COMP 1100
ASSIGNMENT 3

THOMAS LA - TUESDAY 1800
KIRAT ALREJA U7119530

Introduction

Assignment “Othello” builds the functionality of an AI bot which always makes the best possible move by looking ahead into future game states, using the Minimax algorithm. The AI uses a combination of weighted sum, greedy & mobility heuristics to rank moves, always selecting the one with the highest heuristic value.

Program Design

I started by building some basic helper functions, which provide essential utility to the primary AI function

`getUS` & `getOpponent`

These functions return the value of `Player` in the initial `GameState`. This is crucial since the AI can be called either `Player1` or `Player2`; we need to keep track of our & opponent’s value. Since the initial `GameState` always holds our value, `getUs` returns it unchanged while `getOpponent` reverses it.

`takeFirstEle` & `takeSecondEle`

Throughout the entire code, I have used a tuple of type (`Move`,`Int`) to store a Move & its corresponding heuristic value together. However, in many functions, I found myself wanting just the first or second value from this tuple. These functions help achieve the same.

Next, I coded three heuristic functions that help calculate a diverse set of information for any game state.

`mobilityTreeHeuristic`

Mobility heuristic is based on the number of moves a player has in a state. The more we decrease our opponent’s moves &, as a result, maximize ours, the lesser chances they get to flip our coins. This heuristic returns a positive result if it’s our turn & the opposite for our opponent.

`weightsTreeHeuristic`

Weighted sum heuristic is based on the fact that certain positions on the board help increase the stability of our coins, meaning that they secure our coins by making them immune to being flipped or less likely to be flipped.

The helper functions for this heuristic are as follows

`weights`

`weights` is a list of values for each position on the board. Corners have been assigned the highest weightage since they are the most stable positions. On the other hand, the positions adjacent to the corners are vulnerable; hence they have been assigned a high negative weightage. The positions between the adjacent positions of the two corners are pretty stable; hence they have been assigned a weightage that's almost half as the one for corners. Similarly, all the other positions have been given appropriate values.

`gameStateToBoard`

This function returns the value of `Board` which is stored inside a `GameState`.

`boardToPlayer`

`boardToPlayer` converts the positions stored inside the value of `Board` into an integer list with each element as either 1, 0, or -1; depending on whether we own it, it is empty, or our opponent owns it.

`sumZippedWeights`

This function multiplies every element of the weights list with each corresponding element of `boardToPlayer`, & sums the resulting list to give us the score for any `Board`. Hence, if a move allows us to capture stable positions, it will have much more weightage than those that don't. This heuristic will also help us steer away from any baits the opponent may use to lure our AI into unstable positions.

Now, coming to the greedy heuristic

`treeGreedyHeurisitc`

This heuristic calculates the running score of a state. Depending on the `GameState`, various values are considered. If a state allows us to win, it is assigned an extreme value since it provides us with significant benefit. Similarly, if a state allows our opponent to win, it is given an extreme negative value since it is the worst state for us. Lastly, if a state allows our opponent to end the game in a draw, it is assigned a slightly better value since a draw is better for us than a defeat.

`treeGreedyHelper`

This is a helper function for `treeGreedyHeurisitc` & it calculates the running score using `currentScore`.

Finally, the AI uses a heuristic that combines all the three heuristics

`bossHeuristic`

`bossHeuristic` sums up the values obtained from the mobility, greedy & weighted sum heuristics. Each of the heuristics have been assigned a weightage, which have been calculated in a way that does not encourage an individual heuristic to consistently dominate the sum.

- Weighted sum, scaled to 10 : I found weighted sum to be the most optimal heuristic when used by itself
- Mobility, scaled to 3 : Although having higher mobility is essential, I found that the AI ended up picking “quantity” over a “quality”. It avoided corners in certain situations to maximize the mobility, which is not always ideal.
- Greedy, scaled to 1 : I found greedy to be the worst heuristic by itself since it does not strategically use the position logistics like the other heuristics. However, when a win occurs, greedy returns an extreme value which instantly allows the AI to choose the corresponding move. It also saves us from states where our opponent wins. Therefore, it might not perform well by itself but adds pivotal utility in a combination.

