ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

NHẬP MÔN TRÍ TUỆ NHÂN TẠO (CO3061)

Bài tập lớn 3

Chess AI Agent System

Advisor: Vương Bá Thịnh
Students: Phan Thanh Tấn - 2213076.

Thành phố Hồ Chí Minh, tháng 2 năm 2025

MỤC LỤC

1 Introduction

Chess is an ancient intellectual game, considered a benchmark for human logical and strategic thinking. In the digital era, building AI systems capable of playing chess at a high level is not only a scientific challenge but also has great practical significance in artificial intelligence (AI) and machine learning research. This project focuses on developing a chess AI system with multiple agents using various algorithms, including modern machine learning approaches.

2 Theoretical Background

2.1 Overview of AI and Machine Learning

Artificial Intelligence (AI) is the field of study focused on enabling computers to perform tasks that require human intelligence. Machine Learning is a branch of AI that allows computers to learn from data to improve performance without explicit programming.

2.2 Search Algorithms in Chess

2.2.1 Random Agent

Principle: The Random Agent selects a move at random from the set of all legal moves. It does not use any evaluation or lookahead, making it the simplest possible agent.

Advantages: Extremely fast, useful as a baseline for comparison.

Disadvantages: No strategy, performs poorly against any non-random opponent.

Example code:

```
def get_move(self, board):
    import random
    return random.choice(list(board.legal_moves))
```

Listing 1: Random Agent

2.2.2 Minimax Agent

Principle: The Minimax algorithm explores the game tree to a fixed depth, assuming both players play optimally. At each node, it recursively evaluates the best achievable outcome for the maximizing and minimizing player.

Evaluation Function: A common evaluation function is the material count:

$$Score = 1(P) + 3(N) + 3(B) + 5(R) + 9(Q)$$

where P, N, B, R, Q are the number of pawns, knights, bishops, rooks, and queens for each side. More advanced functions add piece-square tables, king safety, mobility, etc.

Step-by-step Example: Suppose White can play $e4$ or $d4$. For each move, the algorithm simulates the move, then recursively evaluates all Black's responses, and so on, up to the given depth. At the leaves, the evaluation function is called. The best move is chosen by back-propagating the scores.

Iterative Deepening: Instead of searching to a fixed depth in one go, iterative deepening searches to depth 1, then 2, then 3, etc., using the result of the previous iteration to improve move ordering. This allows for time control and better pruning.

Transposition Table: A hash table that stores previously evaluated positions to avoid redundant computation. This is crucial in chess, where the same position can be reached by different move orders.

Advanced Example:

```python
def minimax_tt(board, depth, alpha, beta, maximizing, tt):
    key = board.fen()
    if key in tt:
        return tt[key]
    if depth == 0 or board.is_game_over():
        val = evaluate(board)
        tt[key] = val
        return val
    if maximizing:
        value = -float('inf')
        for move in board.legal_moves:
            board.push(move)
            value = max(value, minimax_tt(board, depth-1, alpha, beta, False, tt))
            board.pop()
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        tt[key] = value
        return value
    else:
        value = float('inf')
        for move in board.legal_moves:
            board.push(move)
            value = min(value, minimax_tt(board, depth-1, alpha, beta, True, tt))
            board.pop()
            beta = min(beta, value)
            if beta <= alpha:
                break
        tt[key] = value
        return value
```

Listing 2: Minimax with Transposition Table

Strengths:

- Guarantees optimal play if search is deep enough and evaluation is accurate.
- Deterministic and explainable.
- Can be enhanced with iterative deepening, transposition tables, move ordering.

Weaknesses:

- Exponential growth in computation with depth.
- Relies heavily on evaluation function quality.
- Not practical for very deep searches in chess without further enhancements.

2.2.3 Monte Carlo Tree Search (MCTS) Agent

Principle: MCTS builds a search tree by running many random simulations (playouts) from the current position. It balances exploration and exploitation using the Upper Confidence Bound (UCB1) formula.

Detailed Phases:

1. **Selection:** Starting from the root, recursively select child nodes with the highest UCB1 value until a node with untried moves is reached. This phase balances between exploiting known good moves and exploring new ones.
2. **Expansion:** If the selected node has untried moves, one is chosen at random and a new child node is added to the tree.
3. **Simulation (Rollout):** From the new node, play random (or semi-random) moves until the game ends. The result (win/loss/draw) is used as the value for this simulation.
4. **Backpropagation:** The result is propagated back up the tree, updating the statistics (wins, visits) for each node along the path.

