Kunal Jha, CS76, Fall 2021, PA3

Description + Evaluation

Minimax

My minimax algorithm essentially followed the pseudocode provided in class and the text book. It took a board in as a parameter, and initialized the best move, largest value, and current depth. From there, it iterated through all of the legal moves (represented as a set for low time complexity) given the initial state of the board. It then made a call to the *minValue* function assuming the move had taken place. If the returned value was larger than the largest value, the move was the new best action and its value was stored as the new largest value. Once all moves had been iterated through, the current depth was incremented and the process was repeated until the current depth was equal to the maximum depth. Outside of this loop, the best move was returned.

The *minValue* and *maxValue* function worked similarly. They took in a board, a depth, and a local maximum depth as parameters. If the board had reached a cutoff state (described below), the utility function would be called to evaluate the position, and that value was returned. Otherwise, the depth was incremented, the set of legal moves was iterated through, and the minimal value of the *maxValue* function was stored and returned. The max value had the same "else" condition, except it returned the maximum value of the *minValue* function.

The cutoff function returned true if the depth had surpassed the limit, if there was a checkmate on the board, or if there was a stalemate on the board. Going back to the min and max value functions, if any of these conditions was met the utility function was called. The utility function returned a score of negative or positive infinity if a checkmate had been reached (depending on whether the winner was the maximizing or minimizing player), 0 if a stalemate was reached, and the evaluation of the board if neither of those conditions was reached. The evaluation function found the difference in the number of each piece type on the board for both the white and black players, and weighted them accordingly (pawns worth 1, knights and bishops worth 3, rooks worth 5, queens worth 9, kings worth 200). It then returned the difference.

My minimax algorithm worked correctly, and I verified this through several different games of chess. Against a random agent, Minimax won every time, however, the amount of decisions it required to win varied depending on the depth specified (more detailed below, but briefly, the greater the depth, the fewer the decisions and longer the decision time). Against other minimax agents, the algorithm with the greater depth performed better as it was able to look farther ahead and evaluate moves more accurately.

Alpha-Beta

My AlphaBeta class was an extension of my Minimax class, in that the majority of the methods were duplicated with minor modifications. The utility, cutoff, and evaluation functions were identical to the Minimax class. I kept a dictionary of *(board, list of values and depth)* pairs to implement AlphaBeta recursively. The *AlphaBeta* function took in a board, a depth, an alpha, a beta, and a current Maximum as parameters. It began by checking whether the current state had been visited in the dictionary. If it had been and the depth was less than the depth value stored in the dictionary, the dictionary's board value was returned. If the board had reached a cutoff state, the board's utility was returned. Outside of the base cases, the board's actions were iterated through depending on which player's turn it was. If it was the white player's turn, the maximum value of the AlphaBeta function recursively called with a greater depth and updated board was calculated. The new alpha was the maximum of the old alpha and that value. If the new alpha was greater than the beta, the for loop was broken. Otherwise, the loop repeated and the board was inputted into the dictionary with its maximum value and depth. The black player had a similar statement, except it asked for the minimum value of AlphaBeta and assigned the new beta as the minimum of the old beta and that value. The *noDeepening* function uses standard AlphaBeta pruning, and works by iterating through all possible moves, calling AlphaBeta search at the maximum depth, and returning the maximum possible value for AlphaBeta. Deepening works similarly, however, instead of running AlphaBeta at the maximum depth, it iteratively increments the depth of AlphaBeta from 0 to the maximum depth. The last method was the *sortBoard* method, and the details of its implementation are below. Really briefly, it arranges the list of legal moves for any given state from their lowest Utility to the highest.

In terms of results, when fed the same states in two different examples (a pawn moving and a knight moving), both AlphaBeta and Minimax explored 420 states initially before performing an action. However, AlphaBeta took only 429 states visited to explore the next move while Minimax took 462 states before coming to an action. This is in line with the expected performance.

