Homework 2 - Artificial Intelligence

Introduction:

In the realm of artificial intelligence applied to chess, implementing the Alpha-Beta Minimax search algorithm is pivotal for developing intelligent agents capable of making optimal decisions. The primary goal is to maximize the decision-making performance of the agent, ensuring that each move is evaluated efficiently and accurately.

In this evolutionary implementation, we will explore three stages of Alpha-Beta Minimax algorithm enhancement specifically designed for chess. We will begin by examining how to improve performance by focusing on the most promising states through an H0 static evaluation. Subsequently, we will generalize this approach by considering different HL evaluations to find the optimal parameter l.

Finally, to integrate machine learning into state evaluations, we will develop an H0 function based on specific chess observations. We will introduce a regressor, denoted as R, training it to predict HL values based on H0. This implementation will modify the Alpha-Beta Minimax approach, replacing static evaluations with predictions from the R regressor.

Each stage of this algorithm evolution will be experimented with to evaluate performance compared to the original Alpha-Beta Minimax in the context of chess.

Stage 1: Optimization through H0 "Static" Evaluation

The first stage involves implementing an H0 static evaluation specifically tailored for chess. This approach aims to identify and select the most promising states during the exploration of the game's decision tree, seeking to enhance algorithm efficiency without exploring all possible moves.

Stage 2: Generalization with HL Evaluation

In the second stage, we will expand the concept of H0 static evaluation by introducing an HL evaluation specifically designed for chess. The parameter l will range from 0 to L, where L represents the maximum level of generalization. We will explore the trade-off between precision and computational speed through

experiments with different l values, aiming to pinpoint the optimal choice in a context where L=10 or more.

Stage 3: Introduction of H0 based on Regressor R

The third stage involves integrating machine learning into state evaluations. We will define an H0 function based on specific chess observations and introduce a regressor R trained to predict HL values. This regressor will replace static evaluations in the Alpha-Beta Minimax algorithm, enabling more accurate predictions based on the R model.

Implementation Overview:

In this section, we provide an overview of the implementation approach to address the challenges outlined in the given prompt. The code snippet provided serves as the foundation for our enhanced Alpha-Beta Minimax algorithm tailored for chess.

Heuristic Evaluation:

The implementation introduces a heuristic evaluation strategy to assess the chess board's state efficiently. The `HeuristicChess` class encompasses various static methods that compute heuristic values based on different criteria, such as the overall board status, the number of pieces, and the positioning of pieces.

`check_status_heuristic` Method:

This method evaluates the board status, considering conditions like stalemate, insufficient material, seventy-five moves rule, fivefold repetition, checkmate, and check. The heuristic value reflects the board status, aiding in decision-making during the search process.

`PiecesNumberHeuristic` Method:

The heuristic evaluation based on the number of pieces on the board considers the count of each type of piece for both white and black. The difference in piece counts is used to compute the heuristic value, providing insights into the relative strength of the positions.

`PositionHeuristic` Method:

This heuristic evaluates the board based on the position of each piece. Different weight tables for pawn, knight, bishop, rook, queen, and king contribute to the overall position heuristic. The positioning encourages strategic movements, such as centralizing pawns and knights, avoiding corners with bishops, and placing rooks on the 7th rank.

`evaluate_board` Method:

This method combines the results of the number and position heuristics to provide an overall evaluation of the chess board. The final heuristic value is determined by the combination of these factors, reflecting the agent's perception of the board's strategic advantages.

Machine Learning Integration:

The code further extends the implementation by incorporating machine learning for heuristic prediction. The `MLPRegressor` from the scikit-learn library is used to train a regressor (`R`) for predicting heuristic values based on static board evaluations.

This implementation allows the algorithm to learn from gameplay data, enhancing its ability to evaluate positions accurately and efficiently during the decision-making process.

Agent and Player Implementation:

In this section, we'll examine the implementation of agents and players within the context of chess. The provided code includes the definition of a generic agent and two types of players: the Greedy player and the MinMax player. Subsequently, we'll introduce the MinMax player with optimizations to speed up the search.

`Agent` Class:

The `Agent` class represents a generic agent capable of applying different solution search strategies. Each agent is associated with a player identifier (e.g., 'WHITE' or 'BLACK') and a specific solver.

`GreedyPlayer` Class:

The `GreedyPlayer` class is a type of agent that makes "greedy" choices. In other words, at any given moment, it chooses the BEST move for THAT MOMENT without considering where it might lead in the long term. This player uses a heuristic strategy to evaluate legal moves on the chessboard and selects the one with the highest score.