Rollout Policy: The simplest rollout policy is to play random moves. Stronger policies use shallow evaluation or heuristics to guide the simulation, improving the quality of the value estimate.

UCB1 Parameter Analysis: The exploration parameter c in UCB1 controls the trade-off between exploration and exploitation. Higher c encourages more exploration, lower c focuses on exploitation. Tuning c is important for performance.

Bias and Exploration: If the rollout policy is biased (e.g., always captures), the MCTS may overestimate certain lines. Purely random rollouts can be weak, but too much bias can reduce exploration.

Modern Variants:

- **AlphaZero/Leela:** Use a neural network to provide both value and policy priors, replacing random rollouts with network evaluation (PUCT formula).
- **RAVE:** Shares statistics between similar moves to speed up learning.

Strengths:

- Scales well to complex games.
- Does not require a handcrafted evaluation function.
- Can be combined with neural networks for even stronger play.
- Flexible time management: can stop at any time and return the best move found so far.

Weaknesses:

- Random playouts may not reflect true position value.
- Requires many simulations for strong play.
- Less explainable than minimax.
- Sensitive to rollout policy and UCB1 parameter.

2.2.4 Deep Learning Agent

Principle: The Deep Learning Agent uses a neural network to evaluate board positions or predict move probabilities. The network is trained on a large dataset of chess games.

2.2.4.1 Board Encoding A crucial step is converting the chess board into a numerical format suitable for neural networks. Two common approaches:

- **Bitboard:** Each piece type and color is represented as a separate 64-bit vector. For example, all white pawns are encoded as a 64-bit array with 1s at pawn locations.
- **Planes:** The board is encoded as a $8 \times 8 \times 12$ tensor (12 planes for 6 piece types x 2 colors). Each plane is a binary matrix indicating the presence of a specific piece type and color.

Example: Plane Encoding

```
Plane 0: White Pawns
Plane 1: White Knights
...
Plane 5: White King
Plane 6: Black Pawns
...
Plane 11: Black King
```

A white pawn on e2 is encoded as 1 at (4,1,0) (using 0-based indexing).

2.2.4.2 Additional Features Other features such as castling rights, move count, repetition, and side to move can be encoded as extra channels or appended to the input vector.

2.2.4.3 Data Augmentation To improve generalization, board positions can be rotated (by 180 degrees), mirrored (left-right), or color-inverted. This increases the effective size of the training set and helps the network learn symmetries of chess.

2.2.4.4 Regularization To prevent overfitting, techniques such as dropout (randomly zeroing activations), weight decay (L2 regularization), and batch normalization are used. These help the network generalize better to unseen positions.

2.2.4.5 Network Architectures

- **MLP (Multi-Layer Perceptron):** Flattens the board and passes through several fully connected layers. Fast but ignores spatial structure.
- **CNN (Convolutional Neural Network):** Uses convolutional layers to capture local patterns and spatial relationships between pieces. Widely used in modern chess engines.
- **Residual Networks:** Allow very deep networks by adding skip connections (residual blocks). Used in AlphaZero/Leela Chess Zero.

Example: Residual Block in PyTorch

```
import torch.nn as nn
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.conv1 = nn.Conv2d(channels, channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.conv2 = nn.Conv2d(channels, channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)
    def forward(self, x):
```

```
        residual = x
        out = torch.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += residual
        return torch.relu(out)
```

Listing 3: Residual Block

2.2.4.6 Loss Functions

- **Regression:** Mean squared error for value prediction (e.g., win probability).
- **Classification:** Cross-entropy loss for move prediction (policy head).
- **Combined:** AlphaZero uses a sum of value and policy losses: $L = (z-v)^2 - \pi^T \log p$, where z is the game result, v is the value head output, π is the target policy, p is the predicted policy.

