Finding The Optimal Halma Agent with Artificial Intelligence

Yoav Gochman
yoav.gochman@mail.huji.ac.il

Assa Kariv
assa.kariv@mail.huji.ac.il

Eyal Schaffer
eyal.schaffer@mail.huji.ac.il

Noam Delbari
noam.delbari@mail.huji.ac.il

Final Project, Introduction to Artificial Intelligence 67842, Spring 2020

The Rachel and Selim Benin School of Computer Science and Engineering,
The Hebrew University of Jerusalem,
Jerusalem, Israel

Halma is a strategy board game invented in 1883. In this paper we will compare between several kinds of Artificial Intelligence agents, in order to find the optimal player for this game.

Minimax; Alpha-Beta Pruning; Iterative-Deepening; Monte-Carlo Tree Search; Expectimax; Halma; Chinese Checkers;

I. INTRODUCTION

A. *The Game*

Halma (from the Greek word ἅλμα meaning "jump") is a strategy board game invented in 1883 or 1884 by George Howard Monks, a US thoracic surgeon at Harvard Medical School. His inspiration was the English game Hoppity which was devised in 1854 [1]. A famous variant of this game is "Chinese Checkers" or Stern-Halma.

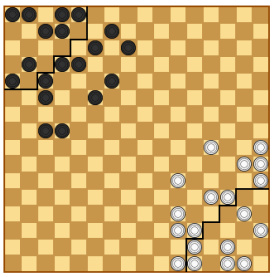

Figure 1.1 - Halma Board [8]

The game is played by two or four players seated at opposing corners of the board. The game is won by being first to transfer all of one's pieces from one's own camp into the camp in the opposing corner. On each turn, a player either moves a single piece to an adjacent open square, or jumps over one or more pieces in sequence[1]. Moving a piece can be done by rolling – moving into adjacent cell, or by hopping, a sequence of jumps over pieces (from both sides).

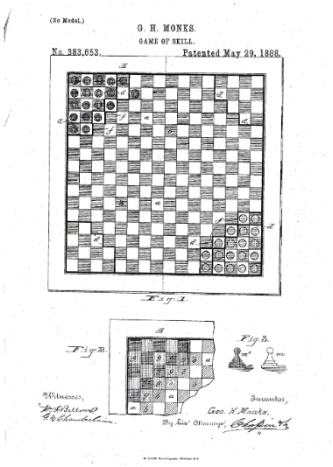

Figure 1.2 - George Monks Original Patent [2]

George Monks (the inventor) identifies 3 stages in a typical Halma game: [2]

- The gambit – the pieces are leaving their base
- The melee – the pieces are "clashing" in the middle of the board
- The packing - the pieces passed their opponent and are now trying to reach the goal

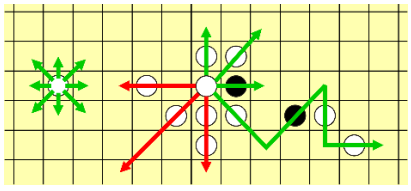

Figure 1.3 - Valid (green) and invalid (red) moves of a white pawn in Halma [2]

B. The Project

We wanted to explore a board game which involves forward thinking and strategic planning. In addition, we realized that the number of different game states in Halma can be huge, and we wanted to take the challenge of modeling the problem in efficient ways and optimizing the algorithms.

Our goal in this project is to find the optimal Halma AI Agent. In order to do so we will analyze different kinds of agents with several heuristics.

II. METHODOLODY

A. Modeling The Problem And Handling The Challanges

The original game of Halma is 16 x 16 board with 19 pieces for each player, hence the space complexity for the state space is:

$$\binom{256}{19,19} \approx 1.47 * 10^{34}$$

In order to reduce the search-space we created a minimized version of the game with 8 x 8 board and 10 pieces for each player. Halma is considered as a "racing" game, and it has two main challenges: First, unlike chess, no pieces are removed from the board as the goal of each player is moving forward into the opposite base. In addition, unlike Go [3] the pieces can go forward and backwards, in every point of the game, and repetitions are allowed. Thus, the number of possible actions is rising quickly until the *packing* phase, and as a result the branching factor increases as well. After this phase, it decreases slowly. This leads into enormous computation times in the search trees, especially in large depths.

