

COMS4995 Final Project: AI Gomoku Player in Haskell

Yunkai Zhu; UNI: yz4129

1 Introduction

Gomoku, also called Five in a Row, is an abstract strategy board game. It is usually played by two players, represented by the white and black Go stones, on a Go board. Players can place stones of their color on empty intersections on the board, represented by $(row, column)$. When a player have placed an unbroken chain of 5 stones, the game stops and that player wins.

2 Project Set-up

In this project, the game board is represented by a 9×9 matrix of integers, where 0 represents empty space, 1 and -1 represent different players. Two AI players are built: the first one is implemented using the MinMax algorithm with alpha-beta pruning; the second one also utilizes the MinMax algorithm, but is implemented in a parallel method.

3 AI Player

3.1 interface

The AI player takes in a board ($[[int]]$) and a side (int) and returns the best move ((int,int)).

3.2 Basic Idea

The AI player implements the MinMax search algorithm. The idea is to assume both players uses the same strategy to play the game, which is to make the move that gives the best outcome. We use recursion to create a tree structure. Alternating levels of the tree represents alternating turns between both players. We populate the tree bottom-up. At each level, the player chooses the move with the best outcome.

The outcome is decided using heuristics, which is implemented in the *scoreBoard* function. Since the heuristics is not the focus of this project, I have randomly chosen one that makes some sense.

3.3 Alpha-beta pruning

What usually comes together with the MinMax search is Alpha-beta pruning. The idea is that when certain conditions are satisfied, we can ignore certain subtrees. However, I think to implement this algorithm, our MinMax search has to be in serial (i.e. search each children of a node in sequence). Therefore, I did not implement this algorithm in this project.

4 Performances

This section shows the performances of both AI player on the same scenario: make a move based on the current board. The AI player with alpha-beta pruning is able to make a predicition within 0.766

seconds for depth 3 and 6.932 seconds for depth 4. The runtime of the parallel AI is shown in Table 1. The results show that the parallel implementation is able to speed up the process significantly: when

Table 1: Performances of two AI players (averaged on 10 runs)

	1 core	4 cores	8 cores
Parallel AI depth 3	2.039	0.807	0.646
Parallel AI depth 4	82.46	30.808	24.684

running in 4 cores, the run time is less than half of the run time in 1 core. However, the run time of the alpha-beta pruning AI is significantly better than this parallel implement.

References

https://www.andrew.cmu.edu/user/rbcarlso/proposal_rbcarlso.html

<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.5904rep=rep1type=pdf>

```

1 module Main where
2
3 import Control.Parallel.Strategies
4
5 main :: IO()
6 main = print(r)
7
8
9
10 {-
11     Make a move on the board
12     Input: board, tar_cor, side
13     Return: board
14 -}
15 makeAMove :: [[Int]] -> (Int,Int) -> Int -> [[Int]]
16 makeAMove board tar_cor side = makeAMoveHelper board tar_cor 0 side
17
18 makeAMoveHelper :: [[Int]] -> (Int, Int) -> Int -> Int -> [[Int]]
19 makeAMoveHelper (x:xs) (tar_x, tar_y) curr_x side
20     | tar_x == curr_x = (makeAMoveRow x tar_y side) : xs
21     | otherwise = x:(makeAMoveHelper xs (tar_x, tar_y) (curr_x+1) side)
22
23 makeAMoveRow :: [Int] -> Int -> Int -> [Int]
24 makeAMoveRow row index side = makeAMoveRowHelper row index 0 side
25
26 makeAMoveRowHelper :: [Int] -> Int -> Int -> Int -> [Int]
27 makeAMoveRowHelper (x:xs) tar_index curr_index side
28     | tar_index == curr_index = side : xs
29     | otherwise = x:(makeAMoveRowHelper xs tar_index (curr_index+1) side)
30
31
32
33
34 {-
35     Score a board for one board
36     Input: board
37     Return: score (int)
38 -}
39
40 scoreBoard :: [[Int]] -> Int
41 scoreBoard board = (scoreBoardOneSide board 1) - (scoreBoardOneSide board
42 (-1))
43
44 scoreBoardOneSide :: [[Int]] -> Int -> Int
45 scoreBoardOneSide board side = (scoreBoardOneDirection board side) +
46 (scoreBoardOneDirection (flipBoard board) side)
47
48 scoreOneRow :: [Int] -> Int -> Int
49 scoreOneRow row side = scoreOneRowHelper row side 0

