Maxwell Carmichael

COSC.076.01 PA3

Fall 2020

Introduction

In this programming assignment, we took a python chess package and created algorithms which, given a game board, returned the best possible move under computationally-limited restrictions. Overall, the algorithms I implemented were a success, in that no obviously irrational moves were taken, and intelligent pin opportunities and guaranteed checkmates were always acted upon.

Description

Evaluation function

Theoretically, if my computer had an impossible amount of processing power, you could represent the states at the end of a chess tree as +1 for your checkmate, -1 for their checkmate, and 0 for a draw. The optimizing first player would be able to go through the entire tree and find the path that preferably gave it +1 (a guarantee checkmate) or a 0.

However, this is a very unrealistic approach given computational constraints. It would be much better to have some cutoff_test function, which returns True if we have reached a deep enough point in the Minimax tree, either at a guarantee checkmate or at some pre-designated depth limit.

Thus, I began implementing my AI with the evaluation function, which tells how good/bad a state of the game is; I based it on chess piece material values. I iterate over all pieces of the chess board, add +1 if it is a white pawn, +3 if it is a white bishop or knight, +5 if it is a white rook, and +9 if it is a white queen; make these values negative for black pieces. I also give a +0.5 if black is in check and -0.5 if white is in check, so that, all else equal, the algorithms prefer to put the opponent in check. If the state is checkmate, I return -10000 if black checkmates white and +10000 if white checkmates black (with a slight modification which I will get to soon). I return 0 if the board is a draw.

I ran into two problems with my evaluation function. The first was an oversight on my part: if the board is in checkmate (or check) and it's white's turn, then you should return -10000 (or subtract 0.5). I made the mistake of thinking that if the board is in checkmate, and it's white's turn, then it is good for white. This was an easy fix.

The second problem was that although my AI would play relatively intelligently (never sacrificing pieces for no gain while also making intelligent pins), when a checkmate was in sight, even one turn away, they would never go for it and eventually randomly move irrelevant pieces around until the game draws. I realized that, although the algorithm found a checkmate, it found it at a later depth than it could get it otherwise. For instance, there could be one possible checkmate in 1 move, but also many more possible checkmates in 3 moves (randomly moving a piece before going in for the kill), so it was far more likely to go after a 3-move checkmate than a 1-move checkmate (given my implementation). I fixed this by simply making current depth a parameter to my evaluation function, so that, in the case of checkmates, the AI prioritized the ones at a sooner depth.

MinimaxAI

I began my implementation of MinimaxAI by implementing the minimax algorithm according to the pseudocode in the textbook. The first thing I noted was that the game and the state could be both represented by a board object, where I updated the board by pushing a move onto it and then undoing that move when I was done with its branch of the DFS tree. Another thing I needed to figure out was how to also allow the choose_move function to return the best move for the black player. The pseudocode assumes that the player is the maximizer, but I did not want to have to reverse my heuristic function based on which player's turn it is. So, I called max_value if it was white's turn and min_value if it was black's turn.

The max_value and min_value recursive functions were implemented like the pseudocode. I kept track of nodes visited in an instance variable so I could better compare Minimax to AlphaBetaAI, which I reset to 0 after printing. Also, since I represented the game and the state in the same object.

One problem my implementation had was that the AI sometimes got into move-loops. Because many moves yielded the same best heuristic value, the algorithm would always go for the first one, which ended up causing loops such as simply moving a rook back and forth. I approached fixing this by implementing an override chance of 10%, where whenever a move yielded the same heuristic value as the previously best found move, there was a 10% chance for this new move to override the previous move (I seeded random to allow for replicability). Ideally, the algorithm would choose one of the many equally optimally good moves in a random distribution, and this is feasible in Minimax, but in Alpha Beta one would have to lose some of the pruning optimization to afford to do this. So I went with a 10% chance to override, over say 50%, as 50% weights the later considered best-case moves much higher than the earlier considered best-case moves.

AlphaBetaAI

To implement AlphaBetaAI, I copied my code from MinimaxAI and modified it slightly, according to the pseudocode in the textbook (adding alpha/beta parameters to the max_value and min_value functions and using them to prune).
Everything else, including my evaluation function, stayed the same, except I also implemented a sorting algorithm to increase the likelihood of pruning.

My sorting algorithm was pretty straightforward, until I realized it was too slow. My idea was to prioritize moves that checkmated the other player, followed by moves that captured pieces of the opponent, followed by all other moves. As it turns out, my material heuristic function provides exactly that: take the list of moves, push a move onto the board, and evaluate the board. If they're white's moves, you want to order the moves with a higher heuristic first, and vice versa for black's moves. I took advantage of python's sorted() method to pass in an anonymous function as the key, which took the current board and a move and returned the heuristic for the move on the board. I simply reversed the list if it was the max (white) player's turn.

As it turns out (I talk about this in Analysis section), this sorting method (which I will refer to as "old sort") actually significantly increases the runtime (in seconds) when making moves from the starting board. So I thought about how I could achieve the same effect in faster time. As it turns out, the chess board object provides a piece_map() method which returns a map from square to piece. So, if the square which a move goes to has a piece on it, then it's a capture! Return the negative value of this captured piece and sort based on this value. Return 0 otherwise. This way, the AlphaBeta algorithm will look at queen captures first, no captures last.

The only disadvantage this new sort has over the old sort is that it cannot detect checks and checkmates, but the benefit (as you shall see) greatly outweighs this disadvantage.

