Fall 2021

COMS W4995: Parallel Functional Programming

Project Report

Gomokururu: A Go-Moku Solver

Kaiwen Xue, Andreas Cheng

{kx2154, hc3142}@columbia.edu

December 22, 2021

1 Abstract

While it is known that Go-Moku can be won by exhaustively searching the possible outcomes[1], such solution has a huge search space, making normal computers takes ages to perform global searching. Normally, to develop a Go-Moku AI, the search tree must be limited to a certain level. Therefore, as a computational-intensive problem, it is viable to parallel the operations in the tree search. In Gomokururu, we implemented a Go-Moku AI in haskell and applied and tested parallelism in various configurations. We found that parallelizing tree search can make minimax more efficient by tweaking various parameters of haskell's parallelism library.

2 Introduction

Go-Moku (or Five in a Row) is a board game where two players take turn to place black and white piece on a board until the board is full or a player completes a five-in-a-row. The game board is formed by $N \times N$ horizontal lines. N is traditionally 15 or 19, where N=19 is an older standard that people Go-Moku on a Go Board. Players should place piece on the intersection of these lines. The player assigned to the black piece plays first. The goal of the players is to form a five-in-a-row, which could be vertical, horizontal, or diagonal.

Minimax algorithm is based on a DFS tree search. Each tree node represents a possible state of the game. Tree leaves can be evaluated using a heuristic functions. Each layer of the tree will attempt to maximize and minimize the final outcome alternatively. Minimax is often used in game decisions where there are 1) opponents, and 2) the heuristic function can reflect the possibility of winning the game. When the tree search is sequential, alpha-beta pruning can cut the branches to prevent evaluating unnecessary game states.

3 Design and Implementation

The codes listed at the end of this report are based on our favored setting, which will be discussed in Experiment Section. Nevertheless, we have tested different settings to compare the performance of different kinds of parallelism. The full code and instructions to reproduce our results can be found at our public GitHub repository: `https://github.com/KevinRSX/gomokururu`.

3.1 Data Structures

We need Go pieces to play Go-moku. Our implementation names it as `Piece`. It represents the state of a grid on the board, so a `Piece` can be black, white, or empty.

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

With the states of grids, we can create a Go-Moku board. The `Board` data structure contains a dimension `dim`, representing how many rows and columns it has, as well as a 2D vector that stores `dim * dim` grids. Note that we use the `Data.Vector` package to create this 2D vector. The reason is that `Vector` can ensure random access time of $O(1)$, making it much master to query.

```
data Board = Board { dim :: Int
                   , getBoard :: Vector (Vector Piece) }
                   deriving (Show)
```

All our codes are based on these two structures.

3.2 Playing the Game

Our implementation follows the general rules in Go-moku. Players take turn to place a piece until the board is full or a player completes a five-in-a-row. The first case (A full board) is a tie; and that player in the latter case is a winner. One interesting part in our implementation is our winning checking function. **We designed a way that could possibly reduce the the complexity of the function. Instead of searching five-in-a-row in the whole board, we simply search it based on the last placed piece.**

Our winning checking function takes three arguments: row, col, and board. row and col are for representing the last placed piece on the board.

With this, we can check the winning status by simply checking the "star lines" with the last placed piece as the centroid. By "star lines," we mean the horizontal, vertical, and diagonal lines that pass through the last placed piece. An ASCII representation of the star lines is shown below in the code snippet.

```
chkBoardWinning row col board
  | Black `elem` res && White `elem` res = error "Tie? IMPOSSIBLE!!!"
  | null res = Nothing
  | otherwise = Just $ head res
```

```haskell
  -- length res > 1 is possible : more than 5 pieces in a row
    where res = mapMaybe whoIsWinning5 sls
          sls = getStarLines row col 5 board

whoIsWinning :: [Piece] -> Int -> Maybe Piece
whoIsWinning line cLen = helper (group line) cLen
  where
    helper :: [[Piece]] -> Int -> Maybe Piece
    helper [] cLen = Nothing
    helper (x:xs) cLen
      | length x >= cLen && head x /= Empty = Just $ head x
      | otherwise = helper xs cLen

whoIsWinning5 :: [Piece] -> Maybe Piece
whoIsWinning5 line = whoIsWinning line 5

{-
Star lines
4   4   4
 3  3  3
  2 2 2
   111
432101234
   111
  2 2 2
 3  3  3
4   4   4

-}
```

3.3 AI

We implement this part with reference to [2], with improvements of the running efficiency and coding style.

3.3.1 Minimax and Alpha-Beta Pruning

To perform minimax, we generate a game tree of a given maximum level using the `Data.Tree` package. Note that we only include neighbors as tree nodes. Unlike in Go, which requires occupying space on the board, it is meaningless not to place a piece where there is no neighbors. This approach significantly reduce the tree size.

Code wise, minimax and Alpha-Beta pruning can be done at the same time, as the latter is just cutting branchese for the former. The minimizer will call the maximizer at the next level, while the maximizer will call the minimizer at the next level. If there exists a value indicating that the rest of the branch can be discarded, the algorithm will return a value.