We decided to handle this by adding an action-time-limit for our agents. For example, in Minimax algorithm, we evaluate the state if it is a leaf node or if the time limit (which is pre-defined) has passed. This ensures we don't modify the original problem drastically but can play a game with reasonable computation times. The time-limit solution did solve the time problem, but the agent played poorly. We realized that with a reasonable time-limit the depth 3 agent only explores one branch in depth 1 and ten branches in depth 2 on average.

This led us into finding other solutions, such as Alpha-Beta pruning, Move-ordering and Move-ignoring. The latter means to completely ignore some actions in the game, for example actions that go backwards. This adaptation reduces the amount of actions significantly, and the runtime, respectively. However, ignoring backward actions comes with a heavy price: some of the actions we ignore might be considered optimal actions from some states of the game. Consequently, we decided to use this adaption only on Minimax agents of depth 3 and higher, and in the MCTS agent. Table 1 shows how the adaptions significantly improved computation time. On the other hand, fig. 2.1 shows the possible bad outcome of Move-Ignoring.

Table 1 – Average single move computation time, depth 3, 8x8 board (seconds)

AGENT	TIME IN SECONDS
MINIMAX	500
MINIMAX W/ ALPHA-BETA	50
ALPHA-BETA W/ MOVE-ORDERING	30
ALPHA-BETA W/ MOVE-IGNORING	4
ALPHA-BETA W/ MOVE-IGNORING AND MOVE-ORDERING	2

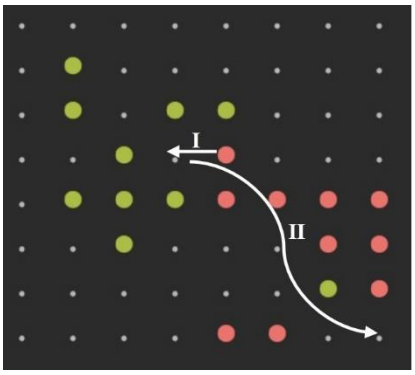

Figure 2.1 - Example for 'Move-Ignoring' Drawbacks – Red player will ignore move I, even though it will probably be benefitial for him

Another challenge is handling deadlocks. Some games of Halma, especially AI vs AI games can cause deadlock. For example, if one of the pieces of player one fortifies in the base, player two can never win. In order to solve these cases, we added two features in our model:

- Our agents pick an action randomly from a list of their best actions. This prevents deterministic games and breaks the deadlock in most cases.
- Games that last longer than 150 rounds are decided by a tie breaker function – the winner is the player who has more pieces after the diagonal half line.

B. Agents

We implemented several agents. Each agent is given a state and returns an action to take.

- **Reflex Agent** – this is a benchmark agent for testing purposes. We wanted it to simulate a professional human player, so it is not completely greedy. It is trying to move the army "as a whole", with the intention to make the furthest advancement.
- **Minimax Agent**

- **Expectimax Agent**
- **Minimax with Alpha-Beta Pruning Agent** – We decided to implement this agent with 'Move Ordering' – sorting the actions from best to worst by a heuristic function, in order to maximize the number of prunes.
- **Iterative Deepening Alpha-Beta Agent** – In specific states of the game, regular Alpha-Beta agent does not take the best action. If one can win the game in one action, or in two actions – both actions get the same heuristic score, and the latter might get chosen. The algorithm (Figure 2.2) will prefer actions to winning states that are closer to the root. That is, states which take less actions to get to.

```
ITERATIVE-DEEPENING-ALPHA-BETA(STATE, DEPTH)

FOR J IN [1,…DEPTH]:

BEST-ACTION = ALPHA-BETA(STATE, J)

IF BEST-ACTION == PROBLEM.IS-WINNER():

    RETURN BEST-ACTION


RETURN BEST-ACTION
```

Figure 2.2 - Iterative-Deepening Alpha-Beta Algorithm

- **Monte-Carlo Tree-Search Agent (MCTS) –** Heuristic search algorithm which aims to approximate the search tree such as Min-Max by running a lot simulation of random games (also known as rolls-outs). While building the tree, it uses a degree of exploration in search for better solutions. One can divide MCTS to four principles:
  Selection: Search through the tree to a leaf following the node with the maximum value of the selection policy function.
  Expand: Expands some of the leaf successors (if it is not a terminal node) and sample a child randomly.
  Simulate: Run a random game from the sampled child up until the game is over or other preconditions are met.
  Back propagate: Update the values of all nodes in the path to the root.

