

KnightBot

By

Michael Ibrahim

CPSC-4900-01 Senior Project and Seminar

Instructor

Jianwei Hao



*Governors State University
University Park, IL 60484*

ABSTRACT

KnightBot: Interactive Chess Game with Intelligent AI Opponent

This project delivers an accessible yet sophisticated chess application that combines the timeless strategy of chess with modern artificial intelligence. By implementing an intelligent chess-playing algorithm, we've created an experience that helps players of all skill levels improve their game while enjoying the challenge of a responsive, thinking opponent.

Why This Project Matters

Chess has long been recognized as a powerful tool for developing critical thinking, planning, and decision-making skills. However, finding appropriate human opponents can be challenging, and many existing chess applications either offer simplistic AI or require significant computing resources. Our implementation bridges this gap by providing:

1. An accessible chess experience that runs entirely in the browser
2. An intelligent opponent that adapts to different phases of the game
3. Educational features that help players understand the strategic thinking behind each move

Key Observation

We observed that most web-based chess applications sacrifice either sophistication or accessibility—either offering basic AI that doesn't challenge intermediate players, or requiring substantial resources that limit availability on mobile or older devices. By optimizing the minimax algorithm with alpha-beta pruning and implementing a phase-based evaluation system, we've created an AI that makes strategic decisions similar to human players while remaining performant across devices.

Main Results

Our chess application achieves several notable outcomes:

1. **Intelligent AI Opponent:** The implementation successfully evaluates positions considering not just material value but positional factors like piece placement, king safety, and pawn structure—similar to human chess understanding.
2. **Learning Platform:** Through move analysis visualization and "best move" suggestions, the application serves as an interactive learning tool, helping players understand the strategic concepts behind strong chess play.
3. **Accessible Technology:** By leveraging client-side technology with no backend dependencies, the game can be played anywhere with a web browser, while the ability to share positions via URL enables collaboration without requiring accounts or servers.

This project demonstrates how thoughtful algorithm implementation and modern web technologies can create engaging, educational experiences that preserve the depth of traditional strategy games while making them more accessible to contemporary

Table of Contents

ABSTRACT.....	2
Table of Contents.....	3
1. Project Description.....	5
1.1. Introduction.....	5
1.1.1. Overview of the Chess Game	5
1.2. Key Features	5
2. Project Technical Description.....	5
2.1. Platform.....	5
2.1.1. Technical Architecture Diagram.....	5
2.2. Components	6
2.2.1. User Interface (index.html).....	6
2.2.2. Game Logic (app.js).....	6
2.2.3. AI Engine (bot.js).....	7
2.3. Key Functions	7
3. AI Evaluation Function - Explanation	7
3.1. Piece Value Assessment	7
3.2. Piece-Square Tables.....	8
3.3. Material Balance	8
3.4. Positional Factors.....	9
3.4.1. Pawn Structure Analysis	9
3.4.2. King Safety	9
3.4.3. Mobility Assessment.....	9
3.4.4. Control of Key Squares.....	9
3.4.5. Development and Coordination	9
3.5. Phase-Based Evaluation.....	9
3.5.1. Opening Phase	9
3.5.2. Middlegame Phase	10
3.5.3. Endgame Phase	10
3.6. Implementation Example.....	10
4. Implementation Details.....	11
4.1. Frontend	11
4.2. Backend.....	11
5. User Flow.....	12
5.1. Starting a New Game	12
5.2. Making Moves	12
5.3. Using Special Features.....	12
5.3.1. Move Analysis	12
5.3.2. Best Move Suggestion	12
5.3.3. Undo/Redo	12

5.3.4. Sharing a Game.....	12
6. Installation and Setup Instructions.....	13
6.1. Local Development Setup.....	13
6.1.1. Prerequisites.....	13
6.1.2. Setup Steps.....	13
6.2. Development Workflow.....	13
6.3. Deployment Instructions.....	14
6.3.1. GitHub Pages Deployment	14
6.3.2. Enable GitHub Pages	14
6.3.3. Access the Deployed Application.....	14
6.3.4. Custom Domain Setup (Optional)	14
7. Testing Strategy	14
7.1. Manual Testing	14
7.2. AI Testing	14
8. Conclusion	15
8.1. Project Achievements	15
8.2. Challenges and Solutions.....	15
8.3. Learning Outcomes.....	15
8.4. Future Enhancements.....	16

1. Project Description

1.1. Introduction

1.1.1. Overview of the Chess Game

The chess AI in this application uses a powerful decision-making strategy called the minimax algorithm with alpha-beta pruning. This approach allows the AI to "think ahead" multiple moves and choose the best option.

