGameMove, and the parent ITreeNode. Then, instead of storing the state, it memorizes the legal moves from that state and it will select one of them when an expansion is needed.

- *ITreeNodeCreator*: it provides a method to generate the root of a tree. It allows the MCTS algorithm to be independent by the actual implementation of the ITreeNode interface.

We provided the UCT implementation of these interfaces with the classes UCTTreeNode and UCTTreeNodeCreator. The class MCTSAlgorithm implements the general MCTS algorithm that relies on the previous interfaces. The code is shown in Source Code A.1.

Source Code A.1: Monte Carlo Tree Search algorithm implementation

```
public IGameMove Search(IGameState rootState, int iterations)
{
    ITreeNode rootNode = treeCreator.GenRootNode(rootState);
    for (int i = 0; i < iterations; i++) {
        ITreeNode node = rootNode;
        IGameState state = rootState.Clone();

        // Select
        while (!node.HasMovesToTry() && node.HasChildren()) {
            node = node.SelectChild();
            state.DoMove(node.Move);
        }

        // Expand
        if (node.HasMovesToTry()) {
            IGameMove move = node.SelectUntriedMove();
            state.DoMove(move);
            node = node.AddChild(move, state);
        }

        // Rollout
        while (!state.IsTerminal()) {
            state.DoMove(state.GetSimulationMove());
        }

        // Backpropagate
        while (node != null) {
            node.Update(state.GetResult(node.PlayerWhoJustMoved));
            node = node.Parent;
        }
    }

    return rootNode.GetBestMove();
}
```

For the Information Set Monte Carlo Tree Search (ISMCTS) algorithm, the things are a little bit different. The game state also need a method to clone and randomize it from the point of view of a given player, we need it to create a determinization of the state. Therefore, we created the interface IISGameState that extends IGameState and adds this method. Also the interface of the node must change, for this purpose we created the interfaces IISTreeNode and IISTreeNodeCreator. Since the node of the ISMCTS tree represents an information set, i.e. a set of states, storing all the states or the current determinization does not make sense. For this reason, some methods defined in IISTreeNode take in input the current determinization in order to calculate the current available moves when they are needed. Also in this case, we provide the Information Set Upper Confidence Bounds for Trees (ISUCT) implementation of these interfaces with the classes ISUCTTreeNode and ISUCTTreeNodeCreator. The class ISMCTSAlgorithm implements the general ISMCTS algorithm that uses the previous interfaces. The code is shown in Source Code A.2.

Source Code A.2: Information Set Monte Carlo Tree Search algorithm implementation

```
public IGameMove Search(IISGameState rootState, int iterations)
{
    IISTreeNode rootNode = treeCreator.GenRootNode(rootState);
    for (int i = 0; i < iterations; i++) {
        IISTreeNode node = rootNode;
        IISGameState state =
                rootState.CloneAndRandomize(rootState.PlayerToMove());

        // Select
        while (!node.HasMovesToTry(state) && node.HasChildren()) {
            node = node.SelectChild(state);
            state.DoMove(node.Move);
        }

        // Expand
        if (node.HasMovesToTry(state)) {
            IGameMove move = node.SelectUntriedMove(state);
            state.DoMove(move);
            node = node.AddChild(move, state);
        }

        // Rollout
        while (!state.IsTerminal()) {
            state.DoMove(state.GetSimulationMove());
        }

        // Backpropagate
```

```
            while (node != null) {
                node.Update(state.GetResult(node.PlayerWhoJustMoved));
                node = node.Parent;
            }
        }

        return rootNode.GetBestMove();
}
```

In order to easily implement the variants of the MCTS and ISMCTS algorithms we proposed for Scopone (Chapter 5), we made the class SoponeGameState take in input four components:

- *IScoponeSimulationStrategy*: it is responsible to generate a simulation move from the state according to a fixed strategy.
- *IScoponeMovesHandler*: it generates the available moves of the state or a subset of them.
- *IScoponeRewardsMethod*: it returns the result of the state.
- *IScoponeDeterminizator*: it is responsible to generate a determinization of the state.

