Gomoku Game in Haskell

Qinhan Zhou (qz2380)
Zheng Yao (zy2388)

Contents

1 Introduction

We implement an AI v.s. AI `Gomoku` game, also called Five in a Row. Here it's played on a board with size 10*10. Black and white players alternate turns to place a stone of their color on an empty intersection.
We use `minimax` search algorithm to depth of three employing `alpha-beta` cut-off strategy to address the two players playing against each other.

2 Implementation

There are three files: Board.hs, AI.hs and Main.hs.

2.1 Board.hs

Define a 10*10 Board and each player's move can denote by a Point. Each point has its color(Black or White) and its corresponding position (Int, Int).
Key methods are `addPoint` and `checkWin`. After each move, we check four directions whether there is already five same color points in a row. There is no need to check all the boards, just four lines those includes the latest point.

2.2 AI.hs

Key method here is where to put the current point to realize a game of competition. We use `Minimax` algorithm to alternate between the two AI players, the player desires to pick the move with the maximum score. In turn, the scores for each of available moves are determined by the opposing player which of its available moves has the minimum score. Scores are calculated as: score 100000 when 5 in a row; 10000 with 4 in a row; 1000 with 3 in a row and 100 with 2 in a row. Build a tree of depth 3 to compare all possible next three moves and pick the most favorable one for current move.
Then we improve `Minimax` by `alpha-beta` cut-off. Each node has a boundary [alpha, beta]. `alpha` means lower boundary and `beta` means the upper boundary. Each time when `beta` $\leq$ `alpha`, we no longer search more sub-trees, This is a process of pruning. We refer to other's codes when we implement the `minmaxAlpha` and `minmaxBeta` methods but we modified it to fit our data structures.
We used parallel strategies in two places. First, we run the minmax algorithm on each child board of the current board in the board tree in parallel and choose the one with highest score as the next move. Second, to rate each board, we use parallel to get the score of current board in each possible directions. In both cases, we use `parMap rdeepseq` as our parallel strategy.
Some steps of alternatively running two AIs on a 10*10 board is shown in the table below. `O` and `X` denotes the two users and _ denote a vacant place in board.

```
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,0,_,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,0,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,X,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,X,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,_,_,_,_,_]
[_,_,0,_,X,_,_,_,_,_]
[_,_,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,X,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,_,_,_,_,_]
[_,_,0,_,X,_,_,_,_,_]
[_,X,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,X,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,0,_,_,_,_]
[_,_,0,_,X,_,_,_,_,_]
[_,X,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,X,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,0,_,_,_,_]
[_,_,0,_,X,_,X,_,_,_]
[_,X,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,0,_,X,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,0,X,0,_,_,_,_]
[_,_,0,_,X,_,X,_,_,_]
[_,X,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,_,0,_,X,_,_,_,_]
[_,_,_,X,0,_,_,_,_,_]
[_,_,_,0,X,0,_,_,_,_]
[_,_,0,_,X,_,X,_,_,_]
[_,X,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,_,_,_,_,_,_,_,_]
[_,_,0,0,_,X,_,_,_,_]
[_,_,_,X,0,_,_,_,_,_]
[_,_,_,0,X,0,_,_,_,_]
[_,_,0,_,X,_,X,_,_,_]
[_,X,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

[_,_,X,_,_,_,_,_,_,_]
[_,_,0,0,_,X,_,_,_,_]
[_,_,_,X,0,_,_,_,_,_]
[_,_,_,0,X,0,_,_,_,_]
[_,_,0,_,X,_,X,_,_,_]
[_,X,_,_,X,_,_,_,_,_]
[_,_,_,_,0,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,_,_,_]

......
```

2.3 Main.hs

Method `gameLoop` turns on the game by repeatedly calling method `moveAI` until method `checkWin` no longer returns `Empty`, i.e. one player wins.

3 Performance

We compare performance on one core and that on four cores. We find that running time on one core is 1.44s and 0.94 on four cores.

