Universität Hamburg
Department Informatik
Knowledge Technology, WTM

Mit welchen Spielbaum-Algorithmen aus der Spieltheorie lassen sich gute Gegner fuer klassische Spiele entwickeln?

Proseminararbeit

Proseminar Kuenstliche Intelligenz

Michal Zlotnik
Matr.Nr. 7054053
michal.zlotnik@studium.uni-hamburg.de

11.02.2019

Abstract

Combinatorial games are subject of ongoing large-scale research in the field of artificial intelligence. In this work, basic concepts and algorithms from combinatorial game theory are presented to aid a design of an opponent that would play optimally.

Contents

1 Introduction

1.1 Motivation

Playing games yields a question on how to play optimally, i.e. maximize the chance of winning. Combinatorial games require analysis regarding next moves and their possible outcomes. Ultimate goal is to outsmart an opponent by ensuring the optimal strategy. Board games with sequential moves provide a fully observable environment that allows a thorough field for their analysis. Rules are strictly defined which enables, to a certain degree, stable environment. Zero-sum games of perfect information where no chance moves are allowed serve as an introduction to more advanced subjects, for instance modern approach utilized by AlphaZero or stochastic games.

1.2 Scope

This work primarily focuses on introductory aspects of game theory, as well as its sub-branch called combinatorial game theory, and most commonly used algorithms. In the succeeding chapter relevant terms are explained. The main part presents methods that are commonly used in the classic games like tic-tac-toe, draughts and chess. Finally, it provides a brief information on the most known computer playing programs.

2 Background

2.1 General information on game theory

Game theory is a formal study of mathematical models of interactions between rational decision-making agents whose choices might posses a potential to affect each other interests. Developed in the 20th century, independently by mathematicians Zermelo and Von Neumann, laid foundations for a scientific investigation of logical decision making in both humans and computers.[1] Furthermore, it has been ever since widely recognized as an important tool in many scientific fields with applications ranging from economics to biology.

In the two subsequent sub-sections main focus is placed exclusively on games types and their representation relevant to choice of classic games, i.e. tic-tac-toe, draughts, and chess.

2.2 Combinatorial games: tic-tac-toe, draughts, chess

Subjective choice of classic games was based on their increasing computational complexity as well as it was influenced by their familiarity to the general audience. This will allow to omit a comprehensive introductory section regarding their governing rules. It is worth mentioning, that, apart from tic-tac-toe, due to their hardness they were recently subject of a extensive research.

Tic-tac-toe is commonly described as a paper-and-pencil, where two players take turns marking fields on a 3x3 grid with noughts and crosses. The first one who manages to obtain three marked spaces horizontally, vertically or diagonally wins a game. Assuming, that X makes the first move and taking into account rotations and reflections, there are: 91 distinct positions won by X, 44 distinct positions won by O and 3 distinct positions are drawn[2]. In total, there are 255,168 unique games[3].

Draughts is a strategy NxN-board game ($N \in \mathbb{N}$). Two players on opposite sides of the board alternate turns and move their pieces, when not capturing, one step diagonally forwards. Pieces are categorized into men and kings. Once reaching the opposite side of the board a man becomes a king and is allowed to move any distance diagonally backwards and forwards provided, that a diagonal is not blocked. If a neighbouring square is occupied by opponent's piece and the one behind is vacant, a capture is mandatory. It involves jumping over opponent's piece. Multiple captures are allowed.

Chess is one of the most widely known as well as researched two-player strategy board game. It is played on a board containing 64 squares which are arranged in an 8x8 grid. Degree of hardness of computational complexity could be illustrated by two estimates 5.3×10^{79} and 10^{120}. Former being the estimated number of atoms in the observable universe[4], the latter is Shannon number which is the lower bound of the game-tree-complexity, i.e. the number of leaf nodes in a decision tree that constitutes a value of an initial position[5].

