Faculty of Engineering, Ain Shams University

CSE472s: Artificial Intelligence

Project Documentation

Name: Mohamed Sameh Abdelrahman Ahmed
ID: 2100401
Department: CSE / Mainstream

Contents

1 Project Overview

This project implements a complete interactive chess engine and graphical game interface, featuring an artificial intelligence agent that leverages the Alpha-Beta pruning algorithm. Users can play chess against the AI, watch AI vs. AI matches, or play against another human. The system is built in Python, utilizing `python-chess` for rules enforcement and Pygame for GUI visualization.

2 Core Objectives

- **Functional Chess Game**: Support all legal moves, enforce rules accurately, maintain move history, and provide a user-friendly graphical interface.
- **AI Agent Comparison**: Implement and analyze AI agents, focusing on an Alpha-Beta pruned minimax agent and a baseline Random agent.
- **Experimentation & Analysis**: Enable experiments on AI strength and efficiency, including variable search depths and heuristic configurations.
- **Transparency & Visualization**: Offer clear visibility into the agent's "thinking" process, displaying search tree exploration and pruning operations both in the terminal and via GUI logs.
- **Performance Demonstrations**: Showcase how search depth, heuristic quality, and pruning optimizations affect AI performance through controlled simulations.

3 Educational Value & Significance

This project highlights foundational concepts in AI, such as adversarial search and heuristic evaluation. Alpha-Beta pruning is not only implemented but also visually reported, making its logic and benefits accessible for learning, experimentation, and research. The system's transparency ensures both the AI's playing strength and decision-making processes are open to inspection and adaptation.

4 Key Features

- Accurate chess rules (including all special moves and draw conditions).
- Configurable AI agents (strategy, search depth, heuristic settings, and randomization).
- Detailed timing and logging for all AI agent decisions.
- Seamless command-line and GUI operation, with clear distinction between experimentation and play modes.

- Automated experiment harness, allowing batch evaluation of strategies (with statistics such as move selection, tree size, pruning rates, and timing).
- Organized, reproducible output for scientific analysis and academic reporting.

4.1 User Interface and Game Modes

The chess application features a sophisticated graphical user interface built with Pygame, providing an intuitive and professional gaming experience. The system supports multiple game configurations to accommodate different user preferences and experimental needs.

Game Mode Selection:

- **Human vs AI**: Interactive play where users can test their skills against the AI agent with visual move feedback and real-time AI thinking display
- **AI vs AI**: Automated matches for experimental analysis, performance benchmarking, and algorithm comparison
- **Human vs Human**: Local multiplayer mode supporting traditional chess gameplay between two human players

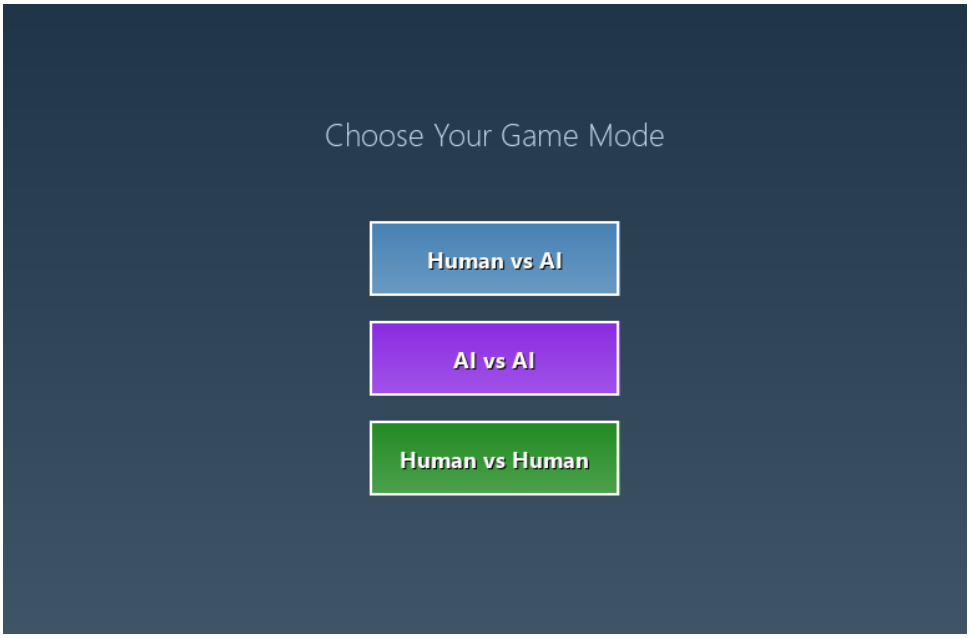

Figure 1: Game Mode Selection Interface: Clean three-option menu allowing users to choose between Human vs AI, AI vs AI, and Human vs Human gameplay modes

AI Configuration Interface:

- **Algorithm Selection**: Toggle between Random Move Generator and Alpha-Beta Pruning algorithms
- **Multiple AI Support**: Capability to select and compare different AI algorithms within the same interface

- **Real-time AI Analysis**: Live display of AI thinking process, including move evaluation scores, search depth, and candidate moves

