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 prediciton 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$_r$*bcarlso.html*

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

```haskell
module Main where

import Control.Parallel.Strategies

main :: IO()
main = print(r)



{-
    Make a move on the board
    Input: board, tar_cor, side
    Return: board
-}
makeAMove :: [[Int]] -> (Int,Int) -> Int -> [[Int ]]
makeAMove board tar_cor side = makeAMoveHelper board tar_cor 0 side

makeAMoveHelper :: [[Int]] -> (Int, Int) -> Int -> Int -> [[Int]]
makeAMoveHelper (x:xs) (tar_x, tar_y) curr_x side
    | tar_x == curr_x = (makeAMoveRow x tar_y side) : xs
    | otherwise = x:(makeAMoveHelper xs (tar_x, tar_y) (curr_x+1) side)

makeAMoveRow :: [Int] -> Int -> Int -> [Int]
makeAMoveRow row index side = makeAMoveRowHelper row index 0 side

makeAMoveRowHelper :: [Int] -> Int -> Int -> Int -> [Int]
makeAMoveRowHelper (x:xs) tar_index curr_index side
    | tar_index == curr_index = side : xs
    | otherwise = x:(makeAMoveRowHelper xs tar_index (curr_index+1) side)




{-
    Score a board for one board
    Input: board
    Return: score (int)
-}

scoreBoard :: [[Int]] -> Int
scoreBoard board = (scoreBoardOneSide board 1) - (scoreBoardOneSide board
(-1))

scoreBoardOneSide :: [[Int]] -> Int -> Int
scoreBoardOneSide board side = (scoreBoardOneDirection board side) +
(scoreBoardOneDirection (flipBoard board) side)

scoreOneRow :: [Int] -> Int -> Int
scoreOneRow row side = scoreOneRowHelper row side 0
```

```haskell
scoreHelper :: Int -> Int
scoreHelper num = case num of
    2 -> 100
    3 -> 200
    4 -> 10000
    _ -> 0

scoreOneRowHelper :: [Int] -> Int -> Int -> Int
scoreOneRowHelper [] side num = scoreHelper num
scoreOneRowHelper row@(x:xs) side num
        | x == side = scoreOneRowHelper xs side (num + 1)
        | otherwise = (scoreHelper num) + (scoreOneRowHelper xs side 0)

scoreBoardOneDirection :: [[Int]] -> Int -> Int
scoreBoardOneDirection board side = sum [scoreOneRow row side | row <- board]

flipBoard :: [[Int]] -> [[Int]]
flipBoard matrix
    | null matrix = [[] | _ <- [1 .. 9]]
    | otherwise = let (x:xs) = matrix in
        [a:b | (a,b) <- (zip x (flipBoard xs))]


{-
    Get all possible moves on the board
    Input: board
    Return: moves [(int,int)]
-}

getAllMoves :: [[Int]] -> [(Int, Int)]
getAllMoves board = getAllMovesHelper board 0

getAllMovesHelper :: [[Int]] -> Int -> [(Int, Int)]
getAllMovesHelper board row_index
    | null board = []
    | otherwise = let (x:xs) = board in
        (getAllMovesOneRow x row_index) ++ (getAllMovesHelper xs (row_index + 1))

getAllMovesOneRow :: [Int] -> Int -> [(Int, Int)]
getAllMovesOneRow row row_index = getAllMovesOneRowHelper row row_index 0

getAllMovesOneRowHelper :: [Int] -> Int -> Int -> [(Int, Int)]
getAllMovesOneRowHelper row row_index curr_index
    | null row = []
    | otherwise = let (x:xs) = row in
        do
            if x == 0
                then (row_index, curr_index):(getAllMovesOneRowHelper xs
```

```haskell
row_index (curr_index + 1))
            else getAllMovesOneRowHelper xs row_index (curr_index + 1)


{-
    AI functions
-}

initializeBestScore :: Int -> Int
initializeBestScore side
    | side == 1 = -100000
    | otherwise = 100000

switchSide :: Int -> Int
switchSide side = -side

chooseBetterScore :: Int -> (Int,(Int, Int)) -> (Int,(Int, Int)) -> (Int,(Int, Int))
chooseBetterScore side (score1, move1) (score2, move2) =
    do
        if (side == 1 && score1 > score2) || (side == -1 && score1 < score2)
            then (score1, move1)
        else
            (score2, move2)

getBestMoveHelper :: [[Int]] -> Int -> Int -> Int -> (Int, Int) -> (Int,(Int, Int))
getBestMoveHelper board side depth curr_depth move
    | curr_depth == depth = (scoreBoard board, move)
    | otherwise = chooseBestMove allResults side
    where possibleMoves = getAllMoves board
          allResults = parMap rdeepseq (parallelHelper side board depth curr_depth) possibleMoves


chooseBestMove :: [(Int,(Int, Int))] -> Int -> (Int,(Int, Int))
chooseBestMove [] side | side == 1 = (-100000, (10000,10000))
                       | otherwise = (100000, (10000,10000))
chooseBestMove results@(x:xs) side = chooseBetterScore side nextR x
                                    where nextR = chooseBestMove xs side

parallelHelper :: Int -> [[Int]] -> Int -> Int -> (Int,Int) -> (Int, (Int, Int))
parallelHelper side board depth curr_depth move = getBestMoveHelper movedBoard (switchSide side) depth (curr_depth + 1) move
                                                where movedBoard = makeAMove board move side

getBestMove :: [[Int]] -> Int -> Int -> (Int, Int)
getBestMove board side depth = snd (getBestMoveHelper board (switchSide side)
```

```haskell
depth 0 (10000, 10000))


-- ================ testing -- ================


board1 = [
        [0,1,1,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,1,1,1,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,-1,-1,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0]