Discussion Questions

When varying the maximum depth of minimax, I noticed that the time required for the program to run grew significantly. This makes sense because the time complexity of minimax is **O(b^m)**, where *m* is the maximum depth of the tree. To illustrate this observation, I will compare the algorithm's performance across a number of different maximum depths. In all of these examples, minimax will be playing against a Minimax AI of depth 2. In the first case of a maximum depth of 3 vs a maximum depth of 2, the algorithm made decisions relatively quickly. It won after 18 calls to minimax. Against an optimal player, the decision time and game duration were both small. In the second case of a maximum depth of 1 vs a maximum depth of 2, the algorithm took less time to come to a decision, but required 42 calls to minimax. Moreover, it stalemated rather than win outright. This indicates that while the time complexity decreased, the total number of decisions increased, making the program more efficient and less effective. Examples at a depth of 4 and greater took too long of a time to choose an action for me to consider using them as examples, but from a brief overview they took longer to choose an action, similar to the jump in time complexity from depth of 1 to a depth of 3. On the surface level, one may think to use a minimax of depth 1 each time if it acted so quickly. However, this is not effective when playing against the optimal player. The algorithm is more capable of beating the optimal player the deeper it searches the tree. Against a random AI, it is not hard to win without thinking too far ahead, however, if you think several moves ahead you have a higher chance of winning albeit over the course of a longer game. This is exactly what we talked about in class with the problems of using Minimax against un-optimal players: it can take too long and may not win under random settings. One other problem I noticed was when the Minimax played against a random AI, it could take significantly more moves to win (a depth of 2 took 70 moves to win against a random AI in one case, but only 25 in another). This again illustrates the problem with Minimax against random agents.

The evaluation function used relied on the point totals for each piece on the board. Pawns were worth 1 point, knights and bishops were worth 3 points, the rooks were worth 5 points, the queens 9 points, and the king 200 points. The function found the difference in the number of white pieces and black pieces, weighted by their point values. A positive score indicated that the white side was winning, while a negative score indicated that the black side was winning. By doing so, the minimax algorithm was able to evaluate the strength of a move even without calculating possible outcomes until the terminal state was reached. The utility function implemented the evaluation function by saying if white won the score would be infinity, if black won the score was negative infinity, if it was a stalemate the score was 0, and if neither of these cases was true the score was determined by the evaluation function. Since I built this evaluation and utility function at the same time I tested the minimax algorithm, the performance across depths is the same as indicated in the previous paragraph.

In AlphaBeta Pruning, the program performed similar to Minimax, however, its time complexity was much less than Minimax. This is because AlphaBeta shortens the tree if it finds values which are invalid in a search (a.k.a. it prunes the tree). As a result, AlphaBeta has a best-case time complexity of **O(b^(d/2))**, where *d* is the depth of the tree. An important note is that this time complexity is in the best case scenario, which means all of the nodes in the tree are ordered perfectly. If the nodes are not ordered properly, then the time complexity could be just as much as that in standard minimax search. To illustrate this, I tested AlphaBeta at a depth of 3 for two cases: when the nodes were sorted and when they weren't. In the first case, I just iterated over the pre-built "legal moves" method from the *board* class. In the second case, I had to design a function called *sortBoard*, which determined the value of all legal moves and sorted them based on their expected value from greatest to least. In comparing the performance, I found that the sorted nodes version of the problem performed qualitatively faster than the standard ordering of the nodes. This makes sense, because the AlphaBeta algorithm can prune branches sooner with sorted nodes than without. Quantitatively, sorted nodes had an average of about 2000 states visited per move, while un-sorted had an average of about 2500. In terms of shortcomings, I am not 100% confident I implemented the ordering of nodes correctly. I ranked each action by Utility, but if there was a more effective way of sorting actions then my program may not have worked as well.

The idea behind iterative deepening is that the best move changes the deeper the algorithm searches. This can be verified by printing the best move during each depth iteration in the program. I ran AlphaBeta at depth 3 with iterative deepening to test this against a random AI. For the initial state, the board returned a best move of g1h3, likely because the first move is arbitrary. However, the second time the AlphaBetaAI was run the best move changed from h3g5 to c2c3 as the depth increased from 1 to 2, and returned c2c3 as the best move for a depth of 3. Later in the game, the program changed the depth from c5e7 to c5d4 as depth increased. I noticed that when the best move changes for different states, the best move from a depth of 2 and depth

of 3 are often times the same. While I am not 100% sure about why this happens, I believe it is because the best move forces the opponent into a series of dilemmas where the final utility of a move will remain the same. Thus, the best move at a depth of 2 could force the opponent into a limited set of outcomes in which the utility at depth 3 is the same. Regardless of why this happens, it is clear that the algorithm is improving its best move as the depth increases, which means iterative deepening is working successfully.