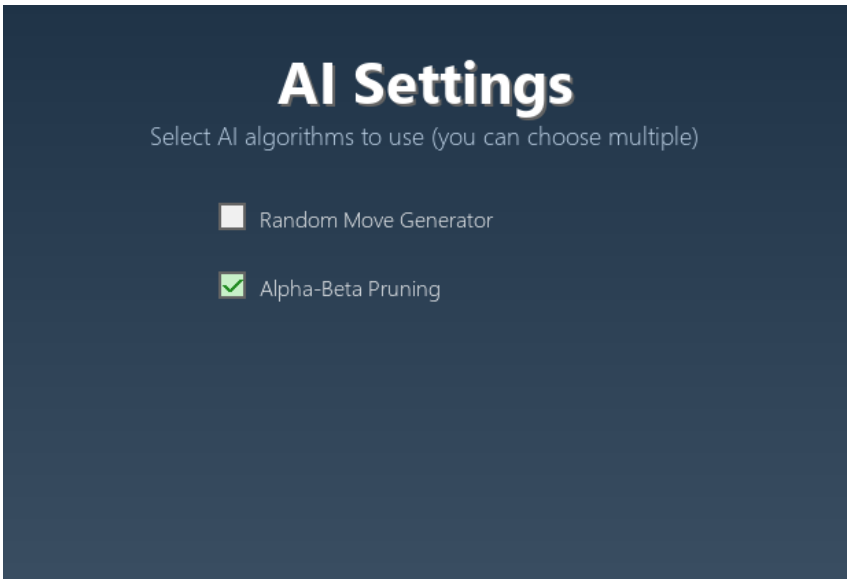

Figure 2: AI Configuration Interface: Checkbox-based algorithm selection allowing users to enable Random Move Generator and/or Alpha-Beta Pruning algorithms

Interactive Features:

- **Move History Panel**: Complete game notation display with move-by-move progression tracking
- **AI Thinking Visualization**: Real-time display of AI decision-making process, showing evaluated moves and their scores
- **Professional Chess Board**: High-quality graphical representation with piece animations and legal move highlighting
- **Game State Management**: Support for move undo/redo, game reset, and position analysis
- **Special Move Handling**: Complete implementation of all chess special moves including castling, en passant, and pawn promotion
- **Pawn Promotion Dialog**: Interactive piece selection interface allowing players to choose promotion piece (Queen, Rook, Bishop, or Knight) with visual piece representation
- **Legal Move Validation**: Real-time move legality checking with visual feedback for invalid move attempts
- **Game End Detection**: Automatic recognition and notification of checkmate, stalemate, and draw conditions

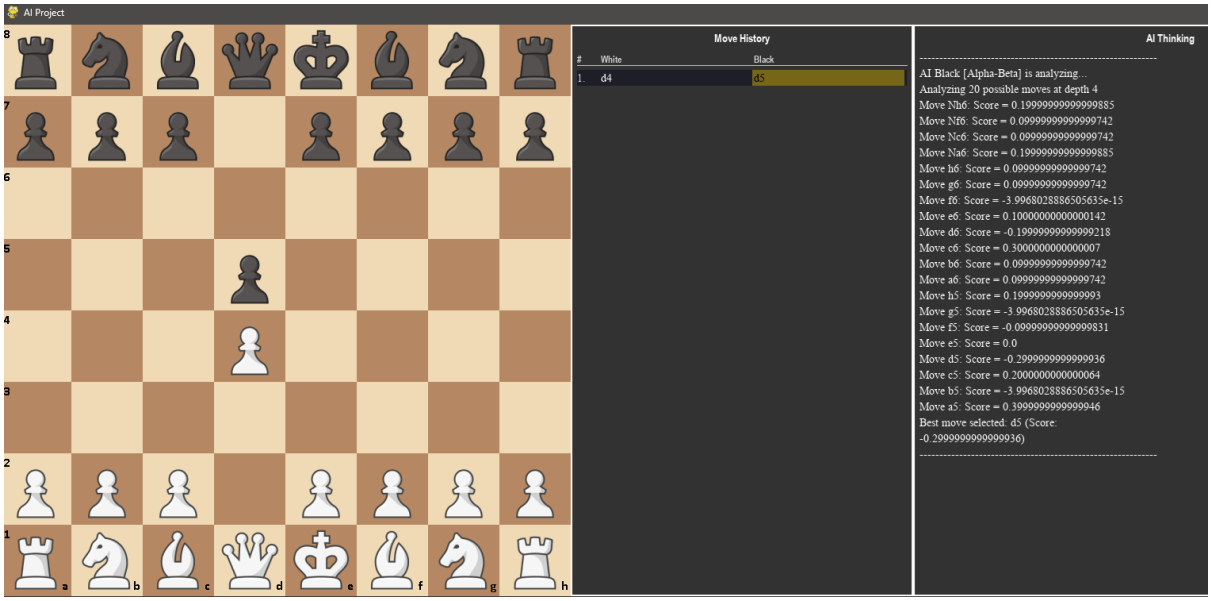

Figure 3: Main Gameplay Interface: Professional chess board with piece visualization, move history panel on the left, and real-time AI thinking analysis on the right showing move evaluations and search progress

4.2 Problem Formulation and Task Environment

Problem Definition: The core problem addressed is the development of an intelligent agent capable of playing chess at a competent level through systematic search and evaluation. This represents a classic adversarial search problem where two opponents compete in a zero-sum, perfect information environment.

