

CSE 4110: Artificial Intelligence Laboratory

Shologuti: An Intelligent Game Engine with Minimax and MCTS Agents

Submitted By:

Choyan Mitra Barua Bijoy

Roll: 2007101

&

Plaban Das

Roll: 2007111

Submitted To:

Md Mehrab Hossain Opi

Lecturer

&

Waliul Islam Sumon

Lecturer



Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh

October, 2025

Contents

List of Tables	ii
List of Figures	ii
1 Introduction	1
1.1 What the Project Does?	1
1.2 How the Project is Organized?	1
1.3 Which Techniques are There?	2
1.4 Objectives	2
1.5 The Traditional One	2
1.5.1 Setup	3
1.5.2 Rules	3
2 Methodology	4
2.1 Minimax Algorithm with Alpha-Beta Pruning	4
2.1.1 Core Minimax Concept	4
2.1.2 Alpha-Beta Pruning Optimization	5
2.1.3 Implementation Details	5
2.1.4 Evaluation Function	6
2.1.5 Configurable Depth	6
2.1.6 Algorithm Implementation - Minimax	7
2.2 Monte Carlo Tree Search (MCTS)	8
2.2.1 MCTS Phases	8
2.2.2 Implementation Details	9
2.2.3 Reward Calculation	9
2.2.4 Algorithm Implementation - Monte Carlo	10
3 Gameplay and Feature Walkthrough	11
3.1 Authentication and Startup	11
3.2 Main Menu	12
3.3 Human vs AI Mode	13
3.4 AI vs AI Mode	14

3.5	Board Interactions and Features	15
3.6	Firebase Authentication Implementation	16
4	Discussion And Conclusion	17
4.1	Discussion	17
4.2	Conclusion	17
	References	17

List of Tables

1.1	Board and starting positions for sholo guti	3
2.1	Visualization of MiniMax Algorithm	5
2.2	Visualization of Monte Carlo Tree Search Algorithm	8
3.1	Login Screen	11
3.2	Create New Account Screen	11
3.3	Main Menu	12
3.4	Human Vs AI Gameplay	13
3.5	AI Vs AI Gameplay	14
3.6	Human Vs AI Gameplay	15
3.7	Firebase Authentication User Management Dashboard	16

Chapter 1

Introduction

Shologuti is an old board game that has been played for many years by people in different cultures. The game involves two players who take turns moving pieces on a board with 37 positions connected by lines. Players win by removing all of their opponent's pieces from the board. The main goal of this project is to recreate this traditional game on a computer using modern programming tools.

1.1 What the Project Does?

This project builds a playable version of Shologuti using Python and Pygame, which are popular tools for creating interactive applications. The project has three main parts. First, it sets up the game board with all the correct rules so that the game plays exactly like the original version. Second, it creates computer opponents that can play against human players. These opponents use different strategies to decide their moves. One strategy looks ahead several moves to find the best option, while the other strategy tries many possible game outcomes to make decisions. Third, the project provides a user-friendly interface where people can play the game, choose different difficulty levels, and watch computers play against each other.

1.2 How the Project is Organized?

The code is organized into separate parts, each handling a different task. One part manages the game board and piece movements. Another part handles the turn system and special rules about capturing pieces. A third part focuses on the computer players and how they make decisions. The project also includes a login system so users can create accounts, though this is optional. The whole application works on different operating systems like Windows, Mac, and Linux.

1.3 Which Techniques are There?

The Shologuti project uses several key techniques to implement the game and create intelligent opponents. The game board is represented as a graph with 37 connected nodes, allowing the system to validate moves and determine capture possibilities. The game logic implements all the traditional Shologuti rules including piece movement, capturing mechanics, and turn management. For the computer opponents, the project uses two different artificial intelligence methods: Minimax with Alpha-Beta Pruning, which thinks ahead several moves to find the best strategy, and Monte Carlo Tree Search, which simulates many random games to discover winning patterns. The user interface is built with Pygame to display the board, show piece positions, and handle player interactions. An optional Firebase authentication system allows players to create accounts and log in to the application.

1.4 Objectives

The objectives of this project are:

- Implement accurate Shologuti game mechanics with proper move validation and capture rules.
- Create two AI agents using Minimax with alpha-beta pruning and Monte Carlo Tree Search algorithms.
- Develop an interactive user interface where players can compete against AI or watch AI versus AI matches.
- Integrate Firebase authentication for secure user account management.
- Demonstrate how traditional games can be digitized while preserving authentic gameplay.

1.5 The Traditional One

Sixteen soldiers or Shologuti is a two-player abstract strategy board game from Sri Lanka. It also comes from India under the name cows and leopards. The game was documented by Henry Parker in Ancient Ceylon: An Account of the Aborigines and of Part of the Early Civilisation (1909) with the name Hēwākam Keliya or the War Game. Parker mentions that the game is also played in India and Bangladesh under the name sōlah guttiya or sixteen balls.

