Adversarial Search problem on the example of FAUhalma

February 21, 2024

Abstract

People have always been curious about the applications of AI in board games [7] [9]. This report gives an overview of different approaches (from basic move strategies without applying heuristics to the application of search algorithms and heuristics) to solve the problem of evaluating the best moves to beat the virtual opponent online, using the FAUhalma game, which has different difficulty levels (strength of the opponent), (see Section 5) as an example. [10]

1 Introduction

Since Deep Blue defeated Garry Kasparov in 1997 [9] interest in the application of AI has continued to grow and more and more algorithms or combinations of algorithms have appeared, e.g. AlphaStar, AlphaGo, etc. [7] Although since 1960s chess is considered the "Drosophila of AI", chess and other board games are not so easy to solve due to a large number of possible game states. [7] Only for Chinese Checkers it is around 1.73×10^{24} [12]. This requires an approach where we only evaluate the best possible moves for a future game state, taking into account the current state of the game. At the same time, we want to maximize our gain and minimize the opponent's gain. [10] To solve the above problem, various adversarial search algorithms such as minimax, alpha-beta search, and Monte Carlo tree search can be applied. These algorithms are designed in such a way that they can be adapted to different adversarial search problems depending on the defined utility. [7]

1.1 Research question

How adversarial search algorithms could be applied to the set of rules of FAUhalma game to find the best combination of moves to win over the opponent all the time?

1.2 Contribution

There are two key contributions to this report:

1. Finding an optimal representation of the non-rectangular grid for FAUhalma
2. Evaluating various strategies for finding the best move to maximize the win of the chosen player.

1.3 Overview

This paper consists of 6 parts. In the next section (2), we discuss the necessary preliminaries and related work that are crucial for understanding the next parts of the paper. Section 3 presents the problem description and the rules of the FAUhalma game. The following fourth section deals with different approaches to accomplishing the task and represents the main part with 4 subsections. Section 5 then presents the evaluation and comparison of the different methods. Finally, section 6 concludes the report.

2 Preliminaries

Terms *Adversarial*, *minimax*, *alpha beta search* and *evaluation function* are important for further understanding of the work. *Monte Carlo Tree Search* extends the overview of available methods for solving the research problem but was not implemented in the main part section.

Definition 2.1 (**Adversarial search**)**.** This is a computational approach in Artificial Intelligence and Game Theory, where the goal is to find an optimal strategy to decide on the next move. Decisions are made by two players (adversaries), who play the game. [3] Adversarial search problems are often known as games [9]

Definition 2.2 (**Minimax search**)**.** A canonical algorithm for determining the optimal strategy in games is the minimax algorithm. In this algorithm, we denote the opponent as "Min" and our player as "Max" (because we always want to win the opponent). We perform a depth-first search in a game tree with "Max" as the maximizing player at the root node. Next, we apply a utility function to the final positions and perform a bottom-up evaluation for each interior node in the game tree. At each level of the tree, we determine whether it is Min's or Max's turn to minimize or maximize the utility values propagated from the successor nodes. Finally, such a move is returned which leads to the maximum utility value for the Max player. [7]

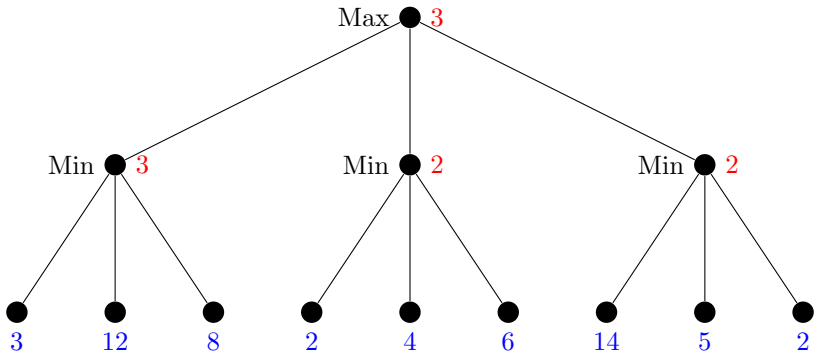


Figure 1: Example of minimax from the lecture notes [7]

Definition 2.3 (**Alpha beta search**)**.** An improved version of minimax search, which reduces the search tree by 1/2 in the best case. [9] While traversing the game tree we always keep two values, α (the best value found so far for the maximizing player) and β (the best value found so far for the minimizing player). Before evaluating the states α is initialized as $-\infty$ while β has an initial value of $+\infty$ If it is determined during the run that the value of a node does not influence the final result (In "Min" node if one of the successors has utility $\leq \alpha$ or in "Max" node if one of the successor nodes has utility $\geq \beta$), it is pruned (not evaluated further). [7]

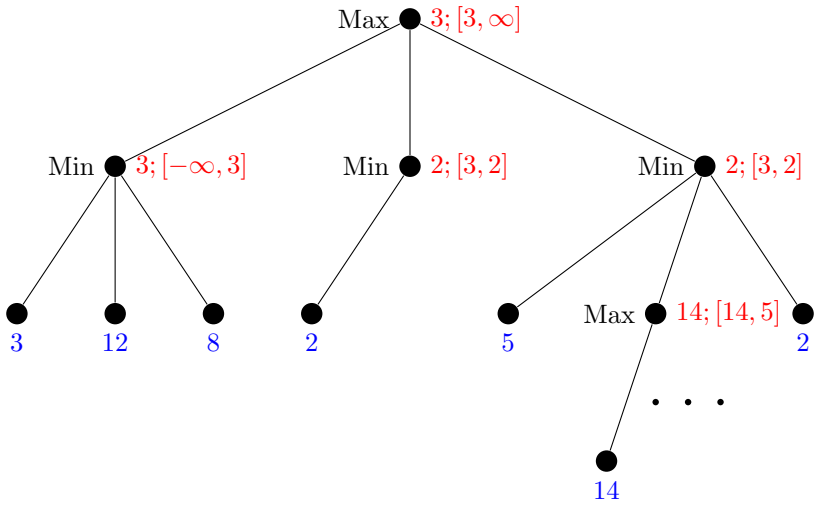


Figure 2: Example of alpha-beta pruning from the lecture notes [7]

