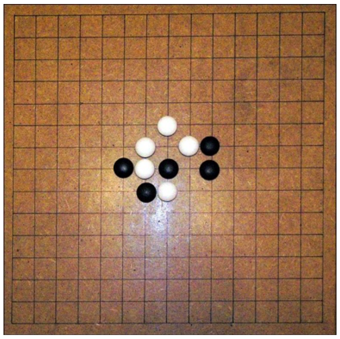
Gomoku in Haskell

Danny Hou; UNI: dh3034

1. Introduction:

Gomoku, also known as Five in a Row, is an abstract strategy board game. It is traditionally played with Go pieces (black and white stones) on a Go board. It is played using a 15 x 15 board while in the past 19 X 19 board was standard. Because pieces are typically not moved or removed from the board, Gomoku may also be played as a paper-and-pencil-game. It is a generalized version of Tic-Tac-Toe. In my project, I have created relevant data structures to implement the Gomoku using Haskell. I used minimax algorithm with alpha-beta pruning in sequential and tested the performance. I also implemented the algorithm again in parallel and achieve some improvement in performance.



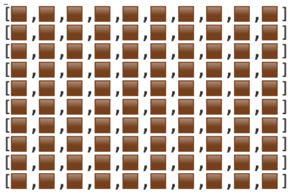
2. Game Rules

There are two players of the game, who own either white color stones or black color stones. Players can place their stones on empty intersections of the 8 x 8 board, represented by (row, column). Usually, player who owns the white stones are the first one to start the game. When one player has placed a serial chain of white stones or black stones, that player wins the game.

3. Implementation & Performance

3a. Prepare the data structures

In `Board.hs` I defined the board with board dimension and color of the stones. The board can be set to any dimension. An empty board will look like this:



```
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
[🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫,🟫]
```

Stone is defined in `Point.hs` and it consists of color and position on the board. I used 'generateMove' function for the Ai's to place the stones on the board and after each turn, the function 'isOver' will check whether there is a winner of the game.

The key algorithm here is minimax algorithm with alpha-beta pruning. We first generate a tree of boards with all possible moves with depth of 3. We compare the score of each board. Since we are trying to get 5 stones with same color in a line, we calculate the scores based on how many stones of same color we can have in a line. If there are 5 stones of same color in a line, score will be 100000; if there are 4 of such stones, score will be 5000; if there are 3 of such stones, score will be 300; and if there are only two of such stones, score will be 10.

After the sequential implementation, I used `parMap` and `rdeepseq` when running miniax algorithm with alpha-beta pruning on child boards in the board tree to evaluate all the child boards in parallel. I also used `parMap` and `rdeepseq` when calculate the score of the board from all directions when a stone is placed on the board.

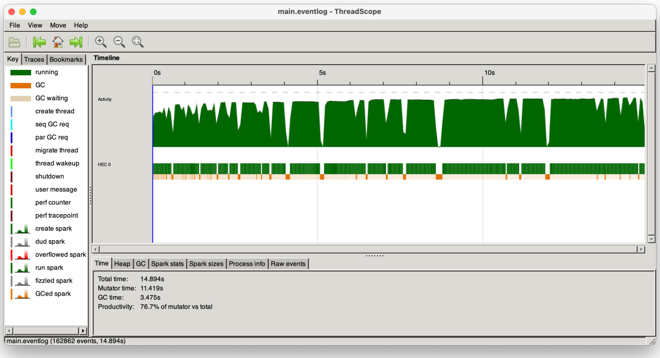
3b. Player vs AI mode

There is also an experimental player vs AI mode. It currently has some problem of judging the winner of the game. In the future I would like to fix the problems and make it works.