1.5.1 Setup

An expanded Alquerque board is used for Shologuti. Two triangle boards are attached to two opposite sides of an Alquerque board. Each player has 16 pieces that are distinguishable from the other player. Pieces are placed on the intersections (or “points”) of the board, specifically on their half of the Alquerque board, and the nearest triangular board.

1.5.2 Rules

1. Players alternate their turns.
2. A player may only use one of their pieces in a turn, and must either make a move or perform a capture but not both.
3. A piece may move onto any vacant adjacent point along a line.
4. A piece may capture an opposing piece by the short leap as in draughts or Alquerque. The piece must be adjacent to the opposing piece, and leap over it onto a vacant point immediately beyond. The leap must be in a straight line and follow the pattern on the board.
5. Captures are not mandatory. A piece can continue to capture within the same turn, and may stop capturing any time.
6. The captured piece (or pieces) is removed from the board.
7. The player who captures all of the other player’s pieces wins.

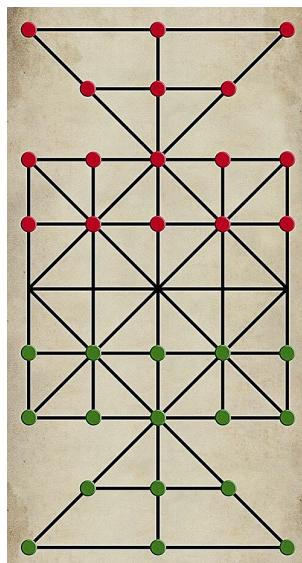


Figure 1.1: Board and starting positions for sholo guti

Chapter 2

Methodology

This section provides an in-depth explanation of the artificial intelligence techniques implemented in Shologuti. The project uses two distinct AI agents: the Minimax Agent with Alpha-Beta Pruning and the Monte Carlo Tree Search (MCTS) Agent. Both agents demonstrate different approaches to game-playing intelligence.

2.1 Minimax Algorithm with Alpha-Beta Pruning

The Minimax algorithm is a classical decision-making technique used in two-player zero-sum games. It operates under the assumption that both players play optimally, with one player attempting to maximize their advantage while the other attempts to minimize it.

2.1.1 Core Minimax Concept

The algorithm works by building a game tree where each node represents a game state and each edge represents a legal move. Starting from the current game position, the algorithm explores all possible moves and their consequences up to a specified depth. At each level of the tree, the algorithm alternates between maximizing and minimizing layers:

- **Maximizing Layer:** When it is the AI's turn, the algorithm selects the move that produces the highest score, assuming the AI wants to maximize its advantage.
- **Minimizing Layer:** When it is the opponent's turn, the algorithm assumes the opponent will choose the move that minimizes the AI's advantage.

The algorithm recursively evaluates positions by propagating values from the bottom of the tree back up to the root. Each non-terminal node's value is determined by either the maximum or minimum of its children's values, depending on whose turn it is.

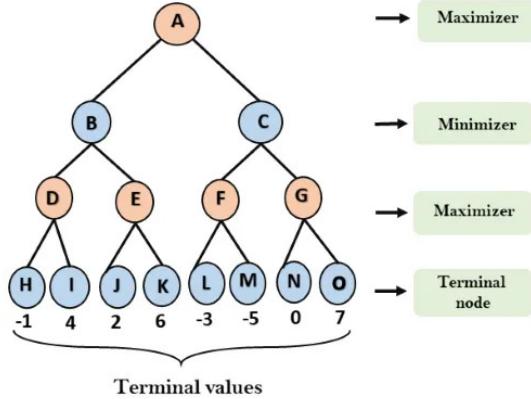


Figure 2.1: Visualization of MiniMax Algorithm

2.1.2 Alpha-Beta Pruning Optimization

Alpha-beta pruning is an optimization technique that significantly reduces the number of nodes evaluated in the game tree without affecting the final result. It maintains two values during the search:

- **Alpha (α):** The best value the maximizing player can guarantee so far. Initially set to $-\infty$.
- **Beta (β):** The best value the minimizing player can guarantee so far. Initially set to $+\infty$.

During tree traversal, when the algorithm determines that a branch cannot possibly affect the final decision (because it is already worse than alternatives found elsewhere), that branch is pruned and not explored further. Specifically:

- If a maximizing node finds a value greater than or equal to β , further exploration of that node's siblings is skipped.
- If a minimizing node finds a value less than or equal to α , further exploration of that node's siblings is skipped.

This pruning can reduce the effective branching factor dramatically, allowing the algorithm to search deeper within the same time constraints.

2.1.3 Implementation Details

The `MinimaxAgent` class implements this algorithm with the following components:

1. **Move Selection:** The `choose_move` method generates all legal moves from the current state and evaluates each by applying the move and calling the recursive minimax function.
2. **Recursive Minimax:** The `_minimax` method implements the core recursive logic. It checks for terminal states (win/loss), evaluates positions at maximum depth using the evaluation function, and recursively explores child nodes while maintaining alpha-beta bounds.
3. **Terminal State Detection:** Before exploring further, the algorithm checks if the game has ended by determining if either player has lost all pieces or if the current player has no legal moves.
4. **Move Generation:** For each state, the algorithm generates legal moves, prioritizing capture moves over simple moves (consistent with game rules). If a forced capture continuation exists, only that piece's capture moves are considered.