Definition 2.4 (**Monte Carlo Tree Search (MCTS)**)**.** This algorithm uses a clever tree search algorithm that carefully balances the exploration of new opportunities with the use of known information. By running random simulations and tracking action statistics, MCTS makes increasingly informed decisions in each iteration. It has proven to be an innovative technique that is particularly effective in combinatorial games. [13] It has 4 important steps: 1) Selection (guided by UCT or UCB), 2)Expansion, 3) Simulation, and 4) Backpropagation which are repeated until the stopping criteria are reached. [6] [7] Authors of the famous AlphaGo have combined MCTS with Neural Networks which produced even better results. [11]

Definition 2.5 (**Evaluation function**)**.** An evaluation function provides an estimate of the expected benefit of the game from a particular position. [9]

3 Problem Description

3.1 Set of game rules

FAUhalma, inspired by the traditional game Star Halma/Chinese Checkers, is suitable for 2 to 3 players. The game board, which usually has a star shape can also be simplified to a rhombus for the sake of simplicity (Figure 4). Each player starts with several checkers in the corners assigned to them (2 corners for 2 players (Figure 5) and 3 corners for 3 players respectively (Figure 6)). The aim of the game is for the players to quickly move their pieces to the opposite corner, which is called "home". The players take turns in an anti-clockwise direction. During their turn, a player has the option of either moving one of their pieces (can be also called *pegs* or *marbles* [4]) to an adjacent unoccupied square or jumping over other pieces (hopping). [10] [12]

Definition 3.1 (**Move**)**.** Any displacement of a marble on the board from the initial position to the n position. [4]

- **Simple move**: a peg can be moved to any adjacent free node. [10]
- **Simple hop**: If there is a peg on an adjacent square, it can "jump over" it if the node behind it is free. [10]
- **Hop chain**: A sequence of consecutive single hops that allows a peg to jump from one node to the next as long as each hop is valid. However, the peg must land on a different square at the end of the chain than at the beginning. **Simple moves cannot be combined with jumps.** [10]
- **Swap rule**: Spaces in the home territory of the moving player on which an opponent's peg is located can be treated as empty for the above rules (except for intermediate spaces within a jump chain). If a peg is moved from S1 to S2 while S2 is occupied by an opponent's peg, the peg is moved from S2 to S1, which means that the pegs are swapped. This rule prevents players from obstructing each other by placing pegs in their opponents' homes. [10]

If there are no valid moves for a player, then the player **loses** the game. The first **winner** is the player, who moves all its pegs into the opposite corner first. The remaining players compete for second place. [10] [1]

3.2 Coordinate system representation

Due to the non-rectangular geometry of the game board [10], I decided to use the 3-coordinates system for directions storing the board positions as nodes of the graph data structure.

3.3 Move representation

Finally, moves are represented using the following convention:

$$[[x_{n-1}, y_{n-1}], [x_n, y_n]]$$

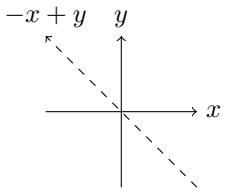


Figure 3: Coordinate system of the positions on the board

. For example, for a single move or single hop we would have

$$[[x_0, y_0], [x_1, y_1]]$$

while for a chain of hops

$$[[x_0, y_0], [x_1, y_1], [x_2, y_2], [x_3, y_3]]$$

[10]

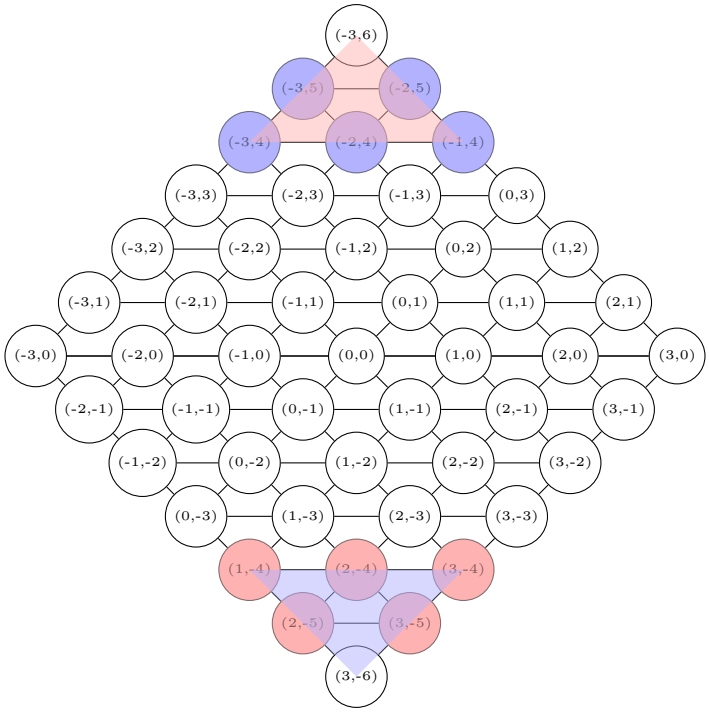


Figure 4: The starting positions of the players for a simplified board variant (rhombus)

4 Main Part

In the following section, we will discuss the translation of the problem into a search graph and different approaches to determine the best strategy for the next move.

