A3: Chess

Kevin King

COSC 76 23F

Minimax

The `MinimaxAI` class uses the minimax algorithm, which provides the foundation for the chess project. Read the text below for a description of how the code flows

- The `choose_move` function is the starting point for selecting the best move for a given chess board state. It initiates a timer to measure the time taken for selecting a move. For each legal move available, it simulates the move, calculates its value using the value function, and keeps track of the move with the highest value. It then outputs the best move, the total nodes searched, and the time taken to decide the move.
- The `value` function determines the worth of a given board state. If the state meets the criteria defined in the `cutoff_test` (i.e., reaching the maximum depth or the game being over), it calculates the value of the board using the `evaluation` function. Otherwise, it recursively explores further moves using either the `max_value` or `min_value` function, depending on whether the current player is maximizing or minimizing.
- The `max_value` function explores all possible moves for the maximizing player (typically white) and returns the highest value obtained from the value function, indicating the most favorable outcome for the maximizing player.
- The `min_value` function does the same for the minimizing player (typically black), returning the lowest value obtained, indicating the most favorable outcome for the minimizing player.
- The `cutoff_test` function determines when to stop the recursive search. It returns True if the specified depth has been reached or the game is over, signaling the algorithm to compute the final value of the board state.
- The `evaluation` function calculates the value of a given board state. It assigns scores based on the pieces present on the board and their corresponding values stored in the `piece_values` dictionary. A higher score indicates a more favorable state for the maximizing player, while a lower score is better for the minimizing player.

Finally, the `choose_move` function prints the selected move, the number of nodes searched, and the time taken to compute the move. The num_calls counter is reset for the next move, and the selected move is returned to be executed on the board.

Evaluation

As mentioned in the previous section, of my report, for my `evaluation` function, I assigned numerical values to each of the chess pieces:

- pawn: 1
- bishop and knight: 3
- rook: 5
- queen: 9
- king: 1000 (an arbitrarily large number)

This function is a crucial component as it estimates the value of a given chess board state. It helps the AI to determine how favorable or unfavorable a particular configuration of pieces on the board is. When the search reaches the predetermined depth or encounters a terminal state (like checkmate or stalemate), the `evaluation` function assesses the desirability of the board state. These evaluations guide the AI's decision-making process, helping it select the move leading to the most favorable outcome.

Here are the components of the function:

- Checkmate situation: The function first checks if the current board state is a checkmate. In the case of a checkmate, it assigns a high positive value if the AI wins and a large negative value if the AI loses. This prioritizes moves leading to a victory and avoids those leading to a defeat.
- Piece Value situation: If it's not a checkmate situation, the function calculates the total value of all pieces on the board for both players. Each type of piece is assigned a specific value, reflecting its importance and strategic role in the game.

The value of the board is calculated by summing the values of all pieces belonging to the AI and subtracting the total value of all pieces belonging to the opponent. This gives a net value of the board from the AI's perspective:

- Positive Value: Indicates that the AI has a material advantage.
- Negative Value: Indicates that the opponent has a material advantage.
- Zero: Indicates an equal material balance between both players.

In the below code from `MinimaxAI.py`, I show how the `evaluation` function is used to evaluate winning and blocking positions at varying depths. Based on the results, several observations can be made:

- Winning Position
  - Move: The AI consistently chooses the move g5g7, capturing the pawn and putting the black king in check.
  - Value: The value of this move is 1008, indicating a winning position. The value is derived from the evaluation function. The AI gives a positive value when the move benefits the white player and a negative value when it benefits the black player. Here, 1000 is the value of the king and 8 represents the accumulated value of the other pieces on the board.
  - Nodes Searched: As the depth increases, the number of nodes searched also increases exponentially. This is expected as the AI explores more possible moves and their outcomes.
  - Time Elapsed: Corresponding with the increase in nodes searched, the time taken for the AI to decide on the move also increases.
- Blocking Position
  - Move: The AI consistently recommends the move d4e5 across all depths. This move is likely chosen because it either counters an immediate threat or creates an opportunity for the AI.
  - Value: There's a slight fluctuation in the value assigned to the chosen move at different depths, likely due to the AI evaluating future positions differently as the search depth increases. However, the selected move remained the same, indicating that, at least up to depth 4, there are no better alternatives discovered.
  - Nodes Searched and Time Elapsed: These also increased significantly with the depth, indicating the growing complexity and the increased number of possible board states the AI has to evaluate.

```
/usr/local/bin/python3.9 /Users/kevin/PycharmProjects/cs76/Assignments/A3 - Chess/MinimaxAI.py
WINNING POSITION
. . . . . .
. . . . . p k  .
. . . . . .
```

```
. . . . . . Q .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . K

________________
a b c d e f g h

Testing winning position at depth 2:
Minimax AI recommending move g5g7 with value 1008
Nodes searched: 163
Time elapsed: 0.007574796676357 42
At depth 2, MinimaxAI recommends move: g5g7, with value 1008


Testing winning position at depth 3:
Minimax AI recommending move g5g7 with value 1008
Nodes searched: 3610
Time elapsed: 0.12944674491882324
At depth 3, MinimaxAI recommends move: g5g7, with value 1008


Testing winning position at depth 4:
Minimax AI recommending move g5g7 with value 1008
Nodes searched: 20461
Time elapsed: 0.8918850421905518
At depth 4, MinimaxAI recommends move: g5g7, with value 1008



______________________

BLOCKING POSITION
r . b . . r k .
p p p p . p p p
. . n . . . . .
. . b . q . . .
. . . P . . . .
. . P . . . . .
P P P . . P P P
R . B Q K B N R

________________
a b c d e f g h

Testing blocking position at depth 2:
Minimax AI recommending move d4e5 with value 9
Nodes searched: 257
Time elapsed: 0.019941091537475586
At depth 2, MinimaxAI recommends move: d4e5, with value 10


Testing blocking position at depth 3:
Minimax AI recommending move d4e5 with value 10
Nodes searched: 7403
Time elapsed: 0.6226732730865479
At depth 3, MinimaxAI recommends move: d4e5, with value 10


Testing blocking position at depth 4:
```

```
Minimax AI recommending move d4e5 with value 8
Nodes searched: 294104
Time elapsed: 24.530704021453857
At depth 4, MinimaxAI recommends move: d4e5, with value 10
```

Iterative Deepening

The `IterativeDeepeningAI` class is an enhancement of the MinimaxAI. It incorporates the concept of Iterative Deepening Search (IDS) to optimize the performance and accuracy of the AI in choosing the best move.

The AI doesn't directly search to the maximum depth. Instead, it starts from depth 0 and incrementally increases the depth limit in each iteration. This is done until it reaches the predefined maximum depth.

`choose_move` Method:

- The AI initializes with a given depth. It sets best_move to None initially.
- The AI iteratively deepens its search. For each depth from 0 to the maximum depth:
  - An instance of MinimaxAI is created with the current depth.
  - The choose_move method of the MinimaxAI instance is called to get the best move for the current depth.
  - This best move is then stored, and the process repeats for the next depth.
- After each depth level is explored, the AI prints out the best move found at that depth.
- The process is repeated until all depth levels up to the maximum are explored.
- The final best move obtained after exploring up to the maximum depth is returned.

While the provided code uses the MinimaxAI, it can be potentially optimized further by incorporating AlphaBetaAI. It will help in pruning the search tree and reducing the computational load, making the search process even more efficient.

I also implemented iterative deepening in the `AlphaBetaAI2` class without needing the separate class to handle iterating through the different depths. I use this class as part of an extension that uses advanced move reordering.

Alpha-Beta Pruning

The `AlphaBetaAI` class builds upon `MinimaxAI`, incorporating alpha-beta pruning to optimize the search process in the game tree.

