Providing an AI for the Game of Tresette

Davide Sforza, **Massimo Spaconi** and **Emanuele Zamattio**

Sapienza University of Rome, Italy

1 Introduction

The goal of our work is to provide one or more AIs for the game of *Tresette*, a card game spread all over Italy. Being a partial information game, we used the determinization technique to adapt full information algorithms: we chose to implement Alpha-Beta Search and Monte-Carlo Tree Search.

To assess the AIs we developed, we used the RandomWalk as a lower-bound and cheating AIs as an upper-bound.

In section 2 we present the game. Following, in section 3 we illustrate the techniques we used. Then in section 4 we explain the experiments we ran. In section 5 we discuss the results we obtained through the experiments. In section 6 we present our future plans. Finally, in section 7 we draw the conclusions.

2 The Game

Tresette is a traditional italian card game. It can be played by two players, one against each other, or by four players divided into two team. In our work we will analize this second variant.

In the current section we first present some historical notes about *Tresette* and explain its rules. Then we analize the game classifying it and providing some computational complexity observations.

2.1 Historical Background

The game appears to have come to the italic peninsula through spanish influence around the 1600, and it quickly became popular around Naples both in high-class establishments and in dive bars near the harbor[9].

It got codified in the treatise "*De regulis ludendi ac solvendi in mediatore et tresseptem*", written in Dog Latin by the christian priest Chitarrella (printed in 1750, Naples) [9, 13].

Although the game is primarily known and played in Italy, it has spread in other areas that were under the Venetian influence, such as Slovenia and Dalmatia.

2.2 Rules

The card game is typically played using italian cards, but a french deck can be easily adapted to fit the game. We need 40 cards of 4 different suits, so that suits will have 10 cards each.

There are 4 people playing, divided into 2 teams of 2 people each. The players are arranged so that each faces the ally and has the opponents on both sides. Note that many variants of the game exist, changing across different regions of Italy. We will consider the central Italy variant, the roman one in particular.

We set the winning threshold at 21 points and the player who starts the round is the one who holds the four of Denari.

Let us first define the terms:

Game A game is a sequence of rounds in which one of the two teams ends up with at least 21 points.

Round A round is a part of a game. It starts with the assignment of the cards to the players and finishes when all cards have been played. A round consists of ten mano.

Mano A Mano is a part of a round. At the beginning of a mano no cards are present on the table, whereas when it

Cards	Points
Three	1/3
Two	1/3
Ace	1
Re	1/3
Cavallo	1/3
Fante	1/3
Seven	0
Six	0
Five	0
Four	0

Table 1: Cards and relative points in the game of *Tresette*. Entries are ordered by card degree.

finishes there are four cards. At the end of the mano, points are assigned to the winning team.

2.2.1 Playing

At the beginning of a round, a player called *Mazziere*, chosen randomly, distribuites ten cards to each of the four players. The player who owns the four of Denari has to start, without any need of throwing the four of Denari as his first card.

The suit of the first card determines the dominating suit of the mano. Then the other players, one after the other and always anti-clockwise, must throw a card. The thrown card must belong to the dominating suit. If no cards are suitable, the player can throw a card belonging to another suit.

At the end of the mano, the player who threw the highest degree card of the dominating suit gets the cards on table, and his team gains all the points associated with them. The points and the degree for each card are shown in table 1. This player becomes the starting player of the next mano.

At the end of the tenth mano, that is the end of the round, all points of the two teams are computed. The player who wins the ending mano gets 1 bonus point. The points gained in each mano are summed and the total gets truncated in his decimal part. If none of the teams reached 21 points, another round is played. Otherwise, that team won the game. If there is a tie, another round is played.

2.2.2 Accusi

It is possible to gain more points through the *accusi*. If a player owns one of the following combination of cards, he must declare it (and the associated cards) in the first mano of each round – as soon as he can play the first card, and his team gains 3 points. The accusi are:

Buongioco It happens when the player has 3 aces, 3 twos or 3 threes. If the player has also the fourth card of the same value, the points the team gains are 4.

Napoli It happens when the player has ace, two and three of the same suit.