2.3 Games types

Games are categorized into two main branches: non- and cooperative. The former models the cases where players are either not allowed or unable to communicate and form alliances. As a result, an analysis of non-cooperative games is related to players' strategic choices, their timing and payoffs that determine a game's outcome.[1]

Zero-sum is a game where the sum of payoffs remains constant, which means that participants' choices may neither increase nor decrease available resources.[6] For games under consideration, it means that a final outcome for any player might be winning, losing or reaching a tie.

Sequentiality and perfect information are related concepts that combine definitions and restrictions regarding choices of actions. Sequential games are games where players are not allowed to move simultaneously. Hence, as their decision might be determined by chosen strategy and current situation they must possess information about all decisions (or moves) that were previously made.

Combinatorial games (shortened as CGT) depict games of high complexity with a large number of possible moves, strategies and possible outcomes. It is said to be a relatively a new field with many areas in need of further research. Furthermore, the concept and its approach are significantly different to the ones introduced by Von Neumann, Morgenstern and Zermelo.[7] In addition, they must meet the following constraints:[7][8]

- exactly two players (”Left” and ”Right”) who alternate their moves
- ensured perfect information (all previous and possible moves are known to both players)
- chance moves are prohibited
- a play will always come to an end and the last move establishes the winner

2.4 Game tree

Previously mentioned sequential games are most suitably represented by game trees. Most commonly, they are pictured as directed graphs, where edges display moves and nodes depict positions. Such representation is often referred to as an extensive form. A complete game tree contains an initial state as well as all possible moves from each position. Due to the complexity of classic games discussed here, partial game trees will be utilized as complete game trees would be too vast to be searched. Partial trees are defined as an integral part of a complete tree. However, they are evaluated to a certain depth.[9]

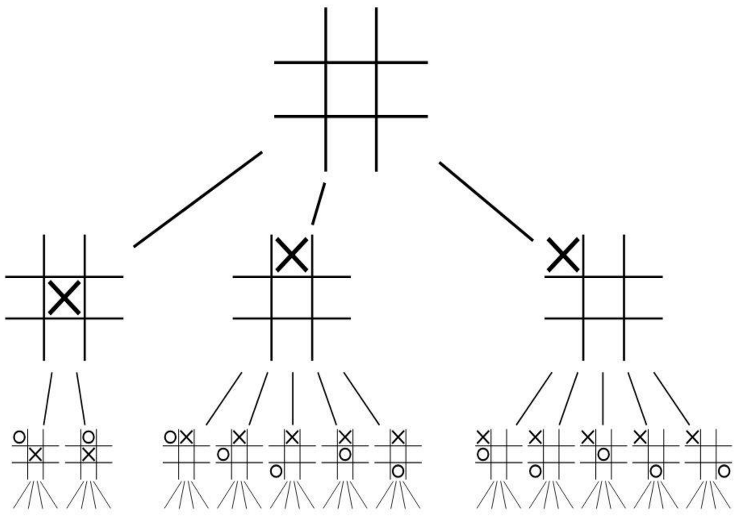


Figure 1: Game tree for tic-tac-toe representing first two moves.

2.5 Solved game

In combinatorial games a solved game is one whose outcome could be correctly determined from any position. Taking into account the constraints listed in the subsection 2.3, there exist three levels of solving a game: ultra-weakly solved, weakly solved, strongly solved[10].

Ultra-weak solution assumes that the outcome of a game is well-established, however there exists no proof for a winning strategy. The outcome of the game is known for both players as well as an algorithm is provided that proves a win for one player or a draw. Both outcomes must assume any possible moves made by an opponent[11]. Strong solution incorporates a known algorithm that determines the outcome of the perfect-play strategy and possible mistakes for all moves performed.

It is worth mentioning that of the classic games listed in the subsection 2.2 tic-tac-toe is strongly solved due to its relatively small game tree. Draughts was weakly solved in 2007, i.e. it was shown that a perfect play on both sides leads to a draw[11]. As a result of an immense game-tree complexity of chess, as mentioned in the section 2.2, it is questionable whether the game of chess will ever be solved.