- Two additional parameters, alpha and beta, are introduced in the value, min_value, and max_value functions. Alpha is the best value that the maximizer currently can guarantee at that level or above, while beta is the best value that the minimizer can guarantee at that level or above.
- The `choose_move` function still iterates through all legal moves, but it also incorporates alpha-beta values to cut off the search where it's not necessary to explore further. The alpha-beta values are updated during the search to reflect the best guaranteed scores for both players as moves are examined.
- In the `max_value` function, if the value found is greater than or equal to beta, the search is cut off because the minimizer has a better option elsewhere, making further search in this branch unnecessary.
- Similarly, in the `min_value` function, if the value is less than or equal to alpha, it cuts off the search as the maximizer has a better option elsewhere. These two functions thus employ pruning to avoid unnecessary calculations, making the AI faster and more efficient.
- The `reorder_moves` function reorders the moves to give priority to capturing moves. This can help in more efficient pruning as capturing moves are often (but not always) better, so exploring them first can lead to earlier cut-

offs. I created a `use_reorder_func` boolean variable that can be toggled based on whether you want to use the function.

See the below output from the `AlphaBetaVsMinimax.py` test file for a comparison of `AlphaBetaAI` and `MinimaxAI` demonstrating that for the same depth, for various positions, alpha-beta explored fewer nodes and yet gave a move (leading to a position satisfying the cut-off test) with the same value:

```
/usr/local/bin/python3.9 /Users/kevin/PycharmProjects/cs76/Assignments/A3 - Chess/AlphaBetaVsMinimax.
Testing at depth 2:

Initial Position:
. . . . . . . .
. . . . . p k .
. . . . . . . .
. . . . . . Q .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . K

Minimax AI recommending move g5g7 with value 1008
Nodes searched: 163
Time elapsed: 0.00800776481628418
Alpha-Beta AI recommending move g5g7 with value 1008
Nodes searched: 163
Time elapsed: 0.006896018981933594
MinimaxAI Move: g5g7, Value: 1008, Nodes Searched: 164, Time Elapsed: 0.008095979690551758s
AlphaBetaAI Move: g5g7, Value: 1008, Nodes Searched: 54, Time Elapsed: 0.006928205490112305s
Move values are the same
AlphaBetaAI searched fewer nodes than MinimaxAI


Initial Position:
r . b . . r k .
p p p p . p p p
. . n . . . . .
. . b . q . . .
. . . P . . . .
. . P . . . . .
P P P . . P P P
R . B Q K B N R

Minimax AI recommending move d4e5 with value 9
Nodes searched: 257
Time elapsed: 0.027943134307861328
Alpha-Beta AI recommending move d4e5 with value 9
Nodes searched: 257
Time elapsed: 0.023318052291870117
MinimaxAI Move: d4e5, Value: 9, Nodes Searched: 258, Time Elapsed: 0.027965307235717773s
AlphaBetaAI Move: d4e5, Value: 9, Nodes Searched: 255, Time Elapsed: 0.023351669311523438s
Move values are the same
AlphaBetaAI searched fewer nodes than MinimaxAI

--------------------
Testing at depth 3:

Initial Position:
. . . . . . . .
```

```
. . . . . p k .
. . . . . . . .
. . . . . . Q .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . K

Minimax AI recommending move g5g7 with value 1008
Nodes searched: 3610
Time elapsed: 0.1346750259399414
Alpha-Beta AI recommending move g5g7 with value 1008
Nodes searched: 1490
Time elapsed: 0.06188607215881348
MinimaxAI Move: g5g7, Value: 1008, Nodes Searched: 3611, Time Elapsed: 0.1347062587738037s
AlphaBetaAI Move: g5g7, Value: 1008, Nodes Searched: 851, Time Elapsed: 0.061916351318359375s
Move values are the same
AlphaBetaAI searched fewer nodes than MinimaxAI


Initial Position:
r . b . . r k .
p p p p . p p p
. . n . . . . .
. . b . q . . .
. . . P . . . .
. . P . . . . .
P P P . . P P P
R . B Q K B N R

Minimax AI recommending move d4e5 with value 10
Nodes searched: 7403
Time elapsed: 0.6274421215057373
Alpha-Beta AI recommending move d4e5 with value 10
Nodes searched: 6720
Time elapsed: 0.595588207244873
MinimaxAI Move: d4e5, Value: 10, Nodes Searched: 7404, Time Elapsed: 0.6274688243865967s
AlphaBetaAI Move: d4e5, Value: 10, Nodes Searched: 6403, Time Elapsed: 0.5956158638000488s
Move values are the same
AlphaBetaAI searched fewer nodes than MinimaxAI

--------------------
Testing at depth 4:

Initial Position:
. . . . . . . .
. . . . . p k .
. . . . . . . .
. . . . . . Q .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . K

Minimax AI recommending move g5g7 with value 1008
Nodes searched: 20461
Time elapsed: 0.8445627689361572
Alpha-Beta AI recommending move g5g7 with value 1008
Nodes searched: 2621
Time elapsed: 0.1431267261505127
MinimaxAI Move: g5g7, Value: 1008, Nodes Searched: 20462, Time Elapsed: 0.8445906639099121s
```

```
AlphaBetaAI Move: g5g7, Value: 1008, Nodes Searched: 1714, Time Elapsed: 0.1431589126586914s
Move values are the same
AlphaBetaAI searched fewer nodes than MinimaxAI


Initial Position:
r . b . . r k .
p p p p . p p p
. . n . . . . .
. . b . q . . .
. . . P . . . .
. . P . . . . .
P P P . . P P P
R . B Q K B N R

Minimax AI recommending move d4e5 with value 8
Nodes searched: 294104
Time elapsed: 24.659616947174072
Alpha-Beta AI recommending move d4e5 with value 8
Nodes searched: 206179
Time elapsed: 18.19606590270996
MinimaxAI Move: d4e5, Value: 8, Nodes Searched: 294105, Time Elapsed: 24.659644842147827s
AlphaBetaAI Move: d4e5, Value: 8, Nodes Searched: 161694, Time Elapsed: 18.19610023498535s
Move values are the same
AlphaBetaAI searched fewer nodes than MinimaxAI

--------------------
```

