COMP1100-Assignment3 Report

Connect X

Name: Yutong WANG

ID: u6293753

(This is assignment I was originally inspired by Rong Xin, and I discussed this assignment with Weijie Tu.

Contents

1 Introduction

The aim of the assignment this time is to implement the ?makeOrders? function using our own strategy to buy or sell stocks thus increase our total without bankrupt.

2 Strategies Explaining

2.1 How to deal with multi dimensional lists

It is really hard to make a choice between building a tree and building a list of lists. Since at the first glance of this assignment, people would think of building a tree directly, but after digging into this assignment, I found out that the constant generation of each possible boards itself is already a tree, and dragging this into a tree structure actually slows down the efficient of the programme. Although list of lists is attracting and interesting, it is a pity that I do not have enough time to figure out how to operate on multi dimensional lists, and all I can do for now is just generate them manually:

```
gen2 b = map genPossibleBoards $ map snd (genPossibleBoards b)

gen3 b = map (\x -> map genPossibleBoards x)
               $ map (\x -> map snd x)
                       $ map genPossibleBoards
                               $ map snd (genPossibleBoards b)

gen4 b = map (\x -> map (\z -> map genPossibleBoards z)x)
               $ map (\x -> map (\z -> map snd z)x)
                       $ map (\x -> map genPossibleBoards x)
               $ map (\x -> map snd x)
                               $ map genPossibleBoards
                                $ map snd (genPossibleBoards
```

It is evident that there is a pattern in the generating of list of lists and what I was trying to do is to write a function that can generate n layer.

2.2 Using Tree Structure

When thinking about generating tree for this assignment, at first I thought that this must be a tree with 'Board' as its nodes which is the easiest way to implement tree structure:

```
genDepthBoards :: Int -> Board -> [(Index, Board)]
genDepthBoards 1 bb = genPossibleBoards bb
genDepthBoards d bb = concatMap (genDepthBoards (d-1))
                          $ map snd (genPossibleBoards bb)
```

```
-- generate the next possible board situation
genPossibleBoards :: Board -> [(Index, Board)]
genPossibleBoards root = zip orderedIndexes (map
                (updateBoardNoScore root) orderedIndexes)
    where   -- order the index in a order that will prioritise
                -- the center columns.
                orderedIndexes = sortBy (comparing (\i -> abs
          $ (fst(dimension root) + 1) `div` 2 - i)) validIndexes
                validIndexes = filter (validMove root)
                                [1..fst(dimension root)]
```

However, if using 'Board' as each Node, then we need to use the function 'updateBoard' or 'updateBoardNoScore' at each Node, and as we all know, both functions works slow, and if we want to increase our chances of winning the game, we need to look ahead as more steps as we can. So, how can we look more steps at such a low speed.

So how about using the index of each moved column to represents the Nodes of the tree? That sounds nice and clear, so how can I know the details of the transformation of the board with only the last index? The answer is to use a list of indexes as each Node, and then I can know the exact way of how the board comes to each states without having to update Board at each Node. The only Board we need is the root board which is the current state, then I use the root board to figure out the valid indexes of the column and generate a list of valid indexes, then append each valid index to its corresponding root indexes, then the path of the board is showing clearly.

```
type Moves = [Index]
```

However, the only Board I know is the Board that passed in the 'makeMove' function and the original root node of it is an empty list. In order to get the valid indexes that we can put pieces in, how can we find out the valid indexes after the first layer without updating Board? The most easiest way is to count the occurrence of each column index in the Moves, if the total occurrence of a index in Moves is greater than the height of the board, then it is invalid. If the above situation does not happen, then check if occurrence of the index that appears in the Moves plus the original height of that index is equal or greater than the height of the board, if it is, then the index is invalid. Also need to check if the root Board itself has already have a column that is full.

After generating the possible indexes, we need to build the sub nodes of each root nodes. We need to append each index of the possible indexes to the root Moves. There is also a small question that is to append an element to a list since haskell does not have the built in function for us to use. At first I wrote the function as below:

```
add ::  [a] -> a -> [a]
add list e = reverse $ a : reverse list
```

However, it uses 'reverse' twice which means it has to go through the whole list twice, thus, I rewrote the function as follows:

```
add :: [a] -> a -> [a]
add list e = list ++ [e]
```

The last part of this section is about control the depth of the tree we generate. The tree has to stop when the depth we want is 0, thus, the situation when depth is 0 is the base case, then what we need to need is just generate tree as normal and passing 'depth-1' every time one layer is created.

