INTRODUCTION TO ARTIFICIAL INTELLIGENCE

- PROJECT FINAL REPORT-

Topic

Chess Game using Minimax Algorithm with Alpha-Beta Pruning

I) Team Information:

Name	Student ID	Email
Hoang Trung Thanh	V202100516	21thanh.ht@vinuni.edu.vn
Tran Anh Vu	V202100569	21vu.ta@vinuni.edu.vn
Nguyen Nhat Minh	V202100570	21minh.nn2@vinuni.edu.vn

II) Contributions:

- Hoang Trung Thanh: Initialized the necessary variables and data types and generated the chessboard using Python libraries. Implemented the minimax algorithm, negamax algorithm with alpha-beta pruning, as well as multiple improvements: MTD(f), Iterative Deepening, Transposition Table, and PV-Move Ordering. Assisted in writing the project proposal, milestone report, and final report.
- Tran Anh Vu: Implemented basic chess games. Implemented move ordering function which sorts the priority of the current move without searching. Assisted in writing the project proposal, milestone report, and final report.
- Nguyen Nhat Minh: Assisted in implementing the basic chess game and the user interface. Explored alternative search algorithms to improve the game's efficiency. Assisted in writing the project proposal, milestone report, and final report.

III) Project Description:

1) Minimum requirements:

- Develop a chess game that can be played in two modes: PvP (Player vs. Player) and Player vs. Computer.
- Ensure that the time spent on choosing the next move by the chess engine does not exceed a reasonable level (3 seconds per move).
- Achieve a winning rate against an average player (approximately 800-900 Elo rating).

2) Further expectations:

- Create a chess engine with a lower time complexity compared to existing chess engines worldwide.
- Design a better algorithm for chess and other zero-sum games.

IV) Implementation Details:

1) Evaluation function:

- We researched to find the proper evaluation function, however, the efficiency with the self-implemented evaluation function did not satisfy our requirements of time complexity in acceptable depth 4 of the minimax search. Therefore, we use the evaluation function to get the utility of a board state based on material balance (which we also use in the move_ordering function), pawns structure, king safety, tactical opportunities, and piece mobility.

2) Minimax Algorithm with Alpha-Beta Pruning:

- We implemented the Minimax algorithm with Alpha-Beta pruning to determine the best move for the chess engine. The recursive nature of the algorithm allowed us to search the game tree and evaluate positions efficiently. Alpha-Beta pruning significantly reduced the number of positions to be evaluated, improving the algorithm's efficiency.

V) Challenges:

1) The complexity of chess rules and special moves in devising the evaluation function:

- Designing an effective evaluation function for chess proved to be time-consuming and complicated due to the game's high complexity
- While we initially relied on the Stockfish engine to determine the board score, its performance was unsatisfactory. Therefore, we needed to find a way to implement our own evaluation function.

2) Low depth value due to high branching factor:

- Chess has a high branching factor, with players having around 20 or more moves in the early and midgame.
- Although Alpha-Beta pruning reduced the branching factor, our machine was still unable to handle more than a depth of 4 within a reasonable time.
- We aimed to reduce the branching factor further to improve the depth and computational efficiency.

VI) Implemented Solution:

1) Negamax Implementation:

- We take advantage of the following equivalence:

$$\max\{\min\{x1, x2,\ldots\}, \min\{y1, y2,\ldots\},\ldots\}$$
$$= \max\{ - \max\{x1, x2,\ldots\}, - \max\{y1, y2,\ldots\},\ldots\}$$

- Each time the depth is reduced by 1, we change our perspective to the other player and recursively maximize the other player’s result. From the maximum results of the other player with each of this player’s moves, we negate all of them and find the maximum from our point of view as the maximizing player. As such, both players are maximizing players, in the sense that they maximize the result from their point of view (If from Black POV, a positive value means black winning, and if from White POV, a positive value means white winning, and vice versa). This can be done as the

result of the chess game being a zero-sum game. Also, with Alpha-Beta, we must change (alpha, beta) to (-beta, -alpha) in the other players' POV. The advantage is that it has the same complexity as Minimax, with easier implementation.

➢ **The implementation is in AlphaBetaWithMemory method of the Agent class in the components.py module**

2) Transposition Table:

- In the process of searching the whole alpha-beta tree, we will be able to come across the same chess formation multiple times. Theoretically, without one side capturing the others' pieces, with each side moving N times, we can encounter up to (N! * N!) the same chess formation in the Alpha-Beta tree. As a result, it is wise to store the board states with the values that we have recorded before in the transition table to save processing time. However, as we are using Alpha-Beta pruning, the value of alpha and beta could be different each time we encounter a visited board, as such, the saved value in the transposition table could only be a lower bound or upper bound in the case of beta-cutoff.
- In Python, we use a Dictionary (Hash Table) for implementing the Transposition Table. The key for each entry is the Zobrist hash value of each chess formation, which is a fast way of hashing a board formation. Collision chance is around 2^(-32), so the chance of collision happens is extremely low, and when it happens, the effect will be negligible in most cases, so we can ignore the collision problem. The entry should save: The depth of search at that time, the flag: upper bound, lower bound, or Exact, the value goes with the flag, and the best move found at that depth,

➢ **The implementation is in the AlphaBetaWithMemory method of the Agent class in the components.py module.**

3) MTD(f) (Memory-Enhanced Test Driver)

- MTD(f) is a clever way of applying Alpha-Beta Search. Normally, we initiate Alpha = -infinity, beta = infinity. However, MTD(f) uses Zero-Window Searches. In Zero-Window Searches, we set alpha = beta - 1, beta = beta. In that way, the window is extremely small, which results in a cutoff in most cases when we perform AlphaBetaWithMemory. However, due to a very small search window, the search can only return the bound on the minimax value. This seems incorrect, however, we take advantage of this by performing Zero-Window Search multiple times and continuously updating beta with the returned value from Zero-Window Searches, as well as the upper bound and lower bound, and when Lowerbound > Upperbound, we return the true value of Minimax.