`MinMaxPlayer` Class:

The `MinMaxPlayer` class represents an agent that employs the MinMax algorithm to perform a search in the move tree. The MinMax algorithm is implemented with alpha-beta pruning to avoid considering values of possible states that are unnecessary. The agent considers legal moves, evaluating the best possible move based on the specified depth in the search.

`MinMaxPlayerSpeedUp` Class:

The `MinMaxPlayerSpeedUp` class extends the `MinMaxPlayer` class and implements optimizations to accelerate the search. In particular, the agent explores only promising subtrees, reducing the number of nodes considered at each level. Optimizations include evaluating and selecting promising nodes based on a specified level (`l`) and the number of best nodes to consider (`k`).

The agent stores observations such as the chessboard state, the chosen move, the best heuristic value, the piece count on the board, the threat count, and the pawn structure. These data can be used for future analysis and evaluations.

Each player implements the `move` method, which returns the chosen move in UCI (Universal Chess Interface) format.

Game and Chess Classes:

The provided code introduces two classes, `Game` and `Chess`, to manage and execute a chess game. These classes orchestrate the interaction between players, the game board, and the flow of the game.

`Game` Class:

The `Game` class is a generic representation of a game, designed to handle different types of games. It is initialized with two players or agents (`A1` and `A2`) and the game board (`Board`). The game keeps track of the game's history, recording moves as the game progresses.

`Chess` Class:

The `Chess` class specifically represents a chess game. It is initialized with two players or agents (`A1` and `A2`) and the chessboard (`Board`). Similar to the `Game` class, it maintains a history of moves made during the game.

Methods:

- `__init__` Method:
  - Parameters:
    - `A1`: The first player or agent.
    - `A2`: The second player or agent.
    - `Board`: The chessboard.
  - Initializes the `Chess` game with the specified players and the chessboard.

- `Begin` Method:
  - Initializes the chess game by making a random number of initial moves.
  - Returns `None`.

- `Play` Method:
  - Plays the chess game until it is over, alternating moves between players.
  - Returns a tuple containing the number of moves made and the result of the game.

- `startGame` Method:
  - Starts the chess game by initializing and playing.
  - Returns the result of the game after playing.

PredictivePlayer and its Variants:

The introduced code extends the concept of a chess player with the `PredictivePlayer` class and its variants, providing a more sophisticated approach by incorporating predictive modeling for move selection.

`PredictivePlayer` Class:

The `PredictivePlayer` class is initialized with a player color (`'WHITE'` or `'BLACK'`) and a regression model (`regressor`). This player employs a predictive approach by using a regression model to estimate the value of a move on the chessboard. The features used for prediction include the mapped board state, piece count, threat count, and pawn structure.

Methods:

- `__init__` Method:
  - Parameters:
    - `player`: The color of the player, either `'WHITE'` or `'BLACK'`.
    - `regressor`: The regression model used for move prediction.
  - Initializes the `PredictivePlayer` with the specified player color and regression model.

- `predictedValue` Method:
  - Parameters:
    - `board`: The current state of the chessboard.
  - Predicts the value of the move using the provided regression model.
  - Returns the predicted value of the move.

- `update_features` Method:
  - Parameters:
    - `board`: The current state of the chessboard.
  - Updates features used for prediction based on the current board state.

- `move` Method:
  - Parameters:
    - `board`: The current state of the chessboard.
  - Gets the predicted best move based on the regression model.
  - Returns the predicted best move.

`PredictivePlayerGreedy` Class:

The `PredictivePlayerGreedy` class is a variant of `PredictivePlayer` that selects the move with the highest predicted value in a greedy manner.

Methods:

- `move` Method:
  - Parameters:
    - `board`: The current state of the chessboard.
  - Selects the move with the highest predicted value in a greedy manner.
  - Returns the selected move.

`PredictivePlayerMinMax` Class:

The `PredictivePlayerMinMax` class is another variant of `PredictivePlayer` that employs the MinMax algorithm to select the move maximizing the predicted value.

Methods:

- `move` Method:
  - Parameters:
    - `board`: The current state of the chessboard.
  - Uses the MinMax algorithm to select the move maximizing the predicted value.
  - Returns the selected move.

