

SSN COLLEGE OF ENGINEERING KALAVAKKAM – 603110

# Department of Computer Science and Engineering UCS2504 – FOUNDATIONS OF ARTIFICIAL INTELLIGENCE

**Team Project**

# III Year CSE (V Sem)

Project Title: Chess Engine Using Negamax Algorithm and Pygame

**Academic Year: 2023 - 24**

**Batch: 2021 - 2025**

**Faculty Incharge: Dr.S. Kavitha**

**Project Team Members:**

| **Naren Srinivasan** | **3122 21 5001 057** |
| --- | --- |
| **Makesh Vaibhav** | **3122 21 5001 048** |
| **Ashwin Ravi** | **3122 21 5001 014** |

**Problem Statement:**

Developing a simple Python Chess Engine utilizing PyGame library of Python, aimed at providing an accessible platform for players to engage in the intricacies of chess. The primary focus is to create an immersive and user-friendly interface that accommodates players of all levels, offering an interactive experience coupled with intuitive controls. Additionally, the engine should feature a robust AI opponent employing the negamax algorithm with alpha-beta pruning, enhancing gameplay by evaluating potential moves, considering various outcomes for efficient decision-making.

**Objectives:**

* To develop a Python Chess Engine with Pygame, aiming to create an immersive and user-friendly platform for players to enjoy the strategic game of chess and a visually engaging interface to facilitate an immersive and strategic gaming experience for chess enthusiasts.
* To implement a sophisticated AI opponent using the negamax/alpha-beta algorithm for challenging gameplay and enhance the engine's decision-making capabilities by evaluating potential moves, considering diverse outcomes, and incorporating a transposition table for efficiency.
* To design a user-friendly interface with intuitive mouse controls, special move handling, and an aesthetically pleasing side menu and to ensure a seamless and enjoyable gaming experience, inviting players of all levels to participate in the timeless game of chess.

**Solution:**

* Python Chess Engine with Pygame: This chess engine is a dynamic Python application developed using Pygame, a powerful library for game development. It provides a visually engaging interface for players to dive into the strategic realm of chess.
* AI Opponent: The engine incorporates an AI opponent using the negamax with alpha-beta algorithm, offering challenging gameplay. It evaluates potential moves, considers diverse outcomes, and employs a transposition table for efficient decision-making.
* User-Friendly Experience: With intuitive mouse controls, special move handling, and an aesthetically pleasing side menu, this chess engine ensures a user-friendly experience. It invites players of all levels to enjoy the timeless game of chess with ease and excitement.

**Software Requirements Specification:**

* **Functional Requirements:**

1. User Interface – pieces and board representation, highlight selected piece, show invalid moves
2. Move Generation – give relevant action space for each piece, rook moves horizontally and vertically, bishop moves diagonally, etc and enable special moves like en passant and castling
3. Gameplay – dragging and dropping pieces to move
4. AI engine – implement the negamax algorithm with alpha-beta pruning for efficient move selection.
5. Knowledge handling – keeping track of game state (current position, available moves, etc), test for terminating conditions (checkmate or stalemate)

* **Non-Functional Requirements:**

1. Performance – response time for AI moves should be reasonable, minimize latency in move generation and decision making.
2. Usability – user interface to be intuitive, providing clear instructions on using the application, ensure smooth visual experience
3. Portability – should work in different operating systems for wider accessibility
4. Reliability – minimize unexpected errors during gameplay sessions

**Design Alternatives Considered:**

1. **Minimax Algorithm:** Minimax is a fundamental algorithm used in game theory and decision making. It explores all possible moves recursively, creating a state space tree until a specified depth (not plausible to search entire tree due to computational constraints) is reached. Without alpha-beta pruning, Minimax can be computationally expensive, examining many redundant nodes and leading to slow decision-making, especially in complex games like chess.
2. **Nega Max Algorithm:** Nega Max is an extension of the minimax algorithm designed specifically for games with two players and perfect information, like chess. Nega Max is better than Minimax because it also considers the opponents best move, hence making it easier to defend pieces which minimax fails to do. However, similar to minimax, without alpha-beta pruning, it suffers from the same computational inefficiency in searching the game tree, potentially limiting its depth and performance against stronger opponents.
3. **Alpha Beta Pruning:** Alpha beta pruning is a technique employed with Minimax and Nega Max algorithms to drastically reduce computational expense. Application of pruning returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