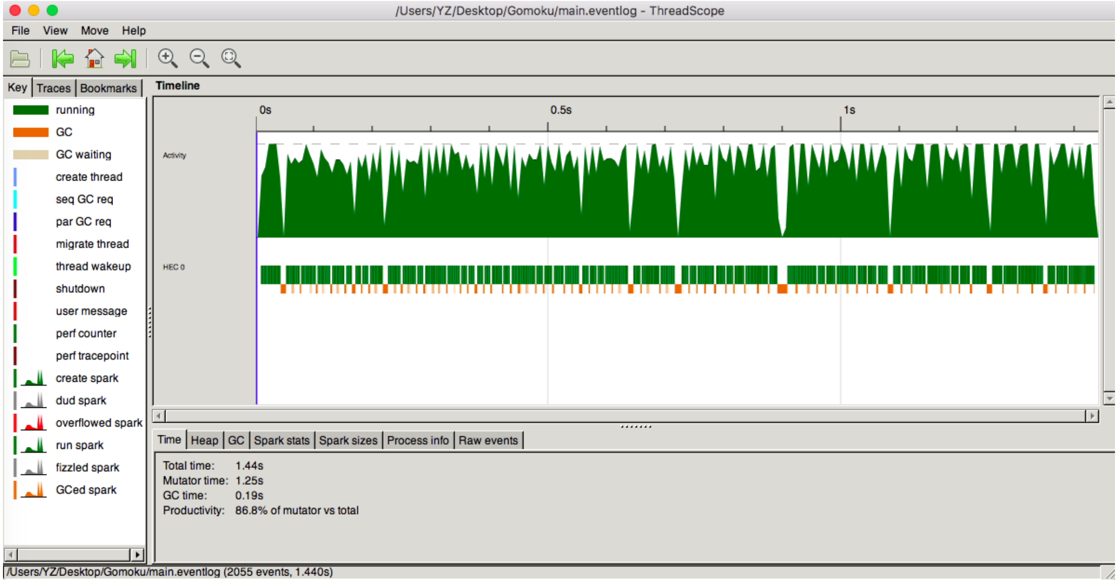

Figure 1: Running on one core

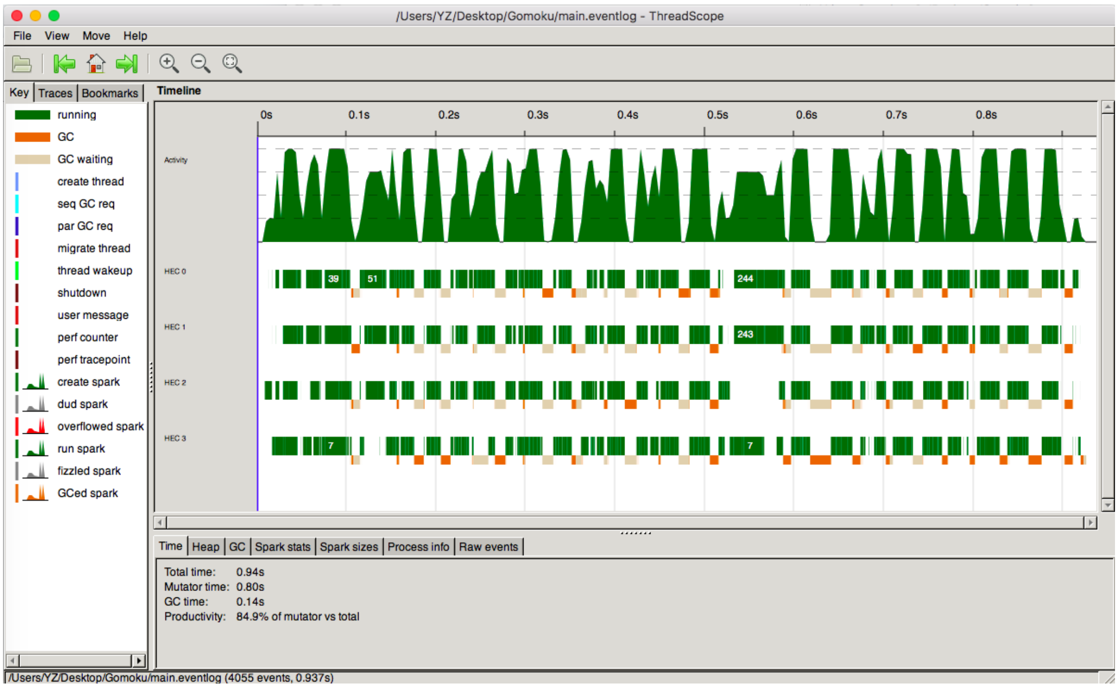

Figure 2: Running on four cores

4 Codes

4.1 Main.hs

```haskell
module Main where

import            AI
import            Board
import            Data.Char

gameLoop :: Board -> Color -> [Board] -> IO ()
gameLoop board color list
  | null curPoint = putStrLn "Tie"
  | checkWin (head curPoint) curBoard == Empty = do
    putStrLn (show curBoard ++ "\n")
    gameLoop curBoard (oppositeColor color) (list ++ [curBoard])
  | otherwise = do
    putStrLn (show curBoard ++ "\n")
    putStrLn (show color ++ "wins")
  where
    curPoint = getCurPoint board curBoard
    curBoard = moveAI board color

main :: IO ()
main = gameLoop initBoard Black []
```

Listing 1: Main.hs

4.2 Board.hs

```haskell
module Board
  ( Color(..)
  , Point(..)
  , Board(..)
  , initBoard
  , oppositeColor
  , filterBoard
  , isEmptyBoard
  , addPoint
  , isValidPoint
  , isVacant
  , checkWin
  , getCurPoint
  ) where

import Data.List

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

```haskell

instance Show Color where
  show Black = "X"
  show White = "O"
  show Empty = "_"

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)

newtype Board = Board [[Point]]