3 Methods and approaches

Designing a good opponent for a chosen range of classic games requires investigation of a series of concepts[9]:

- Initial states and, for the final, i.e. terminal ones, providing terminal test
- Utility function in terminal states
- Min(-imizing) and Max(-imizing) players
- Utilization of algorithms for an optimal move search and decision
- Evaluation function
- Cutting off search and horizon effect

Initial and terminal states yield information on the starting arrangement and the final outcome, respectively.

Binary **terminal test** function evaluates to $true$ or $false$ depending on whether the game comes to an end.

Utility function provides a value for the end result of the game for a player p in the terminal state s. For instance, 1 could be the result of player A winning, -1 losing and 0 drawing. However, upon initial lower and upper bounds are commonly

$$utilityFunction(p, s) = result(\in \mathbb{Z})$$

Given the utility function's result, it could be proposed that a player **Max** performs the first move and aims at reaching a leaf with the highest possible value.

Similarly, a player **Min** makes the second move and aims at the reaching a terminal state with a lowest possible value.

Minimax value aids establishing the optimal strategy. It yields a value for each node in the game tree. Figure 2 presents an example of a minimax value evaluation. Node A is a max node, nodes B, C and D are min nodes. Leafs depict values for terminal states. Max prefers to move to a node with maximal value, whereas Min with minimal.

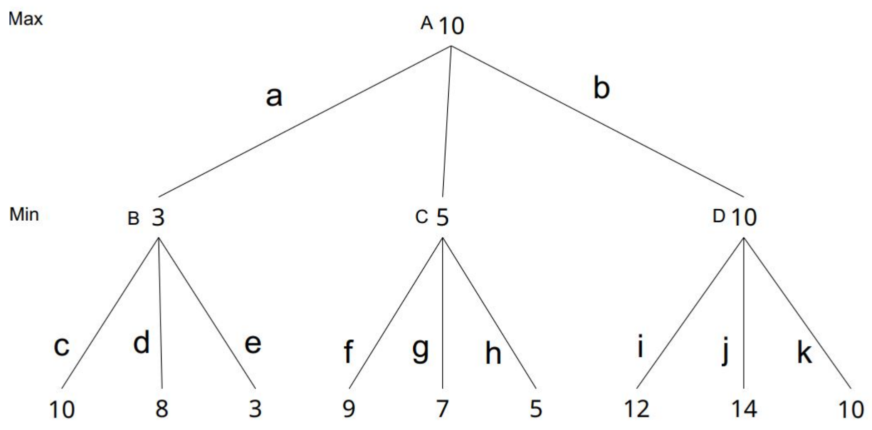


Figure 2: Minimax value evaluation.

```
minimaxValue(state)
    if terminalTest(state)
        utilityFunction(state)
    if currentPlayer(state) == Max
        max(minimaxValue(result(state, action)))
        // "result" yields successor state resulting from
        // performing action "a" in state "s"
    if currentPlayer(state) == Min
        min(minimaxValue(result(state, action)))
```

Sample computation based on figure 2:

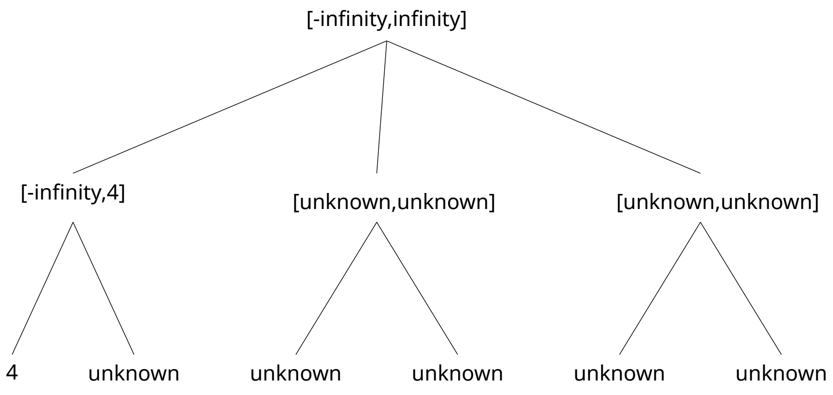
$$\begin{aligned} minimaxValue(A) &= \\ &= max(min(10,8,3), min(9,7,5), min(12,14,10)) \\ &= max(3,5,10) \\ &= 10 \end{aligned}$$