In addition, if the heuristic calculated at a node exceeds the cutoff point, meaning that the game will win or lose if the player choose that step, the rest of the tree search will also be discarded, as this is necessarily the best move. We only include the maximizer here, and the minimizer will be similar, except that it will call the maximizer to get new beta value and choose the minimum max value. The code listed at the end of the report does not perform alpha-beta pruning, as we discard alpha-beta pruning to reach better results for parallelism.

```haskell
maxAlpha :: Piece -> Int -> Int -> Int -> Tree Board -> Int
maxAlpha _ _ alpha _ (Node _ []) = alpha
maxAlpha piece lvl alpha beta (Node b (x:xs))
  | lvl == 0 = curScore
  | canFinish curScore = curScore
  | newAlpha >= beta = beta
  | otherwise = maxAlpha piece lvl newAlpha beta (Node b xs)
  where
    curScore = computeScore b piece
    canFinish score = score > cutoffScore
    newAlpha = max alpha $ minBeta piece (lvl - 1) alpha beta x
```

3.3.2 AI Entry Point

The entry point of the AI is `getNextPos`, which takes a `Board` and the current color of the piece and calls minimax to select the position for the next move.

3.3.3 Heuristic

There are many heuristic functions for Go-Moku with varying performance. In this project, we implemented one that we found online and designed a few on our own. The heuristic we found online is simply based on the connectivity on a piece – the more consecutive pieces there are, the higher connectivity and chance the player will win. However, consecutive pieces blocked by the opponent pieces or the border cannot form a five-in-a-row.

Based on the experience on this faulty heuristic, we then designed our own heuristic. We award no score if both ends are blocked, a low score if one end is block, and a high score if both ends are empty. This can be easily achieved using pattern matching:

```haskell
> [a, b, c, d, e] == [Empty, piece, piece, piece, Empty]
    = 100 + lineScore3 piece (b:c:d:e:xs)
> [a, b, c, d, e] == [reversePiece piece, piece, piece, piece, Empty]
    || [a, b, c, d, e] == [Empty, piece, piece, piece, reversePiece piece]
    = 50 + lineScore3 piece (b:c:d:e:xs)
> otherwise = lineScore3 piece (b:c:d:e:xs)
```

3.4 Parallelism

The tree search nature of the minimax nature makes it possible and suitable to use haskell's parallel techniques. In particular, we implement parallelism at two places: minimax tree search and heuristic calculation

3.4.1 Parallelizing Minimax Tree Search

Haskell's `Data.Tree` package represents the children of a tree in a list, i.e., `[Tree a]`. Minimax's aim is to calculate the maximum value of the score of the first node layer. Therefore, `parMap` comes in handy, as we are supposed to use `Data.List.map` to perform sequential computation.

```haskell
minmax = parMap rdeepseq (minBeta piece searchLevel) children
```

However, this only parallelize the first level of computation. All the subtrees are not parallelized. Indeed, the challenge here is posed by the data dependency of alpha-beta pruning. Alpha-beta pruning assumes a completely sequential operation, as it will always compare the value calculated at one point to the maximum or minimum value in the previous computation in order to decide if one branch can be cut. We therefore decide not to use alpha-beta pruning in our parallel implementation. Nevertheless, not using alpha-beta pruning does not mean we do not benefit from pruning. We will calculate the score for the node beforehand continuing to the next level, and if the score is high enough to ensure a winning condition, the AI will directly choose this node.

In addition, we note that it is also important to control the degree of parallelism. If that degree is too high, the overhead due to creating sparks will exceed the reduced time due to parallelism, making parallelism meaningless. To do this, we only parallelize the first few levels of minimax searching, but keep the rest sequential.

3.4.2 Parallelizing the Computation of Heuristic

The computation of heuristic is also expensive, as we need to apply different functions detecting different numbers of consecutive pieces to the board. In addition, lines to be extracted from the board can be horizontal, vertical, and diagonal. To coordinate this, we use a map-reduce-like method. First, we "map" the board by extracting lines of different directions, and assign workers to calculate the partial heuristic score, and then "reduce" the calculated score by each worker to a final score. The `Eval` monad comes in handy:

```
runEval $ do
  ls2 <- rpar (force (map (lineScore2 piece) pieces))
  ls3 <- rpar (force (map (lineScore3 piece) pieces))
  ls4 <- rpar (force (map (lineScore4 piece) pieces))
  ls5 <- rpar (force (map (lineScore5 piece) pieces))
  rseq ls2
  rseq ls3
  rseq ls4
  rseq ls5
  return (sum ls2 + sum ls3 + sum ls4 + sum ls5)
```

Note that the `map` functions can also be parallized. We will discuss the performance of different configurations in the following section.

3.5 Tweakable Parameters

In order to ensure that both our sequential and parallel algorithm balance efficiency and the smartness of the AI, we parameterize some constants so that we can test for their most appropriate values. First is the number of layers to be searched by minimax, second is the number of layers to be evaluated sequentially to prevent overheads, and the third is the cutoff score for minimax to stop searching for another level.

```
searchLevel :: Int
searchLevel = 2

sequentialLevel :: Int
sequentialLevel = 0 -- level to be evaluated sequentially, must be
                    -- less than or equal to searchLevel

cutoffScore :: Int
cutoffScore = 1000
```

4 Evaluation

Our benchmarks include the following:

1. Running time
2. Spark statistics
3. ThreadScope graphs
4. Subjective observation of whether the AI is smart

We use the following 5-tuple to represent our setup: (`mode, coreNum, searchLevel`, `sequentialLevel`, `parMapHeuristic`). `mode` can be either `seq`, `par`, `seqAB`, meaning we run the sequential version, the complete parallel version, or the sequential version with alpha-beta pruning. `coreNum` indicates with how many cores we run the experiment. `searchLevel` and `sequentialLevel` are tweakable parameters discussed in the previous section. Note that with `seq` or `parAB`, we do not use `sequentialLevel`. `parMapHeuristic` is a boolean, meaning whether nor not we will use `parMap` to parallel inside each worker when computing heuristic. We run each setup for 3 times and calculate the average running time. For the other metrics, we will describe the result from one of the tests.