This allows us to select which variant of the algorithm to use by simply passing to the algorithm a SoponeGameState initialized with the desired components.

A.2 The User Application

In order to let the user play with our Artificial Intelligence (AI), we developed a user application with the cross-platform game creation system Unity. It includes a game engine and the open-source IDE MonoDevelop. The game engine's scripting is built on Mono, the open-source implementation of .NET Framework. It provides a number of libraries written in C#, Javascript, and Boo in order to manage graphics, physics, sound, network, and inputs of a game. One of the greatest advantages of Unity is that it allows the deployment of the same code on several platforms. In fact, it is able to build an application for Windows, Mac OS X, Linux, iOS, Android, BlackBerry, Windows Phone, web browsers, and game consoles. With the recent versions, it allows also to develop a 2D game by means of dedicated libraries and components. For the development of our game we used Unity 4.5.5 with the C# programming language, so that we can use the same code we created for the console application.

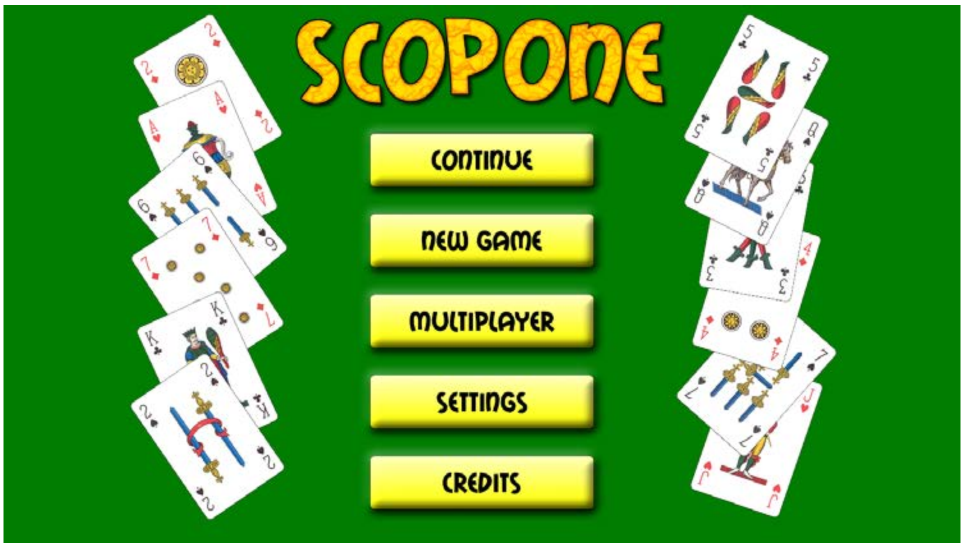


Figure A.1: Menu of the user application

The application starts with a menu (Figure A.1) where you can choose to continue a previous game, start a new game, start a multiplayer session, change the settings, or see the credits. In the settings menu you can change the amount of points to win a game, change the speed of the cards' animation, and change the time that the card, chosen by the AI, freezes before proceeding with the move's animation. If one chooses to start a new game, it will be displayed the AI selection screen, where one can select the AI used by the teammate and the opponents. Then, the round begin with the distribution of the cards. The dealer is chosen randomly and your cards are always at the south position on the screen. During the round (Figure A.2), an icon with two rotating gears is shown on the cards of the player that is thinking on the move to do. When it is your turn, you can just touch the card you want to play and the system will automatically select the captured cards. If more than one combination of captured cards is possible, you have also to touch the cards you want to capture; at each touch, the system will filter out all the combinations that do not include the selected card, until only one combination remains. When the round is finished, the scores and the teams' pile are displayed, and eventually the winner is elected (Figure A.3). The state of the game is saved automatically at each player move, in such a way that, if one exits from the application, then it can continue the game from the point where it left it. In order to collect useful data about the AI, at the end of a game, a survey is displayed and the user can send us its opinion about the AI. By starting a multiplayer session, the program searches for