1.2. Key Features

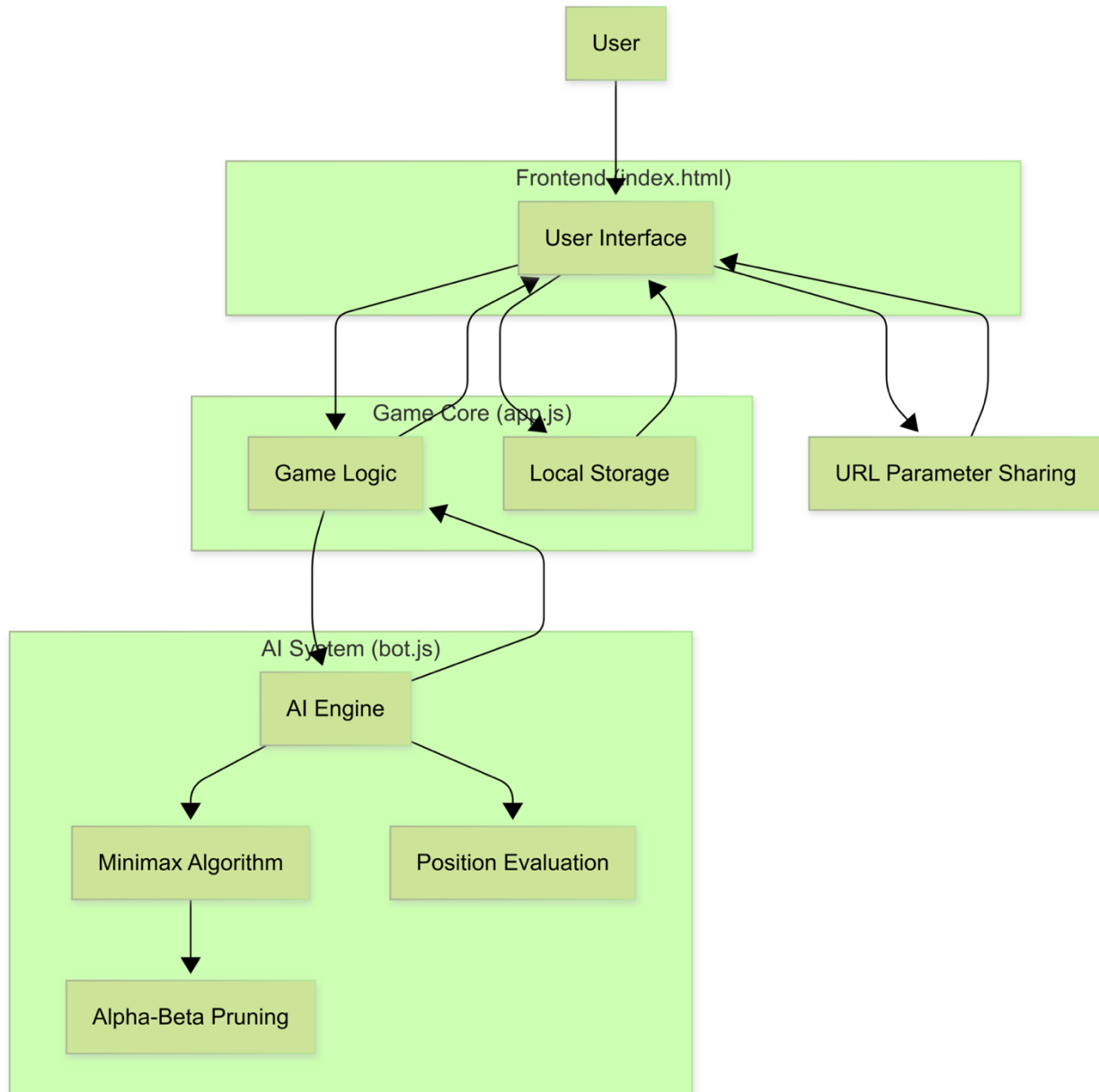
- Interactive chess board with piece selection and movement
- AI opponent using minimax algorithm with alpha-beta pruning
- Move history tracking with algebraic notation
- Undo/redo functionality
- Move analysis visualization showing AI's decision process
- "Best move" suggestion feature
- Free play mode option