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

instance Eq Board where
  (Board points1) == (Board points2) = points1 == points2

initBoard :: Board
initBoard = Board points
  where
    points = [initRow x 10 | x <- [1 .. 10]]
    initRow _ 0 = []
    initRow row col = initRow row (col - 1) ++ [Point Empty (row, col)]

getPoint :: Board -> (Int, Int) -> Point
getPoint (Board points) (x, y) = (points !! (x - 1)) !! (y - 1)

isValidPoint :: Point -> Bool
isValidPoint (Point _ (x, y))
  | x > 0 && x <= 10 && y > 0 && y <= 10 = True
  | otherwise = False

isVacant :: Point -> Board -> Bool
isVacant (Point color (x, y)) (Board points) = curColor == Empty
  where
    (Point curColor (_, _)) = getPoint (Board points) (x, y)

addPoint :: Board -> Color -> Int -> Int -> Board
addPoint (Board points) color x y
```

```haskell
  | isValidPoint (Point color (x, y)) && isVacant (Point color (x, y)) (Board points) =
    add (Point color (x, y)) (Board points)
  | otherwise = Board points

add :: Point -> Board -> Board
add (Point color (newx, newy)) (Board points) = Board newPoints
  where
    newPoints = upperRows ++ (leftCells ++ (Point color (newx, newy) : rightCells)) : lowerRows
    (upperRows, thisRow:lowerRows) = splitAt (newx - 1) points
    (leftCells, _:rightCells) = splitAt (newy - 1) thisRow

checkWin :: Point -> Board -> Color
checkWin (Point color (x, y)) (Board points)
  | winRow (Point color (x, y)) (Board points) /= 0 ||
      (winCol (Point color (x, y)) (Board points) /= 0) ||
      (winDiag (Point color (x, y)) (Board points) /= 0) || (winAntiDiag (Point color (x, y)) (Board points) /= 0) =
    color
  | otherwise = Empty

checkRow :: [Point] -> Color -> Int -> Int
checkRow [] preColor cnt =
  if cnt == 5
    then if preColor == Black
           then 1
           else 2
    else 0
checkRow (head:xs) preColor cnt
  | preColor == Empty = checkRow xs color 1
  | preColor == color && cnt < 4 = checkRow xs color (cnt + 1)
  | preColor == color && cnt == 4 =
    if color == Black
      then 1
      else 2
  | otherwise = 0
  where
    (Point color _) = head

getDiag :: Board -> Board
getDiag (Board points) = Board $ diagonals points

getAntiDiag :: Board -> Board
getAntiDiag (Board points) = Board $ diagonals ((transpose . reverse) points)

diagonals :: [[a]] -> [[a]]
diagonals = tail . go []
  where
    go b es_ =
      [h | h:_ <- b] :
      case es_ of
```

```haskell
        []   -> transpose ts
        e:es -> go (e : ts) es
      where
        ts = [t | _:t <- b]

winRow :: Point -> Board -> Int
winRow (Point color (x, y)) (Board points) = checkRow (newPoints !! (x - 1)) Empty 1
  where
    Board newPoints = addPoint (Board points) color x y

winCol :: Point -> Board -> Int
winCol (Point color (x, y)) (Board points) = checkRow ((transpose . reverse) newPoints !! (y - 1)) Empty 1
  where
    Board newPoints = addPoint (Board points) color x y

winDiag :: Point -> Board -> Int
winDiag (Point color (x, y)) (Board points) = checkRow (diagonals newPoints !! (x + y - 2)) Empty 1
  where
    Board newPoints = addPoint (Board points) color x y

winAntiDiag :: Point -> Board -> Int
winAntiDiag (Point color (x, y)) (Board points) =
  checkRow (diagonals ((transpose . reverse) newPoints) !! (9 - x + y)) Empty 1
  where
    Board newPoints = addPoint (Board points) color x y

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

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

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

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
```

Listing 2: Board.hs

4.3 AI.hs

```haskell
module AI
  ( moveAI
  ) where

import           Board
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

moveAI :: Board -> Color -> Board
moveAI board color
  | isEmptyBoard board = addPoint board color 1 1
  | otherwise = bestMove
  where
    neighbors = possibleMoves board
    (Node node children) = buildTree color board neighbors
    minmax = parMap rdeepseq (minmaxBeta color 3 minInt maxInt) children
    index = fromJust $ elemIndex (maximum minmax) minmax
    (Node bestMove _) = children !! index

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 (addPoint board color x y) (newNeighbors (Point
     c (x, y))) : children ns

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

```haskell
    newAlpha = maximum [alpha, minmaxBeta color (level - 1) alpha beta x]

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

sumScores :: [Int] -> Int
sumScores [] = 0
sumScores (x:xs)
  | x == 5 = 100000 + sumScores xs
  | x == 4 = 10000 + sumScores xs
  | x == 3 = 1000 + sumScores xs
  | x == 2 = 100 + 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)]

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)

possibleMoves :: Board -> [Point]
possibleMoves board = Set.toList $ stepBoard board $ filterBoard board White ++ filterBoard board Black
```

```haskell
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)
  , isValidPoint (Point Empty (x + xDir, y + yDir))
  , isVacant (Point Empty (x + xDir, y + yDir)) board
  ]
```

Listing 3: AI.hs

5 Reference

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

2. `https://github.com/lihongxun945/myblog/issues/14`