Experiments are run on a 2018 MacBook Pro, with 2.7 GHz Quad-Core Intel Core i7 processor. We expect the results to be much faster on desktop machines.

Note that since different setup will result in different number of steps for two AIs finish the game. The run time is the program run time divided by the number of steps in a game.

4.1 Number of Cores

To ensure fairness of different testings, we restrict the number of steps to 20. In this experiment, we only consider the difference among different number of cores in a completely parallel setting.

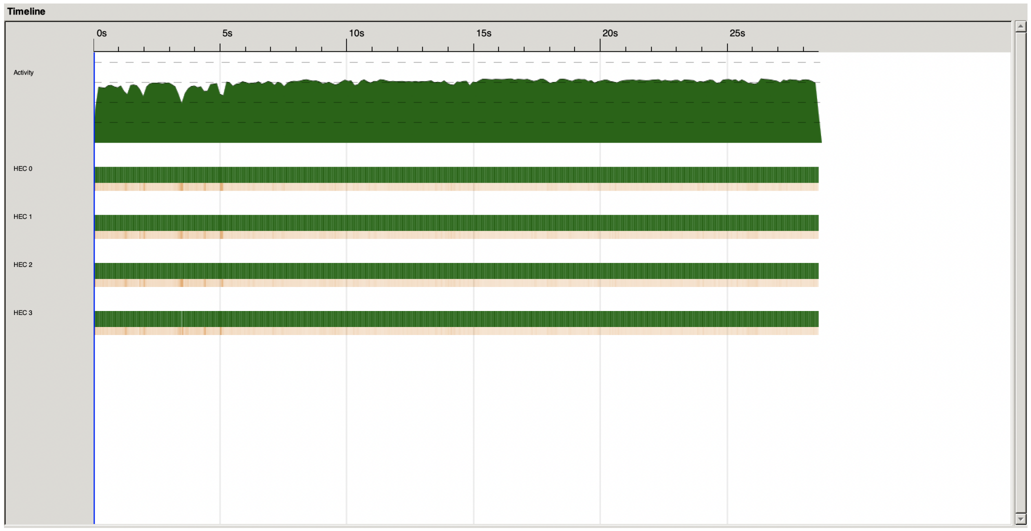
Table 1 shows the comparison of run time and spark statistics for `(par, N1, 3, 1, False)`, `(par, N2, 3, 1, False)`, and `(par, N4, 3, 1, False)`:

Core	Time per step	Total	Converted	GC'd	Fizzled
1	4.04	1238280	0	913176	325104
2	2.19	1238340	1343	904299	332698
4	1.36	1238724	14765	865567	358392

Table 1: Different cores

It can be seen that multi-core leads to a significant amount of performance increase in our implementation. The quad-core system can gain a 2.97× performance increase, and the double-core system can gain a 1.61× performance increase. In addition, the spark conversion rate is also higher in multi-core setting. Doubling the number of cores will increase the conversion rate in a magnitude.

ThreadScope's output also shows that by using `parMap`, the workload is balanced evenly, and the garbage collection does not take majority of time.



4.2 Parallelizing Worker Threads of Heuristic Computation

As mentioned in the above section, the "reduce" part of the heuristic computation can be further parallelized so that each worker thread will call `parMap`. Table 4 shows its result against the previous one in four cores.

Setting	Time per step	Total	Converted	GC'd	Fizzled
`(par, N4, 3, 1, False)`	1.36	1238724	14765	865567	358392
`(par, N4, 3, 1, True)`	1.44	109536668	69214	107421589	204586

Table 2: parMap the heuristic worker threads

As shown in the table, the time per step after parallelize the worker threads even increases slightly, showing that the overhead surpluses the benefit of parallelism. In addition, the number of sparks in this case is 100× the original sequential case. Therefore, this design is undesirable and the worker threads should be sequentially evaluated.

4.3 Depth of Search Level

The knowledge of game is key to winning. Balancing efficiency with knowledge is therefore important. As have shown above, we have picked `searchLevel=3`. Based on our experience playing Go-Moku, we categorize the subjective smartness of the AI in "Good", "Reasonable", and "Bad", where "Good" means the AI is able to make most moves that experienced human players would agree, "Reasonable" means the AI's move might not be consistent with experienced human players' choice but reflect some degree of smartness, and "Bad" means the AI's move is completely unreasonable. We expect our AI to be at least "Good", since the reason to build an AI is to let it compete with human.

In Table 4, we list our comparison results. If we use 4 levels, a step cannot be performed within a reasonable amount of time starting from step 10. With 1 or 2 levels, while the

Setting	Time per step	Smartness
`(par, N4, 4, 1, False)`	N.A.	Good
`(par, N4, 3, 1, False)`	1.36	Good
`(par, N4, 2, 1, False)`	0.14	Reasonable
`(par, N4, 1, 1, False)`	0.01*	Bad

* One player wins before finishing 20 steps

Table 3: Depth setting, runtime, and smartness comparison

computation speed is extremely fast, the smartness is just "Reasonable" or even "Bad". The time per step for 3 levels is within the patience of human. This justifies our choice to use 3 levels as our best setting.