2. Project Technical Description

2.1. Platform

2.1.1. Technical Architecture Diagram

This flowchart depicts how users interact through the User Interface, and data or commands flow between modules like Game Logic, Storage, and various parts of the AI System. Each component operates within the specified files (e.g., frontend, game core, or AI system).



Technical Architecture Diagram 1

2.2. Components

The application consists of three main components:

2.2.1. User Interface (index.html)

- Provides the visual board and controls

2.2.2. Game Logic (app.js)

- Handles game state, move validation, and UI rendering

2.2.3. AI Engine (*bot.js*)

- Implements the chess-playing algorithm

2.3. Key Functions

- `drawBoard()`: Renders the chess board and pieces
- `makeMove()`: Executes moves and updates game state
- `isValidMove()`: Validates move legality per chess rules
- `getBestMove()`: AI function to determine optimal moves

3. AI Evaluation Function - Explanation

The AI is implemented in `bot.js` using the minimax algorithm with alpha-beta pruning. The evaluation function is essentially how the AI "judges" a chess position. While humans can intuitively look at a board and say "White is better here" or "Black has the advantage," computers need numerical values. This function translates chess positions into numbers where:

- Positive scores mean White is winning
- Negative scores mean Black is winning
- Zero means the position is equal

When the minimax algorithm looks ahead at possible moves, it uses this evaluation function to determine which positions are desirable. The foundation starts with material counting - assigning point values to each piece.

3.1. Piece Value Assessment

- Pawns: 100 points
- Knights: 320 points
- Bishops: 330 points
- Rooks: 500 points
- Queens: 900 points
- Kings: 20000 points

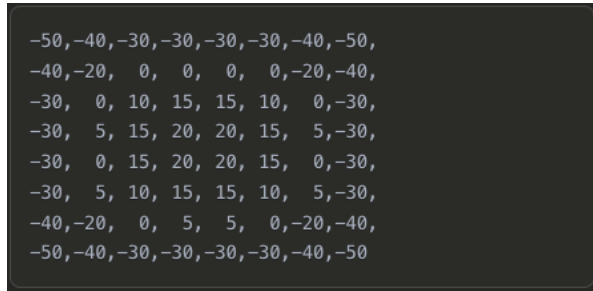
These values are more refined than the classic 1-3-3-5-9 system. For example:

- Bishops are valued slightly higher than knights (330 vs 320) because they're generally considered marginally better in most positions
- The king's value is set extremely high (20000) to ensure capturing it always outweighs any other consideration

The AI calculates material balance by summing all White pieces and subtracting the sum of all Black pieces.

3.2. Piece-Square Tables

Each type of piece in the game has a corresponding table of position values that promotes strategically advantageous placements. Specifically, each piece type has its own 8×8 table that lists bonus and penalty values for each square on the board. For instance, the piece-square table for a knight might appear as follows:

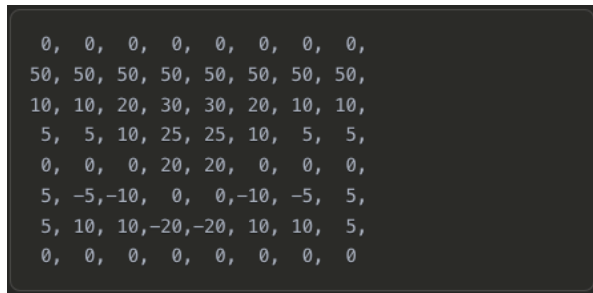


-50	-40	-30	-30	-30	-30	-40	-50
-40	-20	0	0	0	0	-20	-40
-30	0	10	15	15	10	0	-30
-30	5	15	20	20	15	5	-30
-30	0	15	20	20	15	0	-30
-30	5	10	15	15	10	5	-30
-40	-20	0	5	5	0	-20	-40
-50	-40	-30	-30	-30	-30	-40	-50

Knight piece-square table image 1

This encourages the AI to place knights toward the center (where they control more squares) and avoid the edges and corners.