IterativeAI

To implement iterative deepening search, I passed in which AI (Minimax or AlphaBeta) I wanted to use as well as a time value. This time value (in seconds) would ideally be the maximum time we allow the algorithm to take before returning a best move. The IDS best_move algorithm runs by increasing its AI's depth limit, calling the best_move method on the AI, and returning the best_move found so far by the time limit. Unfortunately, I could not figure out how to externally stop a function call in the middle of its call, so I terminated the while loop after whichever function call continued past t seconds and returned the most recent (and thus least naive) best value.

Analysis

To analyze and compare my algorithms, I mainly tested on a normal chess starting board against the AI with my initial move e2e4. I also created a specific case to ensure the black AI would take a guaranteed depth-3 checkmate, and another case with a relatively open board to ensure the black AI would take an unprotected piece.

MinimaxAI

The following is a table representing how many nodes get visited in MinimaxAI for each depth. At a depth of 5, the algorithm takes far too long, and even a depth of 4 is a stretch for a quality human vs. AI experience. Especially since this is the beginning state of the board, with the only possible moves being pawn and knight moves; during the mid-game, the number of possible moves (branching factor) increases, so runtime at a given depth does too. Thus a depth of 3 is preferable for Minimax.

Depth	Nodes Visited	Move	Utility

Depth	Nodes Visited	Move	Utility
1	21	a7a5	0
2	621	a7a5	0
3	13781	a7a5	0
4	419166	a7a5	1

Figure 1. Starting chess board, white moves e2e4, black reacts above (a7a5 in any case). A positive utility is not favorable for black.

AlphaBetaAI

Old sort vs. New sort:
I tested the two sort methods I talked about in AlphaBetaAI, as well as no sorting. As a reminder, old sort is pushing a move, evaluating the heuristic, and popping the move. New sort is seeing if the move captures a piece without pushing the move, and returning the value of the piece for sorting.

Sort type	Time (s)	Nodes visited
No sort	4.96	44889
Old sort	14.82	22573
New sort	3.26	26205

Figure 2. Starting chess board, white moves e2e4, black reacts a7a5 in any case. I timed each one twice on a stopwatch and averaged the times.

As you can see, the old sort method, although it did a wonderful job at increasing pruning, took far long. The new sort method still achieves similar pruning effects but sorting takes substantially less time.

The following figure represents the same test case that I did in the Minimax section, but instead with AplhaBetaAI.

Depth	Nodes Visited	Move	Utility
1	21	a7a5	0
2	70	a7a5	0
3	657	a7a5	0
4	6646	a7a5	1
5	26205	a7a5	-1
6	319701	a7a5	1

Figure 3. Starting chess board, white moves e2e4, black reacts a7a5 in any case.

Notice that at depth 6, AlphaBeta visits less nodes than Minimax at depth 4. This shows the power of pruning! However, at a depth of 6, like the depth of 4 for Minimax, this runtime is probably too long for a quality human vs. AI experience. Especially since this is the beginning state of the board, with the only possible moves being pawn and knight moves; during the mid-game, the number of possible moves (branching factor) increases, thus does runtime. Thus a depth of 5 is probably preferable for AlphaBeta.

For another comparison of AlphaBeta and Minimax, I will compare the test case generated by calling open_board(game) on the ChessGame object. This function sets up the board to a state with four pieces of the same value on each side.

The following is MinimaxAI performing at a depth of 3:

```
Move Utility: -0.5
Maximum Depth: 3
Nodes visited: 11238
MinimaxAI recommending move g8b8
. q . . . . . k
. . . . . n b
. . . . . .
. . . . . .
. . . . . .
. . . . . .
B N . . . . .
K . . . . . Q
---------------
a b c d e f g h

White to move
```

The following is AlphaBetaAI performing at a depth of 3:

```
Move Utility: -0.5
Maximum Depth: 3
Nodes visited: 1463
AlphaBetaAI recommending move g8b8
. q . . . . . k
. . . . . n b
. . . . . .
. . . . . .
. . . . . .
. . . . . .
B N . . . . .
K . . . . . Q
---------------
a b c d e f g h

White to move
```

In this case, AlphaBetaAI also visits significantly less nodes than MinimaxAI (7/8 less!). Note that this does not imply about 7/8 faster, as AlphaBetaAI sorts while MinimaxAI does not. Also note that the move utilities (and the moves themselves) are the same, indicating that, as expected, both algorithms find moves with the same level of optimality.

IterativeAI

I found IterativeAI a useful tool to help analyze the algorithm at different depth limits (I used it to help generate the above figures 1 and 3). For example, in the following start state the AI generated the moves in the following table at increasing depths.

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

Depth	Nodes Visited	Move	Utility
1	17	g8a2	-3.5
2	63	g7e6	3.5
3	1463	g8b8	-0.5
4	2471	g8b8	3
5	33771	g7e6	-3
6	83741	g7f5	3
7	796481	g7e6	-3

Figure 4. Given the above start state, AlphaBetaAI recommends moves which differ depending on the start state.

Although the utility does not increase, we can assume that at deeper depths the AI is making more informed moves, so it makes sense the move changes (so long as we assume the changing of moves isn't due entirely to the seeded randomness). Interestingly, at odd depth moves, black had the final move, and at even depth moves, white had the final move. The utility values at each of these depths is positive for even depths and negative for odd depths, indicating that the game may have a tendency to give greater utility values when white has the final move at the given depth (and vice versa for black).

Other

Although I have two testing functions, I mainly only used open_board(game) for bug/runtime detection and discussed it above, so I will not focus on it here. My other function, called black_checkmate(game) on the ChessGame object makes it so black (the AI) can checkmate in exactly three turns. This board setup is useful for ensuring the algorithm is functioning properly, and that it will always go for a checkmate if it sees one. As it turned out, I initially had my checkmate values reversed, so this was helpful in isolating the problem.