2.2.4.7 Training Pipeline

1. Extract positions and results from PGN game files or self-play.
2. Encode positions as tensors, results as targets (value, policy).
3. Split into training/validation sets.
4. Train using mini-batch gradient descent (e.g., Adam), monitor loss and accuracy.
5. Evaluate on validation set, tune hyperparameters (learning rate, batch size, architecture).
6. Optionally, use self-play to generate new data and continue training (reinforcement learning).

Example: Full Training Loop

```
for epoch in range(num_epochs):
    for batch in dataloader:
        boards, targets = batch
        optimizer.zero_grad()
        outputs = model(boards)
        loss = loss_fn(outputs, targets)
        loss.backward()
        optimizer.step()
    # Validation, logging, checkpointing...
```

Listing 4: Training Loop

2.2.4.8 Practical Issues

- **Data Quality:** Training on high-level games (e.g., grandmaster PGNs) yields better evaluation than random games.
- **Overfitting:** Monitor validation loss, use regularization and augmentation.
- **Hardware:** Training deep networks for chess is computationally expensive and often requires GPUs.
- **Interpretability:** Neural networks are black boxes; it is hard to explain why a move is chosen.

2.2.4.9 Modern Approaches

- **AlphaZero/Leela Chess Zero:** Use deep residual networks, self-play reinforcement learning, and MCTS guided by neural network policy and value heads.
- **Hybrid Models:** Combine classical search (minimax, MCTS) with neural network evaluation for best results.

2.2.4.10 Comparison with Classical Methods

- Deep learning agents can learn complex patterns and generalize to unseen positions, but require much more data and compute.
- Classical methods are more interpretable and require less data, but are limited by the quality of the evaluation function.
- The strongest modern engines combine both approaches.

2.3 Elo Rating System

The Elo rating system is a widely used method for calculating the relative skill levels of players in zero-sum games such as chess. It is also used to evaluate the strength of AI agents by simulating matches between them.

2.3.0.1 Principle Each player (or agent) is assigned a numerical rating. After each game, the ratings are updated based on the game result and the expected outcome. The system assumes that the difference in ratings between two players predicts the expected score.

2.3.0.2 Expected Score Given two players with ratings R_A and R_B, the expected score for player A is:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

Similarly, the expected score for player B is $E_B = 1 - E_A$.

2.3.0.3 Rating Update After a game, the ratings are updated as follows:

$$R'_A = R_A + K(S_A - E_A)$$

where:

- R'_A: new rating for player A
- K: development coefficient (K-factor), controls sensitivity to new results (typical values: 10–40)
- S_A: actual score (1 for win, 0.5 for draw, 0 for loss)
- E_A: expected score

2.3.0.4 Example Calculation Suppose agent A has $R_A = 1600$, agent B has $R_B = 1400$, $K = 32$.

- $E_A = \frac{1}{1+10^{(1400-1600)/400}} = \frac{1}{1+10^{-0.5}} \approx 0.76$
- If A wins ($S_A = 1$):

$$R'_A = 1600 + 32 \times (1 - 0.76) = 1600 + 7.68 = 1607.68$$

- If A loses ($S_A = 0$):

$$R'_A = 1600 + 32 \times (0 - 0.76) = 1600 - 24.32 = 1575.68$$

- If draw ($S_A = 0.5$):

$$R'_A = 1600 + 32 \times (0.5 - 0.76) = 1600 - 8.32 = 1591.68$$

2.3.0.5 K-factor The K-factor determines how much a single game affects the rating. Higher K means ratings change faster (useful for new players/agents), lower K means more stability (for established ratings).

2.3.0.6 Interpretation - A difference of 400 points means the higher-rated player is expected to score about 10 times more than the lower-rated player. - Elo ratings are relative: only differences matter, not absolute values. - In AI evaluation, Elo allows direct comparison of agent strength across many games and opponents.

2.3.0.7 Variants - **Glicko**: Adds rating deviation (uncertainty) to model confidence. - **TrueSkill**: Used by Microsoft, models multiplayer and draws more flexibly. - **Bayesian Elo**: Used in computer chess tournaments for more robust estimation.