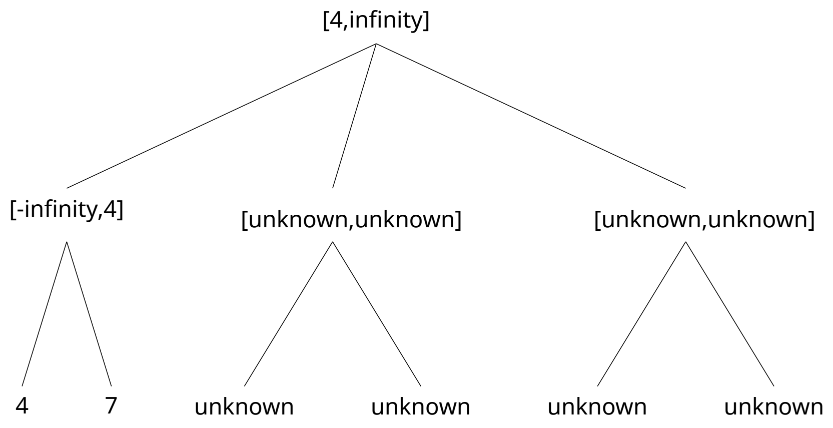
Minimax algorithm yields a decision for the best move from a given state for either a minimizing or maximizing player. It searches a game tree down to the terminal notes. After applying utility function it backs up the corresponding state with an utility value starting recursive process.

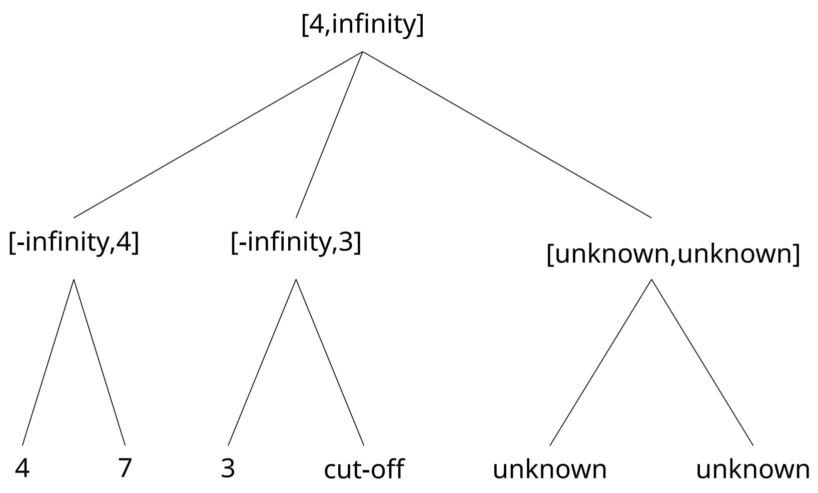
```
minimaxAlgorithm(state)
    maxValue(state)
        if terminalTest(state)
            return utilityFunction(state)
        bestMove = -infinity
        for each action in setOfActions
            bestMove = max(bestMove,
            minValue(result(state, action))
        return bestMove
    minValue(state)
        if terminalTest(state)
            return utilityFunction(state)
        bestMove = infinity
        for each action in setOfActions
            bestMove = min(bestMove,
            maxValue(result(state,action))
        return bestMove
    return decision
```