2.3 Analysis

We inspect some theoretical aspects of the game. First we provide a classification of the game under classical criteria. Then we investigate the complexity of the search problem (we are particularly interested in the search for the best move).

2.3.1 Classification

We provide a classification of the game.

Zero-Sum The game is zero sum. Each point will eventually be assigned to one of the teams. The *accusi* are computed at the beginning of the match, thus not posing a threat to the zero-sum property.

Partially Observable As in most card games, each player only knows part of the state of the game. In *Tresette*, the player knows his cards and the ones that have already been played. In addition, he can know when another player has no more card of a suit. This poses a big threat to the validity of common perfect information algorithms such as alpha-beta search and MCTS. To overcome this issue, a technique called **determinization** has been used.

Multiagent The game is clearly multiagent, as it presents 4 player in 2 teams.

Within each team there is full cooperation and no competition, whereas between the two teams there is full competition and no degree of cooperation. An interesting aspect of the game is the almost complete absence of communication, even between human players. This varies considerably across the regions of Italy (in most places of Northern Italy talking is forbidden and thus a jesture language is sometimes employed).

Deterministic The outcome state does depend solely on the initial state and the action performed.

Sequential As in most card games, past choices affect the current state of the game. Each round could be considered as independent, and so episodic. Within each round, though, there is sequentiality. Therefore the game as a whole must be regarded as sequential.

Static In *Tresette*, when a player is choosing the action to take (i.e. the card to play) nothing can alter the environment.

Discrete There is a finite set of states in which the environment can be.

Known The physics of the game are known to the agents.

2.3.2 Computational Complexity

Even assuming Full Observability, finding the best move is still a NP-Hard problem. Let us now suppose that the initial state of the game is one in which each player has only cards of the same suit. Therefore, each suit belongs only to a single player. Despite being simple in practise, – as the first player's team will surely win, giving no chance to the opponents to get to start – this represents the **Worst Case Scenario** for a search: it presents a depth of 40, and a branching factor that diminishes by one every four moves. The number of nodes $|V|_{max}$ shown in equation 1 illustrates how an exhaustive search in the game state tree is an unfeasible choice.

$$|V|_{max} = \sum_{i=1}^{10} i!^4 \approx 1.734 \cdot 10^{26} \qquad (1)$$

Thankfully, this upper bound is strongly pessimistic. At each round, each player can only play the cards of the same suit of the first card thrown. Therefore, they are only allowed to choose among their entire set either if they are starting the round or if they do not have any card of the ruling suit left. The former happens every four turns, while the latter is usually rare in early stages of the game, – when the branching factor is higher – and it appears more often as the game approaches to an end.

3 Employed Techniques

We chose to use Alpha-Beta Search and Monte-Carlo Tree Search, which are generally utilised in Fully Observable games. In order to use them, we needed to apply a technique called Determinization.

At first, we wanted to implement also the Information Set Monte-Carlo Tree Search algorithm introduced in [4]. This technique applies directly on Partial Information Games, without explicitly relying on Determinization. The poor performances achieved in [4] and in [5] had us reconsider the idea. In the latter, which deals with other italian card games close to *Tresette*, the ISMCTS was heavily outperformed by Determinized MCTS. In the literature, we did not encounter any cases in which ISMCTS defeated Determinized MCTS.

3.1 Determinization

This technique allows us to apply perfect information algorithms to the game of *Tresette*. It consists in producing a set of full information game states, consistent with the observed partial state, and then apply the chosen perfect information algorithm to every resulting state. The most picked action is the one we choose to perform.

In the technique we used, *Flat Monte-Carlo*, the samples are taken uniformly randomly from the possible consistents assignments. Firstly, we assign all known cards to the players (i.e. cards which entail accusi and four of Denari). Secondly, we proceed in assigning random cards to a player, granting that a player who can no longer have a certain suit does not receive cards belonging to that suit.

This techniques has two major downsides:

- ▷ The algorithm will believe that the state is fully known. Therefore, it will never try to make explorative moves (actions that make the player gain knowledge of the actual state)[10], nor will it exploit the opponent's ignorance (e.g. alpha-beta will behave as if the opponents play optimally, and as if he knows what cards the player holds in his hands). With respect to this game, while the absence of explorative actions do not seem to be a problem as they are usually absent in human play as well, the lack of exploitation of the ignorance appears to be seriously constraining to this technique's strength. Still, an algorithm that provides optimal play assuming that the opponent will play optimally, should behave well even in case of opponent's suboptimal play.[4]
- ▷ Some determinization can be very similar and lead to similar game states. But since we need to recompute everything from scratch and run it independently, it results in a recomputation of visited states.