3c. Performance

Sequential:

15.04s on average

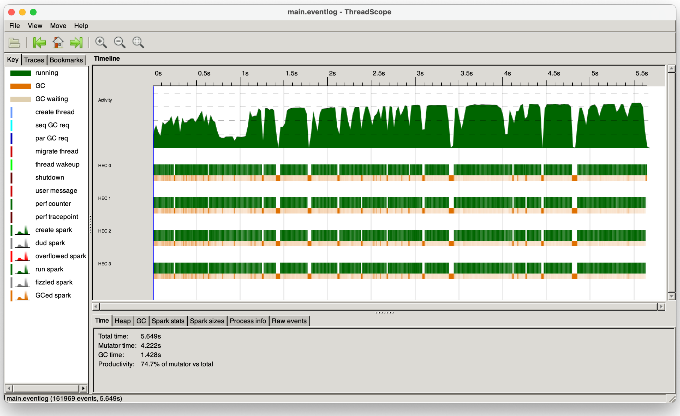


```
Total time:     14.894s
Mutator time:   11.419s
GC time:        3.475s
Productivity:   76.7% of mutator vs total
```

main.eventlog (162862 events, 14.894s)

Parallel:

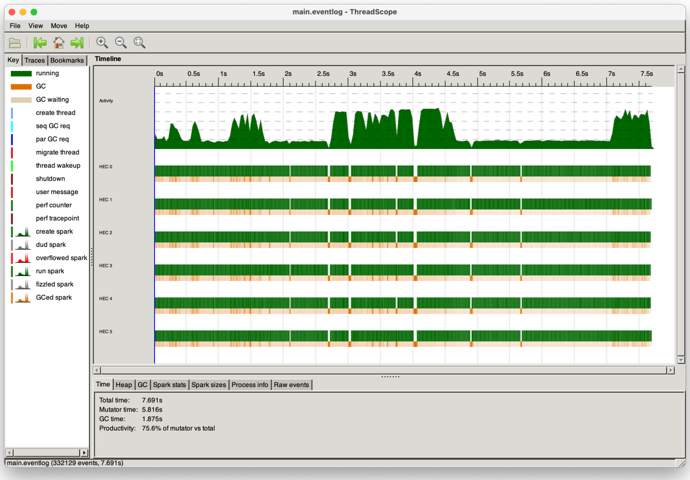
4-core: 6.23s on average



```
Total time:     5.649s
Mutator time:   4.222s
GC time:        1.428s
Productivity:   74.7% of mutator vs total
```

main.eventlog (161969 events, 5.649s)

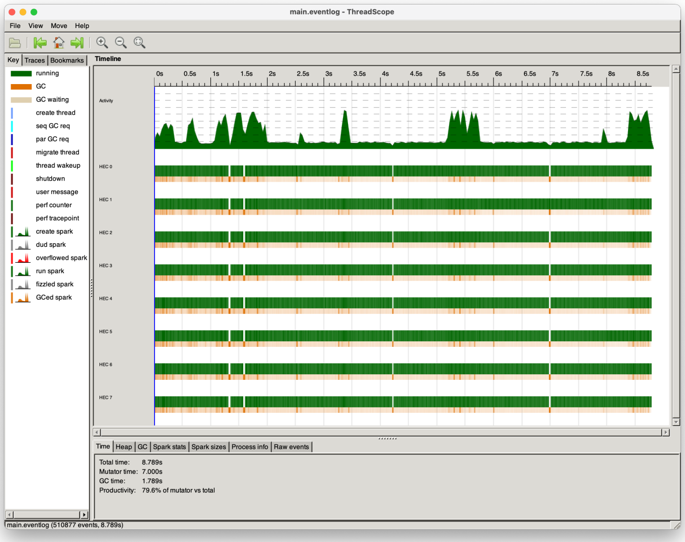
6-core: 8.18s on average



```
Total time:     7.691s
Mutator time:   5.816s
GC time:        1.875s
Productivity:   75.6% of mutator vs total
```

main.eventlog (332129 events, 7.691s)

8-core: 7.198s on average



```
Total time:     8.789s
Mutator time:   7.000s
GC time:        1.789s
Productivity:   79.6% of mutator vs total
```

main.eventlog (510877 events, 8.789s)