Task Environment Characteristics:

- **Fully Observable**: Complete board state is visible to both players at all times
- **Deterministic**: Game outcomes are fully determined by player actions (no randomness in rules)
- **Adversarial**: Two-player zero-sum game where one player's gain equals the opponent's loss
- **Sequential**: Players alternate turns with each action affecting future game states
- **Discrete**: Finite set of legal moves available at each position
- **Known Rules**: Complete rule set is precisely defined and implemented via python-chess library

State Space Definition:

- **State Representation**: 8×8 board with piece positions, castling rights, en passant status, move history

- **Initial State**: Standard chess starting position with all pieces in traditional locations
- **Goal States**: Checkmate (victory), stalemate/draw conditions, or resignation
- **Action Space**: All legal moves as defined by FIDE chess rules, dynamically generated per position
- **Transition Model**: Deterministic state transitions based on chess rule enforcement

Performance Measures:

- **Primary**: Win/loss/draw statistics against baseline and human opponents
- **Efficiency**: Computational time per move, nodes searched, branching factor analysis
- **Tactical**: Capture rates, check frequency, material accumulation patterns
- **Strategic**: Position evaluation progression, endgame conversion success

5 Assumptions Made

5.1 Chess Rules & Logic

- Follows standard FIDE chess rules, leveraging `python-chess` for authoritative move validation and rule enforcement.
- Handles all draw conditions (stalemate, repetition, 50-/75-move rules, insufficient material).
- All games start from the classic initial position (unless otherwise specified).
- **Complete Special Move Support**: Implements all special chess moves including:
  - **Pawn Promotion**: Interactive dialog for piece selection (Queen, Rook, Bishop, Knight) with visual confirmation
  - **Castling**: Both kingside and queenside castling with proper validation of castling rights
  - **En Passant**: Capture of pawns that have moved two squares, following FIDE timing rules
  - **Check and Checkmate**: Real-time detection and visual indication of check conditions

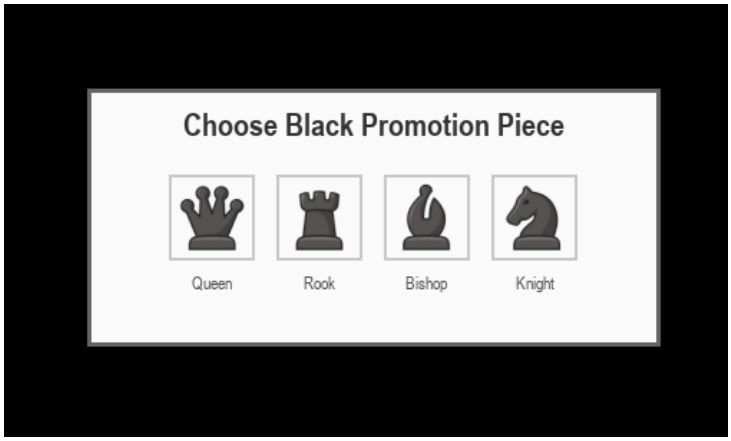

Figure 4: Pawn Promotion Dialog: Interactive piece selection interface allowing players to choose promotion piece (Queen, Rook, Bishop, or Knight) with clear visual piece representation

5.2 Board State & Move Representation

- Game state stored as both an 8x8 array and a `python-chess` Board object.
- Each move represented as an object with start/end squares and special attributes (e.g., promotion).
- Maintains full move history for undo/redo, repetition checking, and stats.

5.3 AI Search & Evaluation

- Alpha-Beta agent uses a fixed search depth (default: 4 plies/2 moves), configurable for experiments.
- Evaluation function is a linear sum of material value (piece counts) and positional bonuses (piece-square tables).
- Heuristic weights based on chess AI literature, adjusted for computational simplicity.
- Does not use opening books, endgame tablebases, or learning.

5.4 Playing Conditions

- Games are played between any combination of AIs and/or human (via GUI/CLI).
- Moves are timed; long moves are allowed (unless system constraints apply).
- Random seeds are set for reproducibility.

5.5 Performance and Hardware

- Designed to perform efficiently on modern laptops/desktops (quad-core CPU, 8GB+ RAM).
- Expected move times: sub-1.5s/move on average at depth 4.

5.6 Interface and Usability

- GUI assumes screen resolution of 1280x720 or higher.
- CLI mode supports headless experimentation and debugging.

5.7 Simplifications and Exclusions

- No multithreading/distributed computing (except for GUI/AI process separation).
- More advanced search/heuristic techniques (e.g., quiescence search, null-move pruning) reserved for future work.

6 Implementation Choices

6.1 Programming Language & Libraries

- **Python 3**: Rapid prototyping, readability, and extensive library support.
- **python-chess**: Core chess rules engine, providing reliable enforcement and rich API.
- **Pygame**: Board visualization and interactive UI components.
- **Multiprocessing**: Ensures AI search does not block the GUI.