4.4 Partial Parallelism of Tree Search

As noted above, parallelizing a part of the tree, while sequentially evaluating the rest, might reduce the performance decay caused by overhead of useless spark creation. In this part, we use a range of different settings by modifying `sequentialLevel` to evaluate different choices of partial parallelism.

Setting	Steps	Time/Step	Total	Converted
`(par, N4, 4, 2, False)`	10	6.08	3160731	11690
`(par, N4, 4, 3, False)`	10	8.74	2808749	227567
`(par, N4, 3, 0, False)`	20	1.37	1379813	11414
`(par, N4, 3, 1, False)`	20	1.36	1238724	14765
`(par, N4, 3, 2, False)`	20	1.57	1226022	35394

Table 4: Tree Searching Performance with spark creation metrics

Note that when `searchLevel == sequentialLevel`, all levels will be evaluated sequentially and the minimax will be completely sequential. Since we only care about `mode=par` here, we do not test such case. Instead, we evaluate it in the following section. When `searchLevel=3`, we can see that both the efficiency and cost of sparks do not differ much. However, we also compute the performance of `searchLevel=4`. We found that evaluating only 1 level in parallel will hurt the performance. Therefore, we conclude that for searching that are in 3 or 4 levels, it is reasonable to use as fewer sequential searching levels as possible, since the overhead will not reversely affect the performance.

4.5 Parallel vs Sequential

An important design choice we made in our parallel implementation is that we discard alpha-beta pruning. Therefore, it is important to test if this choice improvese the performance.

Setting	Time/Step	Smartness
`(par, N4, 3, 1, False)`	1.36	Good
`(seqAB, N4, 3, 3, False)`	1.07	Reasonable
`(seqAB, N1, 3, 3, False)`	0.89	Reasonable
`(seq, N4, 3, 3, False)`	5.10	Good

Table 5: Parallel vs Sequential

The running time per step for a sequential AI without alpha-beta pruning is the slowest. What is interesting is the parallel solution's comparison with `seqAB`. The running time is similar and even slower for parallel solution, while the smartness is better. This is due to the fact that alpha-beta pruning does not gain all the information of the tree of a certain level. In addition, we can see that alpha-beta pruning does not gain any benefit from multicore systems. If we have more than 4 cores, the performance for the parallel solution will potentially be better than the alpha-beta pruning solution.

5 Conclusion and Future Work

Based on the experiment we have performed, we have discovered the most favorable setting for Gomokururu: `(par, N4, 3, 1, False)`. This setting balances performance, smartness of AI, and resource utilization.

However, due to time constraint, one possible improvement is not realized in this project. We can combine alpha-beta pruning with the parallel implementation by implementing the sequential part in alpha-beta pruning. This requires using two sets of minimax functions and the sequential and parallel implementation has to call each other. As all the code are open-sourced on GitHub, we welcome contributions to this implementation.

6 Team Member Contribution

- Andreas Cheng: Data structures, game board, UI, test suite, winning check, heuristic, neighbor finding
- Kaiwen Xue: Data structures, game board, heuristic, tree building, minimax, parallelism, experiment