4. Code

Main.hs

```haskell
module Main (main) where

import AI
import Board
import Data.Char
import System.IO

gameLoopAI :: Board -> Color -> IO ()
gameLoopAI board color
    | isOver curBoard == True = do
        putStrLn (show color ++ "'s turn.")
        printBoard curBoard
        putStrLn (show color ++ " wins!")
    | otherwise = do
        putStrLn (show color ++ "'s turn.")
        printBoard curBoard
        gameLoopAI curBoard (oppositeColor color)
    where
        curBoard = generateMove board color


playerLoop :: Board -> Color -> IO ()
playerLoop board color
    | color == Black = do
        x <- prompt "Enter row: "
        y <- prompt "Enter col: "
        let playerBoard = addPointToBoard (Point color (read x :: Int, read y :: Int)) board
        printBoard playerBoard
        if isOver playerBoard then putStrLn (show color ++ " wins!") else playerLoop playerBoard (oppositeColor color)
    | otherwise = do
        putStrLn (show color ++ "'s turn.")
        let curBoard = generateMove board color
        printBoard curBoard
        if isOver curBoard then putStrLn (show color ++ " wins!") else playerLoop curBoard (oppositeColor color)

prompt :: String -> IO String
prompt text = do
    putStr text
    hFlush stdout
    getLine
```

```
main :: IO ()
main = gameLoopAI (initBoard 8 8) Black
-- main = do
--     let board = initBoard 10 10
--     printBoard board
--     playerLoop board Black
```

Point.hs

```
module Point
  ( Point(..)
  ) where

import Color

data Point =
  Point
    { color    :: Color
    , position :: (Int, Int)
    }

instance Show Point where
  show (Point color _) = show color

instance Eq Point where
  (Point color1 (x1, y1)) == (Point color2 (x2, y2)) = x1 == x2 && y1 == y2 && color1
== color2

instance Ord Point where
  compare (Point _ (x1,y1)) (Point _ (x2,y2)) = compare (x1*10+y1) (x2*10+y2)
```

Color.hs

```
module Color
  ( Color(..)
  ) where

data Color = White | Black | Empty deriving (Eq)


instance Show Color where
  show Black = "⚫"
  show White = "⚪"
```

```
  show Empty = "🟫"
```