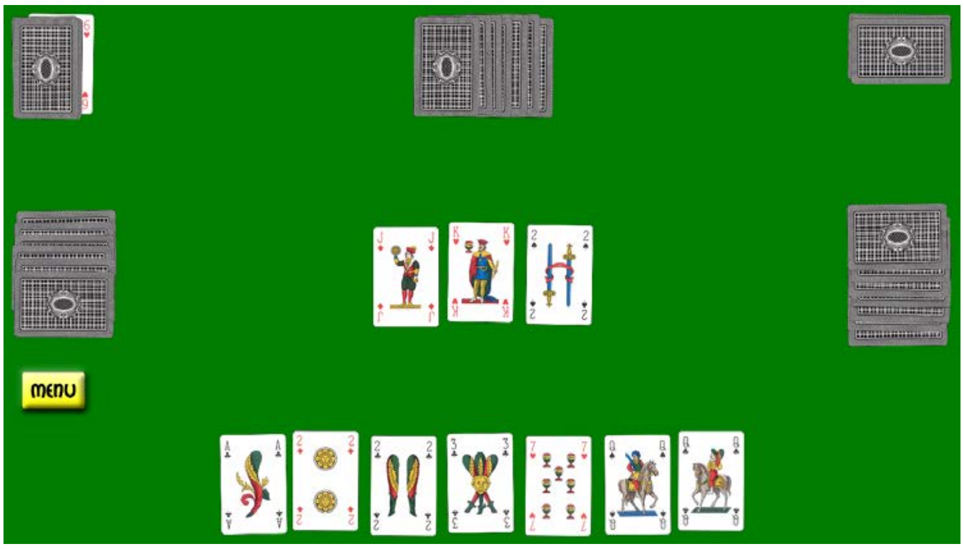
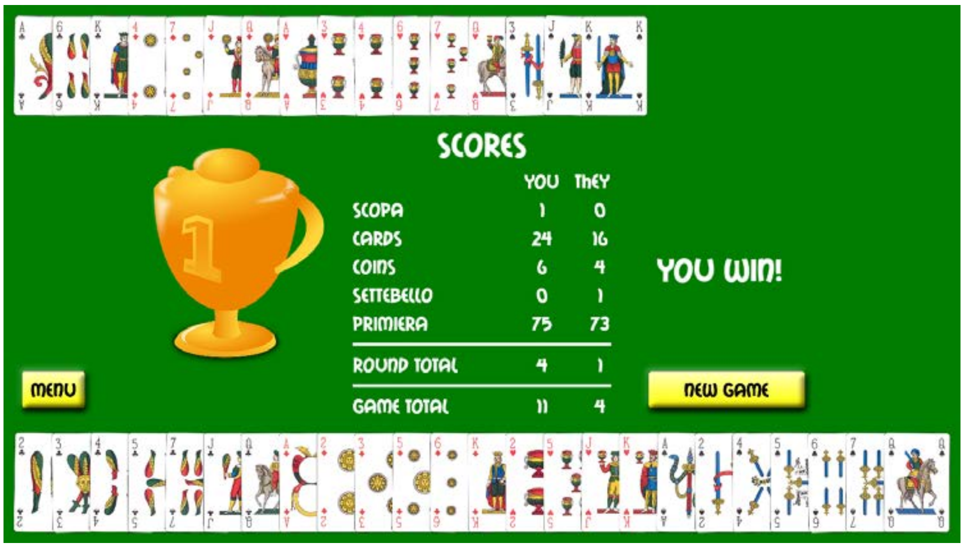


Figure A.2: Round of the user application



SCORES

	YOU	THEY
SCOPA	1	0
CARDS	24	16
COINS	6	4
SETTEBELLO	0	1
PRIMIERA	75	73
ROUND TOTAL	4	1
GAME TOTAL	11	4

Figure A.3: Scores at the end of a round of the user application

available servers registered on a master server. If no server is found, then the program starts a server itself and registers it to the master server. Once all the four players are connected, the game starts and proceeds as usual.

Bibliography

[1] Chess programming. http://chessprogramming.wikispaces.com.