3.2 Alpha-Beta Search

Basic **MinMax** works by searching into the game tree the path that will lead to the higher utility function, while considering his opponent's play as optimal. Considering a game state tree, we can see each game state as a node and moves as edges. Let us fix a team we want to win. In each game state, the player that makes the move will try to maximise or minimise our team's revenue depending on whether he belongs to my team or not. Therefore we can consider the node in which an opponent is playing as a min-node, while our team's nodes are going to be max-nodes.

Minmax algorithm will explore this game tree, and return the move which maximises our final utility, considering that the opponent will play optimally to try to minimise it. It has a time complexity of $O(b^d)$, where b is the branching factor (at most 10 in our case) and d is the maximum depth of the tree (40). This exhaustive search is unfeasible for large games.

Alpha-Beta Pruning helps deal with this problem by cutting off entire parts of the tree by recognising that they are not worth investigating further. This can lead to a time complexity of $O(b^{d/2})$, a significant gain. We also used Killer Moves Heuristic to speed up the search. For terminal states, we rely on the evaluation described in 3.2.1. The size of the game tree does not allow a search as deep as the tree. Therefore we rely on the evaluation for nonterminal states described in 3.2.2. A preudocode of the algorithm is shown in 1.

3.2.1 Evaluating Terminal Nodes

In a terminal node, the value of the state is given by the difference between the scores of our player's team and the opponents' one.

3.2.2 Evaluating Nonterminal Nodes

The search could be infeasible for the amount of time we are given. In such cases, we need to rely on a utility function that mirrors the appeal of a certain game state, even though it is not a terminal state.

In the game of *Tresette*, we are lucky: every four moves – at the end of each mano – some points are assigned to the teams, and no subsequent action can decrease it. Therefore, the score of each team grows in a monotonic manner. We exploit this fact, by evaluating those particular game states in the same way we evaluate terminal states. In our implementation, we always reach a finished mani state, avoiding so the evaluation of mid-round states.

This leaves a problem regarding how to handle all the points that are still at stake. In this, we decided to follow the simplest path, that is to not compute this in the nonterminal nodes value. This translates into believing that the teams will part equally the points still at stake.

We investigated a more sophisticated evaluation of nonterminal nodes, but it resulted in an unfeasible overhead for our algorithm: the time spent evaluating could have been better spent by searching deeper in the tree.

Algorithm 1 Alpha-Beta Search

```
function ALPHABETA(gamestate, depth, $\alpha$, $\beta$, maximise)
    if depth = 0 or is terminal node then
        return the heuristic value of the gamestate
    end if
    if maximise then
        $v \leftarrow -\infty$
        KILLERMOVESSORTER
        for all child of gamestate do
            $t \leftarrow$ALPHABETA(child, depth - 1, $\alpha$, $\beta$, false)
            $v \leftarrow$ MAX(v,t)
            $\alpha \leftarrow$ MAX($\alpha$, v)
            if $\beta \leq \alpha$ then
                break
            end if
        end for
        return v
    else
        $v \leftarrow \infty$
        KILLERMOVESSORTER
        for all child of gamestate do
            $t \leftarrow$ALPHABETA(child, depth - 1, $\alpha$, $\beta$, true)
            $v \leftarrow$ MIN(v,t)
            $\beta \leftarrow$ MIN($\beta$, v)
            if $\beta \leq \alpha$ then
                break
            end if
        end for
        return v
    end if
end function
```

3.2.3 Killer Moves Heuristic

This heuristic helps speed up Alpha-Beta Pruning. It works by holding a cache of a limited number of moves for each level of the game states tree. The moves held in memory are the ones which resulted in an alpha or beta pruning. This follows the rationale that a move which resulted in a pruning at a certain depth is likely to cause a pruning in other nodes at the same level.[1]

In our implementation we used two caches of four moves for each level of the tree. Since different nodes at the same level could belong to different teams (and therefore be max-nodes and min-nodes), the two caches are necessary to differentiate between the two pruning types.