7 Code Listing

7.1 AI.hs

```haskell
module AI
  (
    getNextPos,
    buildTree,
    expandBoard,
    computeScore,
    computeScore2,
  )
where

import Game
import Data.Tree
import Data.Vector as V
  ( (!),
    (//),
    concat,
    toList
  )

import Data.Set as S
  ( fromList,
    toList
  )

import Data.List (elemIndex)
import Data.Maybe (fromJust)
import Control.Parallel.Strategies
import Control.DeepSeq

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

maxInt :: Int
maxInt = 2 ^ 29 - 1

-- Tweakable parameters
searchLevel :: Int
searchLevel = 2 -- must be less than treeLevel

sequentialLevel :: Int
sequentialLevel = 1 -- level to be evaluated sequentially, must be
                    -- less than or equal to searchLevel

cutoffScore :: Int
cutoffScore = 1000

-- getNextPos: AI entry point
-- Assumes there is at least one piece on the board, otherwise buildTree will
-- return an empty tree
getNextPos :: Board -> Piece -> (Int, Int)
getNextPos board piece = boardDiff nextBoard board
  where
    (Node b children) = buildTree piece board neighbors (searchLevel + 1)
    neighbors = expandBoard board
    minmax = parMap rdeepseq (minBeta piece searchLevel) children
    index = fromJust $ elemIndex (maximum minmax) minmax
    (Node nextBoard _) = children !! index

-- Assumptions:
-- 1. two boards have equal dim
-- 2. only differ in one bit
boardDiff :: Board -> Board -> (Int, Int)
boardDiff oldBoard newBoard = (drow, dcol)
  where (drow, dcol) = quotRem diffPos (dim newBoard)
        diffPos = getDiff oldBoard1d newBoard1d 0
        getDiff [] [] _ = error "Invalid parameters"
        getDiff (x:xs) (y:ys) ind | x /= y = ind
                                  | x == y = getDiff xs ys (ind + 1)
        getDiff _ _ _ = error "unknown error"
        oldBoard1d = (V.toList . V.concat . V.toList) (getBoard oldBoard)
        newBoard1d = (V.toList . V.concat . V.toList) (getBoard newBoard)


get8NeighboursPoss dBoard r c = validNbPositions
  where validNbPositions = [
          (r + pr, c + pc) |
            pr <- [-1 .. 1],
            pc <- [-1 .. 1],
            (pr, pc) /= (0, 0),
            inBoundary dBoard (r+pr) (c+pc)
          ]

computeScore :: Board -> Piece -> Int
computeScore db p = csHelper 0 0 db p
  where
```

```
    csHelper :: Int -> Int -> Board -> Piece -> Int
    csHelper r c db p
      | c >= bdim = csHelper (r+1) 0 db p
      | r >= bdim || c >= bdim = 0
      | b ! r ! c /= p = next    -- ignore other piece / empty
      | otherwise = numOfGoodNb + next
        where bdim = dim db
              b    = getBoard db
              next = csHelper r (c + 1) db p
              numOfGoodNb = gnHelper b p nbs
                where nbs = get8NeighboursPoss db r c
                      gnHelper b p [] = 0
                      gnHelper b p (n:ns) =
                        fromEnum (b ! nr ! nc == p) + gnHelper b p ns
                        where (nr,nc) = n

computeScore2 :: Board -> Piece -> Int
computeScore2 board piece = runEval $ do
  ls2 <- rpar (force (map (lineScore2 piece) pieces))
  ls3 <- rpar (force (map (lineScore3 piece) pieces))
  ls4 <- rpar (force (map (lineScore4 piece) pieces))
  ls5 <- rpar (force (map (lineScore5 piece) pieces))
  return (sum ls2 + sum ls3 + sum ls4 + sum ls5)
  where
    pieces = getBoardLines board

lineScore2 :: Piece -> [Piece] -> Int
lineScore2 _ [] = 0
lineScore2 piece l
  | length l >= 4 = lineScore2Helper piece l
  | otherwise     = 0
    where
      lineScore2Helper piece (a:b:c:d:xs)
        | pieces4 == [Empty, piece, piece, Empty] = 10 + next
        | pieces4 == [reversePiece piece, piece, piece, Empty]
          || pieces4 == [Empty, piece, piece, reversePiece piece]
          = 5 + next
        | otherwise = next
        where pieces4 = [a, b, c, d]
              next    = lineScore2 piece (b:c:d:xs)
      lineScore2Helper _ _ = error "unknown error"

lineScore3 :: Piece -> [Piece] -> Int
lineScore3 _ [] = 0
lineScore3 piece l
```

```
  | length l >= 5 = lineScore3Helper piece l
  | otherwise     = 0
      where
        lineScore3Helper piece (a:b:c:d:e:xs)
          | pieces5 == [Empty, piece, piece, piece, Empty] = 100 + next
          | pieces5 == [reversePiece piece, piece, piece, piece, Empty]
            || pieces5 == [Empty, piece, piece, piece, reversePiece piece]
            = 50 + next
          | otherwise = next
          where pieces5 = [a, b, c, d, e]
                next = lineScore3 piece (b:c:d:e:xs)
        lineScore3Helper _ _ = error "unknown error"

lineScore4 :: Piece -> [Piece] -> Int
lineScore4 _ [] = 0
lineScore4 piece l
  | length l >= 6 = lineScore4Helper piece l
  | otherwise     = 0
      where
        lineScore4Helper piece (a:b:c:d:e:f:xs)
          | piece6 ==
              [Empty, piece, piece, piece, piece, Empty] = 1000 + next
          | piece6 == [reversePiece piece, piece, piece, piece, piece, Empty]
            || piece6 == [Empty, piece, piece, piece, piece, reversePiece piece]
            = 500 + next
          | otherwise = next
          where piece6 = [a,b,c,d,e,f]
                next = lineScore4 piece (b:c:d:e:f:xs)
        lineScore4Helper _ _ = error "unknown error"

lineScore5 :: Piece -> [Piece] -> Int
lineScore5 _ [] = 0
lineScore5 piece l
  | length l >= 5 = lineScore5Helper piece l
  | otherwise     = 0
      where
        lineScore5Helper piece (a:b:c:d:e:xs)
          | [a, b, c, d, e] ==
              [piece, piece, piece, piece, piece] = 10000 + lineScore5 piece (b:c:d:e:xs
          | otherwise = lineScore5 piece (b:c:d:e:xs)
        lineScore5Helper _ _ = error "unknown error"

-- Ref: 2019 project
buildTree :: Piece -> Board -> [(Int, Int)] -> Int -> Tree Board
buildTree piece board neighbors lvl = Node board $ children lvl neighbors
```

```
  where children _ []                    = []
        children 0 _                     = []
        children lvl ((row, col) : xs) =
          buildTree (reversePiece piece) newBoard newNeighbors (lvl - 1)
            : children lvl xs
            where newNeighbors = expandBoard newBoard
                  newBoard = placePiece board piece row col

maxAlpha :: Piece -> Int -> Tree Board -> Int
maxAlpha piece lvl (Node b children)
  | lvl == 0 = curScore
  | curScore <= 0 = curScore
  | lvl <= sequentialLevel = maximum $ map (minBeta piece (lvl - 1)) children
  | otherwise = maximum $ parMap rdeepseq (minBeta piece (lvl - 1)) children
  where
    curScore = computeScore2 b piece - computeScore2 b (reversePiece piece)

minBeta :: Piece -> Int -> Tree Board -> Int
minBeta piece lvl (Node b children)
  | lvl == 0 = curScore
  | curScore >= cutoffScore = curScore
  | lvl <= sequentialLevel = minimum $ map (maxAlpha piece (lvl - 1)) children
  | otherwise = minimum $ parMap rdeepseq (maxAlpha piece (lvl - 1)) children
  where
    curScore = computeScore2 b piece - computeScore2 b (reversePiece piece)

-- Get a list (or vector) of points created by the next move
expandBoard :: Board -> [(Int, Int)]
expandBoard db = S.toList $ S.fromList $ ebHelper 0 0 db
  where
    ebHelper r c db
      | c >= bdim = ebHelper (r + 1) 0 db
      | r >= bdim || c >= bdim = []
      | b ! r ! c == Empty = next
      | otherwise = validNbPositions ++ next
      where
        b = getBoard db
        bdim = dim db
        next = ebHelper r (c + 1) db
        validNbPositions = [
          (r + pr, c + pc) |
            pr <- [-1 .. 1],
            pc <- [-1 .. 1],
            (pr, pc) /= (0, 0),
            emptyValid db (r+pr) (c+pc)
          ]
```