[2] History of ai. http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html.

[3] Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte carlo tree search in hex. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):251–258, 2010.

[4] Computer Go Group at the University of Alberta. Fuego. http://fuego.sourceforge.net/.

[5] Franco Bampi. Scopone: Le regole dei maestri. http://www.francobampi.it/franco/ditutto/scopone/regole_dei_maestri.htm.

[6] Paul M Bethe. The state of automated bridge play. January 2010.

[7] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1):201–240, 2002.

[8] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding klondike solitaire with monte-carlo planning. In *ICAPS*, 2009.

[9] Bruno Bouzy and Tristan Cazenave. Computer go: an ai oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[10] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[11] Chitarrella. *Le regole dello scopone e del tressette.* Fuori collana. Dedalo, 2002.

[12] A. Cicuti and A. Guardamagna. *I segreti dello scopone.* I giochi. Giochi vari. Ugo Mursia Editore, 1978.

[13] UK. Computational Creativity Group (CCG) of the Department of Computing, Imperial College London. Monte carlo tree search. `http://mcts.ai/about/index.html`.

[14] Peter I Cowling, Edward J Powley, and Daniel Whitehouse. Information set monte carlo tree search. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(2):120–143, 2012.

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

[16] Ian Frank and David Basin. Search in games with incomplete information: a case study using bridge card play. *Artificial Intelligence*, 100(1–2):87 – 123, 1998.

[17] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[18] Matthew L Ginsberg. Gib: Imperfect information in a computationally challenging game. *J. Artif. Intell. Res.(JAIR)*, 14:303–358, 2001.

[19] Wan Jing Loh. Ai mahjong. 2009.

[20] Jeffrey Richard Long, Nathan R Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. In *AAAI*, 2010.

[21] JAM Nijssen and Mark HM Winands. Monte-carlo tree search for the game of scotland yard. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 158–165. IEEE, 2011.

[22] Marc JV Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating opponent models with monte-carlo tree search in poker. In *Interactive Decision Theory and Game Theory*, 2010.

[23] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[24] G. Saracino. *Lo scopone scientifico con le regole di Chitarella.* Ugo Mursia Editore, 2011.

[25] Frederik Christiaan Schadd. Monte-carlo search techniques in the modern board game thurn and taxis. *M. sc, Maastricht University, Netherlands*, 2009.

[26] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[27] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, 1996.

[28] Federazione Italiana Gioco Scopone. Figs website. `http://www.federazionescopone.it/`.

[29] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1):241–275, 2002.

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

[31] Daniel Whitehouse, Peter I Cowling, Edward J Powley, and Jeff Rollason. Integrating monte carlo tree search with knowledge-based methods to create engaging play in a commercial mobile game. In *AIIDE*, 2013.

[32] Wikipedia. Arthur samuel — wikipedia, the free encyclopedia, 2014. `http://en.wikipedia.org/w/index.php?title=Arthur_Samuel`.

[33] Wikipedia. Artificial intelligence (video games) — wikipedia, the free encyclopedia, 2014. `http://en.wikipedia.org/w/index.php?title=Artificial_intelligence_(video_games)`.

[34] Wikipedia. Chess — wikipedia, the free encyclopedia, 2014. `http://en.wikipedia.org/w/index.php?title=Chess`.

[35] Wikipedia. Computer chess — wikipedia, the free encyclopedia, 2014. `http://en.wikipedia.org/w/index.php?title=Computer_chess`.

[36] Wikipedia. Computer othello — wikipedia, the free encyclopedia, 2014. `http://en.wikipedia.org/w/index.php?title=Computer_Othello`.

[37] Wikipedia. Contract bridge — wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Contract_bridge.

[38] Wikipedia. Deep blue (chess computer) — wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Deep_Blue_(chess_computer).

[39] Wikipedia. Game complexity — wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Game_complexity.

[40] Wikipedia. Maven (scrabble) — wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Maven_(Scrabble).

[41] Wikipedia. Scopa — wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Scopa.

[42] Wikipedia. Solved game — wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Solved_game.