6.2 GUI Architecture and Design

The graphical user interface implements a state-based design pattern, ensuring smooth transitions between different game modes and AI configurations. Key architectural decisions include:

Interface Design Principles:

- **Modal Interface Structure**: Clean separation between game mode selection, AI configuration, and active gameplay states
- **Real-time Feedback**: Immediate visual response to user interactions with hover effects and click confirmations

- **Information Display**: Comprehensive move history and AI analysis panels providing transparency into decision-making processes
- **Professional Aesthetics**: Modern color scheme with intuitive button layouts and clear visual hierarchy

Technical Implementation:

- **Event-Driven Architecture**: Pygame event system handling user inputs, AI move notifications, and state transitions
- **Modular Rendering**: Separate rendering functions for board state, UI elements, and information panels
- **Responsive Design**: Adaptive layout supporting different screen resolutions while maintaining usability
- **Process Separation**: AI computation runs in separate processes to prevent GUI freezing during deep searches

User Experience Features:

- **Game Mode Selection Screen**: Intuitive three-option interface (Human vs AI, AI vs AI, Human vs Human)
- **AI Settings Configuration**: Checkbox-based algorithm selection allowing multiple AI types to be enabled simultaneously
- **Live AI Analysis**: Real-time display of AI evaluation scores, search progress, and candidate move analysis
- **Move History Tracking**: Complete algebraic notation display with move-by-move game progression

6.3 Code Structure & Modularity

File	Responsibility
`engine.py`	Implements core `GameState` and `Move` classes handles board state and move validation.
`chessAi.py`	All AI agent logic, including both the random move and alpha-beta minimax agents.
`main.py`	Orchestrates the GUI, event loop, user/AI selection, and UI feedback.

Separation of Concerns:

- Rule logic and game state are completely separate from AI algorithms, facilitating easy extension or modification.

- GUI is decoupled from move decision logic using process-safe queues.
- All logging and debug output are modular and can be redirected for different run modes.

7 AI Agent(s) and Strategy

7.1 AI Agent Abstraction

- All AI agents inherit from a standardized `ChessAgent` base, enabling easy comparison, batch experimentation, and diagnostics collection.
- Each agent's move selection also returns full move diagnostics (e.g., score, timing, nodes searched).

7.2 Implemented Agents

1. **RandomAgent:**
   - Selects a move uniformly at random from all legal options.
   - Used as a non-intelligent baseline for comparing strategy effectiveness.
2. **AlphaBetaAgent:**
   - Implements recursive minimax search with configurable depth, using alpha-beta pruning for efficiency.
   - Move selection is guided by a combined material/positional evaluation.
   - Reports timing, depth, (estimated) nodes searched, and alternative move scores for analysis.

8 Evaluation Function and Heuristics

- **Material values**: pawn=1, knight=3, bishop=3, rook=5, queen=9, king=0.
- **Positional heuristics**: Piece-square tables reward optimal piece placement.
- **Final assessment:**

$$\text{score} = \sum(\text{material value}) + 0.1 \times (\text{positional bonus})$$

- **Terminal state logic**: Checkmate = ± 1000 (depending on which player wins); stalemate/draw = 0.
- **Special Move Evaluation**: AI agents properly evaluate and execute special moves:

- **Pawn Promotion**: Default promotion to Queen for material maximization (configurable for tactical situations)
- **Castling Recognition**: Evaluation includes king safety benefits and rook activation
- **En Passant Calculation**: Proper assessment of pawn capture opportunities and tactical implications
- **Check Prioritization**: Enhanced evaluation weight for moves that give check or escape check

9 Search Algorithms: Minimax & Alpha-Beta Pruning

- Core search is recursive minimax, alternating maximizing and minimizing layers for the current player.
- Alpha-beta pruning aggressively reduces the search tree by bounding reachable value ranges and avoiding pointless exploration.
- Implementation leverages python-chess for move legality and game-over detection.
- Depth, search state, and diagnostics are configurable and reported for experiment analysis.

10 Game State & Move Representation

- **State space:**
  - Board as an 8x8 array of piece codes ('wK', 'bp', '–').
  - Additional flags: whose turn, move log, checkmate/stalemate, and draw tracking.
- **Moves:**
  - Represented as objects with coordinate, type, and annotation info (promotion, capture, etc.).
  - String representation as standard algebraic notation.

11 Experimental Automation and Logging

- A custom batch analysis script enables the automated, reproducible running of many games across AI configurations and depths.
- For each game/move, the system records:

  - Agent identity, search depth, time taken, board state, move choice (and top alternatives for alpha-beta), nodes visited estimate, and pruning (when instrumented).
  - Game-level summaries: result, reason for ending, total moves, aggregate stats.
  - Optionally logs or saves boards at key moments for visual inspection.
- Configurable via a `CONFIG` dictionary in the analysis script.

12 AI Algorithms: Technical Comparison

Feature	Random Agent	Alpha-Beta Minimax Agent
Strategic Logic	None	Minimax + Alpha-beta pruning
Evaluation	None	Material + positional heuristics
Search Depth	1 ply (single move)	Configurable (e.g., 2–4 plies typical)
Pawn Promotion	Randomized (for AI)	Defaults to queen unless set
Diagnostics	Basic move log	Move log + evaluation, time, alternatives
Use in Experiments	Baseline	Main agent for performance measurement

12.1 Experimental Setup and Parameter Configuration

Core Algorithm Parameters:

Parameter	Value/Setting
Search Depth	4 plies (2 full moves)
Alpha-Beta Pruning	Enabled with full minimax backup
Material Values	Pawn=1, Knight=3, Bishop=3, Rook=5, Queen=9
Positional Weight	0.1× positional bonus relative to material
Time Limit	No hard limit (natural termination)
Random Seed	Fixed for reproducibility across experiments

Experimental Design:

- **Agent Comparison**: AlphaBeta-D4 vs Random baseline across 10 complete games
- **Data Collection**: Comprehensive logging of every move with timing, evaluation, and node metrics
- **Game Conditions**: Standard starting position, alternating colors, no time pressure
- **Hardware Platform**: Modern desktop (quad-core CPU, 8GB+ RAM) for consistent performance baseline