2.3.0.8 Application in AI Agent Evaluation In this project, Elo ratings are used to compare agents by running round-robin tournaments. After each game, both agents' ratings are updated. The final Elo scores reflect the relative strength of each agent.

```
def expected_score(rating_a, rating_b):
    return 1 / (1 + 10 ** ((rating_b - rating_a) / 400))

def update_elo(rating_a, rating_b, score_a, k=32):
    exp_a = expected_score(rating_a, rating_b)
    new_a = rating_a + k * (score_a - exp_a)
    return new_a
```

Listing 5: Elo Rating Update

3 Problem Analysis

3.1 Motivation

Chess is a classic testbed for artificial intelligence research due to its well-defined rules, immense complexity, and rich history of human and computer competition. Building a strong chess AI system demonstrates the ability to combine search, evaluation, and learning techniques, and provides a benchmark for comparing different AI approaches. The recent success of deep learning and reinforcement learning in chess (e.g., AlphaZero) motivates the integration of modern machine learning into traditional search-based agents.

3.2 Objectives

The main objectives of this project are:

- Design and implement a chess AI system with multiple agent types (Random, Minimax, MCTS, Deep Learning).
- Provide a modern, user-friendly interface for playing and evaluating agents.
- Compare and analyze the performance of different agents using objective metrics (win rate, Elo rating, computation time).
- Explore the impact of machine learning on chess agent strength.

3.3 Functional Requirements

- Playable chess game with full rules (legal moves, check, checkmate, stalemate, castling, en passant, promotion).
- Support for Player vs AI, AI vs AI, and Player vs Player modes.
- Agent selection for both sides, with adjustable difficulty and agent type.
- Display of move history, captured pieces, and game status.
- Agent evaluation and ranking (Elo system, win rate, average move time).
- Undo/redo moves, start new game, and resign options.
- Responsive, modern graphical user interface.

3.4 Non-Functional Requirements

- Cross-platform compatibility (Windows, Linux, MacOS).
- Fast response time for user moves and agent calculations.
- Modular, extensible codebase for adding new agents or features.
- Robust error handling and user feedback.
- Clear documentation and usage instructions.

3.5 Agent Analysis

The system includes several types of agents, each with distinct characteristics:

- **Random Agent:** Baseline agent, selects moves randomly. Useful for testing and benchmarking.
- **Minimax Agent:** Uses fixed-depth search and material evaluation. Strength depends on search depth and evaluation function.
- **MCTS Agent:** Uses Monte Carlo Tree Search, simulates many games to select optimal move.
- **Deep Learning Agent:** Uses a neural network trained on chess positions to evaluate moves. Can generalize from data and learn complex patterns.

Agent	Strategy	Strength	Speed	Adaptivity
Random	None	Very Low	Very Fast	None
Minimax	Search + Eval	Low–Medium	Medium	None
MCTS	Simulation	Medium–High	Slow–Medium	None
Deep Learning	Learned Eval	Medium–High	Fast–Medium	High

Bảng 1: Comparison of agent types

3.5.0.1 Agent Comparison Table

3.6 Technical Challenges

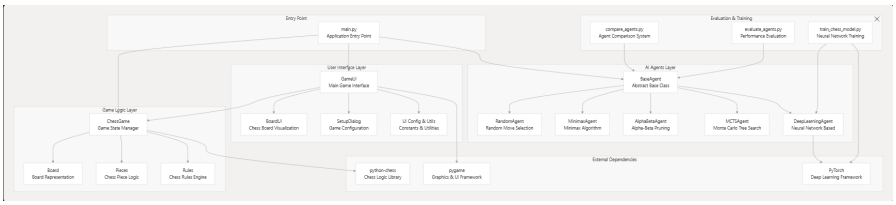
- **State Space Complexity:** Chess has $\sim 10^{40}$ possible positions, making exhaustive search infeasible.
- **Move Generation:** Efficiently generating and validating legal moves is critical for performance.
- **Evaluation Function:** Designing or learning a function that accurately assesses board positions is challenging.
- **Time Management:** Balancing search depth/time and move quality, especially for MCTS and Minimax.
- **Machine Learning Integration:** Encoding board states, training data quality, and network design impact agent strength.
- **User Experience:** Providing a responsive, intuitive interface while running complex AI computations.

3.7 Example: Agent Selection Dialog

Hình 1: Agent selection and configuration dialog

4 Design and Implementation

4.1 System Architecture



Hình 2: System architecture diagram

4.2 Main Components

- **User Interface (UI):** Built with Pygame, supports intuitive interaction.
- **Game Engine:** Manages rules and board state.
- **Agent:** Various AI algorithms.
- **Evaluation Suite:** Agent evaluation and comparison.