4.1 Translating of the problem into a search graph

Many researchers often simplify the star-shaped board to the size of a square 16x16 board or smaller [2] [12] [14], so that even a table can be used as a data structure or a hash list for displaying the board and storing the position. Since we want to have a star-shaped non-rectangular board, the graph representation seemed to me to be the simplest and most practical for several reasons [8]:

1. Graphs enable data visualization, which is always easy to read and understand

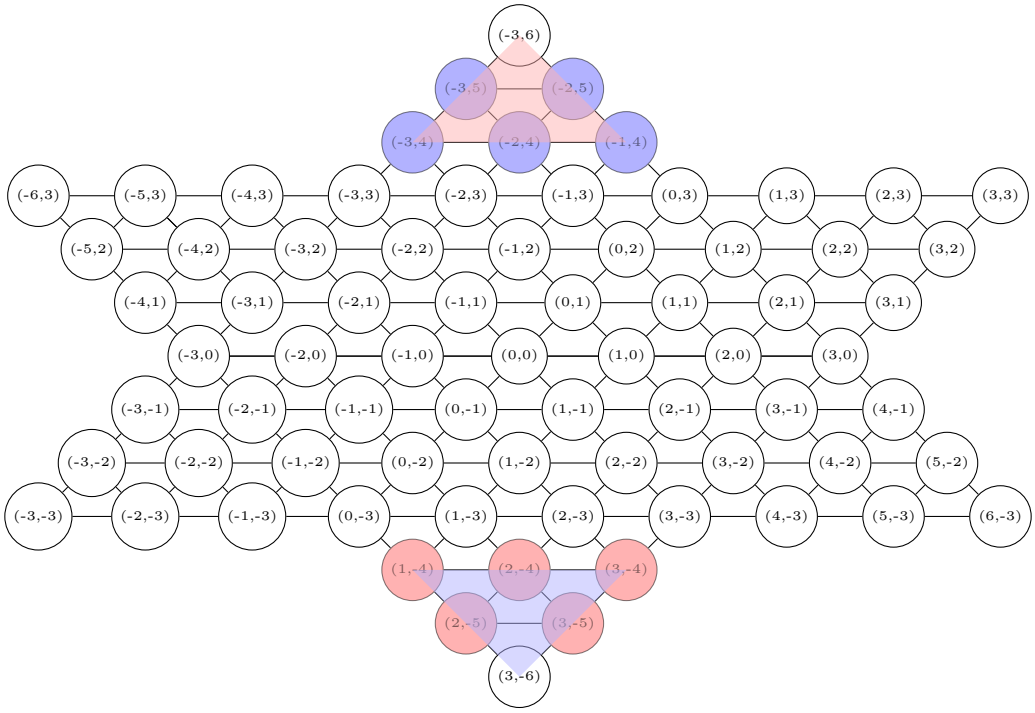


Figure 5: The starting positions of 2 players (simplified scenario) for a star-shaped board

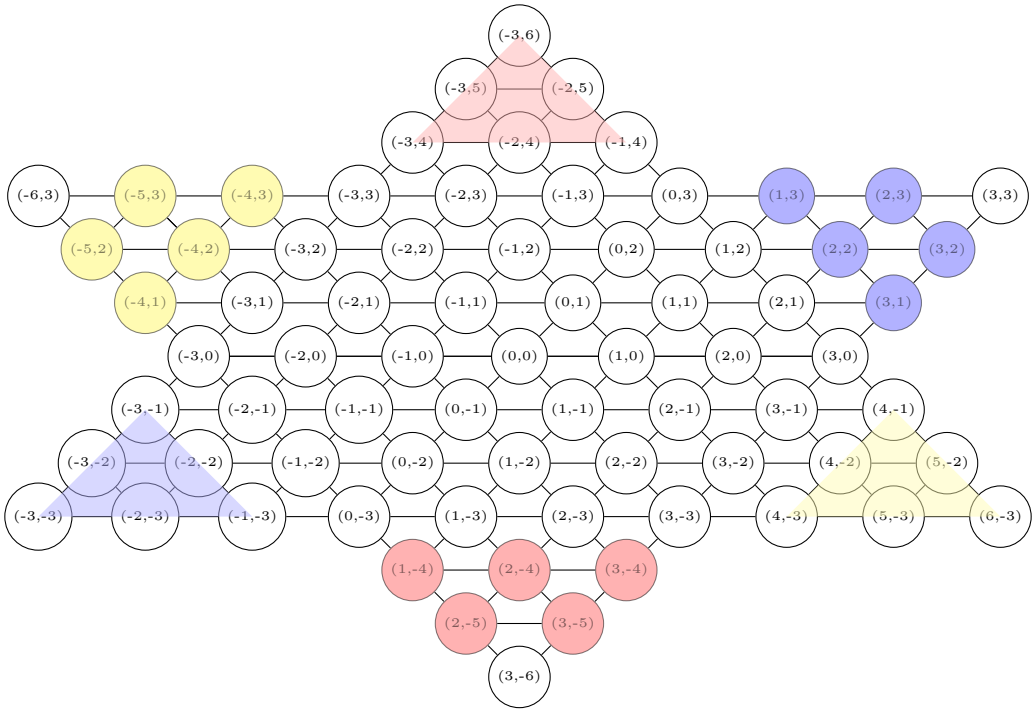


Figure 6: The starting positions of 3 players for a star-shaped board

2. Networkx library in Python offers a lot of methods for operating on graphs (number of nodes, edges, storing attributes, get adjacent neighbors or get the shortest path from n-1 to n which is relevant for calculating the heuristics, and so on) [5].

3. In graphs it is easier to manage non-rectangular structures by defining edges between nodes or when searching for a route we can always check if the edge exists.