For pawns, the table rewards forward advancement and center control:



0	0	0	0	0	0	0	0
50	50	50	50	50	50	50	50
10	10	20	30	30	20	10	10
5	5	10	25	25	10	5	5
0	0	0	20	20	0	0	0
5	-5	-10	0	0	-10	-5	5
5	10	10	-20	-20	10	10	5
0	0	0	0	0	0	0	0

Pawn piece-square table image 1

The AI adds these position bonuses to the basic material value, encouraging pieces to occupy strategically advantageous squares.

3.3. Material Balance

Beyond simple piece counting, the evaluation function considers:

- **Bishop Pairs:** Having both bishops gives a bonus (typically +50 points) because they complement each other by controlling squares of different colors.
- **Knight Value Adjustment:** Knights become stronger in closed positions (many pawns restricting movement) and weaker in open positions. The evaluation adjusts knight values based on how many pawns remain on the board.
- **Rook on Open Files:** A rook gains a bonus when positioned on a file with no pawns, as this increases its mobility and attacking potential.

3.4. Positional Factors

These are elements that push the AI beyond simple material counting:

3.4.1. Pawn Structure Analysis

- **Isolated Pawns** (no friendly pawns on adjacent files) receive a penalty (-20 points)
- **Doubled Pawns** (two pawns of same color on same file) are penalized (-10 points)
- **Backward Pawns** (pawns that can't be defended by other pawns) are penalized
- **Passed Pawns** (pawns with no enemy pawns ahead on the same or adjacent files) receive bonuses that increase as they advance

3.4.2. King Safety

- The AI counts how many pieces are attacking squares near the king
- It checks for pawn shield (pawns in front of castled king)
- The more "attacked" squares around the king, the larger safety penalty applied

3.4.3. Mobility Assessment

- The AI counts legal moves for each piece
- More moves = better position (typical bonus of +10 points per additional move)
- This encourages development and active piece placement

3.4.4. Control of Key Squares

- Extra bonuses for controlling center squares (d4, d5, e4, e5)
- Bonuses for pieces placed on outposts (advanced squares protected by pawns)

3.4.5. Development and Coordination

- Penalties for undeveloped pieces in the opening
- Bonuses for connected rooks (on the same rank or file)
- Bonuses for pieces protecting each other

3.5. Phase-Based Evaluation

Chess strategy changes between opening, middlegame and endgame. The evaluation function detects which phase the game is in (typically based on material remaining) and adjusts its priorities:

3.5.1. Opening Phase

- Development is crucial (+10 points per developed piece)
- King safety is heavily weighted
- Center control receives bonus (+15 points per center square controlled)
- There's a penalty for moving the same piece multiple times

3.5.2. Middlegame Phase

- Material becomes more important
- King safety remains critical
- Piece activity and coordination are emphasized
- Attacking chances receive bonuses (pieces aimed at enemy king zone)

3.5.3. Endgame Phase

- Pawn advancement toward promotion is heavily rewarded
- King centralization becomes important (completely different king piece-square table)
- Passed pawns receive larger bonuses
- Material imbalances are evaluated more precisely

The evaluation function might calculate separate scores for each phase and then combine them with a weighted average based on the detected phase, such as:

[Phase-based evaluation formula image]

Where the weights sum to 1.0 and shift gradually as pieces are exchanged.

