Team Name: Little Green Man

Team Members: Alex Wong, Li-Kai Chi.

Summary:

Our Othello AI works by taking by picking the move with the highest value. This value is based on the difference between number of player's piece and enemy's piece, as well as the possible number of moves, if the move is taken. The amount of depth, or number of steps to look ahead, is dependent on the depth; which can either be fixed from input, or dependent on the time limit given.

In-depth explanation:

Reading Input

Upon receiving input from the tournament program, the play executable file will call main.java; the main java file. In main.java, the scanner will read the input and split the first input string into 5 parts; via .split(" "), which splits string using space as the delimiter. The first part is a string, "game". Because this is standard input, the first part is not processed. The second part should be either "B" or "W". This will define whether the AI will play as black or white. The third part is depth. This will define how deep, or how far, the AI will think ahead. The fourth part is time limit for each move. This will define how long the AI is allowed to think for each move. The last part is time limit for the whole game: this is not processed as the TA told us not to worry about it. These input information is stored as integer by parsing it into an Integer; Integer.parseInt(<string>).

Next, there is an if-else statement that will differentiate the decisions the AI will make, based on the role it plays; either as black or white. If playing as black, then the AI will use alpha beta pruning to decide the value of the game states as it simulates its possible moves. This will be explained later. Then, it will update the game with the new decided move. If it has no possible moves to make, it will pass. If the AI takes too long to think and cannot make a decision within the time limit, it will simply pick the first possible move to

prevent timeout error. Its decision will be printed out to stdout for the tournament program to read. Afterwards, it waits for the tournament program to feed the opponent's move or decision. It reads the next input via the scanner; scan.nextLine(). This is defined into a string, and checked if it is the string, "pass". If the opponent passes, then the AI simply does not do anything. If it does not pass, then it will split the input string into two parts, using space as delimiter, and convert it into two integers (x-axis, y-axis). This information is updated to the game. Then, the loop continues until the game ends. On the other hand, if playing as white, then it will first wait for the input string, and then make its move.

Game.java

There is a game class. The reason for a game class is every move is another possible game state. Whenever each move is made, it moves to the next depth, creates a new game class, and updates the (new) game class with the move it is simulating. Throughout the whole decision making process, there are two different types of game classes created; the actual game (named Game game) and the simulated game (named Game newGame). There can only be one actual game, but there can be many simulated games. Before the actual game makes a move, the AI will run through many simulated games to figure out the best move for the actual game.

At the start of the game, the AI sets up an empty board via instantiating game without input. Then, it instantiates the game again, but inputs the game created and board size to get the game started.

The update method is a key feature in the game class. In short, the update method reads the new move the player, or opponent, is making or made, and updates the possible moves that each player can make. It also updates the number of pieces of both the opponent and player. This sounds simple, but it is actually extremely complicated because of all the checks that it to go through to update the possible moves.

Alpha Beta Pruning

Alpha Beta Pruning moves down each depth by recursion. At each recursion, more game classes (simulated games) are created for each possible move, and also plays as if it has swapped places with its

opponent; the min-max change in minimax at each depth. When it reaches the depth limit (the leaf), it will calculate the value of the state of the game. The value is based on the difference in number of player and opponent pieces on board, and difference in number of possible moves for player and opponent. The weight of the former is 1, and the weight of the latter is 0.5. This value is passed up to the parent node, and processed via minimax and alpha beta pruning. It simply follows the concept of the pseudo code given.

Depth limit VS Time Limit

One thing that the AI has to base its decisions on is making decisions based on the depth limit and time limit. If the time limit is more than zero, it overrides the depth limit. If time limit is zero, then we follow the depth limit. To implement this, I added an if-else statement such that, whenever time limit is zero, then time limit is a huge number. If time limit is not zero, then depth is a huge number. Then, within the alpha beta pruning method, the method will recursively go down depths only if both the depth has not reached depth limit and time taken is not more than time limit minus 100 milliseconds. Thus, the reason why I added the if-else statement in the first place. On a further note, the reason for the 100 milliseconds in the time limit is that it takes the program some time to update after pruning. If it stopped at exactly the time limit, it won't have time to update and choose a move. This would result in timeout error. Additionally, right at the start of the alpha beta method, the start time is recorded. At each depth, the time is recorded and is subtracted from the start time to check if it has passed the time limit (minus 10 milliseconds). Once it has exceeded this, it will return the value of its current game state, and then, the alpha beta pruning's final step; passes the value up the parent node recursively.

Priorities

Additionally, because we know that the corner is the most valuable piece in the game. I added an if statement to override the value of the corners. Whenever the corner is a possible move, the value of the corner will become 99999, which should be highest in the game. This is to ensure the AI choses the corner whenever possible.

Others

Poss_mov & Axis class.

Poss_mov class (stands for possible moves) contains a List of axis (axis is a class). It has a delete and add method, and stores the possible moves in the format of axis. Axis class has an x-axis, y-axis and a value.

Data Structure in the game Class.

Within the game class, there is a poss_mov triple array (moveTable), a poss_mov single array (playercombos), an integer double array board and an integer single array (pieceCount).

moveTable

moveTable records both the possible moves, and the piece (in coordinates) that enables this possible move to be possible. This is very useful because there can be multiple ways to make the same possible move for a given coordinate possible. Upon update, if one of the ways for this given coordinate is made not possible, we need to check if there are other ways to make this given coordinate. If there isn't, then we have to remove the given coordinate from the possible moves. If there is, then the moveTable will show that there is another piece that makes the given coordinate possible. Then, we know not to delete this coordinate from the list of possible moves because it is still a valid move.

playerCombos

playerCombos is similar to moveTable, but instead of storing both the possible move and the corresponding piece that allows the former to be possible, playerCombos only stores the coordinates that is possible for the player, or opponent. It is a poss_mov single array because it only needs to know if the requested information of possible moves is for a black or white player. This is most often used to make decisions.