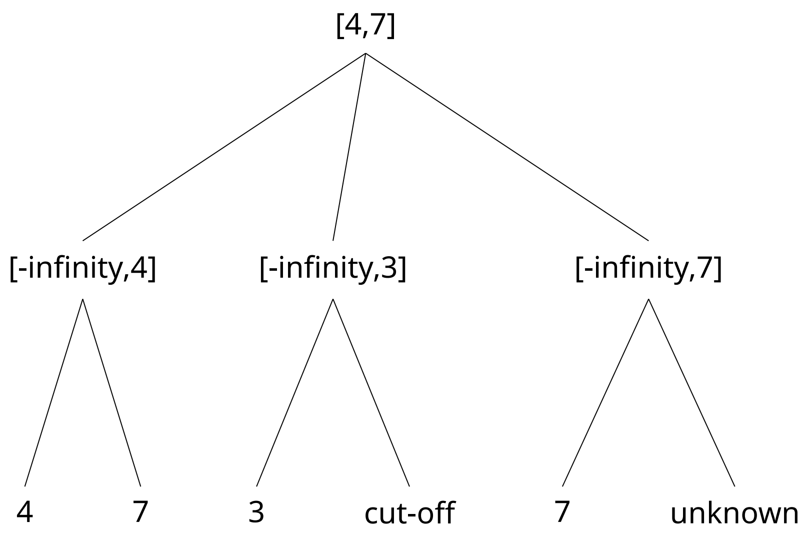
Alpha-beta search algorithm provides a similar technique for decision-making as minimax algorithm However, it utilizes pruning that allows cut off branches of the tree that cannot influence the decision. There two values that are used when conducting a search. Alpha (α) is the highest value found on the path for maximizing player and beta (β) being the lowest value found for the minimizing player. Both get updated during the search.

```
alphabeta(state)
    bestMove = maxValue(state, -infinity, infinity)
    maxValue(state, alpha, beta)
        if terminalTest(state)
            return utilityFunction(state)
        bestMove = infinity
        for each action in setOfActions
            bestMove = max(bestMove,
            minValue(result(state,action),alpha,beta))
            if bestMove >= beta
                return bestMove
            alpha = max(alpha, bestMove)
        return bestMove
    minValue(state, alpha, beta)
        if terminalTest(state)
            return utilityFunction(state)
        bestMove = -infinity
        for each action in setOfActions
            bestMove = min(bestMove,
            maxValue(result(state,action),alpha,beta))
            if bestMove <= beta
                return bestMove
            beta = min(beta, bestMove)
        return bestMove
    return decision
```






[4,infinity]
[-infinity,4]
[-infinity,3]
[unknown,unknown]
4
7
3
cut-off
unknown
unknown


[4,7]
[-infinity,4]
[-infinity,3]
[-infinity,7]
4
7
3
cut-off
7
unknown

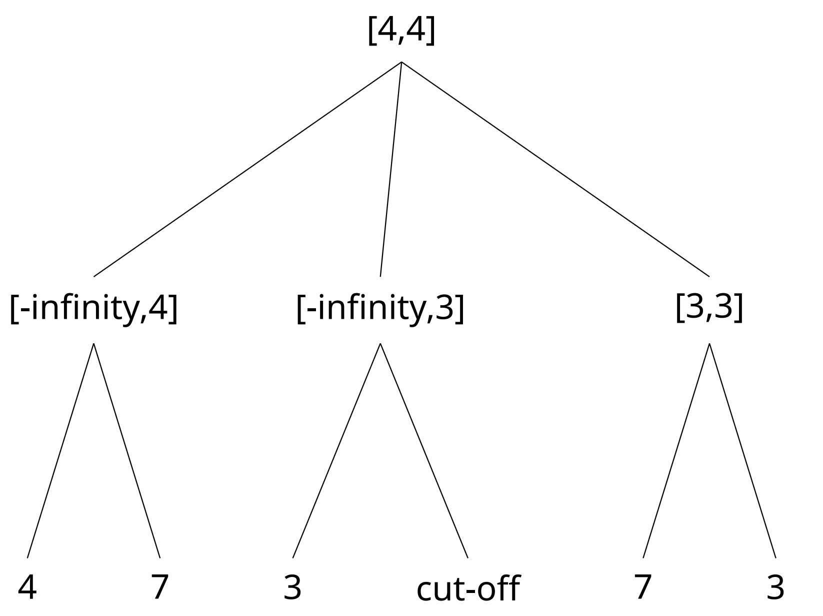


Figure 3: Steps for calculating a decision using alpha-beta algorithm.