7.2 Game.hs

```haskell
module Game
  (
    Piece (..),
    Board (..),
    genBoard,
    putBoard,
    placePiece,
    placePieceFrmTuples,
    placePieceFrmTuplesF,
    whoIsWinning5,
    piece2emoji,
    showStepInfo,
    emptyValid,
    inBoundary,
    reversePiece,
    chkBoardWinning,
    getBoardLines
  )
where

import Data.Vector as V
  (
    Vector,
    map,
    replicate,
    toList,
    fromList,
    slice,
    (//),
    (!)
  )

import Data.List (group)
import Data.Char
import Data.Maybe (mapMaybe)

-- Algebraic data types for Board and Piece
-- TODO (Andreas): Use another branch to try 1D once the first task is done
data Board = Board { dim :: Int
                   , getBoard :: Vector (Vector Piece) }
                   deriving (Show)
```

```haskell
 -- type Board = Vector (Vector Piece)
data Piece = White | Black | Empty deriving (Eq)

instance Show Piece where
  show White = "W"
  show Black = "B"
  show _ = "_"

reversePiece :: Piece -> Piece
reversePiece White = Black
reversePiece Black = White
reversePiece _ = error "invalid argument"

-- full-width alphabets and space are used to align with the emojis
-- if the board's dim goes beyond a specific number, then this becomes ugly
putBoard :: Board -> IO ()
putBoard b = do
  putStrLn $ '' : take (length listifyStrBoard) [''..]
  mapM_ (putStrLn . uncurry (:)) (zip [''..] listifyStrBoard)
  where listifyStrBoard = V.toList $ V.map getVPieceString $ getBoard b

piece2emoji :: Piece -> [Char]
piece2emoji White = ""
piece2emoji Black = ""
piece2emoji Empty = ""

getVPieceString :: Vector Piece -> [Char]
getVPieceString vp = concatMap piece2emoji (V.toList vp)

genBoard :: Int -> Board
genBoard dim = Board dim bd
  where bd = V.replicate dim (V.replicate dim Empty)

showStepInfo :: Piece -> Int -> IO ()
showStepInfo p step = do
    putStrLn $ "\nStep " ++ show step ++ ": " ++ piece2emoji p ++ "'s move"


-- Modifying Board state
emptyValid :: Board -> Int -> Int -> Bool
emptyValid board row col =
  inBoundary board row col && (getBoard board ! row ! col) == Empty
    where db = dim board

inBoundary :: Board -> Int -> Int -> Bool
inBoundary board row col =
  row >= 0 && col >= 0 && row < db && col < db
    where db = dim board

placePiece :: Board -> Piece -> Int -> Int -> Board
placePiece board p row col = Board (dim board) bd
  where bd = b // [(row, updatedRow)]
        updatedRow = (b ! row) // [(col, p)]
        b = getBoard board

placePieceFrmTuples :: Board -> [(Piece, Int, Int)] -> Board
placePieceFrmTuples board [] = board
placePieceFrmTuples board (m:ms) =
  placePieceFrmTuples newBoard ms
    where newBoard = placePiece board p r c
          (p,r,c) = m

-- F stands for FANCY
placePieceFrmTuplesF :: Board -> [String] -> Board
placePieceFrmTuplesF board cList = placePieceFrmTuples board pList
  where pList = helper cList
        helper [] = []
        helper (x:xs) =
          (p,r,c) : helper xs
          where
            (cp:cr:cc:eol) = x
            p = if cp =='B' then Black else White
            r = ord cr - ord 'A'
            c = ord cc - ord 'A'

getBoardLines :: Board -> [[Piece]]
getBoardLines dBoard =
  hLines ++ vLines ++ lhLinesLeft ++ lhLinesRight ++ hlLinesLeft ++ hlLinesRight
  where
    bDim = dim dBoard
    b = dBoard
    max_rc = bDim - 1
    hLines       = [ getLineFrmBoard b r 0 r max_rc | r <- [0..max_rc] ]
    vLines       = [ getLineFrmBoard b 0 c max_rc c | c <- [0..max_rc] ]
    -- [0..bDim] below can be adjusted to ignore diagonal lines in which length <5
      -- Something like: [5..bDim-5]
    lhLinesLeft  = [ getLineFrmBoard b rc 0 0 rc | rc <- [0..max_rc]]            -- ///
    lhLinesRight = [ getLineFrmBoard b max_rc rc rc max_rc | rc <- [1..max_rc]]      --
    hlLinesLeft  = [ getLineFrmBoard b (max_rc - rc) 0 max_rc rc | rc <- [0..max_rc]] --
    hlLinesRight = [ getLineFrmBoard b 0 rc (max_rc - rc) max_rc | rc <- [1..max_rc]] --
```

```haskell
-- Check win
whoIsWinning :: [Piece] -> Int -> Maybe Piece
whoIsWinning line cLen = helper (group line) cLen
  where
    helper :: [[Piece]] -> Int -> Maybe Piece
    helper [] cLen = Nothing
    helper (x:xs) cLen
      | length x >= cLen && head x /= Empty = Just $ head x
      | otherwise = helper xs cLen

whoIsWinning5 :: [Piece] -> Maybe Piece
whoIsWinning5 line = whoIsWinning line 5

chkBoardWinning row col board
  | Black `elem` res && White `elem` res = error "Tie? IMPOSSIBLE!!!"
  | null res = Nothing
  | otherwise = Just $ head res  -- length res > 1 is possible : more than 5 pieces in
    where res = mapMaybe whoIsWinning5 sls
          sls = getStarLines row col 5 board

getColFrmBoard :: Board -> Int -> Int -> Int -> [Piece]
getColFrmBoard board col rf rt
  | col < 0    = error "bad col"
  | col >= bDim = error "bad col: larger than dimension"
  | otherwise  = helper b [rf..rt]
    where b = getBoard board
          bDim = dim board
          helper b []     = []
          helper b (r:rs) = (b ! r ! col) : helper b rs

getLineFrmBoard :: Board -> Int -> Int -> Int -> Int -> [Piece]
getLineFrmBoard board fr fc tr tc
  | fr<0 || fc<0 || tr<0 || tc <0                = error "some coordinates are zeroes"
  | fr>=bDim || fc>=bDim || tr>=bDim || tc>=bDim = error "some coordinates are beyond di
  | fr == tr && fc == tc   = [b ! fr ! fc]  -- A dot
  | fr == tr               = hLine b fr fc tc
  | fc == tc               = vLine board fc fr tr
  | abs (fr-tr) == abs (fc-tc) = dLine b fr fc tr tc
  | otherwise       =
    error $ show fr ++ " " ++ show fc ++ "    " ++ show tr ++" " ++ show tc ++ " genDlin
    where b     = getBoard board
          bDim = dim board
          hLine b row colA colB =
             V.toList (V.slice colA (colB-colA+1) (b ! row))

          vLine = getColFrmBoard

          dLine b fr fc tr tc =
            dHelper b ftr ftc
              where
                ftr = if fr < tr then [fr..tr] else [fr,fr-1..tr]
                ftc = if fc < tc then [fc..tc] else [fc,fc-1..tc]
                dHelper b []  []         = []
                dHelper b _  []          = []
                dHelper b [] _           = []
                dHelper b (r:rs) (c:cs) = b ! r ! c : dHelper b rs cs

{-
Star lines
4   4   4
 3  3  3
  2 2 2
   111
432101234
   111
  2 2 2
 3  3  3
4   4   4

-}

-- Something is wrong with this function
getDrc fr fc dr dc dim len
  | len<=0 || nr < 0 || nc < 0 || nr >= dim || nc >= dim = (fr,fc)
  | otherwise = getDrc nr nc dr dc dim ll
    where nr = fr+dr
          nc = fc+dc
          ll = len-1

-- TODO: create type of Vector (Vector a)
getStarLines :: Int -> Int -> Int -> Board -> [[Piece]]
getStarLines row col llen board =
  [
  -- horizontal
  V.toList (V.slice hi hn (b ! row)),
  -- verticals
  getColFrmBoard board col ra rb,
  -- diagonals
  getLineFrmBoard board tlr tlc brr brc,
  getLineFrmBoard board blr blc trr trc
  ]
  where b    = getBoard board
        bDim = dim board
        hi = if col - llen < 0 then 0 else col - llen + 1
        hn = if hi + hLen > bDim then bDim - hi else hLen
        hLen = llen*2-1

        ra = if row - llen < 0 then 0 else row - llen + 1
        rb = if row + llen >= bDim then bDim - 1 else row + llen - 1

        (tlr, tlc) = getDrc row col (-1) (-1) bDim llen
        (brr, brc) = getDrc row col 1    1    bDim llen
        (blr, blc) = getDrc row col 1    (-1) bDim llen
        (trr, trc) = getDrc row col (-1) 1    bDim llen
```