Every node represents a current position on the board and it has connections (*edges*) to all adjacent nodes around (*to find allowed moves later*). The node can store the attributes, which can be used for example to indicate whether the node is occupied or free. To illustrate it better, I have attached a code for a function in Python, which places the marbles on the board before starting the game.

Listing 1: def place_pegs() function

```
    def place_pegs(self, peg_positions):
        for peg, positions in peg_positions.items():
            for pos in positions:
                if tuple(pos) in self.G.nodes():
                    self.G.nodes[tuple(pos)]['peg'] = peg
                else:
                    print(f"Position {pos} is not on the board.")
```

4.2 Simple strategies

In this subsection, I am going to present simple strategies, that do not require any search algorithm, but prioritize movements by the desired direction or using a simple heuristic *(distance from the target)*, which can be computed using different norms [2].

For the first strategy, I use the function below, which looks for simple moves to neighboring nodes that are not occupied by a peg, or for simple hops behind adjacent nodes (function *update_moves*). It is important to emphasize, that for strategy no.1 we do not search for hop chains in comparison to strategy no.2 where we do evaluate hop chains as well. In addition, the function *is_valid_move* checks whether the new position remains within the board after the move, and $PROHIBITED\_COORDINATES\_BORDER$ is only necessary for strategy no. 2 to prevent navigating into the corners of the star-shaped board. For both simple strategies, we assess only future moves for our player (no assessment of future opponent's moves).

Listing 2: def find_valid_moves_for_A function

```
        def find_valid_moves_for_A(self,board):
        peg_positions = nx.get_node_attributes(board.G, 'peg')
        peg_positions_A = {pos: peg for pos, peg in peg_positions.items() if
            peg == 'A'}
        valid_moves = []
        hops = []

        for start in peg_positions_A:
            neighbors = list(board.G.neighbors(start))
            for end in neighbors:
                if Player.is_valid_move(start, end, board):
                    valid_moves.append([start, end])

        filtered_moves = [move for move in valid_moves if move[1] not in
            PROHIBITED_COORDINATES_BORDER]

        filtered_moves,hops = self.update_moves(board,filtered_moves,hops)

        return filtered_moves, hops
```

4.2.1 Strategy no. 1

This strategy was primarily designed to compete only with one opponent on the rhombus-shaped board. (see Section 5). After generating all valid moves for all pegs of our player, we have to filter them by 3 directions forward (6 directions in total if we consider backward movements as well) (See Subsection 3.2). As we always start in the bottom part of the board (South), our goal is to prioritize movements to the top of the board (North). The priority queue of movements would look as follows: *North-West (NW), North (N), West (W), East (E), South-East (SE) and South (S)*. So if we do not have any moves in the NW direction, we should check if we have moves in the N direction and so

on. NW direction is associated with the intuition for the shortest path. More details regarding the implementation can be found in the Figure 7

4.2.2 Strategy no. 2

It is important to highlight, that this strategy was designed only for a scenario when we have two players only (See Section 5). This strategy extends the previous one by sorting the player's pegs by distance from the goal, which is a very common heuristic for Chinese Checkers [12]. Then we filter only the moves which include the pegs that are the farthest from the goal (-3,6), to keep all the pegs in a shorter distance in between [1]. Finally, we prioritize the moves by directions as in strategy no.1. This strategy shrinks a set of possible moves a lot and might not be suitable for 3 3-player games. More details regarding the implementation can be found in Figure 8.

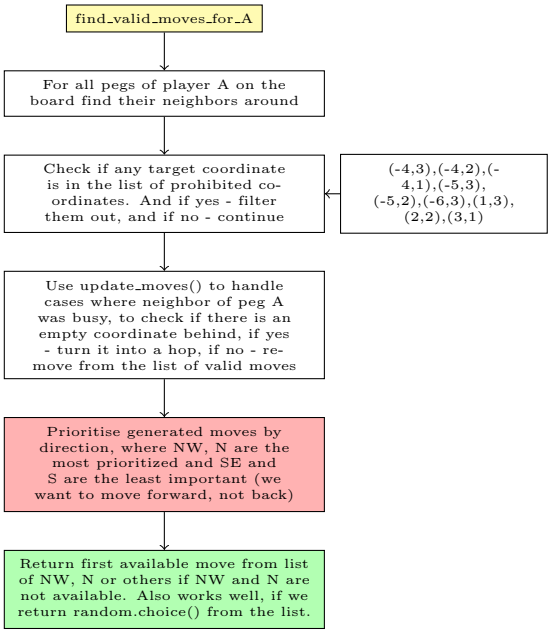


Figure 7: Approach no. 1 for the strategy 1.2.1 [8]

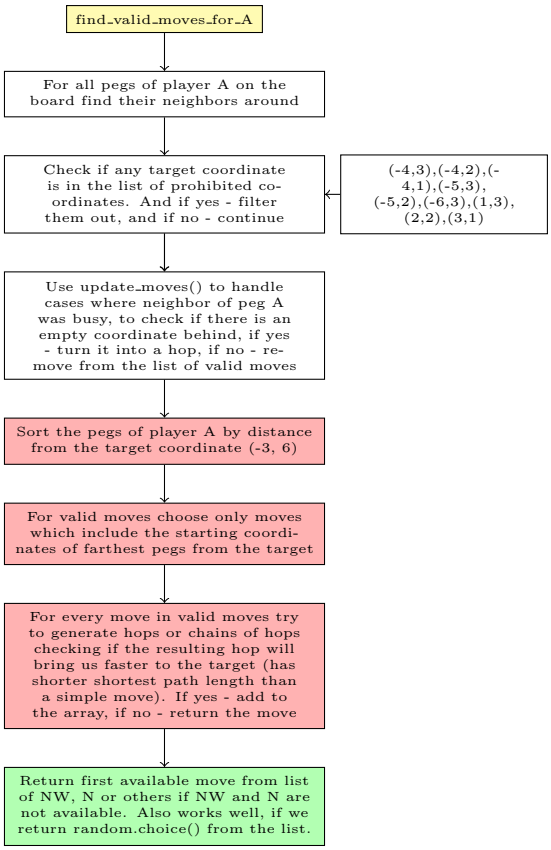


Figure 8: Approach no. 2 for the strategy 1.2.2 [8]