C. *Heuristics*

In order to evaluate a state, we created the following utility functions:

- **Heatmap Forward -** A good rule of thumb is to go towards the goals through the center of the board and prevent the opponent from reaching his goals. We gave each tile on the board a score. We added all the scores of the player and decreased the scores of the opponent.

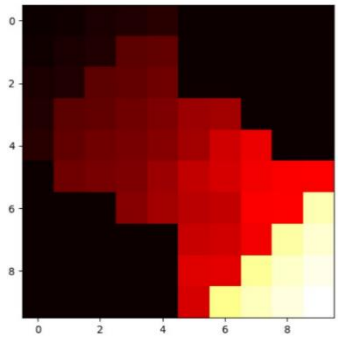

Figure 2.3- Heatmap

- **Euclidean Minimize –** Our goal is to move all pieces from one corner to the opposite. One state can be considered better than another if it minimizes the distance of all the pieces to a respective non occupied goal and maximizes that summation for the opponent. This heuristic translates this logic to an evaluation function of a state using the Euclidean distance metric.

$$EuclideanMinimize(state)$$
$$= \sum_{p \in p2-pieces} \max_{goal \in p2-goals} (distance(p, goal))$$
$$- \sum_{p \in p1-pieces} \max_{goal \in p1-goals} (distance(p, goal))$$

* assuming p1 is maximizing player

- **Avoid-Lone-Wolves** – This heuristic encourages grouping and preventing "lone wolves" that won't be able to hop towards the goal.
- **Diagonal Convergence –** Another good strategy for a "Halma" player is moving your pieces as close to the main diagonal as possible and your opponent's as far from the main diagonal as possible. Having that in mind, we can evaluate a state in the following way:

$$DiagonalConvergence(state) = \sum_{(x,y) \in p2-pieces} |x - y| - \sum_{(x,y) \in p1-pieces} |x - y|$$

* assuming p1 is maximizing player

D. *MCTS Enhancments*

In order to make better evaluations of states we made use of information from previous game trees built by the agent. This

was achieved by storing information about the states and utilizing it in separate parts of the algorithm:

1) Selection policy

One of the important aspects in MCTS is choosing nodes in a path to a leaf. In order to maintain a balance between exploring new paths which might be the optimal solution and exploit information gathered in previous iterations, MCTS scores each node in order to achieve optimal approximation to the search tree. In our implementation, we tried three scoring polices in order to select a child N_i of a node when searching for a leaf.

a) Basic formula:

$$i = argmax_i \left\{ \frac{s_i}{n_i} + ExplorationConst \times \sqrt{\frac{\ln(p_i)}{n_i}} \right\}$$

Variables : s_i *– Score for the i's child of the current node.* n_i *– Number of times the node has been visited in previous selection iterations.* p_i *– Number of visits to the* parent *of the i's node.*
ExplorationConst – Controls the exploration value to have a better trade-off between the first part (which is the exploitation) and the second (which is the exploration). We choose to set its value to $\sqrt{2}$ *as it's seems to be a consensus and ensures asymptotic optimality. [4]*

b) UCT (Upper Confidence Trees)[5]

This method is commonly used today introducing new part to the classical formula which includes more information about the next node. It uses a heuristic with expert knowledge. By adding this term, the algorithm might avoid bad paths.

$$i = argmax_i \left\{ \frac{s_i}{n_i} + ExplorationConst \times \sqrt{\frac{\ln(p_i)}{n_i}} + \frac{h_i}{n_i} \right\}$$

Variables : h_i *– Score for the action that takes the parent node to the current node.*

c) Enhanced MP MCTS[6]

As discussed in Nijsssen's work, the last term is a combination of the progressive bias and the history heuristic. Since in every turn MCTS must build a game tree from scratch, we don't utilize some of the information which might be useful to the next move. By using this policy, we collect a data base of all actions that previously been taken in the game and take into consideration how good some action was in previous games.

$$i = argmax_i \left\{ \frac{s_i}{n_i} + E \times \sqrt{\frac{\ln(p_i)}{n_i}} + \frac{h_i}{n_a} \times H \times \frac{1}{n_i - s_i + 1} \right\}$$

Variables : n_a *– Number of times the action has been played in every game in the past.*
H – Progressive history constant, determines the influence of Progressive History.