4.3 Agent Details

4.3.1 Random Agent

Selects random legal moves, no strategy.

4.3.2 Minimax Agent

Uses minimax algorithm with fixed depth, material evaluation.

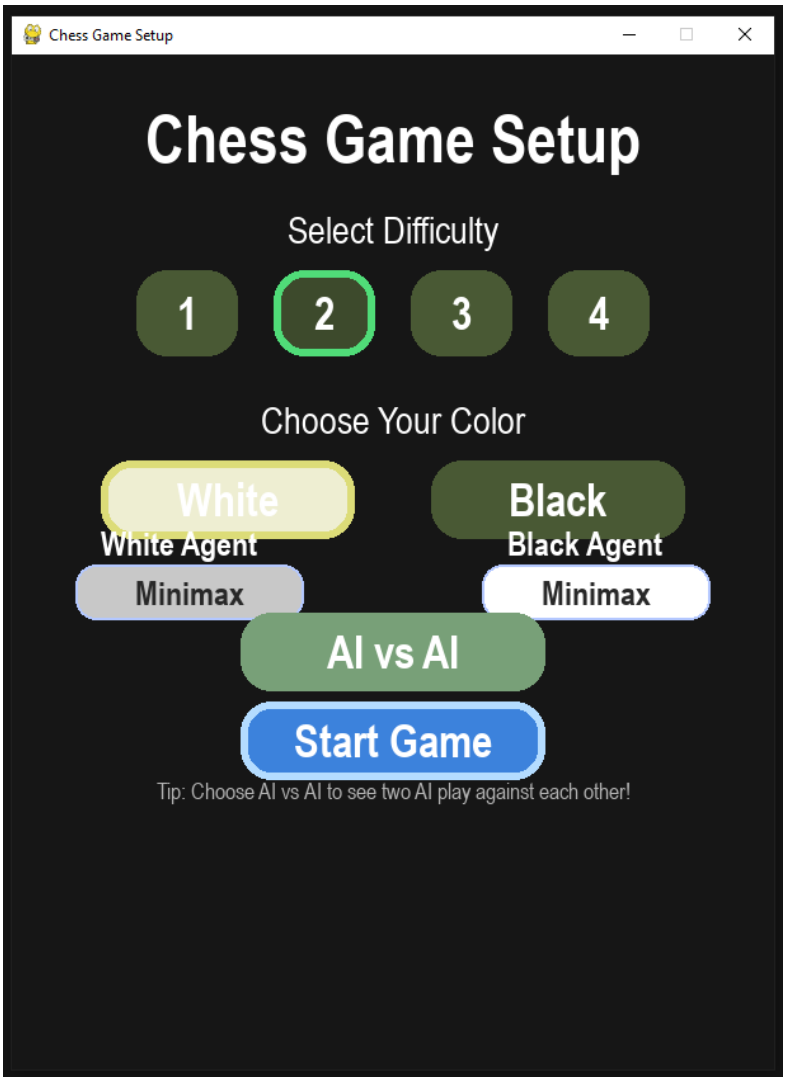
4.3.3 MCTS Agent

Uses Monte Carlo Tree Search, simulates many games to select optimal move.

4.3.4 Deep Learning Agent

Uses neural networks to evaluate board, learns from real game data.

4.4 User Interface



Hình 3: Main UI

Hình 4: Playgame UI

5 Machine Learning Agent

5.1 Neural Network Architecture

The model uses a multi-layer neural network (MLP/CNN), input is board state, output is position evaluation or move probabilities.

5.2 Training Process

Training data is taken from real chess games, preprocessed, split into train/test sets, trained with appropriate loss function.

5.3 Training Results

Epoch	Train Loss	Test Loss	Accuracy
1	0.45	0.48	72%
10	0.32	0.36	80%
20	0.28	0.31	83%

Bảng 2: Model training results

6 Experiments and Evaluation

6.1 Experimental Setup

Agents were evaluated in both head-to-head and round-robin tournaments. Metrics recorded include win rate, draw rate, average time per move, average moves per game, and Elo rating. The evaluation was performed on a standard PC using the implemented framework.