4.3 Minimax approach with alpha-beta pruning

In this subsection, I would like to present the next 3 strategies, which share the same core-bone structure of minimax with a small extension (for three players or using a different evaluation function). Combining minimax with a win-game strategy, i.e. bridge or ladder strategy [14], when we position our pegs in such a way to create hop chains to reach the goal faster in the future, which also brought much better results [1]. But all 3 strategies use concepts of recursion for moving down the search trees of possible moves as well as for generating the chains of hops. Because the minimax algorithm is prone to the horizon problem, I tried to reduce the set of possible states as much as possible, which can be achieved by filtering the valid moves and applying alpha-beta pruning. [7] In the Minimax algorithm, we evaluate the future moves for players A and B in the case of a two-player game and for A, B, and C in the case of 3 players. We run it recursively until we reach the selected depth level or reach the

win state. My implementation of Minimax for strategy no. 3 with alpha-beta pruning in Python can be found in the listing. 3.

Listing 3: def minimax_123() function

```
    def minimax_123(self, board, depth, maximizing_player, alpha=float('-inf')
        , beta=float('inf')):
        if depth == 0 or self.game_over(board):
            return None, self.evaluate_board(board)

        peg_positions_A = board.return_coordinates_of_A_pegs()
        peg_positions_B = board.return_coordinates_of_B_pegs()

        valid_moves_A = self.valid_moves_for_most_far_away_peg(board,
            peg_positions_A)
        valid_moves_B = self.valid_moves_for_most_far_away_peg(board,
            peg_positions_B)

        best_move = None
        if valid_moves_A and valid_moves_B:
            if maximizing_player:
                max_eval = float('-inf')
                for move in valid_moves_A:
                    temp_board = copy.deepcopy(board)
                    self.apply_move(move, temp_board)
                    _, eval = self.minimax_123(temp_board, depth - 1, False,
                        alpha, beta)
                    self.undo_move(move, temp_board)
                    if eval > max_eval:
                        max_eval = eval
                        best_move = move
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
                return best_move, max_eval
            else:
                min_eval = float('inf')
                for move in valid_moves_B:
                    temp_board = copy.deepcopy(board)
                    self.apply_move(move, temp_board)
                    _, eval = self.minimax_123(temp_board, depth - 1, True,
                        alpha, beta)
                    self.undo_move(move, temp_board)
                    if eval < min_eval:
                        min_eval = eval
                        best_move = move
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
                return best_move, min_eval
```

4.4 Heuristic

For approaches 3-5, I have designed a heuristic, which measures the vertical and horizontal distance to the target.

We are going to use the Manhattan Distance or L1 norm to measure the distance from

1. Goal coordinates (-3, 6),(-3, 5), (-2, 5) [for A player]
2. Subgoal coordinates (-3, 4), (-2, 4), (-1, 4) [for A player]
3. Center of the board (0,0)

$$\text{Manhattan Distance} = |x_2 - x_1| + |y_2 - y_1|$$

4.4.1 Goal coordinates

This is a set of coordinates that represent the opponent's home and once all pegs of our player have reached them, the game is considered as won. That is why I set up the weight of this distance to the highest (1.0). [8]

4.4.2 Subgoal coordinates

To motivate the pieces to move from the center of the board to the target, it was important to introduce the sub-target coordinates, which represent a series of positions just before the opponent's home. The weight of this distance is equal to 0.95. [8]

4.4.3 Center of the board

If a player reaches the center of the board as quickly as possible, then that player has a good chance of winning. I therefore decided to include this in the evaluation function to motivate the marbles to leave the starting corner more quickly. As it is an intermediate goal, I chose a weight of 0.5. [8]

4.4.4 Final heuristic

This measure is not a distance, but an additional coefficient that either greatly increases the evaluation value if the player evaluates a winning state (+10), or greatly decreases it if the player loses (-10). For intermediate states, this coefficient is 0 and does not influence the other heuristics. [8]

Combining all the above-mentioned measurements, we get the formula below:

$$\sum_{k=0}^{2} \left[\text{weight} \cdot \sum_{i=0}^{\text{Targets}} \sum_{j=0}^{4} \left(|x_j - x_i| + |y_j - y_i| \right) \right]_{\text{player B}}$$
$$- \left[\text{weight} \cdot \sum_{i=0}^{\text{Targets}} \sum_{j=0}^{4} \left(|x_j - x_i| + |y_j - y_i| \right) \right]_{\text{player A}}$$
$$+ \text{Final Heuristics}$$

, where

- **k** represents the sum of all distance targets (goal, subgoal, center)
- **i** means every coordinate in the set of target distances
- **j** represents every peg (5 in total)

Adjustments of the heuristic for the 3-player game (A, B, C players) [8]:

1. We assign correct goals for each player (different from 2 player game)
2. We calculate the distances for C in the same way as for B and A
3. Summing up the distances for players B and C and taking an average of it

