Halma - Adversarial Search Methods for Two-Player Board Games

V. Bartolomeu, N. Ivakko
FEUP, Faculty of Engineering, University of Porto
Master in Data Science and Enginnering

***Abstract*—In the context of the master study data science and engineering in artificial intelligence we deal with the topic reversarial search for two player board ganes. we analyze different search problem formulations and compare different evaluation functions for intelligent agents for playing Halma - also known as chinese checkers.**
We implement a game interface for human against human, human against computer and computer against computer. Primarily the Minimax algorithm is applied and tried to improve.

***Index Terms*—Artificial Intelligence, Intelligent Agents, Adversarial Search, Minimax, Monte Carlo, Halma.**

I. Introduction

Adversarial search is a subfield of artificial intelligence that involves finding optimal strategies in two-player games where players have opposing goals. Two-player board games provide an excellent platform for testing different adversarial search algorithms as they are well-defined, have clear objectives, and can be used to benchmark different AI techniques. The most common approach to adversarial search is the Minimax algorithm, which searches for the best move for the current player while assuming that the opponent will choose the move that best decreases the overall score of the current player. In recent years, adversarial search has gained significant attention in the context of two-player board games, and various enhancements have been developed to improve their performance.
The game of Halma, is a strategy board game invented around 1883 by George Howard Monks. The name Halma was proposed by his brothers father in law Thomas Hill, referring to the ancient greek word for ”jump”. Although the gameplay is almost identical, ”Chinese Checkers” is better known nowadays, referred to as ”Stern Halma” - ”Star Halma” - in german, which is why the two board games are often confused.

II. Game and Problem Definition

The Halma gameboard is defined by a 16 times 16 checkerboard. The game can be played by two or four players, starting in opponent corners. When played by two players, each player is equipped with 19 pieces, which can be colored stones, wooden ponds or small checkers. For simplicity, Halma is used as a reference for the game with two players. The object of the game is to move all of your pieces to the target area first, which is your opponent's starting position.
The game starts by randomly selecting a player to make the first move. All pieces can move in eight different directions (orthogonally and diagonally). The easiest move for a piece is to move it to one of the adjacent free spaces, resulting in terminating the move. More complex moves include jumping over adjacent pieces of any color as long as a free square is available on the direct opposite side. The same piece can be used to make more jumps or the turn is ended. Once a piece is in the goal area it can move within the goal area but not outside of it. In Halma pieces cannot be captured as in other board games like Chess. The simplicity of the game makes Halma suitable for beginners as well as advanced strategists. The aim of this project is to implement the game of Halma. Game configurations should include Human-Human,

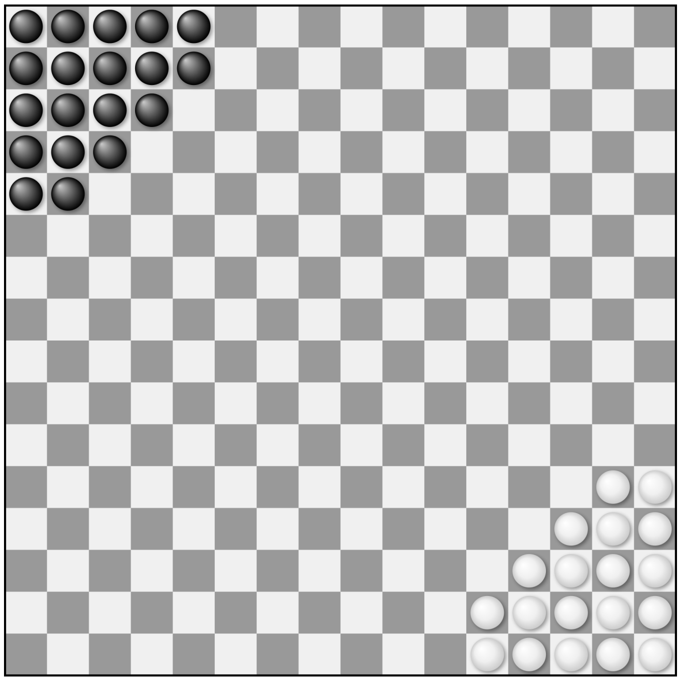
Fig. 1. Halma starting positions for two players.

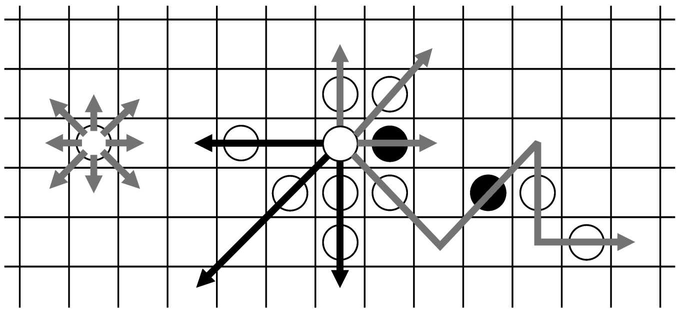
Fig. 2. Halma starting positions for two players.