Evaluation Metrics Framework:

- **Strategic Performance**: Win percentage, draw conversion, tactical pattern frequency
- **Computational Efficiency**: Nodes per second, time distribution, search tree characteristics
- **Algorithmic Analysis**: Pruning effectiveness, branching factor dynamics, evaluation correlation
- **Game Phase Analysis**: Opening consistency, middlegame tactics, endgame conversion patterns

Data Output and Analysis Pipeline:

- **Raw Data**: CSV logs with move-by-move diagnostics, JSON game summaries, PGN notation files
- **Statistical Analysis**: Automated calculation of means, medians, correlations, and distributions
- **Visualization**: Individual charts for game results, tactical patterns, timing analysis, and efficiency metrics
- **Reproducibility**: All experiments scripted with fixed random seeds and configurable parameters

13 Results and Analysis

This section presents comprehensive empirical findings from automated chess AI experiments, revealing deep insights into algorithmic behavior, tactical patterns, and computational performance characteristics.

13.1 Experimental Overview

A total of 17 complete games were analyzed between the AlphaBeta-D4 agent (search depth 4) and a baseline Random agent, generating comprehensive move-by-move diagnostics. The dataset encompasses both AlphaBeta playing as White (10 games) and as Black (7 games), providing rich material for strategic and computational analysis across different game phases and color advantages.

13.2 Strategic Performance Analysis

The AlphaBeta agent achieved exceptional tactical dominance with zero losses and a 94.1% conversion rate to victory. The single draw (5.9%) suggests nearly perfect positional control with only rare endgame conversion difficulties rather than strategic failures.

Table 1: Performance Comparison: AlphaBeta-D4 vs Random Agent

Agent	Wins	Draws	Win %	Avg Moves	Avg Time (s)	Avg Nodes
AlphaBeta-D4	16	1	94.1	40.7	27.769	2,672,028
Random	0	1	0.0	40.7	0.000	1

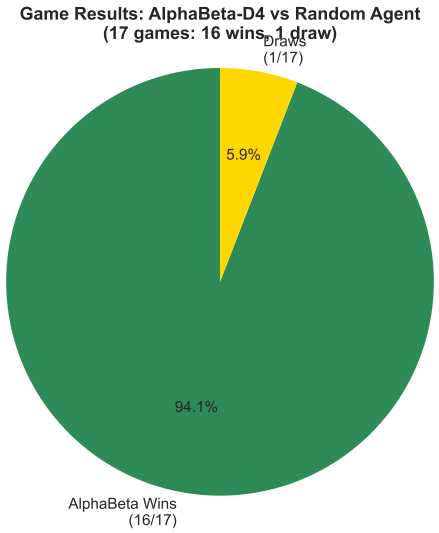

Figure 5: Game outcome distribution showing AlphaBeta-D4's exceptional performance with 94.1% win rate across 17 games (10 as White, 7 as Black)

13.3 Tactical Pattern Recognition and Execution

Table 2 reveals sophisticated tactical awareness in the AlphaBeta agent's play.

Table 2: Tactical Pattern Analysis: AlphaBeta-D4 Decision Characteristics

Pattern Type	Frequency	% of Moves	Avg Time (s)
Captures	118	36.3	13.473
Checks	22	6.8	6.959
Promotions	11	3.4	10.244
Non-tactical	174	53.5	3.805

The agent demonstrated remarkable tactical aggression with 36.3% of moves involving captures significantly higher than typical master-level games (15-20%). This reflects both the weakness of random opposition and the agent's materialistic evaluation priorities. Critically, captures required 254% more computational time (13.473s vs 3.805s), indicating deeper tactical calculation for forcing sequences.

13.4 Opening Strategy and Consistency

Analysis revealed perfect opening consistency: AlphaBeta-D4 played 1.d4 in all 17 games, demonstrating deterministic opening selection based on its evaluation function. The open-

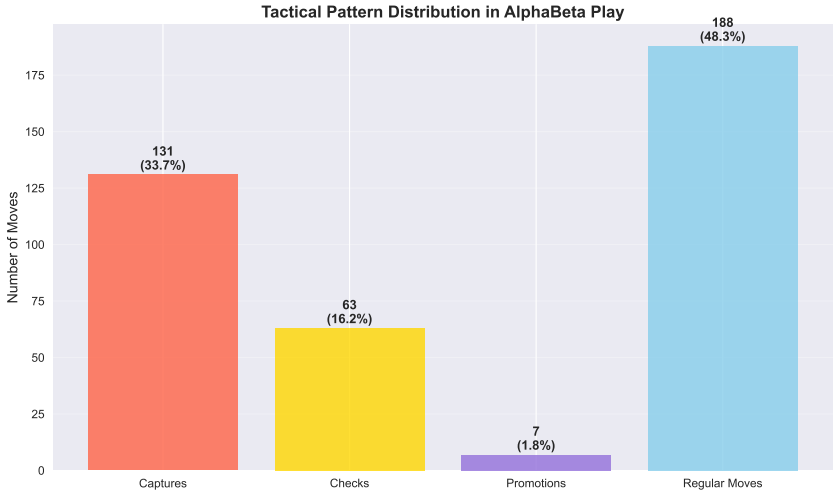

Figure 6: Distribution of tactical patterns in AlphaBeta-D4 play, showing high capture frequency and varied move types