4.4.5 Strategy no. 3

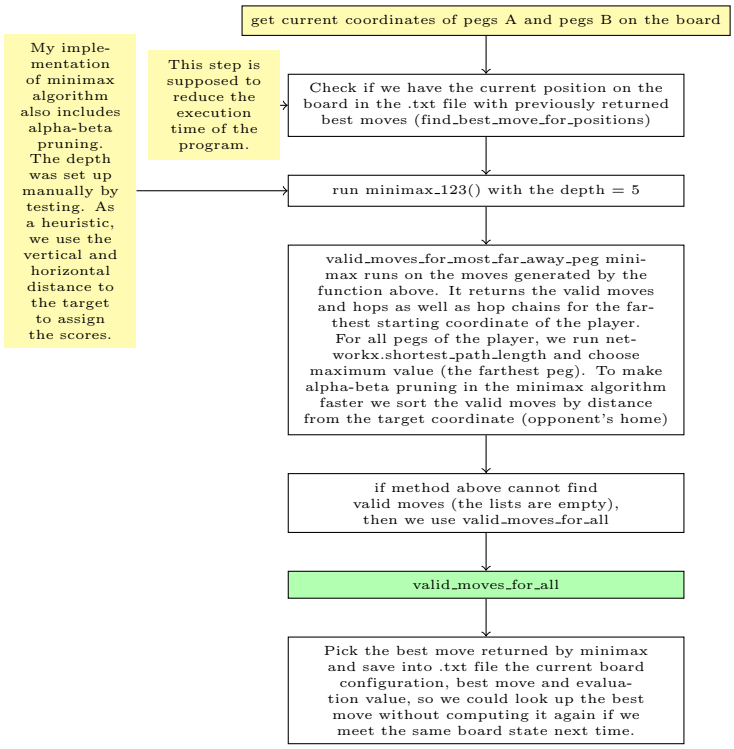
In this approach, we always group our blocks by calculating the shortest path for each marble and sorting in descending order of distance to the target to always select the furthest block and help all blocks stick together. To prevent nodes from de-grouping, we also check the distance between each node in the army to correct it (cannot be bigger than 3). To aid fast progress to the goal, we favor chains of hops or simple hops over simple moves. Finally, to avoid an exhaustive calculation with depth=5, we also save the board configuration and the best move found in a .txt file to look it up for the next games. More details regarding the implementation can be found in Figure 9a.

4.4.6 Strategy no. 4

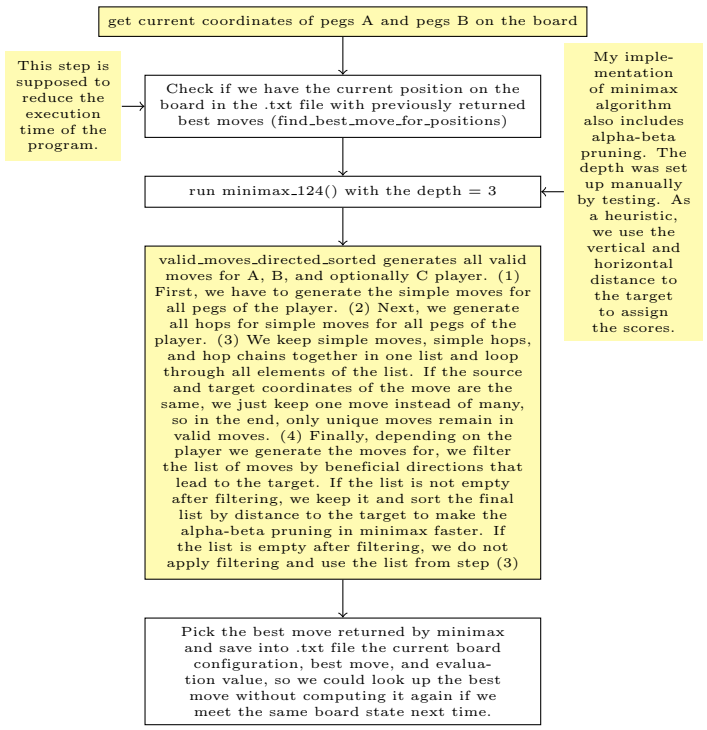
Here we remove the duplicates generated by simple moves and hop functions if they have the same source and destination to have a smaller search space for the minimax. To support minimax's decision I additionally sort the valid moves by distance from the target. We also operate on a lower depth, to have faster computations. More details regarding the implementation can be found in Figure 9b.

4.4.7 Strategy no. 5

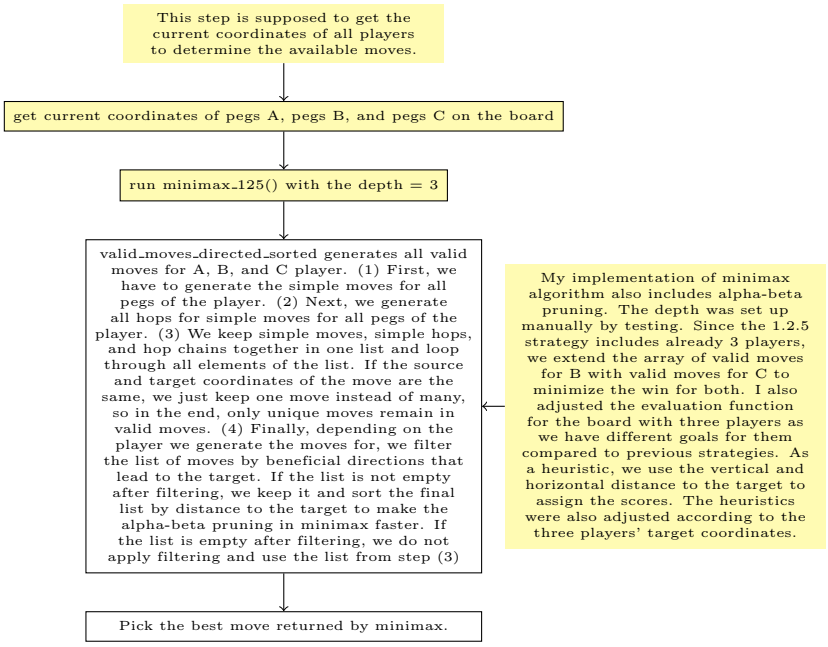
This strategy represents an extension of the previous approach with a focus on 3 players. More details regarding the implementation can be found in Figure 9c.