E- ExplorationConst

2) Exploration policy

In our implementation we chose to add a single node to the tree each iteration. The node is chosen randomly out of all the successive nodes that hasn't added been to the tree yet.

3) Simulation policy

Simulate games were played until a predetermined depth of 17 ply, if the game has not ended yet. We used a heuristic function to evaluate the ending game state by the maximal Euclidean distance of any piece of each player to their opposing camp.

III. Analysis And Results

A. MCTS Expriement

1) Stage 1

In the beginning of our experiments, we tested the running times of a move with the simple MCTS since we knew that other selection polices will increase the running time of each move.

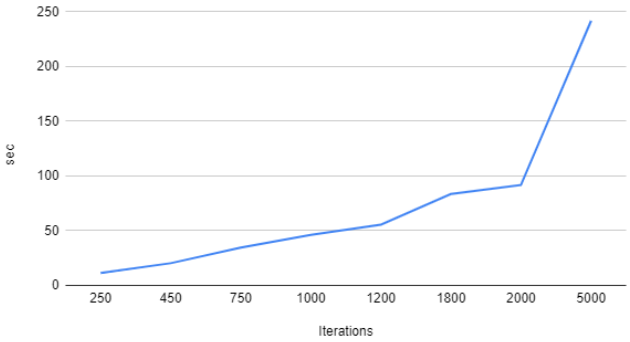

Figure 3.1 100 games average running time of a move, 8x8

It was challenging, since playing a simple game might take up to an hour and we wanted to check its performance over many games in order to get reliable result. We decided to continue the experiments over a smaller 6X6 board.

2) Stage 2

We started by comparing the influence of iterations on the performance of the simple MCTS vs our Reflex agent:

This result (Figure 3.2) was expected and mainly assured us that the agents works properly. The increase in the number of iterations allowed the agent to sample more possible game states and to make sophisticated decisions based on the game tree. We would like to note that there is a peak in its performance when using 2,000 iterations and then its performance decreases when using 5,000. We think it happened because after passing optimal number of iterations, its performance doesn't change by much and results may vary with small standard deviation of ~3% caused by the randomness that algorithm has. We decided to continue our experiments with 2,000 iterations.

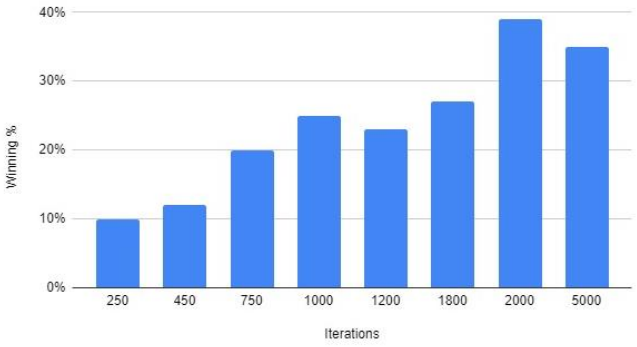

Figure 3.2 MCTS winning rate against reflex player, by iterations

3) Stage 3

In the next stage we wanted to see if the selection policy has a profound effect on the performance of the agent so we tested each policy against others. (Table 2)

Table 2 - Win-rate, policy vs policy

Policy-1	**Policy-2**	**Policy-1 Win-Rate**
UCT	Simple MCTS	59%
Prog. History + Prog. Bias	Simple MCTS	66%
Prog. History + Prog. Bias	UCT	61%

It seems that the selection policy has an effect on the agent, especially against a default MCTS player. In Nijssen paper the winning rate was much higher for a player with the enhanced selection policy against a default MCTS player. We think this difference might be explained by the h_i variable of the action's score, the big number of games simulated and various parameter tunings that could not be infered from the paper.