Transposition Table

The `AlphaBetaAI_TT` class incorporates a transposition table to optimize the alpha-beta pruning algorithm. A transposition table stores the values of previously evaluated board states to avoid recalculating them, making the AI more efficient. Here are the key differences from the basic `AlphaBetaAI` class:

- `trans_table` : The class implements this additional parameter, used to store the evaluated values of board states. This is a dictionary where the keys are hashed board states and the values are the evaluated scores at different depths of the game tree.
- The `value` function checks if the current board state (hashed) is already present in the transposition table at the current depth. If it is, the function returns the stored value instead of recalculating it.
- After evaluating a board state, the function stores the value in the transposition table for future reference.
- Note that the evaluated value is the score based on after the move is made. This was especially evident in the blocking position scenario.

See the below output from `AlphaBetaAI_TT.py` for a comparison of `AlphaBetaAI` and `AlphaBetaAI_TT` :

```
/usr/local/bin/python3.9 /Users/kevin/PycharmProjects/cs76/Assignments/A3 - Chess/AlphaBetaAI_TT.py
WINNING POSITION
. . . . . . . .
. . . . . p k .
. . . . . . . .
. . . . . . Q .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

```
. . . . . . . K

________________
a b c d e f g h

Testing winning position at depth 2:
Alpha-Beta AI recommending move g5g7 with value 1008
        Nodes searched: 163
        Time elapsed: 0.011931896209716797
AlphaBetaAI_TT recommending move g5g7 with value 1008
        Nodes searched: 163
        Time elapsed: 0.028835773468017578
AlphaBetaAI Move: g5g7, Evaluation: 1008
AlphaBetaAI_TT Move: g5g7, Evaluation: 1008
Transposition Table Entries: 25


Testing winning position at depth 3:
Alpha-Beta AI recommending move g5g7 with value 1008
        Nodes searched: 1490
        Time elapsed: 0.11506915092468262
AlphaBetaAI_TT recommending move g5g7 with value 1008
        Nodes searched: 1490
        Time elapsed: 0.29582905769348145
AlphaBetaAI Move: g5g7, Evaluation: 1008
AlphaBetaAI_TT Move: g5g7, Evaluation: 1008
Transposition Table Entries: 163