**Testing different algorithms with Scholar’s Mate:**

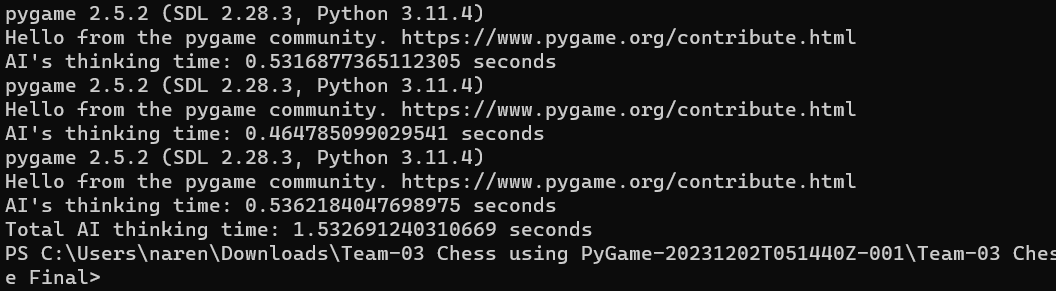
Scholar’s mate:

Scholar's Mate or Mate-in-four is a famous chess play that lets white win the game in just 4 moves. For this, white opens by moving a pawn to e4 and then a bishop to c5. After that, move your queen to h8 and then use it to capture the pawn on f7 to checkmate. We will test this set of moves with 4 different algorithms to choose the best algorithm at depth 2. This test case is chosen as it is the simplest trap that beginners fall for, so we can if the engine is at least better than a beginner chess player. Also the moves played by the user remains the same, so there is no worry of unwanted bias:

i)Minimax without alpha beta pruning:

Not able to defend:  
 No of moves taken: 3

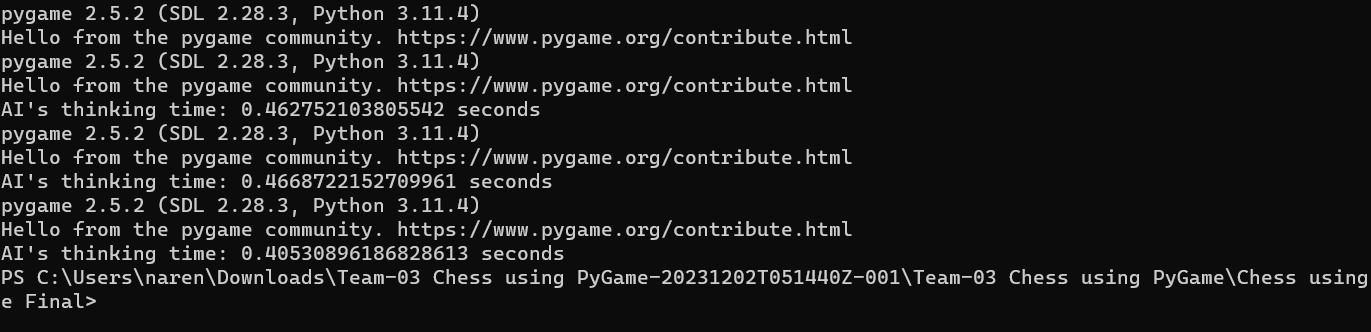
Time taken by AI to think: 1.5327 seconds





ii)Minimax with alpha beta pruning:

Not able to defend: 3  
Time take by AI to think: 1.3349 seconds



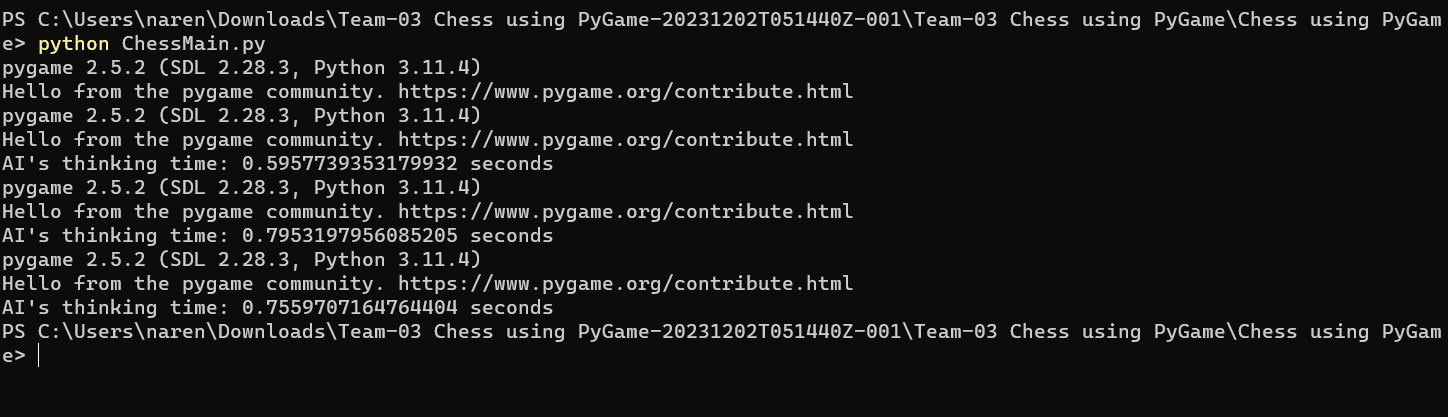


iii)Negamax without alpha beta pruning:

Successfully defended:

No of moves taken: 3

Time taken by AI to think: 2.1471 seconds



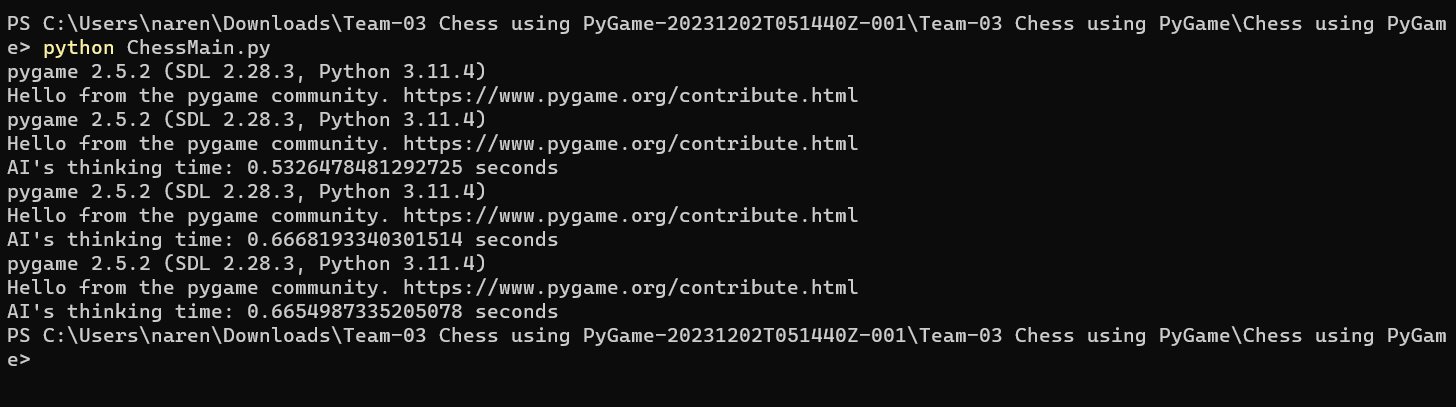


iv)Negamax with alpha beta pruning:

Successfully defended:

No of moves taken: 3

Time taken by AI to think: 1.8649 seconds





**Other Design Alternative:**

* **Deep learning:** Deep Learning involves training neural networks to recognize patterns and make predictions based on input data. In the context of a chess engine, deep learning could be used to evaluate positions or learn strategic patterns from vast amounts of game data. Neural networks could approximate the evaluation function, learning from historical games or simulations. While this approach might enhance evaluation accuracy, training requires extensive data, and the learned patterns might lack human interpretability or understanding of chess principles.

**Algorithm used:**

Each of these implemented design alternatives offers its own advantages and disadvantages. Minimax without alpha-beta pruning provides a foundational understanding of game trees but lacks efficiency. NegaMax simplifies the implementation but faces similar efficiency issues (takes more time than minimax as it also checks for opponent’s best move). After applying alpha beta pruning, there is a visible decrease in time taken for AI to think, so we can conclude that for both minimax and negamx, it is better to perform alpha-beta pruning. We will choose negamax over minimax as negamax is more effective than minimax. For chess, the effectiveness of the algorithm is to be priotized over efficiency as there is no use in being fast if one cannot win the game.

**Negamax algorithm:**

1. Initialization:

* The findBestMove function is the entry point. It initializes the next\_move variable, shuffles the list of valid moves, and calls the findMoveNegaMaxAlphaBeta function.

2. NegaMax Recursive Function:

* The findMoveNegaMaxAlphaBeta function is a recursive implementation of the NegaMax algorithm.
* It evaluates the current game state and explores possible moves up to a certain depth in the game tree.