Human-Computer and Computer-Computer. The Computer will be an intelligent agent following the Minimax algortihm.

III. Related Work

Adversarial search and intelligent agents have been extensively studied in the context of board games, particularly in games like Halma. One of the most commonly used algorithms for adversarial search is the minimax algorithm, which assumes that the opponent plays optimally and searches for the best move accordingly. However, minimax can be computationally expensive and impractical for larger game trees. To address this issue, several optimization techniques have been proposed, such as alpha-beta pruning, transposition tables, and parallelization [1].

Another approach to improving adversarial search is through the use of intelligent agents, which are capable of learning from experience and making decisions based on past interactions. This has been particularly successful in games like Halma, where machine learning algorithms have been used to train agents to play the game at a high level [2]. One example of this is the use of reinforcement learning, where agents learn through trial and error by receiving rewards for making good moves and penalties for making bad moves. This approach has been shown to be effective in improving the performance of Halma agents and has led to several advancements in the field[3]. Overall, the combination of adversarial search algorithms and intelligent agents has proven to be a promising area of research in game theory, with many exciting developments expected in the future.

IV. Formulation of the problem as a search problem

To approach the implementation of intelligent agents the problem has to be defined as a search problem.

A. State Representation

The board will be represented as as a $n \times n$ numpy array of zeros. The pieces will be represented as integer values, such as 1 and 2, for each player respectively, since each piece has the same power.

In addition to using a numpy array to represent the board and integer values to represent the pieces, we could also consider using other data structures and data types to optimize the efficiency and speed of our Halma game implementation. For instance, we could use a bitboard representation of the board, where each bit represents a square on the board and whether it is occupied by a player's piece or not. This could potentially improve the performance of our game implementation by reducing the memory usage and improving the speed of operations such as bit shifts and bitwise operations.

B. Initial state

In the starting position of Halma game for two players with 19 pieces, each player's pieces are arranged in a symmetrical pattern on opposite corners of the board. Assuming the first player's starting position is the top left corner, the first two rows of the board will be occupied by five pieces each, the following three rows by four, three and two pieces, respectively, forming a triangular pattern. This symmetrical arrangement of pieces ensures that both players have an equal opportunity to make their moves and win the game.

C. Objective test

To objectively evaluate the Halma game implementation we will conduct performance and accuracy tests. During the implementation process the algorithm for proposing the possible movements was evaluated, so that cycling or infeasible movements will not occur.

D. Operators

Each operator will be assigned a designated name, such as "up" or "jump_down_right," to denote a straightforward move to a neighboring square above or a jump to an unoccupied square over a piece in a downward right orientation, respectively. The preconditions of the operators aimed at moving towards an adjacent square will be based on verifying if the targeted square is unoccupied and if the direction of movement is confined within the board limits. On the other hand, the preconditions of jump-operators will verify the availability of an adjacent piece, the presence of an unoccupied square behind it, and whether the jump is within the board boundaries.

The program will be designed to detect and analyze sequences of jumps in a player's turn to determine if multiple jumps are possible. When a piece is moved the effect is twofold: the original position of the piece becomes vacant, while the new position becomes occupied by the piece, resulting in a shift of pieces across the board. The cost of a move is determined by its efficiency, which is based on its distance to the goal position. Jumps towards the goal will be assigned a lower cost, while those away from the goal will be penalized the most. The same principle applies to simple moves to adjacent squares.

E. Evaluation Function

Evaluation functions for Halma game with intelligent agents will take into account several factors, including the number and position of the pieces, the distance to the goal positions. For example, the evaluation function might assign higher scores to pieces that are closer to their respective goal positions or that have more mobility to make future moves. It might also assign scores based on the ability to block the opponent's moves or to occupy strategic positions on the board. This will be made by evaluating the sum of all the minimum distances from a piece to the goal area and the capability to minimize this value at each following move at a given depth. The ultimate goal of the evaluation function is to provide a measure of the relative strength of a given board position for a particular player, which can then be used by the intelligent agent to make optimal moves.

V. Implementation

The Project was made using Python with jupyter notebooks. Recursive data structures are useful in situations where there is a hierarchical relationship between data points. These data structures can be represented in the form of trees or graphs, where the parent node has one or more child nodes. In the case of the Halma game, the initial state of the board is represented by the root node, and each subsequent state is represented by its child node. Each child node represents a possible move that a player can make from the parent node state.