```

```

49 scoreHelper :: Int -> Int
50 scoreHelper num = case num of
51     2 -> 100
52     3 -> 200
53     4 -> 10000
54     _ -> 0
55
56 scoreOneRowHelper :: [Int] -> Int -> Int -> Int
57 scoreOneRowHelper [] side num = scoreHelper num
58 scoreOneRowHelper row@(x:xs) side num
59     | x == side = scoreOneRowHelper xs side (num + 1)
60     | otherwise = (scoreHelper num) + (scoreOneRowHelper xs side 0)
61
62 scoreBoardOneDirection :: [[Int]] -> Int -> Int
63 scoreBoardOneDirection board side = sum [scoreOneRow row side | row <- board]
64
65 flipBoard :: [[Int]] -> [[Int]]
66 flipBoard matrix
67     | null matrix = [[] | _ <- [1 .. 9]]
68     | otherwise = let (x:xs) = matrix in
69         [a:b | (a,b) <- (zip x (flipBoard xs))]
70
71
72 {-
73     Get all possible moves on the board
74     Input: board
75     Return: moves [(int,int)]
76 -}
77
78 getAllMoves :: [[Int]] -> [(Int, Int)]
79 getAllMoves board = getAllMovesHelper board 0
80
81 getAllMovesHelper :: [[Int]] -> Int -> [(Int, Int)]
82 getAllMovesHelper board row_index
83     | null board = []
84     | otherwise = let (x:xs) = board in
85         (getAllMovesOneRow x row_index) ++ (getAllMovesHelper xs (row_index +
86 1))
87
88 getAllMovesOneRow :: [Int] -> Int -> [(Int, Int)]
89 getAllMovesOneRow row row_index = getAllMovesOneRowHelper row row_index 0
90
91 getAllMovesOneRowHelper :: [Int] -> Int -> Int -> [(Int, Int)]
92 getAllMovesOneRowHelper row row_index curr_index
93     | null row = []
94     | otherwise = let (x:xs) = row in
95         do
96             if x == 0
97                 then (row_index, curr_index):(getAllMovesOneRowHelper xs

```

```

row_index (curr_index + 1))
97     else getAllMovesOneRowHelper xs row_index (curr_index + 1)
98
99
100 {-
101     AI functions
102 -}
103
104 initializeBestScore :: Int -> Int
105 initializeBestScore side
106     | side == 1 = -100000
107     | otherwise = 100000
108
109 switchSide :: Int -> Int
110 switchSide side = -side
111
112 chooseBetterScore :: Int -> (Int,(Int, Int)) -> (Int,(Int, Int)) -> (Int,
(Int, Int))
113 chooseBetterScore side (score1, move1) (score2, move2) =
114     do
115         if (side == 1 && score1 > score2) || (side == -1 && score1 < score2)
116             then (score1, move1)
117             else
118                 (score2, move2)
119
120 getBestMoveHelper :: [[Int]] -> Int -> Int -> Int -> (Int, Int) -> (Int,(Int,
Int))
121 getBestMoveHelper board side depth curr_depth move
122     | curr_depth == depth = (scoreBoard board, move)
123     | otherwise = chooseBestMove allResults side
124     where possibleMoves = getAllMoves board
125           allResults = parMap rdeepseq (parallelHelper side board depth
curr_depth) possibleMoves
126
127
128 chooseBestMove :: [(Int,(Int, Int))] -> Int -> (Int,(Int, Int))
129 chooseBestMove [] side | side == 1 = (-100000, (10000,10000))
130                          | otherwise = (100000, (10000,10000))
131 chooseBestMove results@(x:xs) side = chooseBetterScore side nextR x
132                                   where nextR = chooseBestMove xs side
133
134 parallelHelper :: Int -> [[Int]] -> Int -> Int -> (Int,Int) -> (Int, (Int,
Int))
135 parallelHelper side board depth curr_depth move = getBestMoveHelper
movedBoard (switchSide side) depth (curr_depth + 1) move
136                                   where movedBoard = makeAMove
board move side
137
138 getBestMove :: [[Int]] -> Int -> Int -> (Int, Int)
139 getBestMove board side depth = snd (getBestMoveHelper board (switchSide side)