2.1.4 Evaluation Function

At leaf nodes, the algorithm uses an evaluation function to estimate the desirability of a position. The evaluation combines multiple strategic factors:

$$\text{Score} = 10 \times \text{Material} + \text{Mobility} + \text{Pending Bonus} \quad (2.1)$$

Where:

- **Material:** The difference between the AI's remaining pieces and the opponent's remaining pieces. Weighted heavily ($10\times$) as piece count is the most critical factor.
- **Mobility:** The difference between the number of legal moves available to the AI and the opponent. Higher mobility indicates better positional control.
- **Pending Bonus:** A small bonus (+1) when the AI has a forced capture continuation in progress, encouraging multi-capture sequences.

2.1.5 Configurable Depth

The search depth can be configured to 1, 3, 5, or 7 ply (half-moves). Higher depths provide stronger play by looking further ahead, but require exponentially more computation time. The relationship between depth and nodes explored is approximately $O(b^d)$, where b is the branching factor and d is the depth.

2.1.6 Algorithm Implementation - Minimax

Algorithm 1 Minimax Part 1: Choose Move

```

1: function CHOOSEMOVE(state,  

    depth)  

2:    moves  $\leftarrow$  Generate legal moves  

3:    best_score  $\leftarrow -\infty$   

4:    best_move  $\leftarrow$  None  

5:     $\alpha \leftarrow -\infty$   

6:     $\beta \leftarrow +\infty$   

7:    for each move in moves do  

8:      child  $\leftarrow$  Apply move  

9:      score  $\leftarrow$  Minimax(child,  

    depth - 1,  $\alpha$ ,  $\beta$ )  

10:     if score > best_score then  

11:       best_score  $\leftarrow$  score  

12:       best_move  $\leftarrow$  move  

13:        $\alpha \leftarrow \max(\alpha, score)$   

14:     end if  

15:     if  $\beta \leq \alpha$  then  

16:       break  

17:     end if  

18:    end for  

19:    return best_move  

20: end function

```

Algorithm 2 Minimax Part 2: Recursive Search

```

1: function MINIMAX(state, d,  $\alpha$ ,  $\beta$ )  

2:    if Game ended then  

3:      return Terminal value  

4:    end if  

5:    if d = 0 then  

6:      return Evaluate(state)  

7:    end if  

8:    val  $\leftarrow -\infty$   

9:    for each move do  

10:     child  $\leftarrow$  Apply move  

11:     val  $\leftarrow \max(val,$   

    Minimax(child, d - 1,  $\alpha$ ,  $\beta$ ))  

12:      $\alpha \leftarrow \max(\alpha, val)$   

13:     if  $\beta \leq \alpha$  then  

14:       break  

15:     end if  

16:    end for  

17:    return val  

18: end function  

19: function EVALUATE(state)  

20:    material  $\leftarrow my\_pieces - opp\_pieces$   

21:    mobility  $\leftarrow my\_moves - opp\_moves$   

22:    return material  $\times 10 + mobility$   

23: end function

```

2.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search is a probabilistic search algorithm that has proven highly effective in games with large branching factors. Unlike Minimax, which requires an explicit evaluation function, MCTS estimates position values through random simulations.

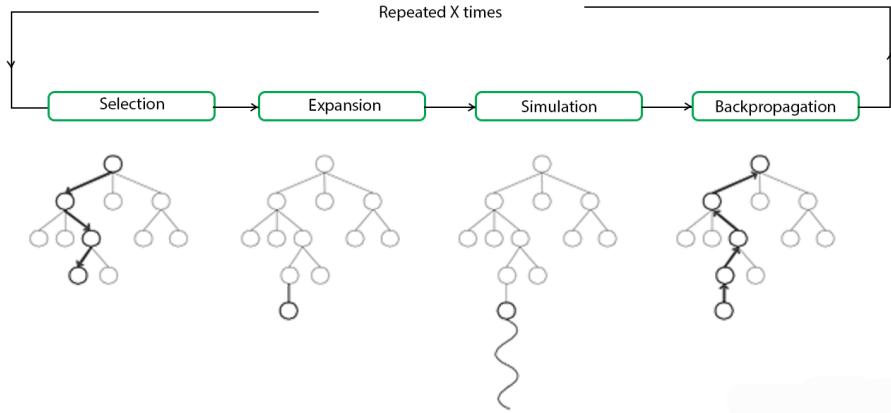


Figure 2.2: Visualization of Monte Carlo Tree Search Algorithm

2.2.1 MCTS Phases

MCTS operates in four distinct phases that repeat iteratively:

1. **Selection:** Starting from the root node (current game state), the algorithm traverses the existing tree by selecting child nodes. The selection uses the UCB1 (Upper Confidence Bound applied to Trees) formula to balance exploration and exploitation:

$$\text{UCB1}(\textit{node}) = \frac{W_i}{N_i} + c\sqrt{\frac{\ln N_p}{N_i}} \quad (2.2)$$

Where:

- W_i = number of wins from node i
- N_i = number of visits to node i
- N_p = number of visits to parent node
- c = exploration constant (default: $\sqrt{2}$)

The exploitation term $\frac{W_i}{N_i}$ favors nodes with high win rates, while the exploration term $c\sqrt{\frac{\ln N_p}{N_i}}$ encourages visiting less-explored nodes.