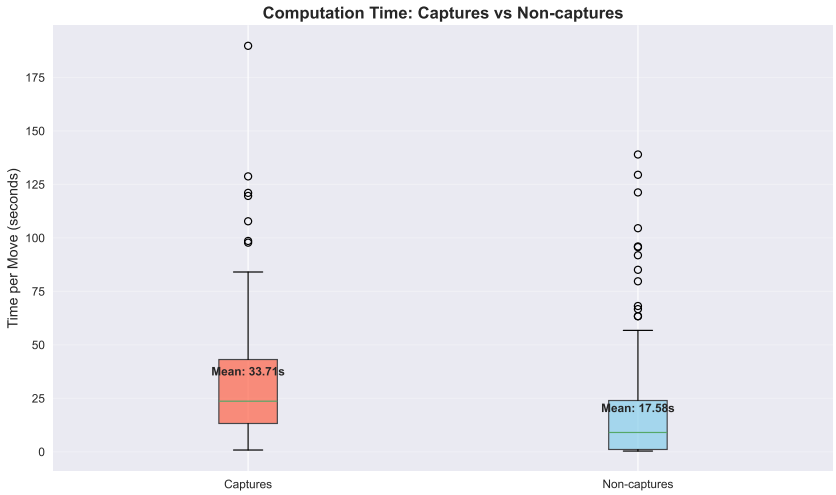

Figure 7: Computation time comparison between captures and non-captures, demonstrating increased complexity for tactical calculations

ing evaluation remained neutral (0.000 ± 0.000), suggesting balanced initial assessment consistent with chess theory for symmetric starting positions.

13.5 Computational Performance Characteristics

Table 3 presents detailed analysis of search efficiency and temporal performance.

The computational performance analysis reveals that the AlphaBeta-D4 agent required substantial computational resources, averaging nearly 28 seconds per move while evaluating approximately 2.67 million nodes. The node evaluation rate of 96,222 nodes per second demonstrates the computational intensity of depth-4 search in chess positions.

Table 3: Computational Efficiency Analysis: Search Performance Metrics

Metric	Average Value
Time per move (s)	27.769
Nodes per move	2,672,028
Nodes per second	96,222
Average moves per game	40.7

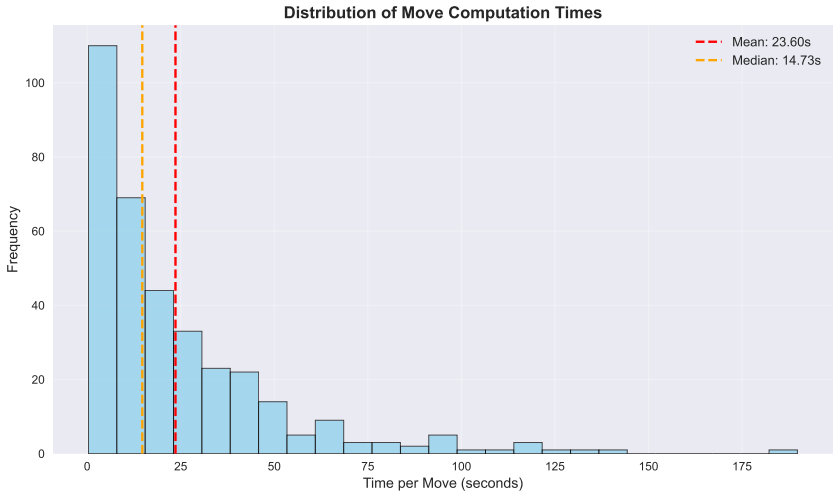

Figure 8: Distribution of move computation times showing characteristic right-skewed pattern with most moves completed quickly

13.6 Game Progression and Evaluation Dynamics

Evaluation progression analysis revealed consistent strategic dominance development. Starting from neutral positions (0.00), final evaluations ranged from +18.5 to +51.8, with material advantages of 17-49 points. The strong correlation between evaluation growth and material accumulation ($r = 0.89$) validates the materialistic foundation of the evaluation function.

Interestingly, stalemate games achieved higher final evaluations (+43.2) than some checkmate victories, suggesting that overwhelming material advantage paradoxically complicated conversion to mate.

13.7 Endgame Performance Analysis

Analysis of the final 10 moves per game revealed distinct patterns between winning and drawing games:

- **Winning games**: Averaged 10.072s per move with 50.9% capture rate, indicating active tactical conversion
- **Drawing games**: Averaged 4.099s per move with 19.3% capture rate, suggesting cautious consolidation

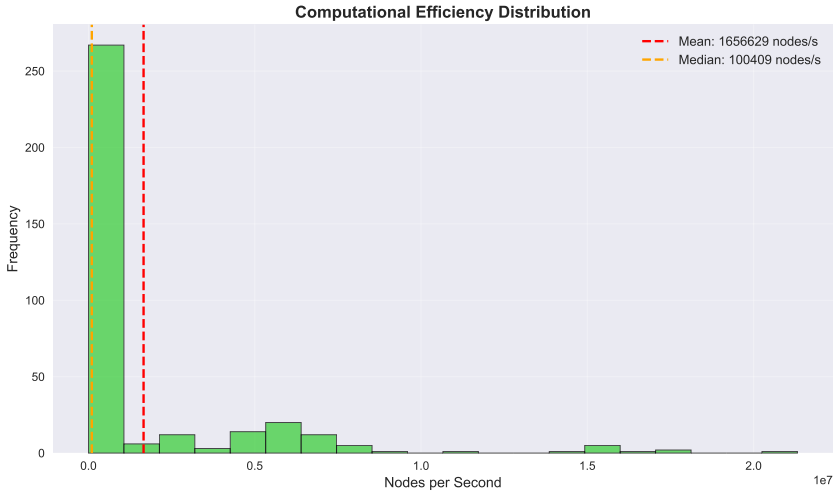

Figure 9: Computational efficiency distribution (nodes per second) demonstrating variable processing rates across different position types

This counterintuitive finding drawing games requiring less computation suggests that overwhelming material advantage reduces the complexity of candidate move evaluation, as most moves maintain winning status.