```

```
depth 0 (10000, 10000)
140
141
142 -- ===== testing =====
143
144
145 board1 = [
146     [0,1,1,0,0,0,0,0,0],
147     [0,0,0,0,0,0,0,0,0],
148     [0,0,0,1,1,1,0,0,0],
149     [0,0,0,0,0,0,0,0,0],
150     [0,0,0,-1,-1,0,0,0,0],
151     [0,0,0,0,0,0,0,0,0],
152     [0,0,0,0,0,0,0,0,0],
153     [0,0,0,0,0,0,0,0,0],
154     [0,0,0,0,0,0,0,0,0]
155 ]
156
157 r = getBestMove board1 1 3
```

```

1 main :: IO()
2 main = print (r)
3
4 {-
5     Make a move on the board
6     Input: board, tar_cor, side
7     Return: board
8 -}
9 makeAMove board tar_cor side = makeAMoveHelper board tar_cor 0 side
10
11 makeAMoveHelper (x:xs) (tar_x, tar_y) curr_x side
12     | tar_x == curr_x = (makeAMoveRow x tar_y side) : xs
13     | otherwise = x:(makeAMoveHelper xs (tar_x, tar_y) (curr_x+1) side)
14
15 makeAMoveRow row index side = makeAMoveRowHelper row index 0 side
16
17 makeAMoveRowHelper (x:xs) tar_index curr_index side
18     | tar_index == curr_index = side : xs
19     | otherwise = x:(makeAMoveRowHelper xs tar_index (curr_index+1) side)
20
21
22
23
24 {-
25     Score a board for one board
26     Input: board
27     Return: score (int)
28 -}
29 scoreBoard board = (scoreBoardOneSide board 1) - (scoreBoardOneSide board
30 (-1))
31
32 scoreBoardOneSide board side = (scoreBoardOneDirection board side) +
33 (scoreBoardOneDirection (flipBoard board) side)
34
35 scoreOneRow row side = scoreOneRowHelper row side 0
36
37 scoreHelper num = case num of
38     2 -> 100
39     3 -> 200
40     4 -> 1000
41     _ -> 0
42
43 scoreOneRowHelper row side num
44     | null row = scoreHelper num
45     | otherwise = let (x:xs) = row in
46         do
47             if x == side
48                 then scoreOneRowHelper xs side (num + 1)
49             else (scoreHelper num) + (scoreOneRowHelper xs side 0)

```

```

49 scoreBoardOneDirection board side = sum [scoreOneRow row side | row <- board]
50
51 flipBoard matrix
52   | null matrix = [[] | _ <- [1 .. 9]]
53   | otherwise = let (x:xs) = matrix in
54     [a:b | (a,b) <- (zip x (flipBoard xs))]
55
56
57 {-
58   Get all possible moves on the board
59   Input: board
60   Return: moves [(int,int)]
61 -}
62 getAllMoves board = getAllMovesHelper board 0
63
64 getAllMovesHelper board row_index
65   | null board = []
66   | otherwise = let (x:xs) = board in
67     (getAllMovesOneRow x row_index) ++ (getAllMovesHelper xs (row_index +
68 1))
69
70 getAllMovesOneRow row row_index = getAllMovesOneRowHelper row row_index 0
71
72 getAllMovesOneRowHelper row row_index curr_index
73   | null row = []
74   | otherwise = let (x:xs) = row in
75     do
76       if x == 0
77         then (row_index, curr_index):(getAllMovesOneRowHelper xs
78 row_index (curr_index + 1))
79         else getAllMovesOneRowHelper xs row_index (curr_index + 1)
80
81 {-
82   AI functions
83 -}
84 initializeBestScore side
85   | side == 1 = -100000
86   | otherwise = 100000
87 initializeAlphaBeta side = -1 * (initializeBestScore side)
88
89 switchSide side
90   | side == 1 = -1
91   | otherwise = 1
92
93 chooseBetterScore side (score1, move1) (score2, move2) =
94   do
95     if (side == 1 && score1 > score2) || (side == -1 && score1 < score2)