2. **Expansion:** When a node with unexplored children is reached, the algorithm selects one untried move randomly and creates a new child node by applying that move to the parent's game state.
3. **Simulation (Rollout):** From the newly expanded node, the algorithm performs a random playout where both players make random legal moves until the game ends or a maximum step limit (200 moves) is reached. The outcome (win, loss, or draw) is recorded.
4. **Backpropagation:** The simulation result is propagated backward through the tree path from the expanded node to the root. Each node along the path increments its visit count and updates its win count based on the simulation outcome.

2.2.2 Implementation Details

The `MCTSAgent` class implements the algorithm with the following structure:

1. **Tree Node Structure:** The `_MCTSNode` class represents a node in the search tree, storing the game state, parent reference, move that led to this state, list of child nodes, untried moves, visit count, and win count.
2. **Iteration Loop:** The `choose_move` method runs the four-phase process for a configurable number of iterations (50, 100, 200, or 300). More iterations generally produce better move quality but require more computation time.
3. **Move Selection Strategy:** After all iterations complete, the algorithm selects the child of the root with the highest visit count (not win rate). This approach is more robust than selecting by win rate alone, as highly-visited nodes have more statistical confidence.
4. **Rollout Strategy:** During simulation, moves are chosen uniformly at random from all legal moves. The rollout terminates when a winner is determined or the 200-move limit is reached (resulting in a draw outcome of 0.5).

2.2.3 Reward Calculation

The reward assigned during backpropagation depends on the simulation outcome from the perspective of the AI player:

- AI wins: reward = 1.0
- AI loses: reward = 0.0
- Draw or timeout: reward = 0.5

2.2.4 Algorithm Implementation - Monte Carlo

Algorithm 3 MCTS Part 1: Choose Move

```

1: function CHOOSEMOVE(state,
   iterations)
2:   root  $\leftarrow$  New node(state)
3:   for i = 1 to iterations do
4:     node  $\leftarrow$  root
            $\triangleright$  Selection Phase
5:     while node fully expanded and
       not terminal do
6:       node  $\leftarrow$  Select best child
7:     end while
            $\triangleright$  Expansion Phase
8:     if node has untried moves then
9:       move  $\leftarrow$  Random untried
       move
10:      new_state  $\leftarrow$  Apply move
11:      new_node  $\leftarrow$  New
       node(new_state)
12:      Add new_node as child
13:      node  $\leftarrow$  new_node
14:    end if
            $\triangleright$  Simulation Phase
15:    reward  $\leftarrow$  Rollout(node.state)
            $\triangleright$  Backpropagation Phase
16:    while node  $\neq$  None do
17:      node.visits  $\leftarrow$  node.visits +
         1
18:      node.wins  $\leftarrow$  node.wins +
         reward
19:      node  $\leftarrow$  node.parent
20:    end while
21:  end for
22:  best  $\leftarrow$  Child with max visits
23:  return best.move
24: end function

```

Algorithm 4 MCTS Part 2: Helper Functions

```

1: function BESTCHILD(node,
   exploration_const)
2:   best_score  $\leftarrow$   $-\infty$ 
3:   best_child  $\leftarrow$  None
4:   for each child in node.children do
5:     exploit  $\leftarrow$   $\frac{\text{child.wins}}{\text{child.visits}}$ 
6:     explore  $\leftarrow c \times \sqrt{\frac{\ln(\text{node.visits})}{\text{child.visits}}}$ 
7:     score  $\leftarrow$  exploit + explore
8:     if score > best_score then
9:       best_score  $\leftarrow$  score
10:      best_child  $\leftarrow$  child
11:    end if
12:   end for
13:   return best_child
14: end function
15: function ROLLOUT(state)
16:   sim  $\leftarrow$  Copy(state)
17:   steps  $\leftarrow$  0
18:   max_steps  $\leftarrow$  200
19:   while steps < max_steps do
20:     if Game ended then
21:       return result
22:     end if
23:     moves  $\leftarrow$  Legal moves
24:     move  $\leftarrow$  Random choice
25:     Apply move to sim
26:     steps  $\leftarrow$  steps + 1
27:   end while
28:   return 0.5 (draw)
29: end function

```

Chapter 3

Gameplay and Feature Walkthrough

This section describes how users interact with Shologuti, from the initial authentication step through the different game modes and available features.