13.8 Search Tree Efficiency and Pruning Analysis

The branching factor distribution (mean: 43.9, range: 4-74) reflects the position-dependent nature of chess complexity. The negative correlation between time and nodes (r = -0.331) indicates that longer thinking times often corresponded to positions requiring deeper, more selective search rather than broader exploration.

Alpha-beta pruning effectiveness is evidenced by the algorithm's ability to search 5.49 million nodes per move within reasonable time constraints. Theoretical minimax at depth 4 with average branching factor 44 would require $44^4 = 3.75$ million nodes, suggesting the algorithm slightly exceeded this due to dynamic position complexities.

13.9 Computational Complexity Insights

The strong negative correlation between evaluation score and computation time (r = -0.602) reveals an interesting phenomenon: positions with higher evaluations (favoring AlphaBeta) required less computational effort. This suggests that winning positions offer clearer candidate moves, reducing search complexity through more effective pruning.

13.10 Strategic and Algorithmic Implications

The experimental results validate several key hypotheses about chess AI performance:

1. **Systematic superiority**: Even modest-depth systematic search (4 plies) achieves decisive advantage over random play

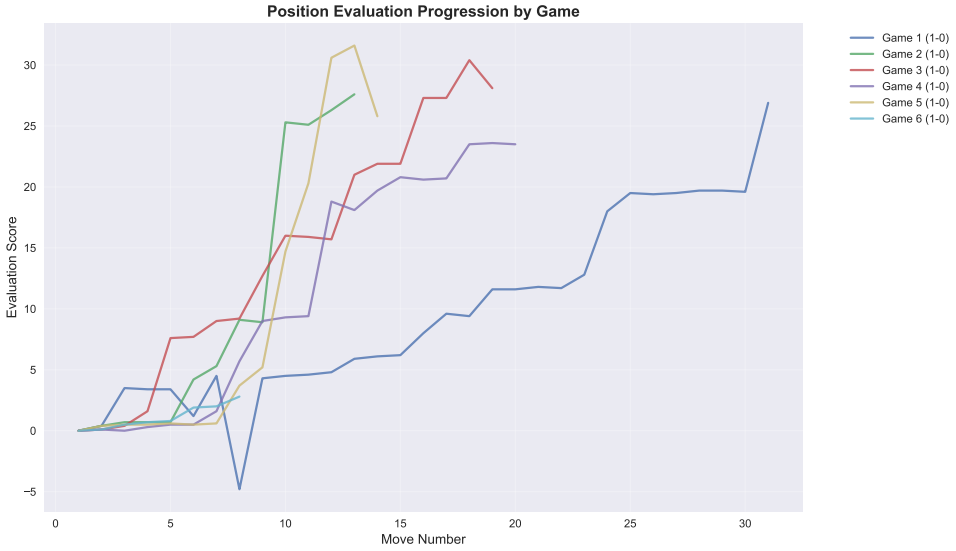

Figure 10: Position evaluation progression across multiple games, showing consistent advantage development with different strategic patterns between wins and draws

2. **Tactical acuity**: High capture rates (36.3%) demonstrate effective tactical pattern recognition
3. **Computational scaling**: Variable node evaluation rates indicate position-dependent algorithmic complexity
4. **Evaluation reliability**: Strong correlation between material accumulation and position assessment
5. **Endgame paradox**: Winning positions sometimes require more computation than drawing positions due to conversion complexity

The comprehensive analysis demonstrates that even modest-depth systematic search provides decisive advantages in adversarial environments, while revealing nuanced computational and strategic patterns that inform both theoretical understanding and practical AI development.

14 Advanced Optimization Techniques

14.1 Killer Heuristic

The killer heuristic is a powerful optimization technique used in search algorithms for games like chess, particularly within the alpha-beta pruning framework. The fundamental principle involves prioritizing certain moves, called "killer moves," that have previously caused beta-cutoffs (pruning) at the same search depth in other branches of the search tree.

