Lauren Kidman
COSC76: Artificial Intelligence 24F
25 October 2024

Report for PA3: Chess

Table of Contents:

The Assignment + *README

In this assignment, I implemented a chess-playing AI using adversarial search techniques, including Minimax, Alpha-Beta Pruning, and Iterative Deepening. The goal was to develop an AI capable of playing chess by making decisions based on game tree exploration.

* To run this code, it is vital that the Python chess package is installed. From there, ensure all files are within the same folder, and run gui_chess.py to get a visual graphic showing the moves of the game. One can modify which type of AI they would like to see play (Random, Minimax, AlphaBeta) by modifying the player1 and player2 initializations in the _main_ of gui_chess,py

Minimax Algorithm, Cutoff Test, and Evaluate:

---

1. Description

The minimax and cutoff test portion of my code is extremely straight forward–I am essentially following the pseudocode provided by the class textbook.

The class first is initiated with the __init__ constructor, which takes the maximum depth for the search (i.e. the depth after which the algorithm stops looking). It stores this depth, along with a placeholder for the current “player”, which will either be True (White) or False (Black). It also stores the best move (which will be elaborated on later) and node count, a counter to keep track of how many nodes (board positions) have been visited during the search and helps in evaluating the efficiency of the algorithm. Cutoff_test, the next function in the class, is very simple: it determines if the search should stop, whether it is due to reaching a terminal state (a win or a stalemate), or reaching the specified depth limit.

Choose_move is the heart of the minimax class–it is the main function that controls the minimax search. The function first resets the node count to zero and stores the current player using board.turn, which is essential for determining whether the AI is playing as White or Black. The search process begins with the max_value function, a helper function; it is called to start the search for the maximizing player, and provides the best move found at that depth. Finally, the function checks if the game is over, before either continuing or exiting the GUI.

Max_value and min_value are two helper functions where the Minimax algorithm truly operates. These functions alternate between simulating the moves of the maximizing player (the AI) and the minimizing player (the opponent), i.e. the two opponents playing against each other. They are called this because the algorithm attempts to MINimize the opponent's score, and MAXimize its own, in order to win. With max_value, if the cutoff condition is not met, the algorithm initializes v (i.e. the utility) to negative infinity (as we are looking for the maximum possible value) and begins looping through all possible legal moves. Each move is applied using board.push(move), which modifies the board state. Then, the min_value function is called recursively to evaluate the opponent's response to this move. The recursive process alternates between the maximizing player and the minimizing player; if a move results in a higher evaluation score (v2), the algorithm updates v and stores the move as optimal_move. After each move is evaluated, it is undone using board.pop(), restoring the board to its previous state. This is crucial from a memory perspective. Similarly, the min_value function works in reverse, as the minimizer aims to reduce the evaluation score for the maximizer. It initializes v to positive infinity and explores the legal moves for the opponent, calling max_value for each move. The goal here is to select the move that minimizes the AI's score, simulating an optimal opponent.

Finally, none of the minimax search could occur without the evaluate function, which is the quantitative method that allows the algorithm to determine its next best step. This is the function that did not ride on any sort of pseudocode; rather I designed the material value myself. Based on online chess discussion pages (I used this), each piece type is assigned a point value:

-Pawns are worth 1 point,
-Knights and Bishops are worth 3 points,
-Rooks are worth 5 points,
-Queens are worth 9 points,
-The King is assigned an arbitrarily high value (since the game is won or lost based on the King's survival)