To store all the possible moves & their values, I chose to use a `Rose` trees data structure. Since the number of moves change with every state, `Rose` trees are a perfect choice as they can store as many children as needed.

`buildGameTree`

This function generates a list of `Rose (Move,Int)` trees for every legal move in the initial state. `(Move,Int)` type has been chosen to store the move & its heuristic value together conveniently.

The function takes two arguments of `GameState`, the first one being dynamic (changes at every level) & the other one being static (original state). The original state is the first state that our AI receives, making it crucial since it stores our `Player` value. Therefore this value is passed on throughout the tree & heuristics, so that our AI can differentiate between us & the opponent – regardless of playing as `Player1` or `Player2`.

`newGameState`

This is a helper function for `buildGameTree`, it returns the new `GameState` after a move is applied using `applyMove.`

`maybeLegalMoves`

This is a helper function for `buildGameTree` & it generates a list of legal moves if the corresponding `GameState` exists.

Now, coming to the core function of the AI

`minimax`

`minimax` chooses a `(Move,Int)` tuple from our list of game trees, assuming the best counterplay at each depth. `MinMaxPlayer` stores a `Min` or `Max` value making the function easier to interpret. When `Max` is playing , `minimax` would choose an optimal value from `Min`'s view at each node. When Min is playing (opponent), `minimax` would do the opposite.

`optimalValue`

`optimalValue` is a helper function for `minimax`. It takes a single tree & returns the optimal `(Move,Int)` from its children, depending on the player. If there are no children, the node value is returned.

`chooseMove`

`chooseMove` chooses the maximum or minimum `(Move,Int)` value from the final list of tuples, which is generated by mapping `optimalValue` over all the trees in `minimax`.

The primary function initiates all the above functions

`minimaxAI`

`minimaxAI` builds a tree of the any depth using `buildGameTree` and applies `minimax` on it to output an ideal move, given a list of all legal moves.

Reflection

It was hard to get my head around the concepts of the Othello, especially understanding what heuristics are. Thankfully, I found this research paper [1] which explained the same.

It talked about various types of heuristics, such as stability, mobility, corners & greedy. I spent time absorbing that information & was eventually able to figure out the Haskell code for them. The paper also inspired me when assigning weights in my weighted sum heuristic.

Another challenge I faced was understanding the minimax algorithm. Upon researching, I found a fantastic blog [2] which explained minimax with some pseudocode. After spending some time working out the steps on paper, I was able to code it successfully.

Lastly, I attempted to extend my minimax with AB pruning, but it did not yield a positive result. Even after trying multiple approaches [3] , the function did not seem to pick the best move & even lost to the standard `minimaxAI`. If I were to do the assignment again, I would spend more time understanding pruning.

Testing

As it was tricky to test some functions with direct outputs in GHCI, I worked out some cases for them on paper. For example, it is pretty challenging to read the output when testing a tree. Hence, I testing them by tracing the function on paper itself. I followed a similar process for `minimax` algorithm. To test the helper functions, I used unit testing.

I played all my bots as player 1 and player 2, to ensure that the code was able to identify us & the opponent successfully. A significant part of my testing process were the heuristics. As I was working to combine three heuristics, I ensured that they could at least beat a basic greedy AI individually.

So, I played matches between all the possible combinations of heuristics & made a spreadsheet of coins lost and won by each one (inspired by [1]). From this spreadsheet, I was able to visualize which heuristic performs well in what situation.

References

[1]https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf

[2] https://thesharperdev.com/implementing-minimax-tree-search/

[3] https://www.javatpoint.com/ai-alpha-beta-pruning