To implement the game using recursive data structures, a Python class representing the state of the game is created. This class contains methods for generating child states and evaluating the current state of the game. The child states are generated by applying each of the possible moves to the current state of the game. The evaluation function assesses the current state of the game and assigns a score based on the difference of the minimum distance of the pieces to their goal target between the two players.

The ConnectHalmaGame class is used to join players and recursively make alternating moves until there's a winner. This is achieved by defining an initial state, which is an instance of HalmaGame, checking who is the current player and accordingly making a move based on the evaluation function defined for the player. After this player decides which is the best move, the board state is updated and turns change until one of these players achieve their goal state. This class also allows to make several game simulations with or without a time limit and has the ability to print the overall results of games.

The Halma Agent player is defined by choosing which evaluation function we are passing into a general-purpose implementation of the minimax algorithm for turn-based games with two players and a max_depth attribute that will limit the depth of the search tree. The minimax function recursively evaluates the game state and generates all possible moves up to the maximum depth. It uses alpha-beta pruning to optimize the search by avoiding unnecessary evaluations of unpromising moves. The execute_move function calls minimax to choose the best move for the current player based on the given evaluation function.

There are other types of agents available like the Random agent and the Human Agent. These agents are defined by their own execute movement functions. In the Random agent, for a given board state it randomly picks a move from the set of all available moves for that state and updates the game board. The Human Agent on the other hand has an interactive input dashboard where the player can see the board state changing after each turn of events and decide which pieces he wants to move. This interface has two different steps:

- An index input that consists on the index of the piece we want to perform the move on. This move index should be written in the form of: 1,1 ; 0,0 ; 2,1 etc...
- The name of the move we want to perform on the selected piece. This name should follow the given structure: ”up”, ”down_left”, ”jump_down”, ”jump_up_right”, etc...

It should be also considered the fact that consecutive jumps may be available and for that case, after a jump is made, there will show an option input asking if the player wants to perform a consecutive jump or not (options: yes, no) and if the player types ”yes” it will show then another input for the name of the move he wants to perform on that piece.

VI. Approach

We will start with a smaller board size of 6×6 with 4 pieces each for proof of concept so the runtimes for the different evaluation functions are lower and we are able to do quicker testing on which are the best representations for the problem.

Our first approach was represent the board state as a matrix so that it would be easier to visualize the board state for each node. This matrix is an attribute of the HalmaGame instance which has also other attributes like:

- Goal board - Which is a matrix that states the final positions for both players
- Current player - Which is the value of the player that is executing the move (can be 1 or 2)
- Parent - Parent object for the given node
- Parent operator - The operator that was used on the parent node and resulted on the given node
- And other parameters to check if there is a jump possibility

This state representation made us able to recursively go from the root node to its children objects and iterate through state tree in the minimax algorithm.

After performing several tests we saw that for depths of 4 and above the time that took for each agent to perform a move was above 2 minutes because at that depth we are creating at a average branching factor of 20 of around 168,421 nodes and given the number of evaluations we perform for each node this is computationally heavy.

Given this we thought of a new way to represent the states of the board that were basically numpy arrays with the indexes for each piece for a each player. This made the operations faster but was harder to code due to lack of representation the state has with indexes in contrary to the previous representation and together with the lack of time we add to develop the algorithm this state implementation wasn't perfected.

Now in order to evaluate the best move we need to adapt the minimax algorithm to the different evaluation functions. The first evaluation function tries to maximize the value of the distance difference for each player by multiplying the evaluation value times -1 if the current player is player 1 and times 1 if the current player is player 2. This adjustment needs to be made because the difference value that the evaluation_function_1 returns is distance of player 1 subtracted by the distance of player 2, so this way player 1 is always pursuing the most negative value possible and player 2 is doing the opposite. By multiplying evaluation values by these constants makes us able to perform a standard maximization of the evaluation function value for both players.

The second evaluation function is just the sum of the minimum overall distances for the current player, then the goal of minimax is simply to minimize this value. Therefore that is where the distinction inside of the execute_minimax_move_aux function comes from.

VII. Results and Analysis

Here is where we are going to state the different runtimes and scores for different evaluation functions and depth

VIII. Conclusions

We could explore the potential of using machine learning techniques to develop intelligent agents for playing Halma. For instance, we could train a neural network to estimate the value function of game states and use it in combination with Monte Carlo Tree Search to develop a strong AI player. Alternatively, we could use reinforcement learning to train an agent to learn the optimal policies for playing Halma through self-play or against human players. By incorporating these advanced techniques, we could develop a more challenging and engaging game that can push the boundaries of our understanding of Halma and artificial intelligence. [4]

References

1 Russell, S. J., *Artificial intelligence a modern approach.* Pearson Education, Inc., 2010.

2 Tesauro, G., "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

3 Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

4 Buro, M., "Monte-carlo search versus minimax in games with hidden information," in *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-03)*, Edmonton, Canada, 2003.