Testing winning position at depth 4:
Alpha-Beta AI recommending move g5g7 with value 1008
        Nodes searched: 2621
        Time elapsed: 0.24382901191711426
AlphaBetaAI_TT recommending move g5g7 with value 1008
        Nodes searched: 2191
        Time elapsed: 1.185347080230713
AlphaBetaAI Move: g5g7, Evaluation: 1008
AlphaBetaAI_TT Move: g5g7, Evaluation: 1008
Transposition Table Entries: 645


____________________
BLOCKING POSITION
r . b . . r k .
p p p p . p p p
. . n . . . . .
. . b . q . . .
. . . P . . . .
. . P . . . . .
P P P . . P P P
R . B Q K B N R

________________
a b c d e f g h

Testing blocking position at depth 2:
Alpha-Beta AI recommending move d4e5 with value 9
        Nodes searched: 257
        Time elapsed: 0.09503698348999023
AlphaBetaAI_TT recommending move d4e5 with value 9
        Nodes searched: 257
```

```
        Time elapsed: 0.1347050666809082
AlphaBetaAI Move: d4e5, Evaluation: 10
AlphaBetaAI_TT Move: d4e5, Evaluation: 10
Transposition Table Entries: 6


Testing blocking position at depth 3:
Alpha-Beta AI recommending move d4e5 with value 10
        Nodes searched: 6720
        Time elapsed: 1.4163529872894287
AlphaBetaAI_TT recommending move d4e5 with value 10
        Nodes searched: 6720
        Time elapsed: 2.665151834487915
AlphaBetaAI Move: d4e5, Evaluation: 10
AlphaBetaAI_TT Move: d4e5, Evaluation: 10
Transposition Table Entries: 257


Testing blocking position at depth 4:
Alpha-Beta AI recommending move d4e5 with value 8
        Nodes searched: 206179
        Time elapsed: 44.821418046951294
AlphaBetaAI_TT recommending move d4e5 with value 8
        Nodes searched: 197795
        Time elapsed: 61.497178077697754
AlphaBetaAI Move: d4e5, Evaluation: 10
AlphaBetaAI_TT Move: d4e5, Evaluation: 10
Transposition Table Entries: 5080
```

BoardHash

- This class is a wrapper around a chess board state that provides a mechanism to hash the board. This is particularly useful for storing and retrieving board states in a transposition table efficiently.
- The `__hash__` method is overridden to return a hash of the string representation of the board. The hash function in Python returns the hash value of the object (if it has one).
- `hash(str(self.board))` converts the board state to a string and then calculates its hash value. Each unique board state will have a unique hash value, but the hash is derived from the string representation of the board state to ensure consistency and uniqueness.
- The equality method `__eq__` is used implicitly when you try to access, insert, or check for the existence of a BoardHash object in a dictionary (which is used for the transposition table).

Extensions (Bonus)

Zobrist Hashing

I implemented Zobrist Hashing in the `ZobristHash` class. For a given board state, if there is a piece on a given cell, we use the random number of that piece from the corresponding cell in the table. You can set the `use_zobrist_hash` variable to True to use the method. Based on the results below:

- The Zobrist hashing doesn't always search fewer nodes compared to the basic `AlphaBetaAI` , but it does start to for cases with higher depths (e.g., depth 4).
- Note that the evaluated value is the score based on after the move is made. This was especially evident in the blocking position scenario.