```



```

96         then (score1, move1)
97     else
98         (score2, move2)
99
100 goThroughMovesHelper moves bestScore bestMove side board depth curr_depth
alpha_beta
101     | length moves == 0 = (bestScore, bestMove)
102     | otherwise = let (x:xs) = moves in
103         let movedBoard = makeAMove board x side in
104             let (newBestScore, newBestMove) = getBestMoveHelper
movedBoard (switchSide side) depth (curr_depth + 1) bestScore in
105                 do
106                     if (side == 1 && newBestScore > alpha_beta)
|| (side == -1 && newBestScore < alpha_beta)
107                         then (newBestScore, x)
108                         else
109                             let (bestScore_, bestMove_) =
chooseBetterScore side (bestScore, bestMove) (newBestScore, x) in
110                                 goThroughMovesHelper xs bestScore_
bestMove_ side board depth curr_depth alpha_beta
111
112 getBestMoveHelper board side depth curr_depth alpha_beta
113     | curr_depth == depth = (scoreBoard board), (-1,-1)
114     | otherwise =
115         let bestScore = initializeBestScore side in
116             let bestMove = (-1,-1) in
117                 let possibleMoves = getAllMoves board in
118                     goThroughMovesHelper possibleMoves bestScore bestMove
side board depth curr_depth alpha_beta
119
120
121 getBestMove board side depth =
122     let alpha_beta = initializeAlphaBeta side in
123         let (_, bestMove) = getBestMoveHelper board side depth 0 alpha_beta
in
124         bestMove
125
126
127 -- ===== testing -- =====
128 boardEmpty = [
129     [0,0,0,0,0,0,0,0,0,0],
130     [0,0,0,0,0,0,0,0,0,0],
131     [0,0,0,0,0,0,0,0,0,0],
132     [0,0,0,0,0,0,0,0,0,0],
133     [0,0,0,0,0,0,0,0,0,0],
134     [0,0,0,0,0,0,0,0,0,0],
135     [0,0,0,0,0,0,0,0,0,0],
136     [0,0,0,0,0,0,0,0,0,0],
137     [0,0,0,0,0,0,0,0,0,0]
138 ]

```

```
139
140 board1 = [
141     [0,1,1,0,0,0,0,0,0],
142     [0,0,0,0,0,0,0,0,0],
143     [0,0,0,1,1,1,0,0,0],
144     [0,0,0,0,0,0,0,0,0],
145     [0,0,0,-1,-1,0,0,0,0],
146     [0,0,0,0,0,0,0,0,0],
147     [0,0,0,0,0,0,0,0,0],
148     [0,0,0,0,0,0,0,0,0],
149     [0,0,0,0,0,0,0,0,0]
150 ]
151
152 board2 = [
153     [1,1,1,1,1,1,1,1,1],
154     [1,1,1,1,1,1,1,1,1],
155     [1,1,1,1,1,1,1,1,1],
156     [1,1,1,0,1,1,1,1,1],
157     [1,1,1,1,1,1,1,1,1],
158     [1,1,1,1,1,1,1,1,1],
159     [1,1,1,1,1,1,1,1,1],
160     [1,1,1,1,1,1,1,0,0],
161     [0,0,0,0,0,0,0,0,0]
162 ]
163
164 board3 = [
165     [0,0,0,0,0,0,0,0,0],
166     [0,0,1,1,1,1,1,1,1],
167     [1,1,1,1,1,1,1,1,1],
168     [1,1,1,1,1,1,1,1,1],
169     [1,1,1,1,1,1,1,1,1],
170     [1,1,1,1,1,1,1,1,1],
171     [1,1,1,1,1,1,1,1,1],
172     [1,1,1,0,0,0,0,0,0],
173     [0,0,0,0,0,0,0,0,0]
174 ]
175
176 r = getBestMove board1 1 3
177 b = getBestMove board1 (-1) 3
```