Evaluation function is a numerical estimator that provides the expected outcome of the game in the current position. States that provide a win must be evaluated to a higher than the ones for draws and losses. It can be expressed as a weighted linear function.

$$\sum_{i=1}^{n} w_i f_i(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$$

Cutting off search helps reducing the search by setting a limit to a depth. Test for cutting off can be applied in order to enable calling the evaluation function. It also must evaluate to *true* for terminal states. Applying it to alpha-beta algorithm yields:

```
alphabeta(state)
    bestMove = maxValue(state, -infinity, infinity)
    maxValue(state, alpha, beta)
        if cuttOff(state, depth)
            return evaluationFunction(state)
        bestMove = infinity
        for each action in setOfActions
            bestMove = max(bestMove,
            minValue(result(state,action),alpha,beta))
            if bestMove >= beta
                return bestMove
            alpha = max(alpha, bestMove)
        return bestMove
    minValue(state, alpha, beta)
        if cuttOff(state, depth)
            return evaluationFunction(state)
        bestMove = -infinity
        for each action in setOfActions
            bestMove = min(bestMove,
            maxValue(result(state,action),alpha,beta))
            if bestMove <= beta
                return bestMove
            beta = min(beta, bestMove)
        return bestMove
    return decision
```

Horizon effect occurs due to the limited search of a game tree. Algorithms, that return an optimal move to a certain depth, might be unaware of existence of a state that is non-optimal and located beyond the set depth limit. In order to mitigate this effect, quiescence search can be applied. It involves extending the search depth in states prior to which high fluctuations in the results of evaluation functions were detected.

Monte Carlo method can applied to the game tree in order to implement decision-making algorithm. It is a probabilistic search algorithm and consists of four stages: selection, expansion, simulation and back-propagation. Selection starts at a root node and proceeds on choosing a child node with a maximum win rate. One of methods used to deduct a child node is Upper Confidence Bound formula. It ensures that optimal branches are played more often than other. Expansion commences when Upper Confidence Bound can be no longr applied to determine child nodes. All possible states are appended from a leaf node and simulation proceeds next. It includes selecting an arbitrary node (state) of a game and carries on simulating a randomized game until its outcome is reached. Back-propagation starts with determining which player won the simulated game. It traverses upward from a terminal state backing up subsequent states visit and win scores if the game was won for a player from that state. Algorithm requires a knowledge on rules governing allowed moves and winning, losing as well as drawing. However, it does not need any further information like numerical values obtained from evaluation function. Its execution can be stopped at any time and it will still be capable of suggesting the succeeding optimal move.

Tic-tac-toe. Based on presented methods, an optimally strategy for tic-tac-toe opponent could be develop. Due to game's relatively low complexity implementation might utilize minimax search algorithm backed by evaluation function. Computation has to evaluate each eight lines on a board: three horizontal, three vertical and two diagonal. Its components might contain:

- negative scores for an opponent
- positive scores for a current player

- +1000 for each three crosses or naughts in line
- +100 for each two symbols in line (provided that the other cell is not occupied)
- +10 for one symbol in line (provided that the other two cells are not occupied)
- 0 otherwise

Draughts. Due to its vast game tree branching alpha-beta search is a more optimal design choice. Furthermore, instead of evaluating a full path that leads to a winning outcome, decisions are made that might lead to one. Evaluation function might be implemented separately for two stages: open-to-middle-game and ending-moves. Former includes calculating values of pieces by assigning higher to kings than men. It also might consider that men that are closer to opponent's side, i.e. might become kings, are more threatening. Hence, men on the opponent's part of the board should be taken into consideration as well. Evaluation function that analyses the end stages might include comparing the number of men as well as kings that are still played by both players. As kings are more powerful, evaluation function can compute a distance between kings and opponent's pieces, taking into account that player in possession of king would like to minimize it, the other maximize, i.e. to avoid a capture. Method that establishes the transition between the two stages can be evaluated by counting the remaining pieces on board.