4) Stage 4:

Now equipped with our best policy we wanted to see if the enhanced agent can do better against other search agents (Figure 3.3)

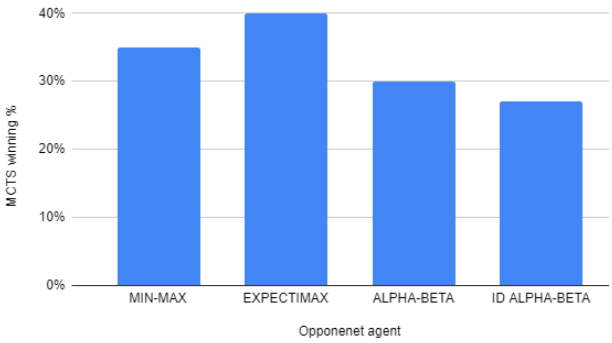

Figure 3.3 MCTS winning rate against Minimax agents with Euclidean heuristic

We saw that even after using a smaller board in order to run 2,000 iterations for each MCTS turn and with the enhanced selection policy, our agent did poorly against all other agents. This was surprising and confusing. Figuring out this failure was a complicated task which we have not found the complete answer to yet. We continued trying other parameters and different mechanics of the agent:

1. Expanding every action from each state and not just forward actions. Expanding forward actions just to a certain point in the game and then expand all actions.
2. Trying bigger thresholds to simulated games and then removing it altogether.
3. Changing the scoring heuristic to our best heuristic that we've found later.
4. Trying to lower the Exploration parameter.
5. Using different roll-out plays (making it play less random and more like the greedy reflex player).

All of these did not improve its performance.

B. *Minimax Approach Expirement*

1) Stage 1: Comparing the different agents

In this stage we compared the win-rate against the benchmark Reflex agent, for all the different agents we built, with the two main heuristics, Heatmap Forward and Euclidean Distance, in depth 2 and 3. The win-rate was calculated out of hundreds of games.

As expected, Alpha-Beta and Minimax gave almost similar results. The Expectimax Agent did not accomplish good results, as it gives the "expectation nodes" the average of the score of its children in the tree, assuming the opponent will take an "average" action. Nevertheless, we know that our Reflex Agent always tries to move forward as much as possible, leading into a good heuristic score both in Heatmap Forward and Euclidean Distance. When the Max node tries to take the maximizing action, it is based on poor evaluations of the expectation nodes, and the agent is not playing well, with around 53% win-rate. The Iterative Deepening gets the best results, winning games that sometimes the regular Minimax Agent fails to win, as mentioned in section II, B.

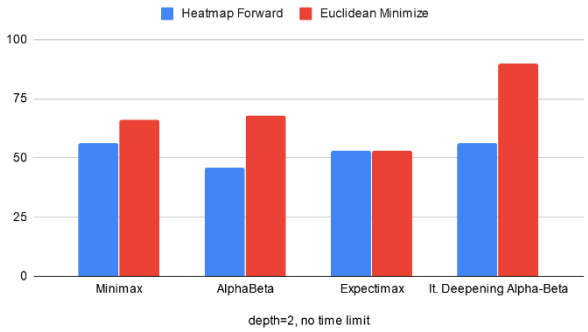

Figure 3.4 - win-rate against benchmark agent – depth 2

2) Stage 2: Combining Heuristics

On our journey towards finding the optimal agent, we decided to move on the next stage of the experiment with the Alpha Beta Agent, and the Iterative Deepening Alpha Beta Agent, as they got the best results from stage 1, and from MCTS experiment.

We needed to find the best combination of heuristics that gives optimal results. In order to find the weight that should be given to each function, we needed to analyze the functions according to the values they return and the importance of the heuristic.

Note: in this paper we will use the following notation for describing linear combination of heuristic funtions:

$$h_1 * w_1 + \cdots + h_n * w_n = h1(w1), \ldots, h2(w2)$$

In stage 2, we created 8 combinations of heuristics, and tuned their weights differently according to the different environment (depth, agent). We wanted to inspect who performs the best versus the bench-mark agent.

