Rudd Johnson
rujo2@pdx.edu
6/2/17
CS 442

Final Minichess Writeup

This project was the implementation of a 5x6 minichess player. The player is written in Java and connects to the IMCS server where it can play opponents. The player utilizes negamax, alphabeta pruning, and iterative deepening to select the best move for it (worst for opponent) using a basic piece valuation heuristic. The general architecture of the program is as follows:

The board is represented as an array of characters in which the capital characters are White and the lower case characters are black. The initial board state is read in from a supplied file, start.txt. And the beginning of each game (from Main). The engine for the player is the class State, which contains the board object in addition to all the functions for move generation and evaluation with with negamax, alphabeta pruning, and iterative deepening. After a user connect to the IMCS server and selects an opponent from the list of opponents, an instance of state is created. In state there are two functions responsible for updating the board, moveOpponent(), and moveAI(). MoveOpponent() is passed the move from the IMCS server and updates the board with that given move. MoveAI() updates the board with the move selected by the player's move generator. The move generator creates a list of available pieces on the board for the side in question by iterating over the entire board and checking. There are Size piece type classes, Bishop, Knight, Rook, Pawn, King, Queen that inherit Piece. Each one contains its perimeters for move generation and is passed an instance of the board as well as the list, such that it can add itself in every legal configuration to the board.

Once this list of pieces and legal moves is created, The optimal move needs to be selected. To do this, the board state, the list of moves, and the side in question are passed to stateEvaluator. StateEvaluator iterates through the entire list of possible moves, and using do and undo modification of the board, passes the new state created with the move to the negamax function where the value of the move at a depth permitted but the time slice provided to that search will permit. Each move cannot take more than 7.5 seconds, so the number of moves to be checked divides 7.5 and thats how long the game tree traversal has to identify the optimal move. More on iterative deepening: An initial search to depth 2 is performed for each piece, and the result store, to ensure a usable return item exist. Next a while loop iterates until a depth of 40 is reached or the time slice of the search expires, doing a depth+1 traversal at every iteration and storing the result.

The negamax function called by the sate evaluator recursively traverses the game tree to the depth provided by the state evaluator. All possible moves for that ply are generated with each recursive call and score is assigned to the move when the leaf is reached. The score is simply the value of each piece (100 for pawn, 300 for bishop and knight, 500 for rook, 900 for queen, and 1000 for king) are added up and the difference taken from each side. The goal of the function is to find the worst possible outcome for the opponent so the score returned from every recursive call is negated (your gain is their loss). Finally alphabeta pruning was implemented so that those subtrees in the game tree would not be explored in which an opponent could force a move worse than an current worse outcome. Once the optimal move for the depth explored is identified in stateEvaluator, it is returned to move generator and subsequently to moveAI where the board state is updated and if the game has not conclude either from a win or reaching ply 40, the opponents turn is taken.

At current, though the player meets the required specifications, it is quite slow due to high memory demands and heavy use of lists. Making each piece it's own class and iterating through a list of potential moves (that each piece represents) was the most natural extension possible of the previous Hexapawn assignment. Unfortunately, because each piece class has multiple functions and data members, instantiating them tens of thousands of times during game tree exploration dramatically increases memory expense and iterating over them in a list dramatically reduces runtime performance. Given another two weeks, I'd store each move as a single character and have a single move generator handle all the different cases for each type of piece, which would be performed in conjunction with use of a transposition table during negamax search. Finally, I'd include an opening book of moves for the player, rather than naively checking every single piece at ply 1.

There was extensive unit testing for move generation. Every combination of legal moves can be reliably generated for all pieces. The Negamax and alpahbeta pruning implementation is extremely hard to test and debug, so all I can say with some certainty is that after stepping through hundred of recursive calls, the function (in those cases) appropriately scored the state and the expected move was selected from the list of move bu the state evaluator.