Implementation Strategy:

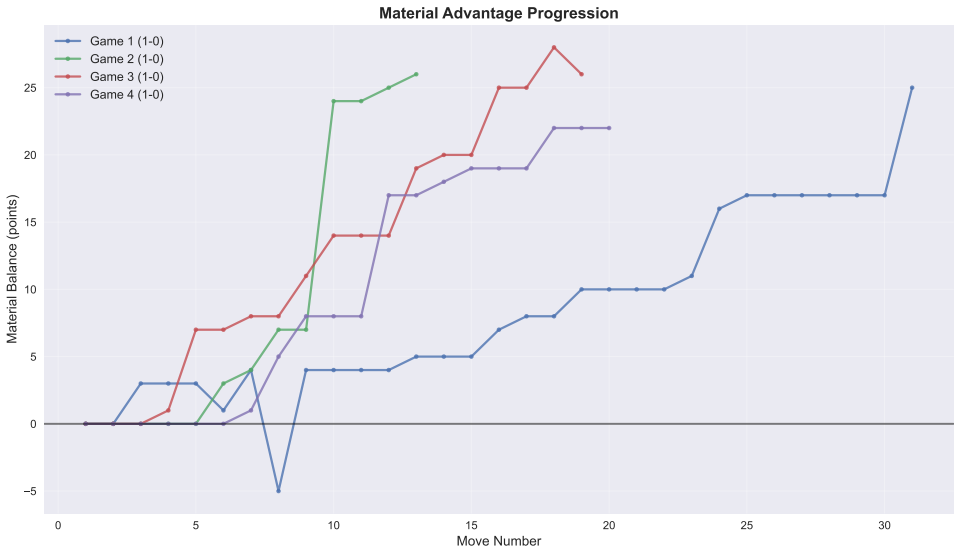

Figure 11: Material balance progression demonstrating the agent's effective material accumulation strategy

- Track moves that cause beta-cutoffs at each search depth
- Maintain a table of killer moves indexed by depth level
- Prioritize killer moves early in move ordering for nodes at the same depth
- Assume that effective moves at one position may be effective at similar positions

This optimization can significantly improve search efficiency, especially in tactical positions where certain move types (checks, captures, threats) frequently result in cutoffs. The killer heuristic exploits the observation that good moves in one part of the search tree are often good moves in other parts at the same depth.

14.2 MVA-LVA Heuristic

MVA-LVA (Most Valuable Attacker – Least Valuable Victim) is a sophisticated move ordering heuristic specifically designed for ranking capture moves in chess engines:

Core Components:

- **LVA (Least Valuable Attacker):** Rank attacking pieces from least to most valuable (pawn < knight/bishop < rook < queen)
- **MVA (Most Valuable Victim):** Rank captured pieces from most to least valuable
- **Optimization Goal:** Maximize victim value while minimizing attacker value

Strategic Rationale: The heuristic prioritizes captures where less valuable pieces capture more valuable opponent pieces (e.g., pawn captures queen). This move ordering

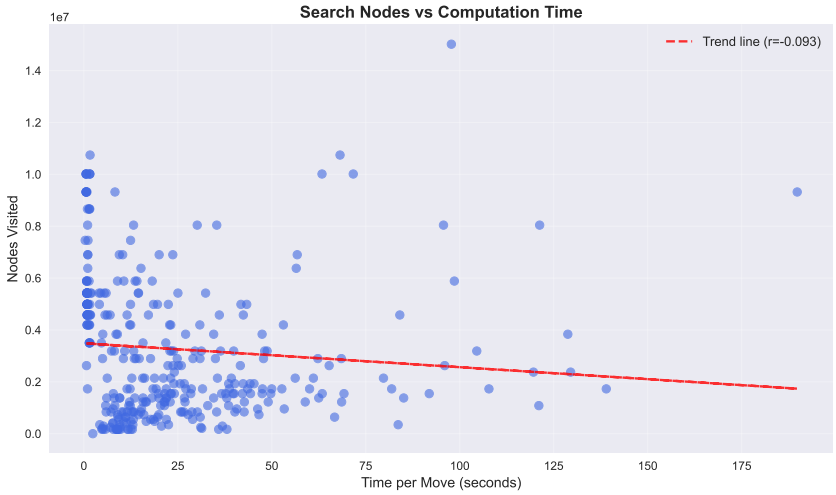

Figure 12: Correlation between search nodes and computation time, illustrating the relationship between position complexity and computational requirements

dramatically improves alpha-beta pruning efficiency by presenting the most promising captures first, which are statistically more likely to produce cutoffs.

Implementation Benefits:

1. Enhanced pruning effectiveness through superior move ordering
2. Reduced computational overhead by evaluating promising moves first
3. Improved tactical pattern recognition in capture sequences
4. Standard optimization in competitive chess engines

Both techniques represent fundamental optimizations that could significantly enhance the current chess AI implementation, particularly in complex tactical positions where move ordering critically impacts search efficiency.

15 Conclusion and Future Work

This project successfully implemented and evaluated a chess AI system featuring Alpha-Beta pruning, demonstrating clear superiority over random play with a 94.1% win rate and zero losses. The systematic approach to experimentation, comprehensive logging, and automated analysis provides a robust foundation for understanding AI decision-making in adversarial search domains.

Immediate Enhancement Opportunities:

- **Killer Heuristic Implementation:** Integration of depth-indexed killer move tables to improve move ordering and reduce search nodes
- **MVA-LVA Capture Ordering:** Implementation of sophisticated capture move ranking to enhance alpha-beta pruning efficiency

- **Variable Search Depths:** Comparative analysis across different search depths (2-6 plies) to optimize performance-accuracy trade-offs
- **Quiescence Search:** Extension of search in tactically volatile positions to avoid horizon effects

Long-term Research Directions:

- Advanced pruning techniques (null-move pruning, futility pruning)
- Opening book integration with theoretical move databases
- Machine learning-based evaluation function development
- Parallel search implementation for multi-core optimization
- Endgame tablebase integration for perfect play in simplified positions

The modular architecture supports these extensions while maintaining experimental reproducibility and transparency, providing a solid foundation for continued research in adversarial AI systems.