Board.hs

```
module Board
  ( Color(..)
  , Point(..)
  , Board(..)
  , initBoard
  , initCol
  , getPoint
  , isEmpty
  , isValid
  , addPointToBoard
  , addPoint
  , filterBoard
  , oppositeColor
  , isEmptyBoard
  , isOver
  , getCurPoint
  , printBoard
  , diagonals
  ) where

import Data.List
import Point
import Color

data Board = Board{row::Int, col::Int, points::[[Point]]}

instance Show Board where
  show (Board _ _ points) = intercalate "\n" $ map show points

instance Eq Board where
  (Board r1 c1 points1) == (Board r2 c2 points2) = (r1 == r2 && c1 == c2 && points1 == points2)

initBoard :: Int -> Int -> Board
initBoard row col = Board row col points
    where
        points = [initCol x col | x <- [1..row]]

initCol :: Int -> Int -> [Point]
initCol row col = if col > 0 then (initCol row (col - 1)) ++ [Point Empty (row,col)] else []
```

```
printBoard :: Board -> IO()
printBoard board = putStrLn $ show board

-- get point from a position
getPoint:: Board -> (Int,Int) -> Point
getPoint (Board _ _ points) (x,y) = (points !! (x - 1)) !! (y - 1)

-- check if a position is empty
isEmpty:: Point -> Board -> Bool
isEmpty (Point _ (x,y)) (Board row col points) = color == Empty
    where
        (Point color (_, _)) = getPoint (Board row col points) (x, y)

-- check if the position we choose is in the board
isValid:: Point -> Board -> Bool
isValid (Point _ (x,y)) (Board row col _) = if (x > 0 && x <= row && y > 0 && y <=
col) then True else False

addPointToBoard::Point -> Board -> Board
addPointToBoard (Point color (x,y)) (Board row col points)
    | (isValid (Point color (x,y)) (Board row col points) && isEmpty (Point color
(x,y)) (Board row col points) ) =
        addPoint (Point color (x,y)) (Board row col points)
    | otherwise = (Board row col points)

addPoint :: Point -> Board -> Board
addPoint (Point color (x,y)) (Board row col points) = Board row col newPoints
    where
        newPoints = upperRows ++ (leftCells ++ (Point color (x, y) : rightCells)) :
lowerRows
        (upperRows, thisRow:lowerRows) = splitAt (x - 1) points
        (leftCells, _:rightCells) = splitAt (y - 1) thisRow


checkRow :: [Point] -> Color -> Int -> Int
checkRow [] prevColor count
    | count == 5 && prevColor == Black = 1
    | count == 5 && prevColor == White = 2
    | otherwise = 0
checkRow (x:xs) prevColor count
    | prevColor == Empty = checkRow xs color 1
    | prevColor == color && count == 4 =
        if color == Black
        then 1
        else 2
    | prevColor == color && count < 4 = checkRow xs color (count + 1)
    | otherwise = 0
    where
```

```
        (Point color _) = x

diagonals :: [[a]] -> [[a]]
diagonals = tail . go [] where
    go b es_ = [h | h:_ <- b] : case es_ of
        []   -> transpose ts
        e:es -> go (e:ts) es
        where ts = [t | _:t <- b]

isOver::Board -> Bool
isOver (Board a b points) =
    if (sum [(checkRow  (points !! x) Empty 1) | x <- [0..b-1] ]/= 0) then True else
        if (sum [(checkRow (( (transpose . reverse) points) !! x) Empty 1) | x <-
[0..a-1] ]/= 0) then True else
            if (sum [(checkRow ((diagonals points) !! x) Empty 1) | x <- [0..b-1] ] /=
0) then True else
                if (sum [(checkRow ((diagonals ( (transpose . reverse) points)) !! x)
Empty 1) | x <- [0..a-1] ] /= 0) then True else False


filterBoard :: Board -> Color -> [Point]
filterBoard (Board _ _ points) color =
  [point | rows <- points, point <- rows, isSameColor point]
  where
    isSameColor (Point c (_,_)) = c == color

oppositeColor :: Color -> Color
oppositeColor color
  | color == White = Black
  | color == Black = White
  | otherwise = error "Invalid opposite color"

isEmptyBoard :: Board -> Bool
isEmptyBoard (Board row col points) = Board row col points == initBoard row col

flatten :: [[a]] -> [a]
flatten xs = (\z n -> foldr (flip (foldr z)) n xs) (:) []

getCurPoint :: Board -> Board -> [Point]
getCurPoint (Board _ _ points1) (Board _ _ points2) = flatten points2 \\ flatten
points1
```

AI.hs

```
module AI where

import           Board
```

```haskell
import           Control.Parallel.Strategies
import           Data.List
import           Data.Maybe
import qualified Data.Set                   as Set
import           Data.Tree

minInt :: Int
minInt = -(2 ^ 29)

maxInt :: Int
maxInt = 2 ^ 29 - 1

generateMove :: Board -> Color -> Board
generateMove board color
  | isEmptyBoard board = addPointToBoard (Point color ((row board) `div` 2, (col board) `div` 2)) board
  -- | isEmptyBoard board = addPointToBoard (Point color (1,1)) board
  | otherwise = bestMove
  where
    neighbors = nextMoves board
    (Node node children) = buildTree color board neighbors
    minmax = parMap rdeepseq (minBeta color 3 minInt maxInt) children
    -- minmax = map (minBeta color 3 minInt maxInt) children
    index = fromJust $ elemIndex (maximum minmax) minmax
    (Node bestMove _) = children !! index


-- generate possible moves for the player
nextMoves :: Board -> [Point]
nextMoves board = Set.toList $ stepBoard board $ filterBoard board White ++ filterBoard board Black

stepBoard :: Board -> [Point] -> Set.Set Point
stepBoard _ [] = Set.empty
stepBoard board (point:rest) = Set.union (Set.fromList (stepFromPoint board point)) $ stepBoard board rest

stepFromPoint :: Board -> Point -> [Point]
stepFromPoint board (Point _ (x, y)) =
  [ Point Empty (x + xDir, y + yDir)
  | xDir <- [-1 .. 1]
  , yDir <- [-1 .. 1]
  , not (xDir == 0 && yDir == 0)
  , isValid (Point Empty (x + xDir, y + yDir)) board
  , isEmpty (Point Empty (x + xDir, y + yDir)) board
  ]

buildTree :: Color -> Board -> [Point] -> Tree Board
```