3.3 Monte-Carlo Tree Search

Monte-Carlo Tree Search is an anytime algorithm that proceeds iteratively until the given time is up. Longer it runs, more accurate will be the prediction of best move. Since it runs random playouts, it does not suffer from the **Horizon Problem** (which affects alpha-beta). Therefore, it can easily deal with high branching factor. UCT, the version of MCTS we use, has been shown to converge to minimax in [2, 8]. During each iteration of MCTS, whose pseudocode is displayed in 2, we do the following steps:

Selection The node to be investigated is chosen. Many ways to do this have been presented in literature, we chose to follow UCT node selection (3).

Expansion The node to be investigated is expanded: children nodes are produced and added to the tree. One among them is chosen at random (4).

Playout The game is randomly played from the chosen game state until a terminal game state is found (5).

Backpropagation The algorithm backpropagates the outcome of the random playout back to the root. It climbs back the tree, updating the visit and winning counters (6).

We used the UCT version of MCTS, which chooses the next node to play out through Upper-Confidence Bound [8]. The priority computation follows the formula:

$$Priority_i = \frac{w_i}{n_i} + c \cdot \sqrt{\frac{\ln(n_{parent})}{n_i}} \quad (2)$$

Where n_i indicates how many times a node has been visited, w_i how many winnings have been produced through this node, n_{parent} the visit count of the parent node. c is the exploration coefficient. Higher it is, more the algorithm will be keen to explore new states. A low c corresponds to no exploration and complete exploitation.

When the time is up, MCTS returns the move whose child node has the highest visit count.

Algorithm 2 Monte Carlo Tree Search

```
function MCTS(currGameState, iterations)
root = NODE(currGameState)
    for all iterations do
        node = SELECT(root)
        node = EXPAND(node)
        winTeam = PLAYOUT(node.gamestate)
        BACKPROPAGATE(node, winTeam)
    end for
    bestNode = MAXNODES(root.children)
    return bestNode.generatingAction()
end function
```

Algorithm 3 Selection

```
function SELECT(node)
    while node has children do
        node = PICKCHILD(node)
    end while
    return node
end function
```

Algorithm 4 Expansion

```
function EXPAND(node)
    if node is terminal then return node
    else
        GENCHILDREN(node)
        return PICKRANDCHILD(node)
    end if
end function
```

4 Experimental Setup

We describe the tests we will perform in order to assess the AIs.

Algorithm 5 Simulation

```
function PLAYOUT(gamestate)
    while is not terminal(gamestate) do
        gamestate = SUCCESSOR(gamestate)
    end while
     return WINTEAM(gamestate)
end function
```

Algorithm 6 Backpropagation

```
function UPDATE(node, winTeam)
    while node is not root do
        node.visit += 1
        if node.maxTeam = winTeam then
            node.win += 1
        end if
    end while
end function
```

4.1 Cheating vs Random

Tests on cheating AIs are needed to assess the performance upper-bound for the game. By cheating we mean that the AI knows the actual assignment of cards to the players.

4.2 Determinized vs Random

With these tests, we inspect how the algorithm behaves against a RandWalk player. We expect that our AIs will beat a player that plays at random most of the time. If an AI gets defeated by RandomWalk, it is probably not worth investigating further.

4.3 Determinized vs Cheating

These tests compare a Partial Information AI against a Cheating AI. Likely, the cheating player will win most of the time, since he knows the current cards of all players. Nontheless, a good AI should be able to resist and perform better than the random one against the cheating player.

4.4 Tests against Humans

At the moment of writing, no tests were run against human players. Still, we prepared a Graphic User Interface to allow humans to play against AIs on equal terms. We don't want a human player to be penalised by the

Depth	Win Rate	Confidence
2	0.91500	± 0.02733
4	0.92000	± 0.02659
6	0.91750	± 0.02696
8	0.93250	± 0.02459
10	0.95000	± 0.02136
12	0.92250	± 0.02620
14	0.94750	± 0.02186
16	0.95500	± 0.02032

Table 2: Cheating AlphaBeta vs RandomWalk (confidence intervals at 95%)

fact that he's playing against virtual opponents instead of real world ones. A screenshot of the GUI we developed is shown in figure 1.