ChessDatasetProcessor Class:

The `ChessDatasetProcessor` class processes a given chess dataset in pandas DataFrame format. It includes methods for converting FEN strings, splitting strings into lists of characters, and converting values to floats. The class processes the dataset by applying a Stockfish evaluation to each FEN position, creating a DataFrame with separate columns for each position, and converting categorical features using predefined mappings.

Methods:

- `__init__` Method:
  - Parameters:
    - `dataset`: The input dataset in pandas DataFrame format.
  - Initializes the `ChessDatasetProcessor` with the provided dataset.

- `toZeroes` Method:
  - Parameters:
    - `string`: The FEN representation of the chess position.

  - Converts a string representation of a chess position by replacing numbers with zeroes.
  - Returns the modified FEN representation.

- `split` Method:
  - Parameters:
    - `string`: The input string.
  - Splits a string into a list of characters.
  - Returns a list of characters.

- `convert_to_float` Method:
  - Parameters:
    - `value`: The input value.
  - Converts a value to a float.
  - Returns the converted float value.

- `convertPieces` Method:
  - Converts categorical features in the dataset using predefined mappings.

- `process_dataset` Method:
  - Processes the chess dataset by applying Stockfish evaluation, creating a DataFrame with separate columns for each position, and converting categorical features.
  - Returns the processed dataset.

Regressor Section:

The Regressor section involves using a Multi-Layer Perceptron (MLP) Regressor for predicting chess move scores based on input features.

Data Preprocessing:
- Fill NaN values with the mean of each column.

Extract Input and Target Columns:
- Extract the target variable (`Score`) and the input features (`X`) from the dataset.

Split Data into Training and Testing Sets:
- Split the data into training and testing sets, allocating 80% for training and 20% for testing.

MLP Regressor Training:
- Initialize an MLP Regressor with two hidden layers of 300 neurons each, a maximum of 200 iterations, and verbose output during training.
- Fit the MLP Regressor to the training data.

Main Class:

The `Main` class orchestrates the execution of multiple chess matches between two players and provides an overview of the results.

Initialization:
- `player1`: The first player or agent.
- `player2`: The second player or agent.
- `game`: The game instance (e.g., `Chess`) used for matches.
- `matches`: The number of matches to be played.
- `view`: A flag indicating whether to display detailed match information (default is `False`).

Methods:
- `play()`: Initiates and oversees the playing of multiple matches.
  - Tracks wins for white and black players, draws, total moves, and average moves per game.
  - Prints detailed match information if the `view` flag is set.
  - Displays execution time and average time per match.
  - Calls the `plot_results()` function to visualize match outcomes.

Statistical Analysis and Results

In this section, we will analyze and interpret the results of different chess match scenarios involving various agents and strategies.

PredictiveMinMax vs. GreedyPlayer

Agent Configuration:
- Agent1: PredictivePlayerMinMax ("WHITE") with a depth of 3 and a regression model (mlp).
- Agent2: GreedyPlayer ("BLACK") with a heuristic-based chess solver.

Results:
- WHITE Player Wins: 3

- BLACK Player Wins: 0
- Draws: 97
- Total Moves: 4099
- Average Moves per Game: 40.99
- WHITE Player Win Percentage: 3.0%
- BLACK Player Win Percentage: 0.0%
- Draw Percentage: 97.0%
- Execution time: 4496.16 seconds
- Average Time per Match: 44.96 seconds

Interpretation:
In this scenario, the PredictivePlayerMinMax (mlp) with a depth of 3 achieved a small percentage of victories (3.0%), while the GreedyPlayer (heuristic-based) failed to secure any wins. The high draw percentage indicates a balanced play between the two agents.

PredictiveMinMax vs. MinMaxPlayerSpeedUp

Agent Configuration:
- Agent1: PredictivePlayerMinMax ("WHITE") with a depth of 3 and a regression model (mlp).
- Agent2: MinMaxPlayerSpeedUp ("BLACK") with a heuristic-based chess solver, depth 3, level 2, and considering 8 nodes.

Results:
- WHITE Player Wins: 21
- BLACK Player Wins: 28
- Draws: 51
- Total Moves: 4132
- Average Moves per Game: 41.32
- WHITE Player Win Percentage: 21.0%
- BLACK Player Win Percentage: 28.0%
- Draw Percentage: 51.0%
- Execution time: 4561.65 seconds
- Average Time per Match: 45.62 seconds

Interpretation:
In this matchup, PredictivePlayerMinMax demonstrated improved performance, securing a higher percentage of wins (21.0%) compared to MinMaxPlayerSpeedUp (28.0%). The relatively high draw percentage suggests a competitive balance between the two strategies.