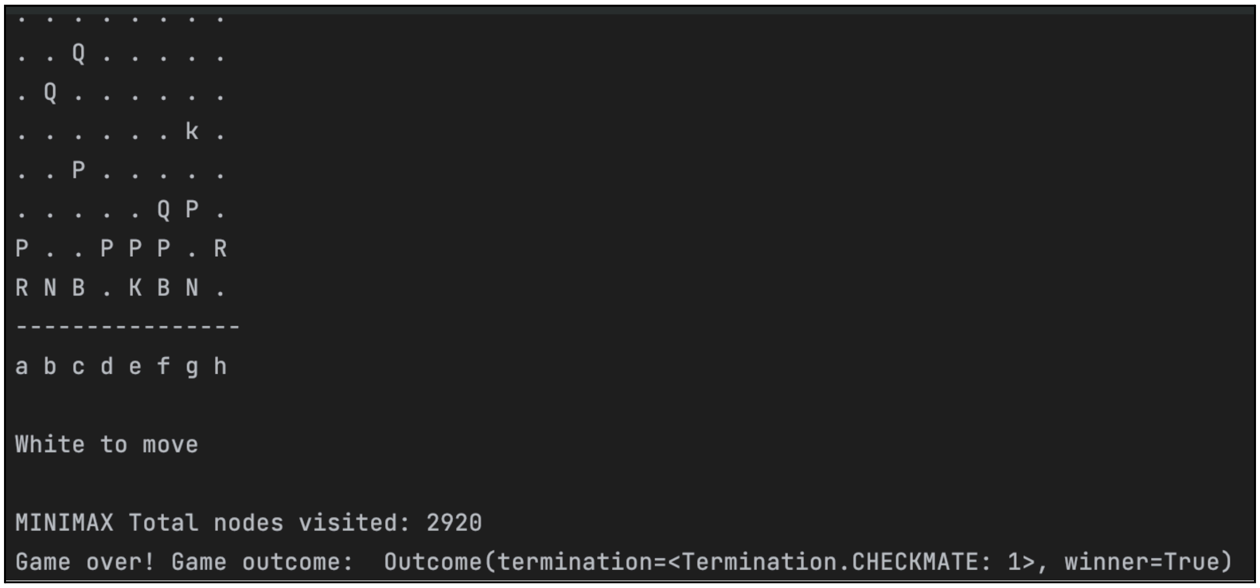
Note: I initially had the King's score set to 0, however after running multiple tests of the chess GUI, I realized that the algorithm put essentially no thought or effort into the movements of the King, which were highly scattered and irrational. With a score of 0, the evaluation function does not recognize the importance of protecting or attacking the King, making it unable to prioritize avoiding checkmate.

Based on these scores, the function sums the material on the board for both players. It calculates the difference between the AI's and the opponent's material balance–

if it's the AI/maximizer's turn, the function adds up the values of the AI's pieces and subtracts the values of the opponent's pieces; if it's the opponent's turn, the evaluation is reversed. This heuristic allows the Minimax algorithm to make informed decisions, prioritizing moves that maintain or increase total point value.

2. Evaluation

My Minimax class (including evaluation and cutoff test) worked very well. Off the bat, the class threw no errors and ran as expected. Even at a max depth of two, the MinimaxAI would beat RandomAI (either checkmate or stalemate), or single out its king, every time. Here is an example outcome of a test I ran:



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

White to move

MINIMAX Total nodes visited: 2920
Game over! Game outcome:  Outcome(termination=<Termination.CHECKMATE: 1>, winner=True)
```

There were essentially no unnecessary lags in printing. In terms of assessing the evaluate function in particular, the scores I utilized in the function are very standardized in real life, so I did not really need to change anything there and it worked great.

3. Discussion Questions

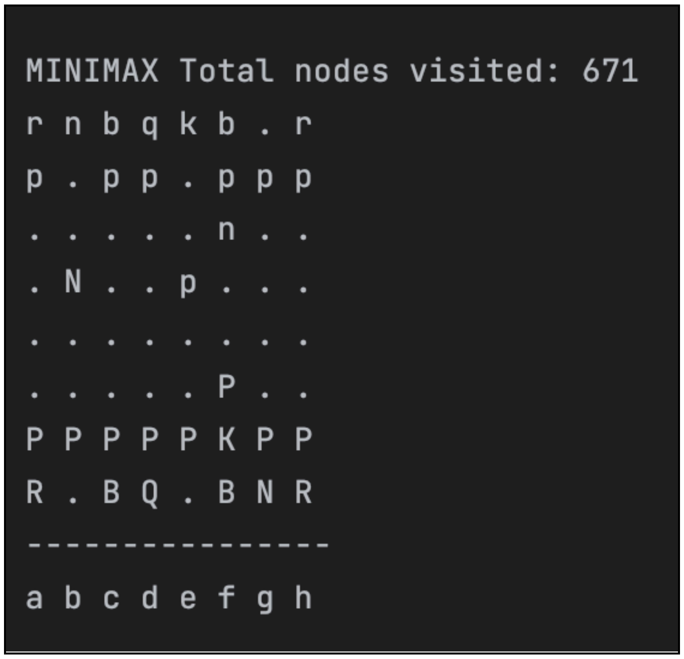
a. Vary maximum depth to get a feeling of the speed of the algorithm. Also, have the program print the number of calls it

made to minimax as well as the maximum depth. Record your observations in your document.

When calling Minimax versus a RandomAI, it would work essentially instantaneously with depth limits up to 3. Once depth was equal to 4, the algorithm would take multiple seconds to determine the next move. This makes sense–comparing nodes visited, for a depth of 3 an example total nodes visited was = 9765, yet for a depth of 4, total nodes visited = 216369. The more the algorithm has to think ahead (and therefore visit more nodes), it is extremely predictable that the algorithm would exponentially increase in time.

b. Describe the evaluation function used and vary the allowed depth, and discuss in your document the results.

Reiterating what I said in the first discussion section of the report, my evaluation function, in essence, creates a quantitative score for each move based on the number of pieces the maximizer and the minimizer have–it sums up not only the number of pieces on the board, but provides (pre-determined, from online) weights to each piece based on their strategic value (based on mobility) and role in the game in order to ultimately reflect which player has a material advantage. When changing the maximum depth, however, I noticed that this particular evaluation does not "think" the best for the early moves of the game when the maximum depth is low. This does make sense–at the beginning of the game when both sides have all of their pieces, it is unrealistic that any piece capturing will occur for at least two or three moves, or more. With low maximum depths like 2, that means these first moves are essentially arbitrary. Furthermore, again with low maximum depths, sometimes the algorithm will make moves that are objectively irrational in the grand scheme of the game. This is because, again since at low maximum depths it is not thinking ahead very far, it prioritizes immediate, short-term gain without worrying about putting other pieces at risk. For example, in the game below, the algorithm brought the King out of its originating spot seemingly for no apparent reason:



```
MINIMAX Total nodes visited: 671
r n b q k b . r
p . p p . p p p
. . . . . n . .
. N . . p . . .
. . . . . . . .
. . . . . P . .
P P P P P K P P
R . B Q . B N R
---------------
a b c d e f g h
```

In general, testing at different depths showed that deeper searches improved the AI's play. For instance, at depth 2, the AI often missed simple captures or defensive moves, while at depth 4, it could block threats and plan captures far in advance.

Alpha-Beta Pruning:

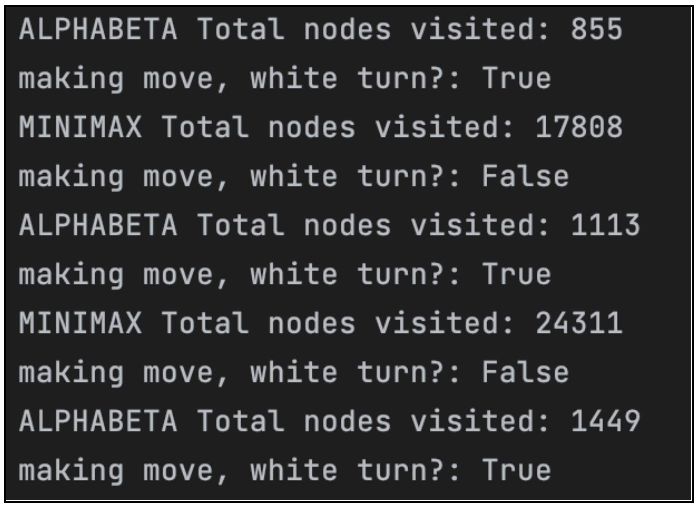
1. Description

The AlphaBetaAI class implements the Alpha-Beta pruning algorithm, which is an optimization of the Minimax algorithm designed to reduce the number of nodes that need to be evaluated. Most of the functionality/code of this class is identical to Minimax–the key difference between Alpha-Beta pruning and standard Minimax is that Alpha-Beta uses two values, alpha and beta, to prune branches of the search tree that will not influence the final decision. Thus, the only functions within the class that are altered are the two helper functions to choose_move, max_value and min_value.

The max_value function is called, initializing the alpha and beta values to -infinity and +infinity, respectively. Like Minimax, the function explores the maximizing player's (the AI's) best possible moves. It finds the maximum value move, updating alpha in the process. However, now, if v (the value/utility of the current move) becomes greater than beta, the function prunes the rest of the branch, meaning it stops evaluating further moves because they cannot improve the result. The min_value function uses the same logic. It simulates the opponent's (the minimizing player) moves, evaluates each move, and updates beta with the minimum value found. Now if v becomes less than alpha, the branch is pruned.

2. Evaluation

The AlphaBeta class worked exactly as intended–it produced the same optimal moves as the standard Minimax algorithm when evaluated at the same depth, yet the key advantage is that it explored fewer nodes, making it faster and more efficient. This can be seen in the example test screenshot below:



```
ALPHABETA Total nodes visited: 855
making move, white turn?: True
MINIMAX Total nodes visited: 17808
making move, white turn?: False
ALPHABETA Total nodes visited: 1113
making move, white turn?: True
MINIMAX Total nodes visited: 24311
making move, white turn?: False
ALPHABETA Total nodes visited: 1449
making move, white turn?: True
```

AlphaBeta explored significantly fewer nodes at each turn. In fact, according to the Lecture 9 slides, AlphaBeta's best scenario shows the time complexity moves from O(b^m) to O(b^m/2)--this definitely looks somewhat true here based on the numbers above. Furthermore, because AlphaBeta explores such fewer nodes, it is able to run extremely quicker than Minimax at higher maximum depth levels. It was able to run a depth of 4 and 5 almost instantaneously, while Minimax took multiple seconds.

3. Discussion Question

a. **Record your observations on move-reordering in your document**

I was inclined to introduce move-reordering into my classes when I began to notice that in many cases, if a game was approaching a stalemate, an opponent's king or other remaining piece would move back and forth between two spots over and over. This was particularly frustrating because it meant the game could not truly end with a definitive outcome declaration, when a stalemate was what clearly had occurred. So, with the advice of Prof. Vosoughi, I implemented random.shuffle(moves) to mitigate this and ensure that the algorithm did not fall into patterns of movement. I think this worked–those back and forth cases definitely occurred less. In the case of AlphaBeta pruning specifically, my move-reordering method of choice (random shuffle) helped pruning in some cases, while had no noticeable effect in others. This makes sense, as the randomness of move ordering means that the effect is not always predictable. Overall, random shuffling introduces variety and potential efficiency gains, and I personally feel it did help to address movement repetition well. In the future I would be curious to see how a more strategic/sophisticated move-reordering system affects algorithm efficiency.

Iterative Deepening:

---

1. Description

Iterative deepening is a search technique used in combination with algorithms like Minimax and Alpha-Beta pruning; it has also been used in the two precious PAs, Fox Problem and Mazeworld. Instead of searching the game tree to a fixed depth all at once, iterative deepening performs a series of depth-limited searches, starting from a shallow depth and gradually increasing the depth until a maximum depth is reached. Each depth-limited search provides a progressively more accurate result, with the best move thus far being stored in a best_move variable.

The implementation of IDS to the Chess problem was very similar to PA1, and very straightforward. Essentially, only the choose_move function is modified, while the remaining Minimax functions are the same. Instead of setting value, move = self.max_value(board, self.depth) on its own, it wraps the statement in a for loop that gradually increases the maximum depth at each iteration. In addition, an if-statement check is included to store the best move found so far. This ensures that the AI can return a valid move even if the search is interrupted.

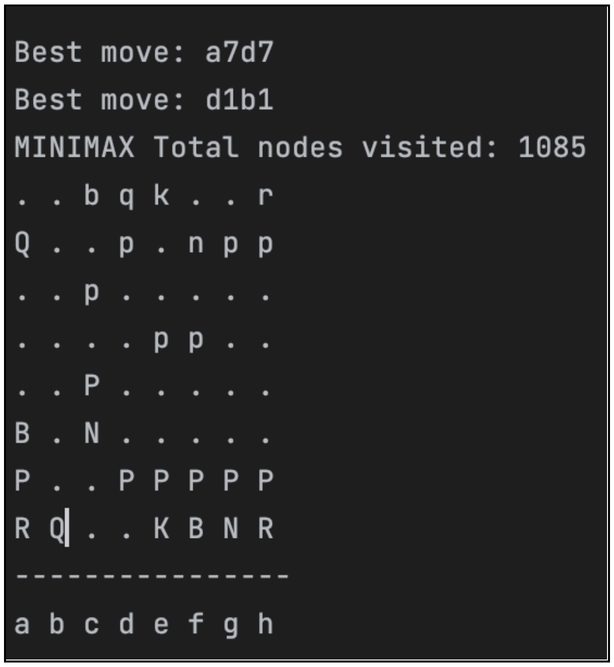
2. Evaluation

Considering implementing IDS in this case is very simple, the evaluation is very simple–the program runs very well, and is able to provide a more reasonable/well-thought out move at shallower depths.

3. Discussion Question

a. **Verify that for some start states, best_move changes (and hopefully improves) as deeper levels are searched. Discuss the observations in your document.**

Utilizing IDS shows improvements even between a max depth of 1 and 2. As mentioned previously, lower depth levels have less insight because they do not plan ahead as far–so even a jump as small as between 1 and 2 adds tremendous knowledge to the AI. For example, take the turn movements from example test run shown below:



```
Best move: a7d7
Best move: d1b1
MINIMAX Total nodes visited: 1085
. . b q k . . r
Q . . p . n p p
. . p . . . . .
. . . . p p . .
. . P . . . . .
B . N . . . . .
P . . P P P P P
R Q . . K B N R
---------------
a b c d e f g h
```

As you can see, at depth 1 the program prioritized short term gain–it wanted to move the queen at a7 to take out a singular pawn at d7, which would have put it in immediate risk of being captured by the opponent. Once it moved to depth 2, it realized how ineffective that move would be (as it would ultimately sacrifice a queen, a super valuable piece), and instead decided to move its queen from d1 to b1, thus saving the queen at a7. This example confirms that as search depth increases, so does move quality.