Chinook is the most widely known draughts playing program. It famous for being the draughts engine that won the world champion title in a competition against human players. Algorithm utilizes alpha-beta search combined with a library of opening and ending moves played by draughts grandmasters. Furthermore, it was programmed by its creators without the use of methods like reinforcement learning.

Chess. Its large complexity causes a need for a more thorough approach in order to design the optimally playing opponent. Similarly to draughts, alpha-beta search might be used and evaluation function could be broken down to opening and ending stages. Furthermore, library containing opening and ending moves would allow further optimization. To improve a simple evaluation function, piece-square tables could be applied. They contain tables for each piece of each color with values assigned to each square on the chess board.

The first chess-playing program to win a game and a match with a reigning world champion (Garry Kasparov) was Deep Blue developed by IBM. It utilized alpha-beta search and relied on its large computational power being capable of evaluating 200 million positions per second. On average it analyzed the game tree to a depth of between six to eight moves. Evaluation functions were based on a large number of grandmaster games.

Stockfish is a renown chess engine and considered to be the strongest open-source engine in the world. Initially released in 2008 it won a couple of unofficial world computer chess championships. It utilizes highly optimized alpha-beta search (with a significantly great search depth limit) and a data structure called bitboard for piece-square tables.

AlphaZero is a computer playing program developed by Alphabet. It uses Monte Carlo method backed by neural networks. After 24 hours of self-play it reached a superhuman performance. Furthermore, in 2018 in a 1,000-game match it managed to defeat Stockfish 9 with astounding results. Out of 1,000 games, 155 were won by AlphaZero and 6 by Stockfish [12].

4 Conclusion

The aim of this work to present an overview of the introductory aspects of combinatorial game theory as well as commonly used methods for opponent design. In games with a relatively small search tree the minimax algorithm seems to be a sufficient approach. With an increasing game tree complexity, it is computationally insufficient to perform the search all the way down to the terminal states. Hence, alpha-beta search with a thorough cut-off test could be used to minimize time required for decision making process regarding choosing an optimal move.

Modern approach, like the one used by AlphaZero chess program, seems to constitute a field for future work and research. Instead of relying on a large number of simulations at any stage of game, research focus could be placed on improving methods that would contribute to the increase the accuracy and efficiency self-play. Another great strength of self-learning is its potential versatility to a range of combinatorial games.

References

[1] G. Bonnano. *Game Theory.* o.V., 2018.

[2] T. Bolon. *How to never lose at Tic-Tac-Toe.* Book Country, 2013.

[3] `http://www.half-real.net/tictactoe/`. Accessed: 2019-02-07.

[4] Peter AR Ade, N Aghanim, M Arnaud, M Ashdown, J Aumont, Carlo Baccigalupi, AJ Banday, RB Barreiro, JG Bartlett, N Bartolo, et al. Planck 2015 results-xiii. cosmological parameters. *Astronomy & Astrophysics*, 594:A13, 2016.

[5] Claude E Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer, 1988.

[6] Theodore L. Turocy and Bernhard von Stengel. Game theory. In Hossein Bidgoli, editor, *Encyclopedia of Information Systems*, pages 403 – 420. Elsevier, New York, 2003.

[7] Michael Albert, Richard Nowakowski, and David Wolfe. *Lessons in Play: An Introduction to Combinatorial Game Theory, Second Edition.* A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2016.

[8] Eric Duchêne. Combinatorial games: From theoretical solving to ai algorithms. In Steven Schockaert and Pierre Senellart, editors, *Scalable Uncertainty Management*, pages 3–17, Cham, 2016. Springer International Publishing.

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[10] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence.* Ponsen & Looijen Wageningen, 1994.

[11] Jonathan Schaeffer. Game over: Black to play and draw in checkers. *ICGA Journal*, 30(4):187–197, 2007.

[12] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.