(a) Approach no. 3 for the strategy 1.2.3 [8]



(b) Approach no. 4 for the strategy 1.2.4 [8]



(c) Approach no. 5 for the strategy 1.2.5 and 1.2.6 [8]

Figure 9: Approaches 3-5 [8]

5 Comparison of approaches

For evaluation of every strategy I was using AI Systems Project's server, which offers 8 levels of FAU Halma game's difficulty [10]:

1. Rhombus environment (simplified board shape, 2 players only)
2. Two-player star environment (easy)
3. Two-player star environment (medium)
4. Two-player star environment (difficult)
5. Three-player star environment (easy)
6. Three-player star environment (medium)
7. Three-player star environment (difficult)
8. Three-player star environment (hardcore)

Each approach was developed to cope with a certain level of difficulty in the game. Approach No. 1 was designed so that the players can only play on the diamond-shaped board, while Approach No. 2 already assumes a star-shaped board and only two players. These conditions must be taken into account when evaluating the methods. Due to the large number of combinations (strategy-game difficulty pairs) and the conditions mentioned above, it was decided to evaluate all implemented strategies under the most robust (star-shaped and 3 players) but least complicated (in terms of opponent's strength).

	Time Metric ↘	*Win Rate Metric* ↗
Strategy no. 1	2.31s	0.20
Strategy no. 2	2.38s	0.55 [(1)]
Strategy no. 3	20.52s [(2)]	0.40
Strategy no. 4	1.34s	0.70
Strategy no. 5	2.13s	0.75

Table 1: Performance summary of Time and Win Rate metrics for all strategies tested on the difficulty level 5

From the measurements in Table 1 we can observe, that strategies **4** and **5** produce the best metrics' results: the least time and the highest win rate. As the win rate has a higher weight for us than time in case of a small variability, strategy 5 would be the best to win the opponent at different difficulty levels.

AI Systems Project's server allows us to evaluate the performance by measuring the win rate per 50 games, which also gives us an understanding that strategy 5 is working the best for difficulty level 5 (Table 2), but as the difficulty continues to increase at next levels, the win rate decreases.

5.1 Time

Time metric calculates how much time we need to compute the best move before the opponent takes his turn. It is measured in seconds and represents the average time during the whole game since in minimax-based strategies the time for calculating the best move is strongly correlated with the board configuration (the more possible moves and the greater the depth, the more time we need to evaluate the best move). We want **to minimize** this criterion first and foremost.

[(1)] (Combined with another strategy) Because the method was designed for two players only, thus for some of the games of 1.2.5 difficulty level it returned "Out of moves" error and had to continue with another strategy (i.e. no. 1) to get out of this situation.

[(2)] (Preloaded states are used) Because the depth of minimax is 5 computing the best move for every board state might take a lot of time, that is why I decided to save every best move for every board configuration to save the time and optimize the performance.

	Best rating (from the server)	*Performance (%)*
Strategy no. 1 for difficulty level 1	1.0/1.0	100%
Strategy no. 2 for difficulty level 2	0.9/1.0	90%
Strategy no. 3 for difficulty level 3	0.8/1.0	80%
Strategy no. 4 for difficulty level 4	0.5/1.0	50%
Strategy no. 5 for difficulty level 5	1.52/2.0	76%
Strategy no. 5 for difficulty level 6	1.02/2.0	51%
Strategy no. 5 for difficulty level 7	0.92/2.0	46%
Difficulty level 8	Not evaluated	Not evaluated

Table 2: Performance summary of strategies (evaluated at the server from assignment [10]) for different difficulty levels.