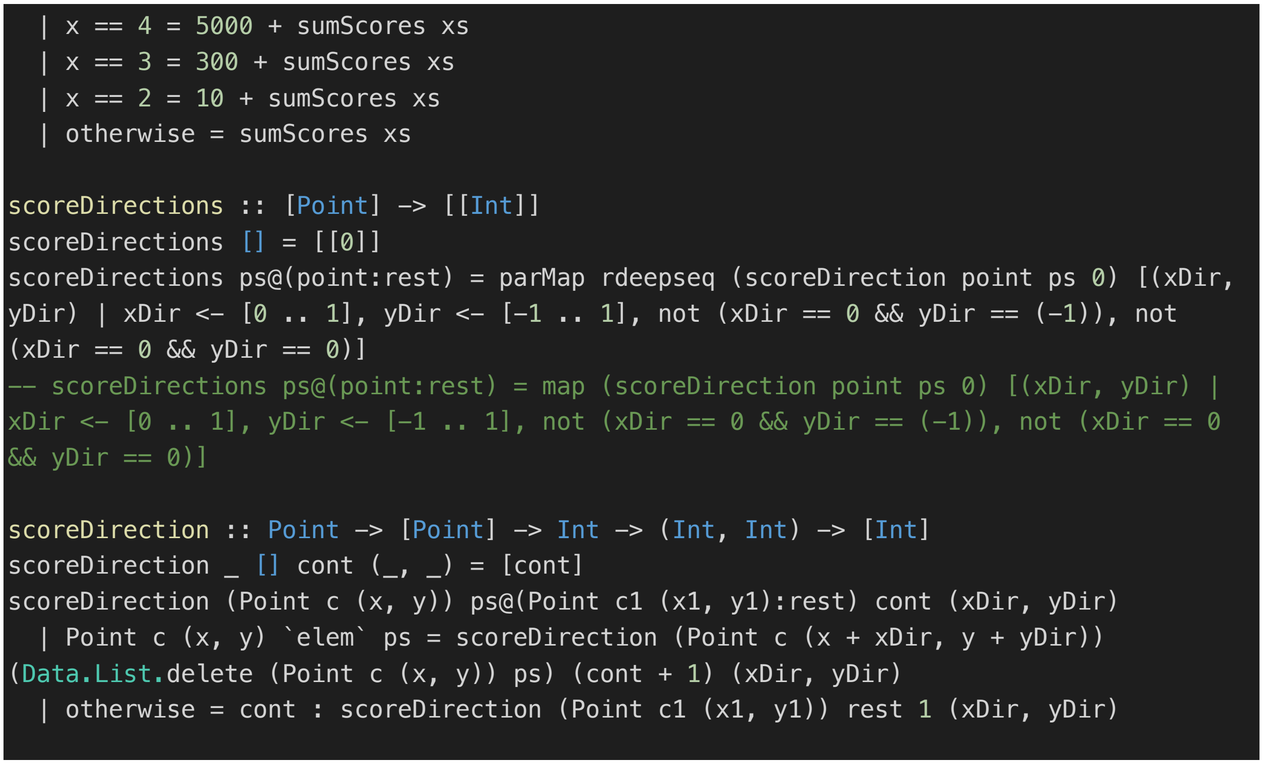
```
buildTree color board neighbors = Node board $ children neighbors
  where
    newNeighbors point =
      Set.toList $
      Set.union (Set.fromList (Data.List.delete point neighbors)) (Set.fromList
(stepFromPoint board point))
    oppoColor = oppositeColor color
    children [] = []
    children (Point c (x, y):ns) =
      buildTree oppoColor (addPointToBoard (Point color (x,y)) board) (newNeighbors
(Point c (x, y))) : children ns

maxAlpha :: Color -> Int -> Int -> Int -> Tree Board -> Int
maxAlpha _ _ alpha _ (Node _ []) = alpha
maxAlpha color level alpha beta (Node b (x:xs))
  | level == 0 = curScore
  | canFinish curScore = curScore
  | newAlpha >= beta = beta
  | otherwise = maxAlpha color level newAlpha beta (Node b xs)
  where
    curScore = scoreBoard b color
    canFinish score = score > 100000 || score < (-100000)
    newAlpha = maximum [alpha, minBeta color (level - 1) alpha beta x]

minBeta :: Color -> Int -> Int -> Int -> Tree Board -> Int
minBeta _ _ _ beta (Node _ []) = beta
minBeta color level alpha beta (Node b (x:xs))
  | level == 0 = curScore
  | canFinish curScore = curScore
  | alpha >= newBeta = alpha
  | otherwise = minBeta color level alpha newBeta (Node b xs)
  where
    curScore = scoreBoard b color
    canFinish score = score > 100000 || score < (-100000)
    newBeta = minimum [beta, maxAlpha color (level - 1) alpha beta x]

scoreBoard :: Board -> Color -> Int
scoreBoard board color = score (pointsOfColor color) - score (pointsOfColor
$ oppositeColor color)
  where
    -- score points = sum $ map sumScores $ scoreDirections points