2.3 Minimax

This section is about the implementation of the minimax algorithm. The top of the tree is alway the max layer, since we want our score the highest, then the next layer is the min layer since the opponent tries to minimise our score. First, we need to define the 'evaluate' , the output of which is a score, which will be utilised when using the minimax algorithm to analyse to leaf node of the tree:

```
evaluate :: Board -> Board -> Score
evaluate boad bd = (myGetScore bd (turn boad)) -
                   (myGetScore bd (otherPlayer $ turn boad))
```

Then we need to think about the algorithm of minimax. It is clear that after the max layer, the next layer must be the min layer. For a tree, that means, if we apply max to the root, then we will need to apply min to its children. In that case, we need to define two functions, one is max algorithm and the other is min algorithm and they need to call each other recursively until the end of the tree which is the leaf node:

```
-- using max algorithm as example
maxAlg maxb (Node _ ls) = maxOfTuple $ map (minAlg maxb) ls
```

At the end of the tree, obviously we can not call the other function anymore, it has to end, so we will define a base case:

```
 -- using max algorithm as example
maxAlg maxb (Node x []) = (evaluate maxb
                                    (myUpdateBoard maxb x), x)
```

Since what the minimax algorithm depends on the score of each board, there is no doubt that we need to evaluate the score of the Board of leaf nodes using the evaluation function above. Moreover, since in order to get the score of the board, we need to update the Board from the original root using the Moves we have, and a simple recursive will do that for us.

```
myUpdateBoard :: Board -> Moves -> Board
myUpdateBoard grid (y : ys) = myUpdateBoard
                              (updateBoardNoScore grid y) ys
myUpdateBoard grid [] = grid
```

The result of the minimax functions is in the form of (Score, Moves). And what we need to do here is to only compare the Score of each child node and choose the tuple that has the largest or the smallest first element depending on which algorithm they use. The detailed illustration of this is more readable by drawing it on the paper.

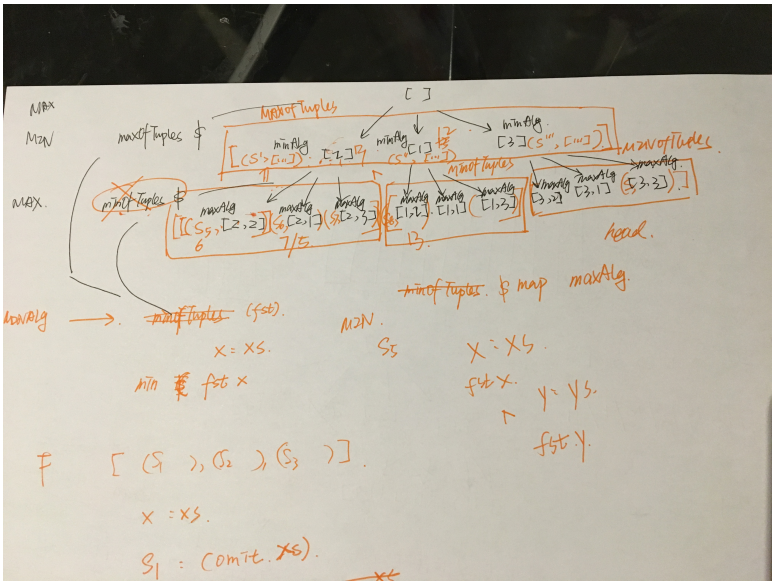

Figure 1: the minimax algorithm visualization

2.4 Alpha Beta Pruning

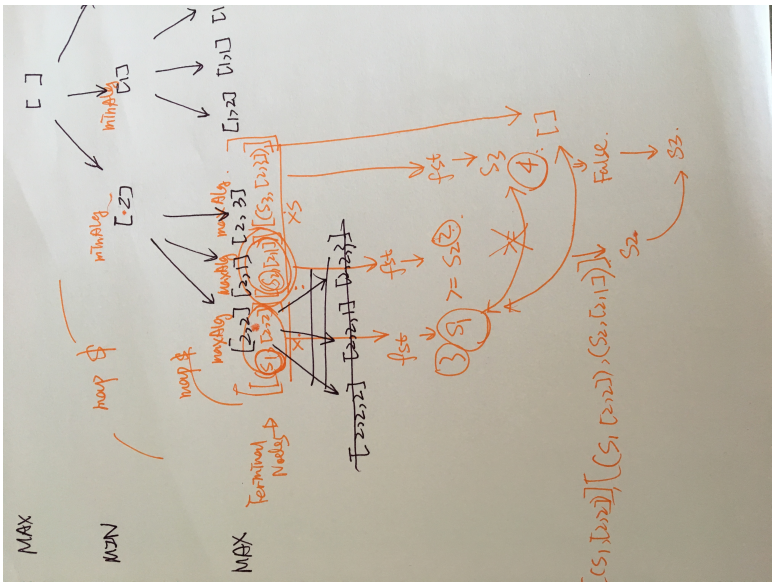

Figure 2:

This section is the illustration about the implementation of alpha beta pruning. The implementation of alpha beta pruning might have some mistakes, since the rank of my programming dropped from top 30 to far behind, however, I decide to use it anyway, since it can really improve the efficiency of the programme: from the fastest speed at around 0.3 to now the fastest speed can achieve around 0.002 and is able to look 8 steps ahead. Since alpha beta pruning is used within the minimax or negamax algorithm, so it is easy to imagine that it might need to modify the original minimax functions. As the picture showing below, the first child node will pass the value to the root node, then the parent node will check if the value greater than or smaller than the value they hold depending the max or min layer it is, then that is the value they want, they would take it, then compare the root node value with the next child node. From my perspective, it is actually the comparison between each child node from the left to right, since the original root node does not have value, or more precisely, the root node only have the most undesirable value, so that means it will always take the value of its first child and then compare it with the rest of the children until it is an empty list, then we should stop, and the empty list situation is obviously the base case. I'm not sure if the implementation of the alpha beta pruning is correct, but for me, at least it is easy to image in this way, moreover we do not need to concern about how to compare the value between the two layer in this case.

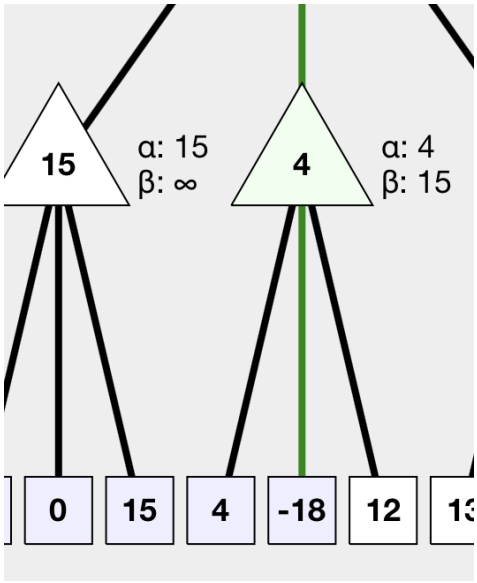

Figure 3: a snapshot of normal alpha beta process

After applying minimax functions, it will eventually reach the base case, and evaluate the children nodes into form like this: (Score, Moves) Now, we will take the first one of the evaluated children lists and compare its Score with the score of the rest of the children nodes.

```
minCompare :: (Score, Moves) -> [(Score, Moves)] -> Bool
minCompare _ [] = False
minCompare s (y:ys) | fst y <= (fst s) = True
                    | otherwise = minCompare s ys
```

Take minCompare as an example, we take the first element of the child node, and compare its Score with the next one, if it is smaller than or equal to the first Score, we will keep this first Score, and compare it with the next one, if it meets a value that is greater than itself, then it will return straight forward and never look at the rest of the children nodes.

3 Bottlenecks

This section shows the bottlenecks and potential bottlenecks when writing this assignment.

3.1 Check the height of the original Board

After implementing the minimax algorithm, there is an error occurs-"failed to make decision in time". The reason of this is because I forgot to check the original root Board if there are already a column that is full, thus it generates nodes that is invalid, and that's why when a column is full, the programme would go wrong. For example:

```
>>> genTree Board{board = [[],[Blue,Red],[Blue,Red,Red,Blue],[]],
                          blueScore = 0, redScore= 0, turn = BlueBot,
                          dimension = (4,4),connect = 3} 1 []
Node {root = [], subForest = [Node {root = [2], subForest = []},
                Node {root = [1], subForest = []},Node {root = [3],
                          subForest = []},Node {root = [4],
                                  subForest = []}]}
```

3.2 Filter hasWon

Imagine that if we use 'placePiece' in 'myUpdateBoard' function instead of 'updateBoardNoScore', what will happen? The assumption is that it will generate more nodes that is invalid, since the 'placePiece' function simply update the Matrix Cell and won't check the situation when the board has won like what 'updateBoardNoScore' would do. And that would not only slow down the speed of the programme as the game goes further, but also may mislead the evaluation function.