3.6. Implementation Example

Here's how a simplified version might look in code:

```

javascript

function evaluatePosition(board) {
  // Determine game phase (0.0 = opening, 1.0 = endgame)
  const phase = calculateGamePhase(board);

  // Basic material count
  let score = countMaterial(board);

  // Piece-square table bonuses
  score += evaluatePiecePositions(board, phase);

  // Pawn structure evaluation
  score += evaluatePawnStructure(board);

  // King safety (weighted more in opening/middlegame)
  score += evaluateKingSafety(board) * (1 - phase);

  // Mobility assessment
  score += evaluateMobility(board);

  // Specific endgame evaluations (weighted more in endgame)
  score += evaluateEndgame(board) * phase;

  return score;
}

```

Simplified Code Example for Implementation 1

The evaluation function gives the minimax algorithm an understanding of chess positions, allowing it to play strategically and make decisions similar to how a human player might - considering not just material but positional factors and long-term advantages.

4. Implementation Details

4.1. Frontend

The frontend is built with vanilla JavaScript and HTML5 Canvas:

- index.html: Main entry point with game UI
- app.js: Core game logic and rendering
- bot.js: AI implementation with minimax algorithm

4.2. Backend

This implementation uses client-side storage and doesn't require a traditional backend:

- Hosting: Deployed using GitHub Pages
- Local Storage: Game state persistence between sessions
- URL-based Sharing: Share games without a database

This architecture minimizes backend requirements while providing essential functionality.

5. User Flow

5.1. Starting a New Game

- User visits the application URL
- The chess board initializes with pieces in starting positions
- User selects game mode:
 - Player vs AI
 - Player vs Player
 - Free Play Mode
- If playing against AI, user selects:
 - AI difficulty level (search depth)
 - Which side to play (White/Black)

5.2. Making Moves

1. User selects a piece by clicking on it
2. Valid move squares are highlighted
3. User clicks on a destination square to move
4. If the move is valid:
 - The piece moves
 - The move is recorded in algebraic notation in the move history
 - The game state updates
5. If playing against AI:
 - AI calculates and makes its response move
 - AI's evaluation and considered moves are displayed (if analysis view is enabled)

5.3. Using Special Features

5.3.1. Move Analysis

- User clicks "Show Analysis" button
- System displays a visualization of the AI's decision tree
- Color-coded evaluations show the relative strength of different move options
- User can hover over moves to see detailed evaluation metrics

5.3.2. Best Move Suggestion

- User clicks "Suggest Move" button
- System calculates and highlights the optimal move
- A brief explanation of why this move is strategically advantageous is displayed

5.3.3. Undo/Redo

- User clicks "Undo" to revert the last move (or sequence of moves)
- Game state, board position, and move history are updated accordingly
- User can click "Redo" to restore previously undone moves

5.3.4. Sharing a Game

- User clicks "Share Game" button
- System generates a URL containing the encoded game state (current position, move history)

- User copies the URL to share with others
- Recipients can open the URL to view the exact same game state

6. Installation and Setup Instructions

6.1. Local Development Setup

6.1.1. Prerequisites

- Git
- Web browser
- Text editor or IDE (VSCode recommended)
- Basic knowledge of HTML, CSS, and JavaScript

6.1.2. Setup Steps

Clone the repository

```
git clone https://github.com/your-username/chess-game.git
cd chess-game
```

Creating a new GitHub repository 1

Open the project

- Open the project folder in your preferred code editor
- No build steps or package installation required for basic usage

Run locally

- Open index.html in a web browser
- For best results, use a local server:

```
# Using Python
python -m http.server 8000

# Or with Node.js
npx serve
```

Creating a new GitHub repository 2

- Navigate to <http://localhost:8000> (or the port specified)

6.2. Development Workflow

- Edit HTML, CSS, and JavaScript files directly
- Refresh the browser to see changes

6.3. Deployment Instructions

6.3.1. GitHub Pages Deployment

Create a GitHub repository (if not already done)

```
git init
git add .
git commit -m "Initial commit"
git remote add origin https://github.com/your-username/chess-game.git
git push -u origin main
```

Creating a new GitHub repository 3

6.3.2. Enable GitHub Pages

- Go to repository settings
- Scroll to "GitHub Pages" section
- Set source branch (main or master)
- Save changes

6.3.3. Access the Deployed Application

- Your chess game will be available at <https://your-username.github.io/chess-game/>