    score points = sum $ parMap rdeepseq sumScores $ scoreDirections points
    pointsOfColor = filterBoard board

sumScores :: [Int] -> Int
sumScores [] = 0
sumScores (x:xs)
  | x == 5 = 100000 + sumScores xs
```



```
  | x == 4 = 5000 + sumScores xs
  | x == 3 = 300 + sumScores xs
  | x == 2 = 10 + sumScores xs
  | otherwise = sumScores xs

scoreDirections :: [Point] -> [[Int]]
scoreDirections [] = [[0]]
scoreDirections ps@(point:rest) = parMap rdeepseq (scoreDirection point ps 0) [(xDir,
yDir) | xDir <- [0 .. 1], yDir <- [-1 .. 1], not (xDir == 0 && yDir == (-1)), not
(xDir == 0 && yDir == 0)]
-- scoreDirections ps@(point:rest) = map (scoreDirection point ps 0) [(xDir, yDir) |
xDir <- [0 .. 1], yDir <- [-1 .. 1], not (xDir == 0 && yDir == (-1)), not (xDir == 0
&& yDir == 0)]

scoreDirection :: Point -> [Point] -> Int -> (Int, Int) -> [Int]
scoreDirection _ [] cont (_, _) = [cont]
scoreDirection (Point c (x, y)) ps@(Point c1 (x1, y1):rest) cont (xDir, yDir)
  | Point c (x, y) `elem` ps = scoreDirection (Point c (x + xDir, y + yDir))
(Data.List.delete (Point c (x, y)) ps) (cont + 1) (xDir, yDir)
  | otherwise = cont : scoreDirection (Point c1 (x1, y1)) rest 1 (xDir, yDir)
```

Reference:

https://www.youtube.com/watch?v=l-hh51ncgDI&ab_channel=SebastianLague

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/?ref=rp

https://github.com/sowakarol/gomoku-haskell

http://www.cs.columbia.edu/~sedwards/classes/2021/4995-fall/reports/Gomoku.pdf

http://www.cs.columbia.edu/~sedwards/classes/2021/4995-fall/reports/Gomokururu.pdf

http://www.cs.columbia.edu/~sedwards/classes/2019/4995-fall/reports/gomoku.pdf