ASSIGNMENT 3

Name: Tay Shao An

UID: u7553225

Laboratory time: Wednesday 5:00pm

Tutor: Pramo Samarasinghe

INTRODUCTION

Assignment3 is a Consecutive-Dots AI which seeks to produce the best move based on a provided game board. Within the code there are two available AIs (a greedy AI and a Minimax one) which use different methods when analysing a game board.

PROGRAM DESIGN

> Important datatypes:

- Gamestate - Keeps track of the current state of the game
    - Sores:
        - An integer: How many in a row a player needs to win
        - A turn datatype: Which player's turn it is,
        - An (Integer, Integer) tuple: Size of the board
        - A list of Column datatypes: Position of each piece on the board
        -
- RoseTree - A single unit of measurement used by the Minimax AI
    - Stores:
        - A (Gamestate, integer) tuple: The integer is the priority of a certain Gamestate.
        - A list of RoseTree datatypes

PROGRAM DESIGN: AIGREEDY

> AIgreedy is an AI with no lookahead and determines its next move by looking at the current state, and placing a piece where the number of pieces it has in a row will be the greatest.

> getLocations

- Used in both aiGreedy and aiMinimax
- Obtains all available locations the AI can place a piece given a GameState

> locCheck

- Used in both aiGreedy and aiMinimax
- Determines whether a piece, if placed at a given location, would allow the Ai to win.

> checkNum

- Used by aiGreedy
- Calls getLocations on a GameState to obtain all possible locations.
- For each location it calls locCheck, and if it allows them to win, it returns a move.

> aiGreedy

- The main for aiGreedy
- Takes the current gamestate, calling checkNum repeatedly, but reducing the number of pieces it takes to win.
- When checkNum returns a move, it outputs the move.

PROGRAM DESIGN: AIMINIMAX

> An AI that generates a RoseTree and compares various routes to determine the move with an optimal “priority”. The functions used by aiMinimax are split into RoseTree generation and RoseTree processing.

> priorityFind

- Used for RoseTree generation
- Priority:
    - An Int which determines the number of pieces in a row, a piece, if placed at a given location, will yield.
    - If the RoseNode is storing the GameState of the enemy's move, the priority is a negative value.
- priorityFind calls locCheck repeatedly until a GameState is returned, and outputs the number of times it calls locCheck as the priority.

> rosetreeCalc

- Used for RoseTree generation
- Generates the next "layer" of a RoseNode
- Calls getLocations on the GameState stored within a RoseNode
- Determines:
    - The priority of placing a piece at each of the locations by calling priorityFind.
    - A new GameState of when a piece is in that new location
- Stores each priority as well as each new GameState in a RoseNode, and stores each RoseNode in the original RoseNode.

> roseMap

- Used for RoseTree generation
- Applies a function on every RoseNode in a RoseTree with no other RoseNodes stores within it (i.e the bottom of each RoseTree).

> roseGen

- The "main" function used for RoseTree generation
- Takes a base RoseTree, and "roseMaps" rosetreeCalc on itself repeatedly (depending on the lookahead), and outputs the resulting RoseTree.

> worstNode

- Used for RoseTree processing
- Determines the enemy's move based on which location would give it the most number of pieces in a row.
- Takes a list of RoseNodes and returns which one has the lowest priority

> nodeRunner

- Used for RoseTree processing
- Returns the accumulated priorities of all routes within a RoseNode
- When called on a RoseNode with positive priority, it calls worstNode on the list of negative RoseNodes within it,
- Adds the priority of the negative RoseNode and the positive RoseNode and sends this value forward to the next layer of positive RoseNodes and calls nodeRunner on them.
- When it reaches the end of the RoseTree, it adds the accumulated priority to a list of all other accumulated priorities.

> quicksort

- Used for RoseTree processing
- Sorts a list of values from big to small
- Taken from: https://stackoverflow.com/questions/19082953/how-to-sort-a-list-in-haskell-in-command-line-ghci

> valueFind

- Used for RoseTree processing
- Returns which index in a list has the greatest value

> priorityCheck

- The "main" function for RoseTree processing

- Taking a RoseTree, it maps nodeRunner on each of the RoseTrees stored within it.
- It then calls quicksort to sort the routes from highest to lowest priority and takes the route with the highest priority.

> aiMinimax

- The main function for aiMinimax
- Generates a RoseTree by calling RoseGen, and calls priorityCheck on it.

TESTING

> Testing individual functions:

- AITest.hs contains basic tests to ensure that most functions run as they should
- I used a basic GameState with only 3 rows and 3 columns as a sample test.
- Using this simplified GameState I was able to better visualise its layers of RoseTrees for when I tested RoseGen functions like rosetreeCalc and roseMap
- For NodeRunner, I calculated the priorities of each of its routes by hand and asserted it to be the correct output for the relevant input.
- priorityFind was tested by providing a GameState with a clear location to increase the number of pieces in a row and asserting its output with the ideal GameState.
- worstNode was tested by providing a list of RoseNodes, and observing if the output node had the lowest priority
- Simple functions such as quicksort can be tested by running it and observing the output (i.e input: quicksort [5,4,1,1,10], output: [10,5,4,1,1])

> Testing AIs:

- aiGreedy's next move can be predicted easily, as it will only ever try to increase the number of pieces it has in a row

- aiMinimax had to be tested more rigorously as their decision-making is dependent on multiple layers of lookahead
- The weightages of the positive and negative nodes, when calculating the accumulated priorities of each route, when changed around would create different scenarios in which the AI would become more defensive or offensive
- I tested aiMinimax's effectiveness by running it against aiGreedy, and observing the result.

REFLECTION

> Encountered problems:

1. Despite my best efforts, there are times in which the aiMinimax makes less than optimal decisions. This is ultimately due to how the weightages for the priorities are calculated and although I have adjusted them as best I could, there are most likely better configurations
2. The RoseTree generation took up the majority of my time working on this assignment. Initially, I kept running into problems handling edge cases, and places where moves would be illegal, and the priorities generated were incorrect.
3. Working with Maybe datatypes was tricky at times, as I had to find a way to parse Nothing values.
4. My RoseTree processing algorithm had to be changed more than once, as the algorithm for comparing accumulated priorities was wrong. Initially, I was only comparing priorities layer by layer (instead of as a whole), and the end result was something similar to aiGreedy.