H1	DIAGONAL(0.25), EUCLIDEAN(1)
H2	DIAGONAL(0.25), HEATMAP(1)
H3	DIAGONAL(0.25), EUCLIDEAN(1), LONE-WOLF(0.25)
H4	DIAGONAL(2.5), HEAT-MAP(1), LONE-WOLF(0.2)
H5	EUCLIDEAN(1),HEATMAP(0.25), DIAGONAL(0.25)
H6	EUCLIDEAN(1), LONEWOLF(0.14)
H7	HEATMAP(1), LONEWOLF(1.9)
H8	EUCLIDEAN(2.2), HEATMAP(1)

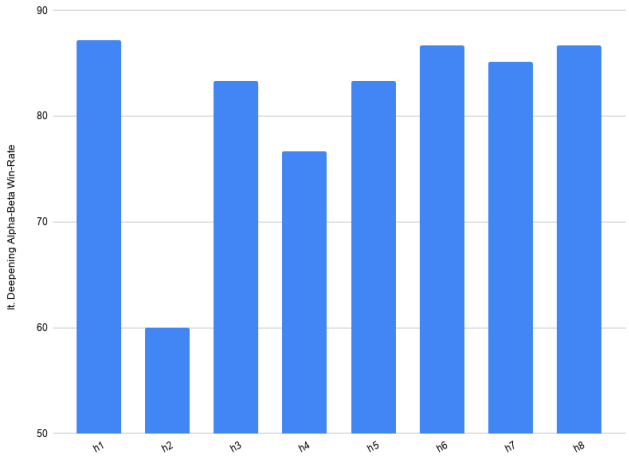

Figure 3.5 - Iterative Deepening Alpha-Beta depth 3 win-rate (in %)

For example, figure 3.5 shows the win-rate for the Iterative Deepening Agent, depth 3 with the different combinations. It is clear to see that the combination between the Main heuristics and the helper heuristics has led to good results and the win-rate percentage is much higher compared to single-heuristic functions. We also saw that too complex functions involving 3 or more heuristics performed worse. We finished this stage with 4 best contestants, that got the highest scores:

- *Euclidean(1), Lone-wolf(0.14), depth 2 – 100%*
- *Diagonal (0.25), Euclidean(1), depth 3 – 87.2%*
- *Heatmap (1), Euclidean (2.2), depth 3 – 86.6%*
- *Heatmap (1), Lone-wolf(1.9), depth 3 – 85.1%*

All used by **Iterative Deepening Alpha-Beta** agent.

We were specifically interested about *Euclidean(1), Lone-wolf(0.14), depth 2*, and tried to understand why the depth-2 (100% win-rate) agent preforms better than the depth-3 agent (74% win-rate). After watching several games of them playing against each other, we realized that the depth-3 agent, which uses the Move-Ignoring feature, sometimes skips good moves, according to this specific

evaluation function, which is exactly the 'heavy price' discussed in section II, a.

In most cases depth-3 agents performed better than their respective depth-2 agents, which is not totally trivial due to the fact we used Move-Ignoring feature in depth 3.

3) Stage 3: Finding the Optimal Agent

We could have finished at stage 2 and announce a winner, yet we wanted to make sure our observations are correct even when we let the agents play one against each other and not only versus the benchmark agent. Our experiment this time was a tournament between all the 4 agents from stage 2. Each one of the agents played 300 games against the others, and the winner of the tournament is the one who simply wins in most games.

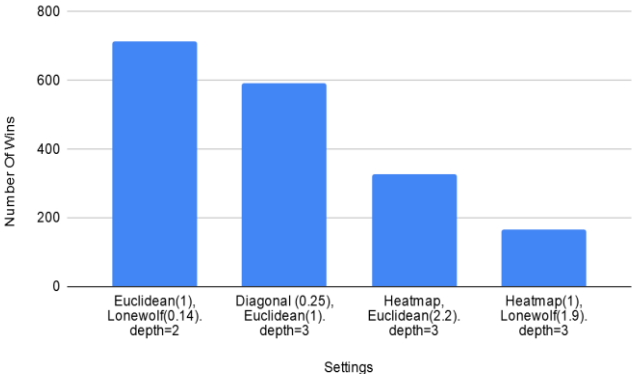

Figure 3.6 – Stage 3 tournament - number of wins

The results were the same, with *Euclidean(1), Lone-wolf(0.14), depth 2* winning over 89% of its games.