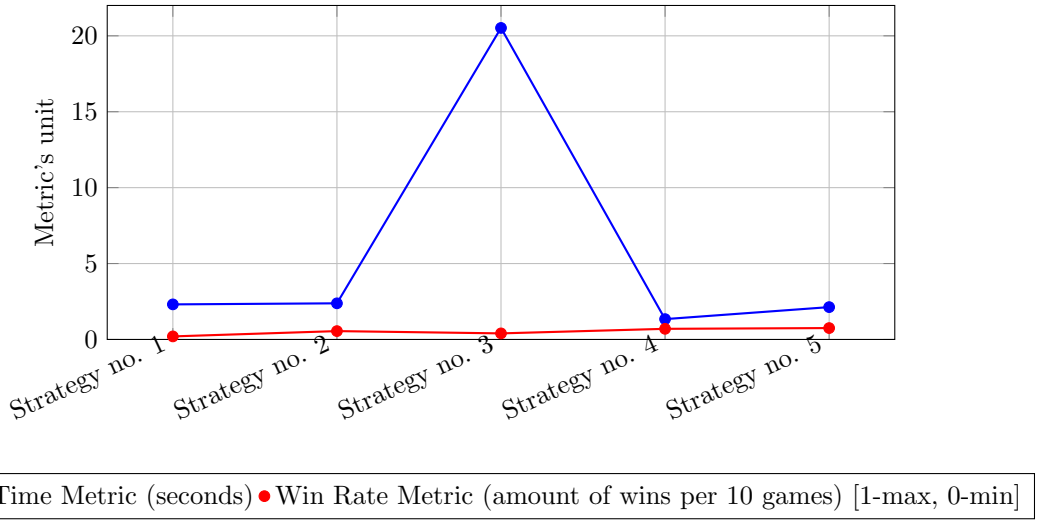


Figure 10: Comparison of metrics for 5 strategies

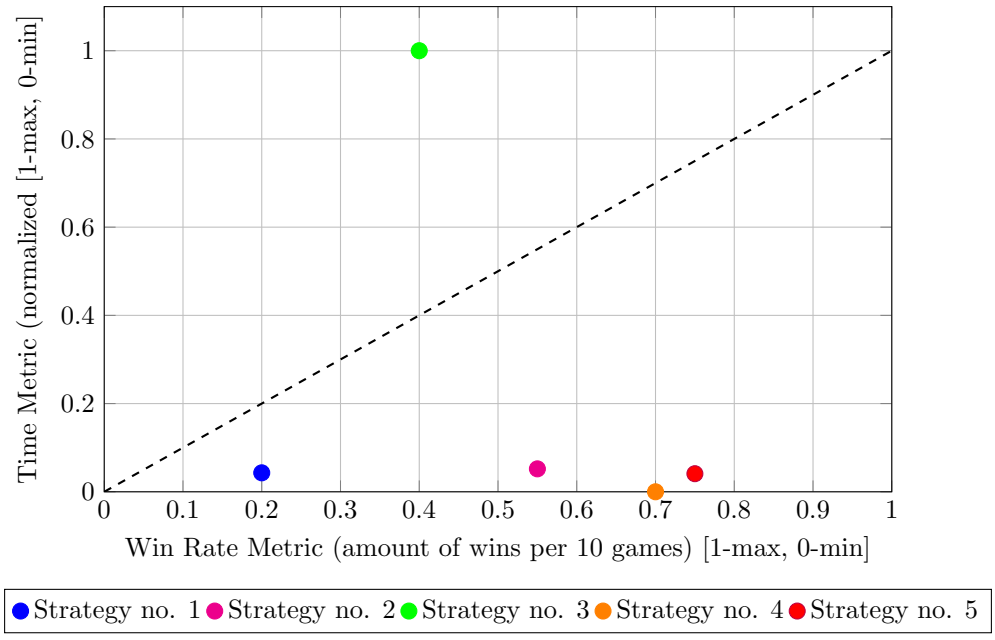


Figure 11: Scatter plot: Win Rate vs. Time (normalized [1-max, 0-min])

5.2 Win rate

To subjectively understand whether the current approach introduces an improvement, we need to look at the number of games the agent has won over time. Win rate takes into account amount of wins within 10 games scope. Since there are three players, we have two options: either we win first (in which case the agent receives 1.0 points according to the table above) or we win second (the agent receives 0.5 points). In case of a lost game, the agent gets 0.0 points. We aim **to maximize** this parameter.

In Figure 11 we can observe that the best combination of both metrics can be found in the bottom right corner of the graph (minimum time and highest win rate). Due to the particularly small number of samples, it may be difficult to identify the trend, which can be better represented with non-scaled metrics on a piece of paper.

6 Conclusion

The implementation has proven that FAUHalma game can be solved using an appropriate heuristic function for the appropriate game difficulty level.

After mapping the game's object onto the search graph, one can use simple heuristics for less strong opponents such as:

- Direction
- Distance from the target
- Following the shortest path
- Creating hop chains (ladder/bridge strategy)

Alternatively for a stronger opponent, one can plan its movements by looking several steps ahead in the future using the minimax approach with alpha-beta pruning. As an evaluation function, one can use:

- Complex heuristics: Vertical and horizontal distance from the target coordinates, close to the target area and center of the board.
- Complex heuristics combined with simple strategies (filtering moves by direction or distance from the target or limiting the search space by removing the duplicates).
- Complex heuristics combined with moves sorting makes alpha-beta pruning even more effective and reduces the time for computing the best move.

For future areas of research, it is recommended to apply Iterative Deepening for minimax search. Alternatively, new methods could be explored, such as:

- Combining the existing methods with neural networks would potentially
- Solving the problem using Monte-Carlo-Tree Search
- Designing even more robust heuristics

References

[1] Chinese checkers master: Rules & strategy. Online. Accessed on February 18, 2024.

[2] George I. Bell. The shortest game of chinese checkers and related problems. In *arXiv.org*, 2008.

[3] K. R. Chowdhary. *Adversarial Search and Game Theory*, pages 303–335. Springer India, New Delhi, 2020.

[4] Nicholas Fonseca. Optimizing a game of chinese checkers. BSU Honors Program Theses and Projects, Item 129, 2015. Copyright © 2015 Nicholas Fonseca.

[5] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. *NetworkX Reference Manual*, 2008.

[6] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[7] Michael Kohlhase. Artificial intelligence 1 winter semester 2023/24 – lecture notes, 2024.

[8] Pavel Kostarev. Assignment 1.2: Play fauhalma - evaluation. https://gitlab.rrze.fau.de/wrv/AISysProj/ws2324/a1.2-play-fauhalma/team725/-/blob/main/Evaluation_1.2.md?ref_type=heads, 2023. Accessed on February 18, 2024.

[9] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A modern approach*. Prentice-Hall, 2010.

[10] Jan Frederik Schaefer. Assignment 2: Play fauhalma. https://kwarc.info/teaching/AISysProj/WS2324/assignment-1.2.pdf. Accessed on February 18, 2024.

[11] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[12] Nathan Sturtevant. *On Strongly Solving Chinese Checkers*, pages 155–166. 12 2020.

[13] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023.

[14] Wu Yisi, Mohd Nor Akmal Khalid, and Hiroyuki Iida. Analyzing the sophistication of chinese checkers. *Entertainment Computing*, 34:100363, 2020.