PA 3 - Chess

CS 76 AI, Fall 2021

Melissa Valencia

Overall Design

`MinimaxAI.py` implements the depth-limited minimax search. `AlphaBetaAI.py` adds Alpha Beta Pruning to the functions of `MinimaxAI.py`. `MoveOrderingAlphaBetaAI.py` utilizes the functionality of `AlphaBetaAI.py` with a change of node ordering to improve the time complexity. `IterativeMinimax.py` impliments an iterative deepening approach to the functionality of `MinimaxAI.py`.

MiniMax Algorithm Implementation

`MinimaxAI.py`contains the implementation of a depth-limited minimax search with a cut-off test. This algorithm takes a recursive approach as it computes each node's minimax value. To write the functions for the MinimaxAI agent, I followed the pseudocode for minimax in the book. In my main decision making function, `choose_move`, by iterating through all the legal moves, constantly checking the value at each move and returning the best move possible at the given depth. The value at each move is calculated by the function `value` which then determines who the player is and whether to call the `min_value` function and `max_value` function to then determine the value at this state.

Discussion and Test Cases

In order to ensure that my Minimax agent and cutoff test implementations were functioning as expected, I tested using varied maximum depths. I started off testing with a max depth of 1 then increased it to 2. The maximum depth this minimax implementation can be ran on is 2, at depths greater than 2 the game runs slower as it has much more nodes to explore. I first ran it against the RandomAI() player, then began testing against the other agents to ensure they were performing as expected.

Evaluation Function Implementation and Discussion

The evaluation function I used, `utility`, follows the implementation of a material value heuristic described in the book. Within this function, I calculate the scores of each piece in the chess board: the pawns, the rooks, the queens, and the kings to then calculate a state (board) score. This evaluation function is used in all of the agents, specifically in the `min_value` fucntion and `max_value` function as it is called when the cutoff test is reached. Upon using the evaluation function within the minimax implementation, the test cases ran faster.

Alpha Beta Pruning Implementation

`AlphaBetaAI.py` contains the implementation of pruning in order to eliminate branches that are not part of the move decision making which is done by alterating the functions in `MinimaxAI.py`. Alpha Beta Pruning keeps track of two variables, alpha representing the value of the best choice (maximum value) we have found so far at any choice point along the path for the max and beta representing the value of the best choice (minimum value) we have found so far at any choice point along the path for the min. The search then updates the values of alpha and beta once it has found a value that is essentially less than or greater than the current alpha and beta for max/min, as it goes through and prunes the branches of this node.

Test Cases

In order to test my Alpha Beta Pruning agent, I ran the game against the original minimax agent at various depths. This first version of Alpha Beta Pruning visited significantly less nodes than the initial minimax implementation.

Alpha Beta Pruning with Move Reordering Implementation

To improve the time complexity of the implementation of Alpha Beta Pruning, I altered the order in which the moves are evaluated in these functions. In `MoveOrderingAlphaBetaAI.py`, I implemented a function for reordering the legal moves list. The function `move_ordering` goes through the current legal moves list and sorts them into a new list based on best board scores. Thus, the time complexity is improved as generally the best moves are chosen.

Discussion and Test Cases

The move reordering implementation in `MoveOrderingAlphaBetaAI.py` further led to less nodes being visited in the search for the best move as opposed to utilizing the depth-limited minimax search implementation and the regular alpha beta pruning. The reordering based on which are the best moves allows for much more efficient pruning and searching as unnecessary searches of certain nodes are prevented.

Iterative Deepening Implementation

A further implementation of the depth-limited minimax search is this iterative deepening approach in `IterativeMinimax.py`. In this implementation, the decison making function, `choose_move`, iterates through every depth based on the max depth given, starting at 1. It then returns the best move at anyone of these depths.

Discussion and Test Cases

The implementation of iterative deepening on the depth-limited minimax search allowed for the search of the best move based on iterations up to the max depth given. In order to test the functionality of this agent, I ran it on depths of up to 3, against the Alpha Beta Pruning agent and the Move ordering Alpha Beta Pruning agent. Upon analyzing my test case results, I noted how the best move was updated for some states if a better move was found at a deeper depth.