➢ **The implementation is in the MTDF method of the Agent class in the components.py module.**

- One of the strongest and most significant features of MTD(f) is that it requires us to input an initial guess (f) for the minimax value. The closer (f) is to the true minimax value, the faster MTD(f) is. That's when Iterative Deepening Framework shows its power.

4) Iterative Deepening Framework

- Normally, Iterative Deepening Framework is only used for taking advantage of both Depth-First Search and Breadth-First Search: The speed of BFS and the memory saving of DFS. However, when we combine IDF with MTD(f), it becomes a very powerful tool to increase efficiency. The idea is: When we search at a lower depth using IDF, we save the return value of MTD(f), and use it for the guess of a higher-depth MTD(f). The idea is that the most recent lower-depth minimax value (MTD(f) value) will likely have a value similar to the next-depth minimax value:

MTD(f, depth - 1) ≈ MTD(f, depth)

- Because the lower depth of MTD(f) is calculated in little time, so the actual time taken decreases massively, by having a strong first guess for the final depth MTD(f).

➢ **The implementation is in the Iterative Deepening method of the Agent class in the components.py module.**

5) PV-Move Ordering and Heuristic Move Ordering: Re-order the list of available moves to fasten the cut-off process

a) PV-Move Ordering:

- We see that, in the transposition table, we save the best move in a specific chess formation. The idea is that we will consider that move first the next time we perform on the same formation. That will likely be one of the best moves, and it will lead to a massive cutoff as we consider better moves first in the tree.

b) Heuristic Move Ordering: Evaluate moves based on simple criteria:

- Value of moves: We assign values to pieces (Pawns: 10, Bishop: 30, Knight: 30, Rook: 50, Queen: 90) except the Kings because they can not be captured. The value of moves is calculated by

Extra_Value = [Value of piece being captured - Value of piece capturing] + 100

(to guarantee that moves are still prioritized even if it captures a less valuable piece and the subtraction returns a negative value)

- Potential Occupied Space: Prioritize moves that can create more moves in the next turn

Extra_Space = len(List of moves in next turn) - len (List of legal moves)

- Ability to occupy the center: Occupying the board's center will be an advantage because it widens the set of available moves as well as opportunities to get to the end-game situation so that being in the center gives the agent more choices of tactics.
- Ability to give a check, promotion, or castling: Prioritize moves that lead to a promotion, castling, or check to the opponent
- Final equation:

Priority = Extra_Value*10 + Extra_space*5 + 50 if giving a check, promotion, or castling

➢ **The implementation is in the move_ordering function in the components.py module.**

VII) Results & Analysis

- We collect 50 random chess formations in the form of Forsyth-Edwards Notation (FEN) strings and we have benchmarked our original Alpha-Beta minimax algorithm and the improved AlphaBeta Search algorithm (with Transposition Table, Heuristic Move Ordering, MTD(f), Iterative Deepening, PV-Move ordering, Negamax)

Depth	Average number of nodes traversed			Average time taken (s)		
	Pure Alpha-Beta	Improved Alpha-Beta	Ratio	Pure Alpha-Beta	Improved Alpha-Beta	Ratio
3	8587.14	1827.64	4.7	4.1009	0.7812	5.25
4	64048.5	3922.92	16.33	33.1538	1.781	18.62
5	Too high	30168.58		Too long	14.33	

Time Analysis between standard and improved Alpha-Beta Search Algorithm

- We have also tested our AI agent with the strongest chess engine such as Stockfish, Lc0, etc. The results are quite impressive: The accuracy of the chess engine falls into the range between 95 and 99, which translates to the ELO at least in the range of 2000-2200, which is the Candidate Masters (CM) Level. As a result, it has far exceeded our expectations, which is a player in the range of 900 ELO.

VIII) Conclusion & Future Prospect

- In conclusion, our project aimed to build a chess game with an AI chess engine using the Minimax algorithm with Alpha-Beta pruning. We successfully implemented the basic chess game, the evaluation function, and the Minimax algorithm, together with lots of improvements, and the agent's performance exceeded our expectations.
- In the future, we aim to study more thoroughly the evaluation function. We have not been able to implement an evaluation function, instead relying on Stockfish's, as the process is tremendously complicated. We also want to port from Python to C to improve the performance, and change the board representation to bitboards for better performance also.

IX) References

https://homepages.cwi.nl/~paulk/theses/Carolus.pdf - Theoretical Reference for Chess Algorithm

https://github.com/GitHub-Traveler/Chess-AI-project - GitHub Repository

http://people.csail.mit.edu/plaat/mtdf.html#abmem - MTD(f) reference