3.1 Authentication and Startup

To play Shologuti, users must create an account or log in with existing credentials. The application uses Firebase Authentication, a cloud-based authentication service, to securely manage user accounts and credentials. The application provides two authentication options at startup:

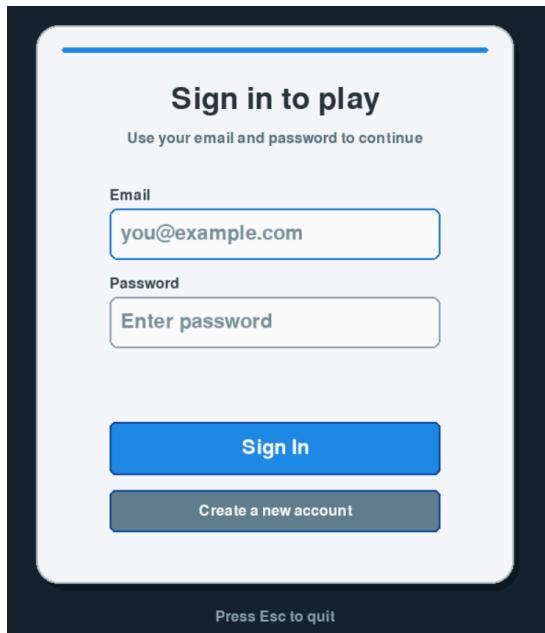


Figure 3.1: Login Screen

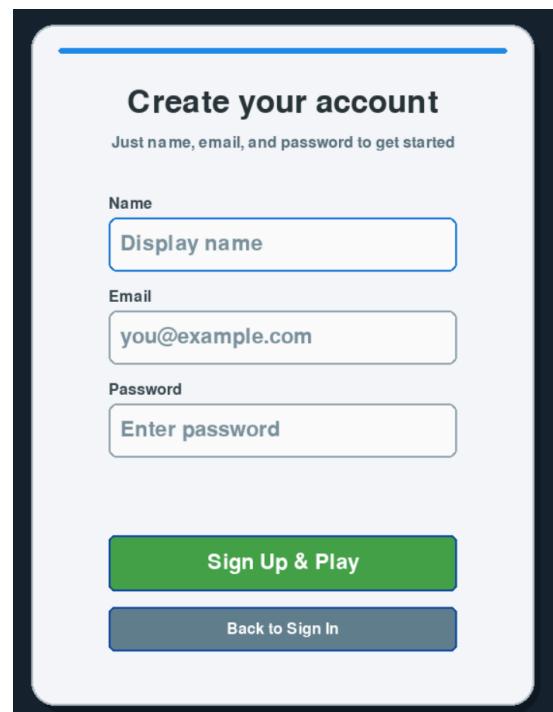


Figure 3.2: Create New Account Screen

3.2 Main Menu

Once the application is running, users reach the main menu screen. From here, three game mode options are presented:

- **Human vs AI:** Play a match against one of the computer opponents.
- **AI vs AI:** Watch two computer agents compete against each other.
- **Logout:** Exit the current session and return to the authentication screen.