3. Base Case - Depth Limit:

* If the depth limit is reached (depth == 0), the function returns the evaluation score of the current game state multiplied by the turn\_multiplier.
* The turn\_multiplier is 1 for white's turn and -1 for black's turn.

4. Move Ordering (TODO):

* The comment indicates a placeholder for move ordering, a technique to prioritize certain moves for more efficient pruning. This step is marked for future implementation.

5. Iteration Through Valid Moves:

* The function iterates through the provided list of valid moves.
* For each move, it makes the move on the game state, retrieves the next set of valid moves, and recursively calls itself.

6. Score Calculation:

* The score for the current move is calculated as the negation of the score returned from the recursive call.
* If the calculated score is higher than the maximum score (max\_score), it updates max\_score and, if at the initial depth, updates the next\_move variable.

8. Alpha-Beta Pruning:

* Alpha-Beta Pruning is applied to optimize the search. It maintains alpha and beta values to prune branches that won't affect the final decision.
* If the max\_score is greater than the current alpha value, alpha is updated.
* If alpha is greater than or equal to beta, the loop breaks, as further exploration is unnecessary.

9. Final Move Selection:

* The next\_move variable stores the best move found at the initial depth.
* The function returns the max\_score, representing the optimal score for the current game state.

**Alpha Beta Pruning:**

1. Initialization:

* The findBestMove function initializes the next\_move variable and shuffles the list of valid moves. It then calls the findMoveNegaMaxAlphaBeta function.

2. NegaMax with Alpha-Beta Pruning:

* The findMoveNegaMaxAlphaBeta function is a recursive implementation of the NegaMax algorithm with Alpha-Beta Pruning.

3.Base Case - Depth Limit:

* If the depth limit is reached (depth == 0), the function returns the evaluated score of the current game state multiplied by the turn\_multiplier.

4. Move Ordering (TODO):

* There is a placeholder for move ordering, marked as //TODO. Move ordering can be implemented later for more efficient pruning.

5. Alpha-Beta Pruning Loop:

* The function maintains an alpha and beta value to prune branches that won't affect the final decision.
* The alpha value represents the best value found so far for the maximizing player along the path to the root.
* The beta value represents the best value found so far for the minimizing player along the path to the root.
* The max\_score variable is used to track the best score.

6. Iteration Through Valid Moves:

* The function iterates through the provided list of valid moves.
* For each move, it makes the move on the game state, retrieves the next set of valid moves, and recursively calls itself.

7. Score Calculation:

* The score for the current move is calculated as the negation of the score returned from the recursive call.
* If the calculated score is higher than the maximum score (max\_score), it updates max\_score and, if at the initial depth, updates the next\_move variable.

8. Undo Move:

* After exploring a move, the function undoes the move on the game state to backtrack for further exploration.

9. Update Alpha and Beta:

* If the max\_score is greater than the current alpha value, alpha is updated to be the same as max\_score.
* If alpha is greater than or equal to beta, the loop breaks, as further exploration is unnecessary.

10. Return Maximum Score:

* The function returns the max\_score, representing the optimal score for the current game state.

**Technological improvement (weak AI):**

1.Improvements to the User Interface: The graphical interface can be made more user-friendly by highlighting the alternative moves when a piece is picked, so that user’s cognitive load can be minimized.

2.AI Algorithm Optimization: Improvement in the main algorithm, since it is currently being used with alpha-beta pruning. Introduction of deep learning will surely increase the decision-making abilities of the engine.

3.Levels of Difficulty: Offer varying degrees of difficulty for the gamer on the computer. A hard mode with additional optimization, a medium mode using the present technique, and an easy option with the computer making simpler movements could be made available so that user may increase the level of difficulty as he progresses.

4.Design that is responsive: Making the interface and even the algorithm responsive and working for different UI layouts and different screen sizes for faster and better usage.

5.Music and sound effects: When moves, captures, and other in-game actions occur, add sound effects. To improve the play experience, background music can be added to elevate user’s experience.

**Learning Outcomes:**

1.Achieved an understanding of the Negamax algorithm, a crucial decision-making tool in two-player games like chess.

2.Solved the undertaken problem in manageable incremental fashion.

3.Improved problem-solving knowledge by successfully tackling the difficulties of game logic, rules, and interactions within the chess-playing project.

4.Applied the learnt basic concepts of artificial intelligence to arrive at effective solutions in the context of game-playing.