7.3 Spec.hs

```
module Main where
import Text.Printf
import Data.Tree

import Game
import AI


redANSI = "\ESC[31m"
greenANSI = "\ESC[32m"
defANSI = "\ESC[0m"
redify s = redANSI ++ s ++ defANSI
greenify s = greenANSI ++ s ++ defANSI

putCheckRes :: (PrintfArg p, Eq a) => p -> a -> a -> IO Bool
putCheckRes caseName eRes res = do
    putStrLn $ prettyCaseName ++ " " ++ passStr
    return match
    where match = res == eRes
          passStr = if res == eRes then  greenify "passed" else redify "failed"
          prettyCaseName = printf "%-40s" caseName

showTree :: Tree Board -> IO ()
showTree (Node board children) = do
  putBoard board
  putStr "\n"
  showTreeHelper children

showTreeHelper :: [Tree Board] -> IO ()
showTreeHelper [] = do return ()
showTreeHelper (c:cs) = do
  showTree c
  showTreeHelper cs


boardPlacementTest :: IO ()
boardPlacementTest = do
    putBoard $ genBoard 10
    putStr "\n"
    let board1 = placePiece (genBoard 10) White 4 5
    putBoard board1
    putStr "\n"
    let board2 = placePiece board1 Black 3 4
    putBoard board2


testWhoIsWinning :: IO Bool
testWhoIsWinning = do
    c1 <- putCheckRes
        "No one is winning (5 Empty): "
        Nothing
        (whoIsWinning5 $ replicate 4 Black ++ replicate 4 White ++ replicate 5 Empty)

    c2 <- putCheckRes
        "Black is winning (5 Black): "
        (Just Black)
        (whoIsWinning5 $ replicate 4 White ++ [Empty, White] ++ replicate 5 Black)

    c3 <- putCheckRes
        "White is winning (5 White): "
        (Just White)
        (whoIsWinning5 $ replicate 4 Black ++ replicate 5 White)

    return $ c1 && c2 && c3

testChkBoardWinning :: IO Bool
testChkBoardWinning = do
    let t = placePieceFrmTuplesF (genBoard 17) ["BKJ","WKJ","BLM","BBC",
    "WOM", "WON","WOO","WOP","WOQ"]
    c1 <- putCheckRes
        "White won on row 0 in a board: "
        (Just White)
        (chkBoardWinning 14 14 t)

    let t = placePieceFrmTuplesF (genBoard 17) ["BAA", "BBB", "BCC", "BDD", "BEE",
                                                 "WAQ", "WBP", "WCO", "WDN", "WEM",
                                                 "WQA", "WPB", "WOC", "WND", "WME",
                                                 "BQQ", "BPP", "BOO", "BNN", "BMM"]

    -- let t = placePieceFrmTuplesF (genBoard 17) ["BAA"]

    c2 <- putCheckRes
        "Black wins (AA-EE): "
        (Just Black)
        (chkBoardWinning 0 0 t)

    c3 <- putCheckRes
        "White wins (AQ-EM): "
        (Just White)
        (chkBoardWinning 1 15 t)

    c4 <- putCheckRes
        "White wins (QA-ME): "
        (Just White)
        (chkBoardWinning 16 0 t)

    return $ c1 && c2 && c3 && c4

buildTreeTest :: IO ()
buildTreeTest = do
    putStrLn "Running buildTreeTest"
    let t = placePieceFrmTuplesF (genBoard 15) ["BGG"]
    showTree $ buildTree White t (expandBoard t) 2

main :: IO ()
main = do
    -- boardPlacementTest
    -- buildTreeTest

    putStrLn "> Running testcases..."

    let res = sequence [
                testWhoIsWinning,
                testChkBoardWinning
                ]
    allPassed <- and <$> res
    if allPassed then
        putStrLn "> Done"
    else
        error $ redify "Some test cases failed"
```