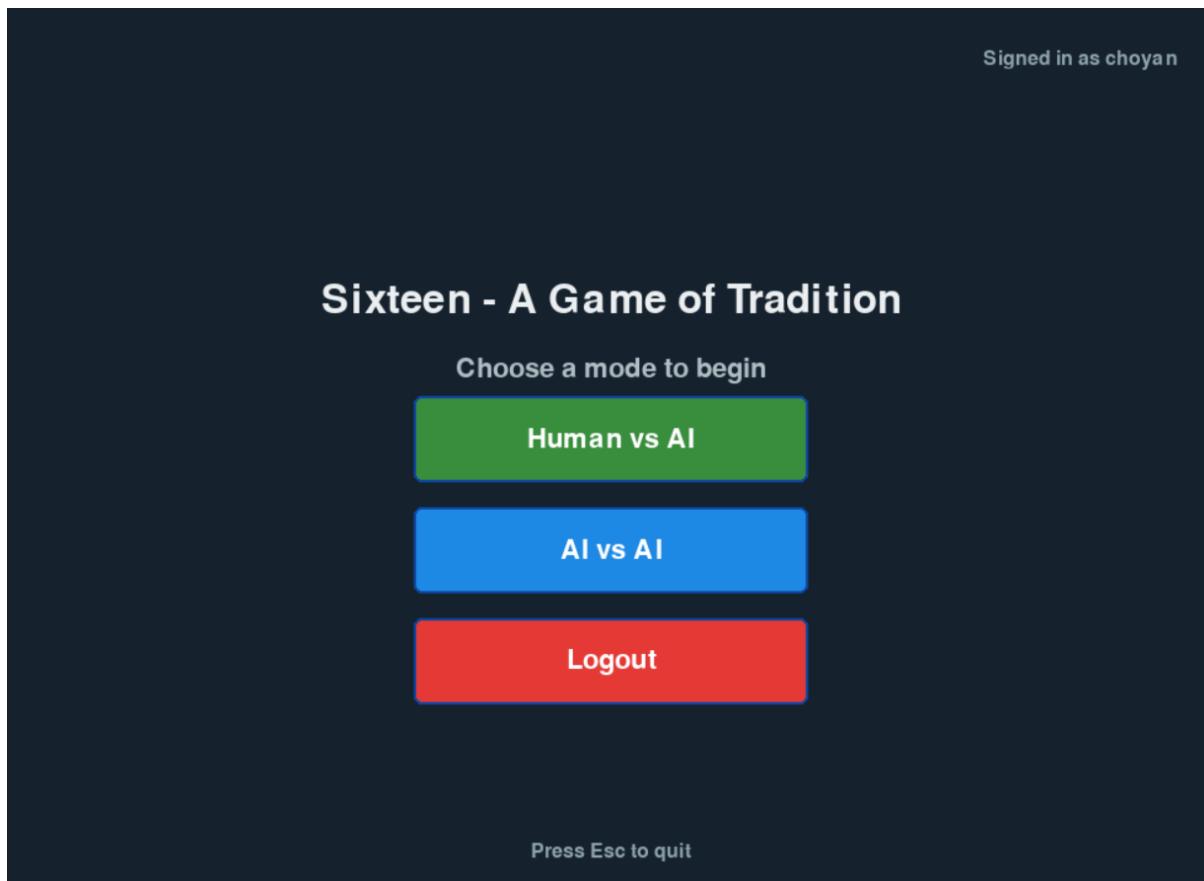


Figure 3.3: Main Menu

Users can press Escape at any time to return to the main menu. Pressing Escape again from the main menu exits the application completely.

3.3 Human vs AI Mode

In Human vs AI mode, a player competes against the computer opponent using the Minimax algorithm. Before starting the match, the player selects which color to control. Green plays from the bottom of the board and moves first. Red plays from the top of the board.

During gameplay, several control buttons appear on the sidebar:

- **New Game:** Reset the board to the beginning and start a fresh match.
- **Undo:** Take back the previous move and return to the prior board state.
- **Adjust Difficulty:** Change the Minimax search depth to 1, 3, 5, or 7 levels. Higher levels produce stronger play but require more computation time.
- **Toggle Color:** Switch control between Red and Green pieces during the game.
- **Return to Menu:** Exit the current match and return to the main menu.

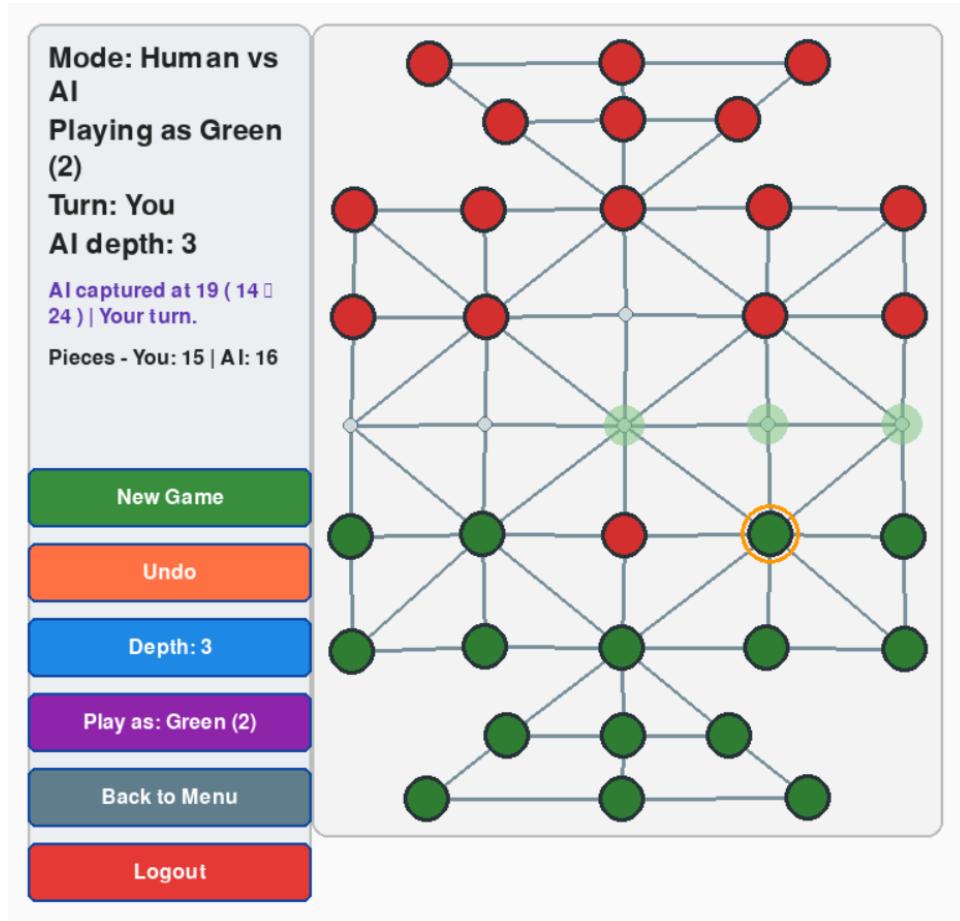


Figure 3.4: Human Vs AI Gameplay

The computer opponent respects all game rules, including forced capture requirements. When a capture move is available, the AI automatically performs it. If a capture lands on a position where another capture is possible, the AI continues capturing until no more captures remain from that piece's position.

