Australian National University

ASSIGNMENT COVER SHEET

ANU College of Engineering and Computer Science
Australian National University
Canberra ACT 0200 Australia
www.anu.edu.au

This coversheet should be filled electronically when possible and attached as the front page of your report in PDF format.

Student ID For group assignments, list each student's ID	u6361796		
Course Code	comp1100		
Course Name	Programming as problem solving		
Assignment number	3		
Assignment Topic	ConnectX		
Lecturer	Katya Lebedeva		
Tutor	Debashish Chakraborty		
Tutorial (day and time)	Monday 12-2Pm		
Word count	1531	Due Date	23/10/2017
Date Submitted	23/10/2017	Extension Granted	

I declare that this work:

- ✓ upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- ✓ is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the course outline and/or Wattle site;
- ✓ is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- ✓ gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- ✓ in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

Signatures

For group assignments, each student must sign.

Wenbo Du

Report

U6361796
Wenbo Du

Abstract

In general, the assignment requires us the use a strategy to play game ConnectX and try to win as much as possible. In my program, there are two main method to maximum my possibility to win: center control and minimax.

Function Decompositions:

getColumn: obtain the numbers of columns for the board.
BoardRange: obtain the range of columns where bot can place discs (include invalid move within the board)
controlCenter: obtain which column is in the center of board (e.g. for a 9*10 board, it is 5)
redTurn:check if the turn is red's, if it is, return True. Else returns False.
minusScore: the score for blue bot minus the score for red bot. Here I did not use getScore instead I used record since getScore do not include the win bonus win some player has won.
allValidMove:obtain the possible next move. For example, in a 9*10 board, if columns 6 is already full, the output will be [1,2,3,4,5,7,8,9] which means the bot can only place discs on column1 ,2,3,4,5,7,8,9.
decision: from a list of scores, filter the greatest elements and add number indicates if the bot get this score where they need to place the s. For example, if the input is [1,0,5,4,5,3,2,5], first the function will filter the greatest values ([5,5,5]), then add their columns number in the original list [(3,5), (5,5), (8,5)]. Here, columns are the (index+1).
decision': nearly the same as the function decision, the only different is here we filter smallest values
FinalMove: the function handles this kind of situation: in the next Move, the player will get the same scores (best result) in several columns, where should they put discs? The function will make player make the best choose: put their discs as near center as possible. For Example, if a player will maximum the possibility to when they put discs on 1, 2, 6,7 columns (on a 9*10) board, it is better when they put discs on 6 columns since 6 is near to the center of boards.
a series of newBoard: newBoard1 return a list of Board, which indicate the potential move for the next 1 steps. NewBoard2 returns a list of lists, and all element are Board, which indicates the potential move for the next 2 steps. Accordingly, other functions are the same. In particular, for newBoard1, in the first situation, it means if in the next 1 move, either players will win, the new Board is the same as input.
a series of score: get the potential scores(minusScore) for these potential moves.
a series of lay: these function use minimax strategy. For lay2, score2 two indication the all potential scores for the next two moves. Lay2 get the minimum score from these a list of lists containing all possible score, and this minimum score form a list. For lay3, first get maximum then get minimum. For lay4, first get minimum then maximum then minimum. The same for remaining functions.
a series of lay': the opposite of the series of lay. This time the function first gets maximum score from a list of lists containing all possible scores for next 2 moves. Then, minimum, accordingly.

Minimax Strategy

Connect X is a zero-sum game. If red player wants to win, it has to maximum its scores and minimum blue player's score. And the same as blue player. Here I used the function minusscore to simplify the situation. If red player wants to win, its needs to minimum the scores (blueScore -redScore). If blue player wants to win, its needs to maximum the scores (blueScore - redScore). For red player, in the first layer of minimax tree, it needs to find the smallest elements and make move according finalMove function(decision'). For the blue player, in the first layer of minimax tree, it needs to find the greatest elements and make move according to finalMove function(decision). The same as layer2 and layer3. That is why I need different functions for different players (decision and decision', layN and layN')

Control Center Strategy

The player may face this kind of situation: if they only look ahead one move, they find if they want to maximum their possibility to win, they can put discs on several columns causes these choose gives them the same result based on their ability to look ahead. In this situation, my strategy is make the player more likely to control center. That is, if putting discs on several columns gives them to same results (all of them are best results), it is better if the player put their discs as near center as possible. Cause if so, they are more likely to have more opportunities to connect their discs (both left and right) to win the game. The same as when they look ahead 2 or more, because they must choose where to put discs in the next move. Control Center strategy can be used too.

Function Efficiency

Unlike the top players in the ranking board, I failed to use alpha beta pruning strategy, so my bot cannot look ahead many moves. In majority of times, it looks ahead 3-8 moves. But I do find my code works so slow for the first several move (just look ahead 1 or 2). It becomes faster and faster as the numbers of discs increase on board. I did not get the reason for it. But I do find sometimes I can win these who can look ahead lots of moves. From my observation, the advance of my bot is that it can try to put discs near the centre of board as much as possible, so even I failed the "predict" the future of the game, I can try to maximum the possibility to win by the maximum the possibility to connect more discs. I do notice I get an error when place with one player. That is because function finalMove do not have pattern match when the list is empty, however, I did not fix it because it will influence my performance in other situations.

Further Implementations

Alpha-beta pruning: the best strategy for this kind of zero-sum game is alpha-beta pruning. I success implement part of this strategy but fail to apply them on connectX: alpha-beta pruning used the character for Haskell named 'lazy evaluation'.

1. To achieve this, we need to replace build in function maximum by a function like this: Input Int and [Int], for element in [Int], from left to right, if any element >Int, return True. Else, return False. Using these kind function, the computer will not evaluate unnecessary element. For example, if input 1 and [1,2,3,2,1] the function will only evaluate 1,2 in the list. Compared with build-in function, this kind of evaluation saves lots of time especially when the length of list is too large. The same as minimum build in function.
2. Next step is omitting unnecessary list: replace map minimax or map maximum function with new function with same the usage. These functions take [[Int]] and return [Int], the output will be different from map minimax or map maximum, but when evaluate to the next step (shallower layer), the output will be the same.

The reason why I failed to use alpha-beta pruning is because I failed to the implement use a chain of mapMinimax (the alpha beta pruning version of map minimax) and mapMaximum. The different from alpha beta pruning and minimax is alpha beta cannot be used in a single layer. I failed to use several layers of alpha beta pruning and cannot implement where to move when the player need to make decision.

Function redundant: My code looks so long and kind of redundant. This is because every look ahead, I made a new tree represented as nested lists (because we only need the value on the roots, other node is nothing). Another way is used build-in function like splitAt: it will input a board and output a list of board([Int]), which indicates all roots in the tree. Then in other functions split them then the depth of list can indicate the look ahead moves. Another part is about the issues of the lay and lay'. Blue score minus red score in every board return the values in nodes and minimax tree. Because every time the player is different, so I build two different minimax strategies for different players. This makes my code too long.

Conclusion

For this assignment, I had a better understanding for minimax and alpha beta pruning. Furthermore, like assignment2, this assignment is about use program to solve problem in real world. Basically, my code tried to maximum to possibility to win although it is not perfect. In future, I will try to use alpha beta pruning and have a deeper understand of zero-sum game. Apart from this, after my knowledge for Haskell increases, I would like to use more build in function to make my code more readable and understandable. In particular, the use of mapM and map_M can be used the reduce the number of maps in code. But I fail to apply them cause the lack of knowledge for monad. Complexity of code is also an important to influence to the efficiency of functions.