7.4 Main.hs

```
module Main where

import Game
import AI
import Data.Char

main :: IO ()
main = do
    let t = placePieceFrmTuplesF (genBoard 15) ["BHH"]
    gameLoop t White 0 20

getPair :: IO (Int, Int)
getPair = do
    line <- getLine
    let (rl:cl:_) = line
    let rlRet = ord (toUpper rl) - ord 'A'
        clRet = ord (toUpper cl) - ord 'A'
    return (rlRet, clRet)


takeTurn :: Board -> Piece -> Int -> IO (Board, Int, Int)
takeTurn board piece step = do
    putStrLn $ "It's " ++ piece2emoji piece ++ " turn."
    putStrLn $ "Please place your piece (e.g. KF):"
    -- putStrLn £ "Input " ++ piece2emoji piece ++ " row and col (e.g. FK):"
    (row, col) <- getPair
    if not (emptyValid board row col)
        then do
            putStrLn "Invalid placement, try again."
            takeTurn board piece step
    else do
        let newBoard = placePiece board piece row col
        showStepInfo piece step
        putBoard newBoard
        return (newBoard, row, col)
```

```haskell
takeTurnAI :: Board -> Piece -> Int -> IO (Board, Int, Int)
takeTurnAI board piece step = do
    putStrLn $ "It's " ++ piece2emoji piece ++ " (AI)'s turn."
    let (row, col) = getNextPos board piece
        newBoard = placePiece board piece row col
    showStepInfo piece step
    putBoard newBoard
    return (newBoard, row, col)

gameLoop :: Board -> Piece -> Int -> Int -> IO ()
gameLoop board piece step totalSteps = do
    putStrLn "\n====Current Board===="
    putBoard board
    putStrLn "====================="

    (newBoard, row, col) <- case piece of
        Black -> takeTurnAI board piece (step + 1)
        White -> takeTurnAI board piece (step + 1)

    putStrLn $ "Score for step " ++ (show $ step + 1) ++ ": " ++
    (show $ computeScore2 newBoard piece)

    if step + 1 >= totalSteps then do putStrLn $ "Game ended at step limit."
    else do
        case chkBoardWinning row col newBoard of
            Nothing -> gameLoop newBoard (reversePiece piece) (step + 1) totalSteps
            (Just piece) -> do putStrLn $ piece2emoji piece ++ " wins!\nGame ended."
```

References

[1] *Go-Moku Solved by New Search Techniques*. URL: https://www.aaai.org/Papers/Symposia/Fall/1993/FS-93-02/FS93-02-001.pdf.

[2] *Gomoku Game in Haskell*. URL: http://www.cs.columbia.edu/~sedwards/classes/2019/4995-fall/reports/gomoku.pdf.