IV. Conclusions

Optimizing the problem was not an easy task. From our results we could learn that using Alpha-Beta Pruning, Move-Ordering, and Move-Ignoring reduced the computation time significantly and allowed us exploring deeper in the game-tree.

Furthermore, the fact that in most cases depth-3 agents performed better than their respective depth-2 agents leads us to the following conclusion: in most cases, using Move-Ignoring did not affect depth 3 search in the Minimax algorithms, the computation-time vs quality of search trade-off was beneficial. In other words, ignoring some of the actions has a price, but in most cases, it was worthwhile, allowing the agent the compute its moves in reasonable amount of time.

The use of Iterative Deepening approach performed better than the regular minimax approach in all our experiments. We conclude that this observation is relevant not only for the Halma game but for all "race" game with same features, where reaching the winning state can sometimes be achieved in several ways, and we would take the shortest option.

We hypothesized couple reasons for Monte Carlo Agent relative failure: Even with our data base, MCTS doesn't utilize a lot of information it acquired over previous turns. Not having such useful prior knowledge can lead the agent to constantly changing the values of previously not promising states in every turn and for some reason choose them. This can also be described as the agent 'forgetting' how bad some states and actions were and mistakenly choose them when exploring new nodes. Moreover, being unable to take advantage of previous information might lead the agent to do greedy moves. This was evident especially playing against the reflex player – MCTS' start was promising, it was advancing its pieces similarly to the reflex agent, but when the game reached its middle, we saw that MCTS usually does locally good moves by advancing its front pieces quickly to the opposing camp, but left the rest scattered behind.

Our optimal agent is depth-2 Iterative Deepening Alpha-Beta Agent with evaluation function: *Euclidean(1), Lone-wolf(0.14)* winning over 89% of its games in the final tournament. This heuristic function combines between the strategy of minimizing the distance to the goal as well as avoiding from pieces staying "alone" on the board. Monks, the inventor of the game, warns in his original patent from leaving stragglers behind[2], as this leads to a single piece advancing the entire board with no hops. We find it fascinating to see that our winning agent somewhat demonstrates it more than hundred years later.

V. Summary and Future Work

In this project we have approached the problem of building an AI Agent for the game of Halma. We have shown the challenges of handling huge state space and slow-decreasing action space, and how we solved them. We have presented different approaches to solve the game using different AI agents and discussed the advantages and disadvantages of them. We combined real life strategies into sophisticated, weighted heuristic functions.

Future research might extend the explanations of the following topics:

- Weights tuning – using learning algorithms, local search algorithms such as Gradient Ascent, or using genetic algorithms.
- Monte Carlo – improving the performance of the MCTS agent with neural network state evaluation.
- Transposition Tables [7] for storing states and reduce computation time.

References

[1] En.wikipedia.org. 2020. *Halma*. [online] Available at: <https://en.wikipedia.org/wiki/Halma>

[2] D. G. Walker, "A book of historic board games", 2009.

[3] Z. Liua, M. Zhoua, W. Caoa, Q. Qua, H. Wing Fung Yeunga, V. Yuk Ying Chung, "Towards understanding chinese checkers with heuristics, monte carlo tree Search,and deep reinforcement learning", 2006.

[4] Kocsis, Levente ,Szepesv´ari, Csaba, “Bandit based Monte-Carlo Planning”, Computer and Automation Research Institute of the Hungarian Academy of Sciences. Budapest, 2006.

[5] WINANDS, MARK H.M. A, and CHASLOT, GUILLAUME M.J-B., “PROGRESSIVE STRATEGIES FOR MONTE-CARLO TREE SEARCH”, MICC-IKAT, Games and AI Group, Faculty of Humanities and Sciences Universiteit Maastricht. Maastricht, 2008.

[6] Nijssen, J (PIM) A. M., and Winands, Mark H. M., “Enhancements for Multi-Player Monte-Carlo Tree Search”, Games and AI Group, Department of Knowledge Engineering. Maastricht, 2010

[7] Chessprogramming.org. 2020. *Transposition Table - Chessprogramming Wiki*. [online] Available at: <https://www.chessprogramming.org/Transposition_Table>

[8] Ludii.games. 2020. *Ludii Portal*. [online] Available at: <https://ludii.games/details.php?keyword=Halma>