Chess AI

By Andy Xu

Minimax and Cutoff Test

To implement depth-limited minimax search, I followed the pseudocode from the textbook. I defined three instance variables: `depth` which represents the maximum depth, `num_nodes` which represents the number of calls to minimax, and `use_ids` which determines whether iterative deepening search should be used or not (discussed more later).

I implemented the `max_value` and `min_value` functions as described in the textbook. I also implemented the `cutoff_test` function that returns true if the terminal state (game over) or the maximum depth has be reached. Finally, I wrote the `utility` function that calculates the utility of given board through a point system of all the pieces on the board.

I noticed that that any maximum depth greater than 4 would slow down the algorithm significantly. The ideal maximum depth seemed to be 3, as it generated more effective moves at a relatively fast speed. At maximum depth 3, the algorithm would make about 10,000 calls to minimax while it would make almost 20000 calls at maximum depth 4.

Evaluation Function

I implemented an evaluation function that utilizes a point system to determine the heuristic. It iterates through the pieces on the board and adds or subtracts the appropriate points for each piece depending on its color.

After implementing the evaluation function, the computer was certainly a lot better at playing chess. It would capture pieces that should be captured and made smart moves. Increasing the depth led to even smarter gameplay but exponentially slower run time which made the game a lot less interesting to play.

Iterative Deepening

I implemented iterative deepening in the `MiniMaxAI` class. To implement this, I added a boolean instance variable called `use_ids`, which is set to true or false depending on the given parameter when a new instance of MiniMaxAI is made. If use_ids is set to true, when the `choose_move` function is called, it will call the `ids_choose_move` function instead of what it usually does when use_ids is false. The `ids_choose_move` function will use a for loop to repeat the minimax algorithm at different depths less than the maximum depth and return the best move out of all the iterations. Through testing and gameplay, it is clear that the computer makes smarter moves as deeper levels are searched, giving it a greater chance of winning.

Alpha-Beta Pruning

I implemented alpha-beta pruning by adding the necessary code described by the textbook to my MinimaxAI algorithm. I also implemented move-reordering by sorting the possible moves based on their utility. Thus, when iterating through each possible move, the algorithm would see the ones with the highest utility first which helps it quickly narrow down the best possible move, saving a lot of time. This was revealed through the fact that the number of nodes also drastically decreased compared to MinimaxAI. With MinimaxAI of maximum depth 3, the algorithm made about 10000 calls for every move. With Alpha-beta pruning of the same depth, the algorithm made between 1000 and 3000 calls for every move.

At the same depth and given the same initial position, both AlphaBetaAI and MinimaxAI returns a move with the same value. For example, when presented with the following state and depth 3:

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . P . . .
P P P P . P P P
R N B Q K B N R
---------------
a b c d e f g h
```

Both AlphaBetaAI and MinimaxAI will make the same move resulting in the following state:

```
r n b q k b . r
p p p p p p p p
. . . . . . . n
. . . . . . . .
. . . . . . . .
. . . . P . . .
P P P P . P P P
R N B Q K B N R
---------------
a b c d e f g h
```

However, while MinimaxAI visited 13754 nodes, AlphaBetaAI visited only 638.

Another example can be seen when the initial position is:

```
r n b q k b . r
p p p p p p p p
. . . . . . . n
. . . . . . . .
. . . . . . . .
. . N . . N . .
P P P P P P P P
R . B Q K B . R
---------------
a b c d e f g h
```

Both AlphaBetaAI and MinimaxAI will make the same move resulting in the following state:

```
r n b q k b . r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . n .
. . N . . N . .
P P P P P P P P
R . B Q K B . R
---------------
a b c d e f g h
```

At this point, AlphaBetaAI visited 1795 nodes while MinimaxAI visited 21388. This pattern is true for many cases. I will not display all of them in this report but it can be clearly seen through testing and gameplay.

Transposition Table

I implemented the transposition table in the `AlphaBetaAI` class. To implement it, I utilized a Python dictionary and the hash function `hash(str(game.board))`. In the utility function, I first check if the hash value of the current board is present in the transposition table. If it is, then I simply return the corresponding utility value stored in the transposition table. If it is not, then I calculate the utility value using the piece point system and create a new entry in the table with the hash and utility value. After testing the transposition table, I did not notice a significant decrease in calls since there were not many cases where the AlphaBetaAI algorithm would generate a previously seen position. I would assume that a transposition table would be more useful for algorithms like MinimaxAI with iterative

deepening which would generate more repeat positions.

Extension: Zobrist Hash Function

As an extension, I implemented the Zobrist hash function in the `AlphaBetaAIZobristEC.py` class. To do this, I added a new field variable `zobrist_table`. I initalize the `zobrist_table` using a function I wrote called `init_zobrist`, which decomposes the chess board and generates a random bitstring for each possible combination of square, piece, and color. I store all of this in the zobrist_table dictionary which I use to calculate zobrist hash values.

Instead of using the function `hash(str(game.board))` to calculate hash values for the transposition table when I calculate heuristics in the `utility` function, I call the function `zobrist_hash`, passing the current board to it. The `zobrist_hash` function then iterates through every square of the board, XORing the corresponding bitstring values from the zobrist_table for each piece on the board. The zobrist_hash function provides a fast and efficient hash function with very limited collisions.

Extension: Related Work Section

Paper: “Analysis and Comparison of Chess Algorithms” by Vesela Trajkoska and Gjorgji Dimeski (https://repository.ukim.mk/handle/20.500.12188/27381)

This paper analyzes the effectiveness of three chess algorithms, specifically the genetic algorithm, Monte Carlo, and Minimax. The Elo rating system was used to rank each of the algorithms based on games played against the Stockfish chess engine. The results revealed that every algorithm performed poorly against Stockfish’s extensively trained and optimized model. Out of the three, Minimax performed the best by winning two games against Stockfish, proving its suitability towards zero-sum games. The paper mentions alpha-beta pruning as an effective method of optimizing minimax search, and expresses the wish to perform research on more optimization techniques as minimax requires high processing power and memory.