3.4 AI vs AI Mode

In AI vs AI mode, two computer agents compete against each other while the player observes the match. Green uses the Minimax algorithm and Red uses the Monte Carlo Tree Search (MCTS) algorithm. This mode demonstrates different AI strategies and decision-making approaches.

The difficulty of each agent can be adjusted independently:

- **Green (Minimax):** Search depth can be set to 1, 3, 5, or 7 levels. Higher depths produce stronger and more deliberate play.
- **Red (MCTS):** Iteration count can be set to 50, 100, 200, or 300 simulations. More iterations produce better decision quality.

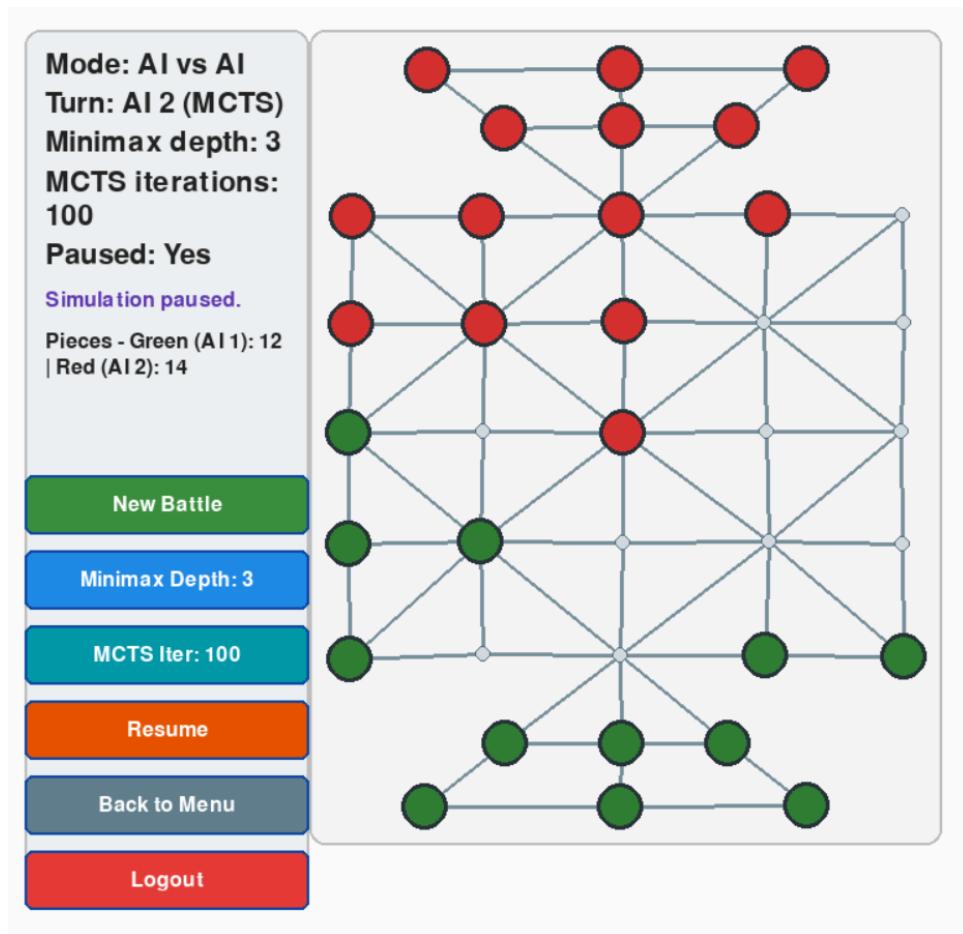


Figure 3.5: AI Vs AI Gameplay

The sidebar provides controls to manage the AI match:

- **New Match:** Start a fresh game with the current AI settings.
- **Adjust Green Difficulty:** Change the Minimax search depth.
- **Adjust Red Difficulty:** Change the MCTS iteration count.
- **Pause/Resume:** Stop or resume the automatic move execution to examine the board.
- **Return to Menu:** Exit the AI observation and return to the main menu.

3.5 Board Interactions and Features

The board interface provides several interactive features to guide gameplay and provide visual feedback:

- **Move Selection:** Click on your own piece to reveal all legal moves available for that piece. Capture moves are highlighted in red, while simple moves to empty adjacent positions are highlighted in green.