```
WINNING POSITION
. . . . . . . .
. . . . . p k .
. . . . . . . .
. . . . . . Q .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . K

________________
a b c d e f g h

Testing winning position at depth 2:
Alpha-Beta AI recommending move g5g7 with value 1008
        Nodes searched: 163
        Time elapsed: 0.016601085662841797
AlphaBetaAI_TT (using Zobrist) recommending move g5g7 with value 1008
        Nodes searched: 163
        Time elapsed: 0.02512526512145996
AlphaBetaAI Move: g5g7, Evaluation: 1008
AlphaBetaAI_TT Move: g5g7, Evaluation: 1008
Transposition Table Entries: 25


Testing winning position at depth 3:
Alpha-Beta AI recommending move g5g7 with value 1008
        Nodes searched: 1490
        Time elapsed: 0.1071169376373291
AlphaBetaAI_TT (using Zobrist) recommending move g5g7 with value 1008
        Nodes searched: 1490
        Time elapsed: 0.1774749755859375
AlphaBetaAI Move: g5g7, Evaluation: 1008
AlphaBetaAI_TT Move: g5g7, Evaluation: 1008
Transposition Table Entries: 163


Testing winning position at depth 4:
Alpha-Beta AI recommending move g5g7 with value 1008
        Nodes searched: 2621
        Time elapsed: 0.23105216026306152
AlphaBetaAI_TT (using Zobrist) recommending move g5g7 with value 1008
        Nodes searched: 2191
        Time elapsed: 0.31595396995544434
AlphaBetaAI Move: g5g7, Evaluation: 1008
AlphaBetaAI_TT Move: g5g7, Evaluation: 1008
Transposition Table Entries: 645


____________________
BLOCKING POSITION
r . b . . r k .
p p p p . p p p
. . n . . . . .
. . b . q . . .
. . . P . . . .
. . P . . . . .
P P P . . P P P
R . B Q K B N R
```

```
----------------
a b c d e f g h

Testing blocking position at depth 2:
Alpha-Beta AI recommending move d4e5 with value 9
        Nodes searched: 257
        Time elapsed: 0.036427974700927734
AlphaBetaAI_TT (using Zobrist) recommending move d4e5 with value 9
        Nodes searched: 257
        Time elapsed: 0.05936884880065918
AlphaBetaAI Move: d4e5, Evaluation: 10
AlphaBetaAI_TT Move: d4e5, Evaluation: 10
Transposition Table Entries: 6


Testing blocking position at depth 3:
Alpha-Beta AI recommending move d4e5 with value 10
        Nodes searched: 6720
        Time elapsed: 0.9892561435699463
AlphaBetaAI_TT (using Zobrist) recommending move d4e5 with value 10
        Nodes searched: 6720
        Time elapsed: 1.723728895187378
AlphaBetaAI Move: d4e5, Evaluation: 10
AlphaBetaAI_TT Move: d4e5, Evaluation: 10
Transposition Table Entries: 257


Testing blocking position at depth 4:
Alpha-Beta AI recommending move d4e5 with value 8
        Nodes searched: 206179
        Time elapsed: 30.14487624168396
AlphaBetaAI_TT (using Zobrist) recommending move d4e5 with value 8
        Nodes searched: 197795
        Time elapsed: 49.0148720741272
AlphaBetaAI Move: d4e5, Evaluation: 10
AlphaBetaAI_TT Move: d4e5, Evaluation: 10
Transposition Table Entries: 5080
```

Advanced Move Reordering

I implemented the `reordered_moves_advanced` function in the `AlphaBetaAI2` class

The `reordered_moves_advanced` function is a refined approach to sort the legal moves for a given board state. The aim is to improve the efficiency of the alpha-beta pruning algorithm by considering the most promising moves first. It's vital because it can lead to more frequent cut-offs, saving computational time. In the enhanced version of `AlphaBetaAI2`, the `reordered_moves_advanced` function utilizes the evaluations stored in the transposition table, avoiding the recalculations and thus speeding up the search process.

I used `test_chess.py` and set player2 to the `RandomAI` with a seed of 1 (to ensure same moves each time)

See below for the amount of time to checkmate for the different classes:

AlphaBetaAI (with `use_reorder_func` set to False):

- `Total Gametime: 67.70714378356934`

AlphaBetaAI (with `use_reorder_func` set to True):

- `Total Gametime: 66.00256991386414`

AlphaBetaAI2:

- `Total Gametime: 13.735483169555664`