4 Future implementation

1.Order the tree One way to make the programme more efficient is to order the tree by placing the possible desired result to the left, since if the root node are more likely to get the most suitable value from the beginning, the chances of pruning is increase, and if the alpha beta pruning can prune more nodes, the programme can definitely work more efficiently.

2. 'myGetScore' Function Improving the 'myGetScore' function is also a good way to make the programme smarter. Since both the minimax algorithm and the alpha beta pruning all depend on the evaluation result to make the right decision. Unfortunately, I don't have enough time to dig into the get score function to improve the performance of the programme due to my improper time arrangement. There are just a few situations

I wish I have implemented: 1) get only the top piece of each not empty column and one empty piece above them

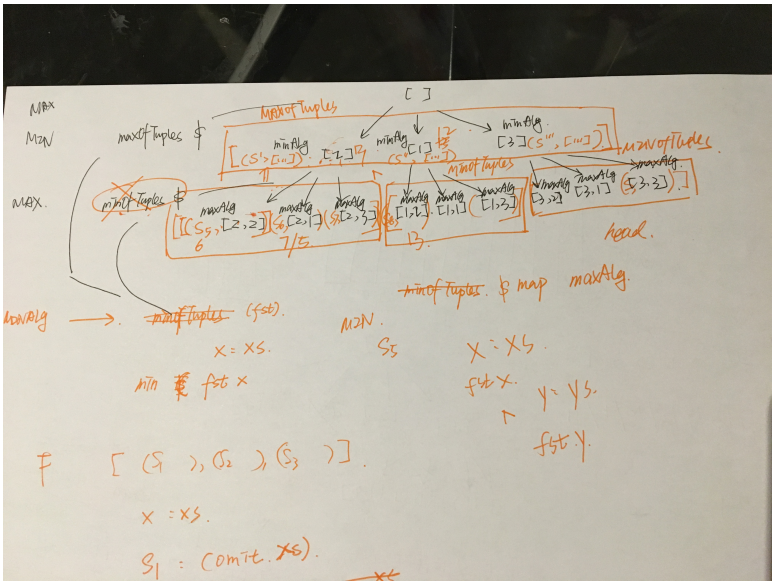

Figure 4:

2) implement the strategy that forces the opponent to make a move when he would rather make no move at all, and in order to achieve this, the player should be able to lead the way odd and even squares are divided up among players. 3) If there is only one empty piece between two parts of player's bot, and the total number of these bots is equal to streak -1

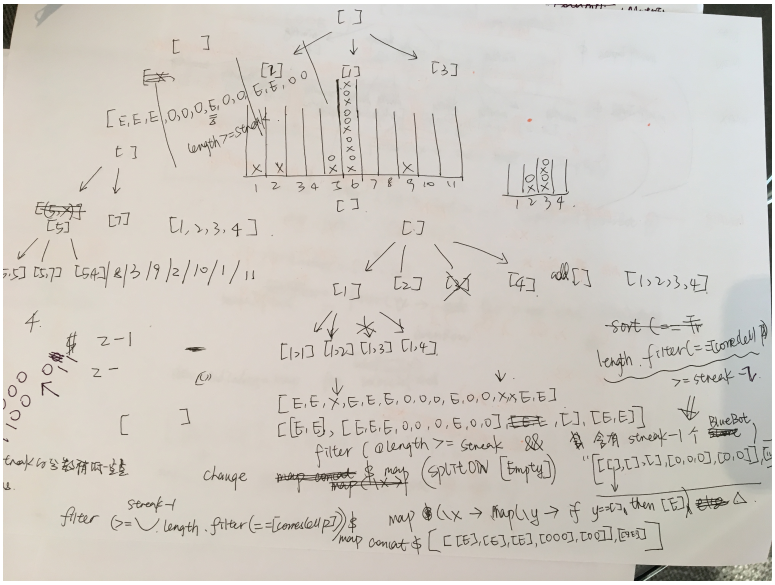

Figure 5:

5 Conclusion

In conclusion, this assignment is really interesting and the more time I spend on it, the more detailed improvement or possible mistakes I can realise. From my perspective, the difficult parts of this assignment are the implementation of minimax, possible pruning, and also the rating of each board. Moreover, it is really useful to write down your idea on a piece of paper and go through the whole process before typing codes straight forward. Furthermore, since I did not properly arrange my time, there are many details I do not have a chance to figure out, in the future studying, I will arrange my time properly.