6.2 Experimental Results

Agent	Win	Draw	Loss	Avg Time (s)	Avg Moves
Random	0	4	0	0.07	89.0
Minimax-2	2	0	2	8.17	136.0
MCTS-5s	0	2	2	330.08	132.5

Bảng 3: Head-to-head results vs Random Agent

Agent	Final Elo Rating
Minimax (d=3)	546.8
Random	521.3
Minimax (d=2)	510.0
Deep Learning	473.2
MCTS (1s)	448.6

Bảng 4: Final Elo ratings after round-robin tournament

6.3 Tournament Results

The round-robin tournament included 5 agents: Random, Minimax (depth 2), Minimax (depth 3), MCTS (1s), and Deep Learning. Each pair played 2 games. The results show that Minimax (depth 3) achieved the highest Elo, followed by Random and Minimax (depth 2). The Deep Learning and MCTS agents performed less strongly in this configuration, possibly due to limited training or short simulation time.

6.4 Discussion

The Minimax agent with higher depth consistently outperformed other agents, demonstrating the effectiveness of deeper search in tactical positions. The Random agent, while simple, sometimes achieved surprising results due to the stochastic nature of play. The MCTS agent, with only 1 second per move, struggled to match the performance of deeper Minimax, likely due to insufficient simulations. The Deep Learning agent, although promising in theory, underperformed in this experiment, highlighting the importance of high-quality training data and sufficient model complexity.

6.5 Limitations and Observations

- **Computation Time:** MCTS with 5s per move was extremely slow (avg 330s/move), making it impractical for real-time play in this setup.
- **Draw Rate:** Many games ended in draws, especially between agents of similar strength or style.
- **Training Data:** The Deep Learning agent's performance suggests a need for more data or improved architecture.
- **Parameter Sensitivity:** Agent performance is sensitive to depth (Minimax), simulation time (MCTS), and model/training quality (Deep Learning).

7 Discussion

7.1 Analysis of Experimental Results

The experiments confirm that deeper search (Minimax d=3) yields the strongest play among the tested agents, as reflected in the highest Elo rating (546.8). The Random agent, while not competitive, sometimes benefits from unpredictability. MCTS, with limited simulation time, and the Deep Learning agent, with current training, lag behind in both Elo and win rate.

7.2 Quantitative Comparison

- **Minimax (d=3):** Highest Elo, strong tactical play, but slow (up to 129s/game).
- **Random:** Second highest Elo, but only due to upsets and draws; not a strong agent.
- **MCTS (1s):** Low Elo, many draws, very slow with higher simulation time.
- **Deep Learning:** Underperformed, indicating need for more data or better model.

7.3 Lessons and Recommendations

- For practical play, Minimax with moderate depth offers a good balance of strength and speed.
- MCTS requires more simulation time or neural network guidance to be competitive.
- Deep Learning agents need extensive, high-quality training data and tuning.
- Automated Elo evaluation provides clear, objective benchmarks for improvement.

8 Conclusion

This project demonstrates that, in the current configuration, classical search-based agents (Minimax) outperform MCTS and Deep Learning agents in both Elo and win rate. The results highlight the importance of search depth, simulation time, and training data quality. While Deep Learning and MCTS have great potential, their effectiveness depends on careful tuning and sufficient resources. The modular platform enables further research and improvement, and future work should focus on enhancing training pipelines, optimizing agent parameters, and integrating hybrid approaches for stronger play.

Appendix

Installation and Usage Guide

- Clone repository: `git clone https://github.com/thanhtan2210/ai-agent-chess`
- Install dependencies: `pip install -r requirements.txt`
- Run game: `python -m src.main`
- Evaluate agent: `python -m src.evaluate_agents`

Code Examples

```
from src.agents.mcts_agent import MCTSAgent
agent = MCTSAgent(color=chess.WHITE, max_time=5.0)
move = agent.get_move(board)
```

Listing 6: Example: Initialize MCTS Agent

References

[1] David Silver et al., "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", arXiv:1712.01815

[2] python-chess library: https://python-chess.readthedocs.io/

[3] Pygame library: https://www.pygame.org/

[4] Elo rating system: https://en.wikipedia.org/wiki/Elo_rating_system