PredictiveMinMax vs. PredictiveGreedy

Agent Configuration:
- Agent1: PredictivePlayerMinMax ("WHITE") with a depth of 3 and a regression model (mlp).
- Agent2: PredictivePlayerGreedy ("BLACK") with the same regression model (mlp).

Results:
- WHITE Player Wins: 13
- BLACK Player Wins: 0
- Draws: 87
- Total Moves: 4087
- Average Moves per Game: 40.87
- WHITE Player Win Percentage: 13.0%
- BLACK Player Win Percentage: 0.0%
- Draw Percentage: 87.0%
- Execution time: 4490.8 seconds
- Average Time per Match: 44.91 seconds

Interpretation:
In this scenario, PredictivePlayerMinMax outperformed PredictivePlayerGreedy, securing a higher percentage of wins (13.0%). However, a significant draw percentage suggests a cautious and balanced playing style.

PredictiveGreedy vs. MinMaxPlayerSpeedUp

Agent Configuration:
- Agent1: PredictivePlayerGreedy ("WHITE") with a regression model (mlp).
- Agent2: MinMaxPlayerSpeedUp ("BLACK") with a heuristic-based chess solver, depth 3, level 2, and considering 8 nodes.

Results:
- WHITE Player Wins: 0
- BLACK Player Wins: 54
- Draws: 46
- Total Moves: 2855
- Average Moves per Game: 28.55
- WHITE Player Win Percentage: 0.0%
- BLACK Player Win Percentage: 54.0%
- Draw Percentage: 46.0%

- Execution time: 917.94 seconds
- Average Time per Match: 9.18 seconds

Interpretation:
In this matchup, PredictivePlayerGreedy failed to secure any wins against the aggressive play of MinMaxPlayerSpeedUp, resulting in a high percentage of victories for the latter. The low average moves per game indicate a relatively quick resolution of matches.

These analyses provide insights into the performance and dynamics of different agent configurations and strategies in chess matches.

Conclusions

The presented chess match scenarios involving various agents and strategies offer valuable insights into the strengths and weaknesses of different approaches. Here are some key conclusions drawn from the analyses:

1. PredictivePlayerMinMax Performance:
   - In matchups against GreedyPlayer and MinMaxPlayerSpeedUp, PredictivePlayerMinMax demonstrated a competitive performance.
   - Achieved a noticeable win percentage (21.0%) against MinMaxPlayerSpeedUp, showcasing its effectiveness in strategic decision-making.

2. PredictivePlayerGreedy Considerations:
   - PredictivePlayerGreedy, while efficient in some scenarios, struggled against the aggressive play of MinMaxPlayerSpeedUp.
   - Failed to secure wins in the matchup against MinMaxPlayerSpeedUp, resulting in a high percentage of losses (54.0%).

3. Draw Prevalence:
   - Across various scenarios, a significant number of matches ended in draws, indicating a balanced and cautious playing style.
   - High draw percentages in scenarios involving PredictivePlayerMinMax and PredictivePlayerGreedy suggest a more conservative approach.

4. Execution Time Observations:
   - Longer execution times were observed in scenarios involving PredictivePlayerMinMax and GreedyPlayer matchups.
   - MinMaxPlayerSpeedUp demonstrated quicker match resolutions, potentially due to its heuristic-based decision-making.

5. Strategic Diversity:
   - The diverse strategies employed by PredictivePlayerMinMax, PredictivePlayerGreedy, GreedyPlayer, and MinMaxPlayerSpeedUp contribute to varied and interesting match outcomes.
   - The combination of predictive modeling and heuristic-based approaches offers a spectrum of playing styles.

6. Agent Tuning Impact:
   - The depth parameter in MinMaxPlayerSpeedUp and PredictivePlayerMinMax significantly influences their performance.
   - Agent tuning, such as adjusting the depth parameter and using regression models like mlp, plays a crucial role in achieving desirable outcomes.

In conclusion, the presented analyses provide valuable insights into the dynamics of chess matches between diverse agents. The results emphasize the importance of strategic diversity, agent tuning, and the interplay between predictive modeling and heuristic-based decision-making in achieving competitive outcomes. Continued exploration and refinement of these approaches hold the potential for further advancements in AI-driven chess gameplay.

Lorenzo Tucceri Cimini
N. 294425