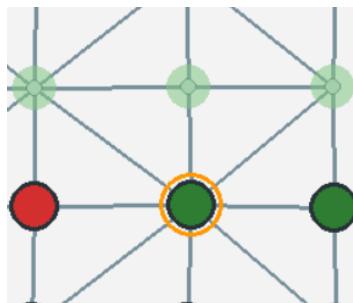
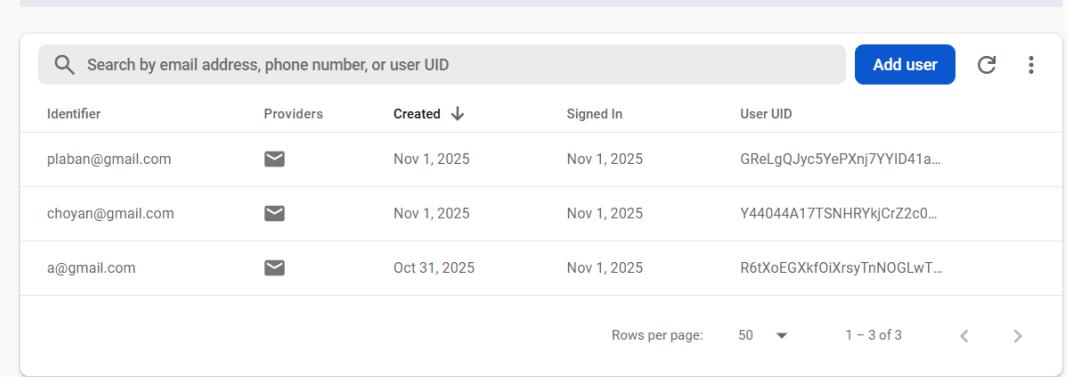


Figure 3.6: Human Vs AI Gameplay

- **Forced Capture Chains:** When a capture move lands on a position where another capture is possible, the piece is locked for further capturing. The player cannot switch to a different piece until the capture chain ends. The system prevents illegal moves by automatically limiting the selection to the capturing piece.
- **Move History and Undo:** The application maintains a history of the last 40 game states. In Human vs AI matches, players can use the undo button to take back moves one step at a time and explore alternative strategies without penalty.

3.6 Firebase Authentication Implementation

Shologuti uses Firebase Authentication to manage user accounts and login credentials. Users can register with email and password, which are stored securely in the Firebase database. The system assigns each user a unique identifier (UID) and manages authentication tokens for session management. This centralized approach ensures secure credential management and allows users to access the application from different devices.



Identifier	Providers	Created ↓	Signed In	User UID
plaban@gmail.com	✉	Nov 1, 2025	Nov 1, 2025	GReLgQJyc5YePXnj7YYID41a...
choyan@gmail.com	✉	Nov 1, 2025	Nov 1, 2025	Y44044A17TSNHRYkjCrZ2c0...
a@gmail.com	✉	Oct 31, 2025	Nov 1, 2025	R6tXoEGXkfOixrsyTnNOGLwT...

Figure 3.7: Firebase Authentication User Management Dashboard

Chapter 4

Discussion And Conclusion

4.1 Discussion

In this project, we developed Shologuti, a digital implementation of a traditional board game with intelligent computer opponents. We implemented accurate game mechanics on a 37-node board with proper move validation and forced capture rules. Two AI agents were created: one using Minimax with alpha-beta pruning and another using Monte Carlo Tree Search, both providing different playing styles and difficulty levels. The game features a Pygame user interface with interactive board controls, move highlighting, and sidebar settings. We integrated optional Firebase Authentication for user account management. The project demonstrates how traditional games can be successfully digitized while maintaining authentic gameplay and showcases practical application of advanced game AI techniques and clean software architecture.

4.2 Conclusion

This project successfully implements Shologuti as a playable game with intelligent AI opponents and an interactive interface. Future work will include implementing a Deep Q-Network agent that uses machine learning to improve its play through experience. Additionally, a fuzzy logic system will be integrated to automatically adjust the AI difficulty based on player performance, ensuring the game remains challenging but fair for players of all skill levels.

Bibliography

- [1] “How to Play Sholo Guti.” Roll the Dice. Available at: <https://rollthedice.in/pages/how-to-play-sholo-gutti>
- [2] “Sixteen Soldiers - Learn How To Play.” GameRules.com. Available at: <https://gamerules.com/rules/sixteen-soldiers/>
- [3] “Sixteen Soldiers.” Wikipedia. Available at: https://en.wikipedia.org/wiki/Sixteen_soldiers
- [4] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. “A Survey of Monte Carlo Tree Search Methods.” IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, 2012.
- [5] Pygame Foundation. “Pygame: Cross-platform set of Python modules.” Available at: <https://www.pygame.org>
- [6] Google Firebase. “Firebase Authentication: Secure user access.” Available at: <https://firebase.google.com/docs/auth>
- [7] Guido van Rossum and the Python Software Foundation. “Python Programming Language.” Available at: <https://www.python.org>