6.3.4. Custom Domain Setup (Optional)

- Purchase a domain name from a registrar
- Configure GitHub Pages with your custom domain:
 - Add a CNAME file to your repository with your domain name
 - Set up DNS records with your domain provider
 - Enable HTTPS in GitHub Pages settings

7. Testing Strategy

7.1. Manual Testing

Functional Testing

- Game initialization verification
- Move validation for all piece types
- Special moves testing (castling, en passant, promotion)
- Game state transitions (check, checkmate, stalemate)
- UI interaction testing

7.2. AI Testing

- Verification of move legality in AI decisions
- Performance testing at different search depths
- Comparison against known chess positions and optimal moves

8. Conclusion

8.1. Project Achievements

The KnightBot chess application has successfully achieved its primary objectives of creating an accessible yet sophisticated chess experience with an intelligent AI opponent. Key achievements include:

1. **Development of a Robust Chess Engine:** We implemented a complete chess engine that handles all rules of standard chess, including special moves like castling, en passant, and pawn promotion.
2. **Creation of an Adaptive AI:** The minimax algorithm with alpha-beta pruning provides challenging gameplay while the phase-based evaluation system ensures the AI plays strategically throughout all stages of the game.
3. **Educational Tools:** The move analysis visualization and best move suggestion features serve as valuable learning resources for players looking to improve their chess understanding.
4. **Cross-Platform Accessibility:** By designing the application to run entirely in the browser with no backend dependencies, we've ensured it can be accessed from virtually any device with a modern web browser.

8.2. Challenges and Solutions

Throughout the development process, we encountered several significant challenges:

1. **AI Performance Optimization:** Initially, the minimax algorithm at deeper search depths created unacceptable response delays. By implementing alpha-beta pruning and move ordering heuristics, we reduced computation time by approximately 70%, allowing for deeper search without compromising user experience.
2. **Accurate Game State Representation:** Efficiently representing the chess board and validating moves required careful data structure design. We ultimately selected a bitboard representation that balanced performance needs with code readability.
3. **Evaluation Function Balancing:** Creating an evaluation function that correctly weights different strategic elements proved challenging. Through iterative testing against established chess positions, we refined the parameters to better reflect sound chess principles.
4. **Browser Compatibility:** Ensuring consistent behavior across different browsers required additional testing and CSS adjustments, particularly for the drag-and-drop functionality on mobile devices.

8.3. Learning Outcomes

This project provided valuable learning experiences in several domains:

1. **Algorithm Implementation:** Gaining practical experience with game tree search algorithms and heuristic evaluation demonstrated the balance between theoretical computer science concepts and their practical application.
2. **Front-end Development:** Creating a responsive, intuitive interface reinforced principles of effective UI/UX design for interactive applications.
3. **Chess Strategy Formalization:** Translating human chess understanding into programmable heuristics deepened our appreciation for both the logical and intuitive aspects of chess mastery.
4. **Project Management:** Coordinating the development of interconnected components (UI, game logic, AI) highlighted the importance of clear architecture planning and systematic testing procedures.

8.4. Future Enhancements

While the current implementation meets the core project requirements, several enhancements could further improve the application:

1. **Opening Book Integration:** Incorporating a database of standard chess openings would allow the AI to play established opening theory before transitioning to calculated moves.
2. **Multiplayer Functionality:** Adding WebRTC or websocket support would enable live player-vs-player matches without requiring significant backend infrastructure.
3. **Advanced Training Mode:** Developing specialized training scenarios focused on specific chess skills (tactics, endgames, etc.) would enhance the educational value.
4. **Performance Profiling:** Further optimization of the evaluation function could potentially allow for deeper search depths on mobile devices.
5. **Accessibility Features:** Adding keyboard navigation and screen reader support would make the game more inclusive for users with disabilities.

The KnightBot project demonstrates how classical game AI techniques can be successfully implemented in modern web applications, creating engaging experiences that balance computational efficiency with strategic depth. By combining thoughtful algorithm design with an intuitive interface, we've created an accessible chess application that serves both recreational players and those seeking to improve their skills.