    ]

r = getBestMove board1 1 3
```

```haskell
main :: IO()
main = print (r)

{-
    Make a move on the board
    Input: board, tar_cor, side
    Return: board
-}
makeAMove board tar_cor side = makeAMoveHelper board tar_cor 0 side

makeAMoveHelper (x:xs) (tar_x, tar_y) curr_x side
    | tar_x == curr_x = (makeAMoveRow x tar_y side) : xs
    | otherwise = x:(makeAMoveHelper xs (tar_x, tar_y) (curr_x+1) side)

makeAMoveRow row index side = makeAMoveRowHelper row index 0 side

makeAMoveRowHelper (x:xs) tar_index curr_index side
    | tar_index == curr_index = side : xs
    | otherwise = x:(makeAMoveRowHelper xs tar_index (curr_index+1) side)




{-
    Score a board for one board
    Input: board
    Return: score (int)
-}
scoreBoard board = (scoreBoardOneSide board 1) - (scoreBoardOneSide board
(-1))

scoreBoardOneSide board side = (scoreBoardOneDirection board side) +
(scoreBoardOneDirection (flipBoard board) side)

scoreOneRow row side = scoreOneRowHelper row side 0

scoreHelper num = case num of
    2 -> 100
    3 -> 200
    4 -> 1000
    _ -> 0

scoreOneRowHelper row side num
    | null row = scoreHelper num
    | otherwise = let (x:xs) = row in
        do
            if x == side
                then scoreOneRowHelper xs side (num + 1)
            else (scoreHelper num) + (scoreOneRowHelper xs side 0)
```

```haskell
scoreBoardOneDirection board side = sum [scoreOneRow row side | row <- board]

flipBoard matrix
    | null matrix = [[] | _ <- [1 .. 9]]
    | otherwise = let (x:xs) = matrix in
        [a:b | (a,b) <- (zip x (flipBoard xs))]



{-
    Get all possible moves on the board
    Input: board
    Return: moves [(int,int)]
-}
getAllMoves board = getAllMovesHelper board 0

getAllMovesHelper board row_index
    | null board = []
    | otherwise = let (x:xs) = board in
        (getAllMovesOneRow x row_index) ++ (getAllMovesHelper xs (row_index + 1))

getAllMovesOneRow row row_index = getAllMovesOneRowHelper row row_index 0

getAllMovesOneRowHelper row row_index curr_index
    | null row = []
    | otherwise = let (x:xs) = row in
        do
            if x == 0
                then (row_index, curr_index):(getAllMovesOneRowHelper xs row_index (curr_index + 1))
            else getAllMovesOneRowHelper xs row_index (curr_index + 1)


{-
    AI functions
-}

initializeBestScore side
    | side == 1 = -100000
    | otherwise = 100000
initializeAlphaBeta side = -1 * (initializeBestScore side)

switchSide side
    | side == 1 = -1
    | otherwise = 1

chooseBetterScore side (score1, move1) (score2, move2) =
    do
        if (side == 1 && score1 > score2) || (side == -1 && score1 < score2)
```

```haskell
            then (score1, move1)
        else
            (score2, move2)

goThroughMovesHelper moves bestScore bestMove side board depth curr_depth alpha_beta
    | length moves == 0 = (bestScore, bestMove)
    | otherwise = let (x:xs) = moves in
                   let movedBoard = makeAMove board x side in
                       let (newBestScore, newBestMove) = getBestMoveHelper movedBoard (switchSide side) depth (curr_depth + 1) bestScore in
                           do
                               if (side == 1 && newBestScore > alpha_beta) || (side == -1 && newBestScore < alpha_beta)
                                   then (newBestScore, x)
                               else
                                   let (bestScore_, bestMove_) = chooseBetterScore side (bestScore, bestMove) (newBestScore, x) in
                                       goThroughMovesHelper xs bestScore_ bestMove_ side board depth curr_depth alpha_beta

getBestMoveHelper board side depth curr_depth alpha_beta
    | curr_depth == depth = ((scoreBoard board), (-1,-1))
    | otherwise =
        let bestScore = initializeBestScore side in
            let bestMove = (-1,-1) in
                let possibleMoves = getAllMoves board in
                    goThroughMovesHelper possibleMoves bestScore bestMove side board depth curr_depth alpha_beta


getBestMove board side depth =
    let alpha_beta = initializeAlphaBeta side in
        let (_, bestMove) = getBestMoveHelper board side depth 0 alpha_beta in
            bestMove


-- ================ testing -- ================
boardEmpty = [
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0]
    ]
```

```haskell

board1 = [
        [0,1,1,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,1,1,1,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,-1,-1,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0]
    ]

board2 = [
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,0,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,0,0],
        [0,0,0,0,0,0,0,0,0]
    ]

board3 = [
        [0,0,0,0,0,0,0,0,0],
        [0,0,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1],
        [1,1,1,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0]
    ]

r = getBestMove board1 1 3
b = getBestMove board1 (-1) 3
```