5 Discussion of the Results

5.1 Cheating AI Performance

The tests provide an upperbound for the performance of determinized algorithms. We observe that even an algorithm who knows the actual assignment of cards can fail against a random walk. This is because of the nature of the game. Some good cards can mean a win, despite the stupidity of the player. A highly skilled player can get easily defeated if he gets assigned poor cards, as he will find himself with little or no power against the opponent. This issue is mitigated by the threshold the team has to reach in order to win a game. The probability of teams to get weaker cards in 3 games in a row is smaller than the one in a single game.

The performances of the two cheating algorithms against RandomWalk are presented in tables 2 and 3. We see that AlphaBeta outperforms MCTS against RandWalk. This may be because of the low branching factor this game presents.

We observe that in both cases, after a certain threshold there is a plateau in performance. In AlphaBeta, after depth 8 the results are equivalent, while in MCTS this happens after 1k.

Iterat.	Win Rate	Confidence
5	0.60500	± 0.03030
20	0.77200	± 0.02600
200	0.89300	± 0.01916
500	0.90800	± 0.01791
1000	0.90800	± 0.01791
4000	0.90100	± 0.01851
8000	0.89400	± 0.01908
16000	0.90300	± 0.01834
32000	0.89400	± 0.01908

Table 3: Cheating MCTS vs RandomWalk (confidence intervals at 95%)

Det	Depth	W.Rate	Conf.
100	2	0.82250	± 0.02648
100	4	0.81650	± 0.02684
100	6	0.82125	± 0.02655
100	8	0.82000	± 0.02662
100	10	0.82500	± 0.02633
100	12	0.81375	± 0.02698

Table 4: Determinized AlphaBeta vs RandomWalk (confidence intervals at 95%)

5.2 Partial Information AI Performance

5.2.1 Against RandomWalk

Determinized AlphaBeta has a win rate against the random player exceeding 80%. The results are presented in table 4. Determinized MCTS instead is not as successful, beating RandWalk in 70% of cases on average. Results in Table 5. Apparently the depth and the iteration number does not affect so much the performance. This is something we noticed also in the cheating version.

Det	Iter.	W.Rate	Conf.
100	500	0.72500	± 0.04376
100	700	0.69500	± 0.04512
100	1000	0.70750	± 0.04458
100	1200	0.70000	± 0.04491
100	1500	0.71500	± 0.04424
100	2000	0.70750	± 0.04458
100	3000	0.72000	± 0.04400
100	4000	0.70000	± 0.04491

Table 5: Determinized MCTS vs RandomWalk (confidence intervals at 95%)

