COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

ASSIGNMENT COVER SHEET

Course Code:COMP1100

Course Title:Programming as Problem Solving

Assignment Number:3

Tutorial ***(e.g. mon10-12)***:Thursday 3 pm to 5 pm

Tutor's Name:Debashish Chakraborty

Student's Name:SUKAI HUANG

Student ID#:U6492211

Hours spent on this assignment: 26 hours

Pledge: All parts of code and report for the assignment should be written by myself alone.

Alpha-beta pruning codes

1. **Slightly different structure**

Basically I follow the handout given from the lab and consequently construct the alpha-beta pruning algorithm code. Nevertheless, there are some structure differences between mine and the one from the handout. I use a Boolean input to signify whether it is a maximizing-value state or minimizing-value state.

2. **My own 'foldl' function**

Moreover, since Haskell is not able to make "for loop", I create my own 'foldl' function to fulfill the pruning mechanism. In my own 'foldl' function, alpha and beta value are updated and compared after each fold and thereby fulfilling the pruning mechanism.

3. **Decision function**

The decision function is slightly different from the one in Lab09. Since we consider board as implicit tree, I can only start the alpha-beta pruning at second depth and then use 'zip' function to zip each heuristic value with the corresponding column index. Note that I also create my own 'map' function in the decision-making function so as to pass the alpha beta value to the next node.

One thing to mention is that initially I did not pass the alpha value from a 'depth-2' node to the next 'depth-2' node (Note that only alpha value is transmitted at 'depth-2' level). Theoretically if I do not transmit alpha value at 'depth-2' level, the efficiency will decrease. However I do compare the time used and I do not find out big time difference.

Heuristic Value code

I do ponder over how to make the heuristic value code. The current code for heuristic value is just the value that my score minus the opponents' score. However, I once tried to slightly adjust the ratio of my score and the opponent score (e.g. value = my score – 2*opponent score) to make my algorithm more defensive or other way round. However this adjustment does not significantly change my rank.

Index List

Normally the column index start from 1. However I modified my indexList so that the algorithm will calculate the middle moves first. I made this modification because I believe usually the moves at the middle of the board are more important than the moves at the side of the board. Therefore I think it helps to make alpha-beta pruning process faster.

Avoid invalid move

1. Guard in alpha-beta pruning function

The current method to avoid invalid move in my code is to check whether the updated board is as same as the initial board. Since the updated board will directly become the input of the alpha-beta pruning function, I make a guard in the alpha-beta pruning function and if the updated board is as same as the initial board, the alpha-beta pruning function will directly return extremely low heuristic value (-2000), and thus the algorithm will definitely skip this move.

2. Drawback of this method

One drawback of this method is that since alpha-beta pruning function will recursively run itself until it reaches the bottom end, this guard will be called again and again. It actually is not efficient.

Actually we can just make the checking-invalid-move function at the making-decision state (which is at depth 2 level). In that case, the checking-invalid-move function will be called just once and thus increase the efficiency.

However I do not do that initially because it is hard to implement this checking-invalid-move function in my making-decision function. In my making-decision function, I zip the column index and the corresponding heuristic value of that move and then I use a supporting function to pick the tuple with higher value and then extract the index information from the tuple. It is thus quite hard to implement the invalid-move-checking function. Let's say if I check there is an invalid move and I delete that move, I need to also delete the corresponding column index from the index list. And this will become quite messy.

ConnectX2.0

When I do this assignment I do think I am investigating a black box. (Maybe it is because I have not yet studied I/O function in Haskell). I often questioned myself: did I make my algorithm correct? How come it is still so slow?

In the future I do want to see the inner process of my code so that I can check whether it fulfills my expectation and also debug it. Maybe I/O function will allow me to do that (just like print() function in python). Checking the process is really important for this kind of assignment. Let's say if I cannot check the inner process, I will never know whether my alpha-beta pruning indeed works.