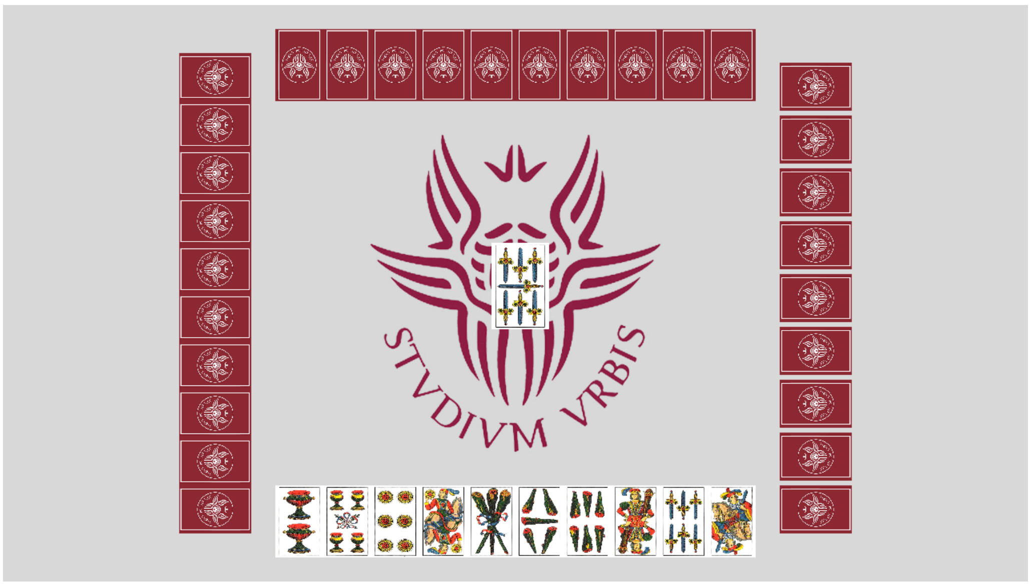

Figure 1: The Graphic User Interface we developed for tests against Humans

5.2.2 Against Cheating AI

Since Cheating AlphaBeta outperformed Cheating MCTS, we are going to use the former as a benchmark.

Table 6 shows the performance of Determinized Alpha-Beta against the Cheating one. Table 7 shows the performance of Determinized MCTS against Cheating Alpha-Beta. We observe how the winning rate of both Determinized Algorithms does not seem to depend on the determinization number.

We note a difference between the two algorithm's performances: AlphaBeta is in the vicinity of 20%, while MCTS achieves around 15%. It appears that AlphaBeta behaves better. MCTS could likely be overlooking some winning moves, while Alpha-Beta doesn't.

The determinized algorithms are far from being competitive against a cheating AI. We don't have clues about the competivity against human players, but we believe that the Flat Monte-Carlo determinization is not accurate enough for such purpose.

Det	Win Rate	Conf.
5	0.20000	± 0.02479
20	0.20100	± 0.02484
50	0.19800	± 0.02470
80	0.20500	± 0.02502
100	0.21200	± 0.02533
200	0.20800	± 0.02516
500	0.19000	± 0.02431
1000	0.22800	± 0.02600

Table 6: Determinized Alpha-Beta (depth 10) vs Cheating Alpha-Beta (depth 10). Confidence intervals at 95%.

Det	Win Rate	Conf.
5	0.14923	± 0.01937
20	0.16077	± 0.01997
50	0.13462	± 0.01855
80	0.14692	± 0.01924
100	0.13385	± 0.01851
200	0.13000	± 0.02084
500	0.14200	± 0.02163

Table 7: Determinized MCTS (it. 1000) vs Cheating Alpha-Beta (depth 10). Confidence intervals at 95%.

6 Future Works

The main weakness of our algorithms is surely the determinization. Skilled players observe the behaviour of the others and infere what their behaviour means. We can do a similar thing by studying the behaviour of other players and then building a Belief Network. Since a human games dataset for tresette is missing, we could learn it by playing against cheating AIs.

7 Conclusions

In the game of *Tresette*, the initial assignment is crucially important. A poor set of cards causes even cheating AIs to lose against RandomWalk.

In all cases we looked into, Alpha Beta outperforms MCTS. In cheating algorithms, there is a slight advantage. It becomes more significant when we deal with determinized algorithms. We attribute this to the low branching factor of the game and to the monotonic behaviour of point assignment.

Both Determinized AIs win by far against the RandomWalk player, but they never achieve Cheating AIs' performances. Against Cheating Algorithms, they win in 20% and 15% of the cases, for AlphaBeta and MCTS respectively.

References

[1] Selim G Akl and Monroe M Newborn. The principal continuation and the killer heuristic. In *Proceedings of the 1977 annual conference*, pages 466–473. ACM, 1977.

[2] 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. *IEEE Transactions on Computational Intelligence and AI in games*, 4 (1):1–43, 2012.

[3] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

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

[5] Stefano Di Palma. Monte carlo tree search algorithms applied to the card game scopone, December 2014. http://hdl.handle.net/10589/102246.

[6] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[7] Raphael A Finkel and John P Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19(1):89–106, 1982.

[8] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[9] Rodolfo Pucino. *Il tressette nei tempi moderni e secondo le nuove tecniche.* Guida, 2005. ISBN 9788871889085.

[10] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited, 2016.

[11] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE transactions on pattern analysis and machine intelligence*, 11(11):1203–1212, 1989.

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[13] Treccani. Tressette. http://www.treccani.it/enciclopedia/tressette. Accessed on 2018-09-21.