COMS 4995: Parallel Functional Programming
Parallel Hex AI with MinMax

Eric Feng: ef2648
Cesar Ramos Medina: cer2178

December 2020

1 Introduction

Hex is an intriguing board game invented by Piet Hein and John Hash in the 1940s independently. The game is played out by two players on a board in the shape of a parallelogram. The parallelogram is made of hexagons of equal length and width. The two players take turns placing pieces on their board. Prior to the game, each player should claim two opposing sides of the board. The winning condition is to have an uninterrupted path of tiles from one end of the chosen side to the other. Observe the blue player's winning path in the figure below.

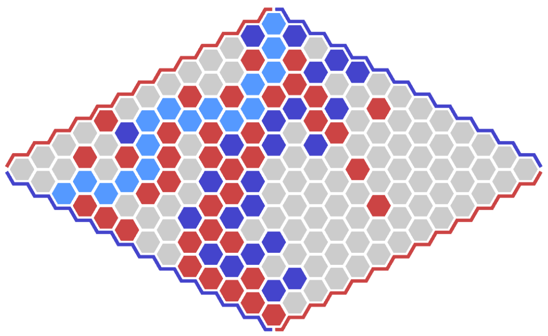


Figure 1: A game of Hex in its finished state. The light blue path indicates the winning path by the blue player.

Hex has some unique properties unlike other board games. For example, it cannot end in a draw due to the topology of the board. Hence, a clear winner always emerges victorious from a match. A proof of this phenomenon may be found here.

We present a Hex Agent implemented in Haskell with the MinMax algorithm. In the following pages, we will describe how we implemented our agent, key decisions, and our findings in attempting to parallelize the agent.

Assumptions : player 1 take top and bottom, player 2 takes left and right. no swap rule

2 Implementation

2.1 Internal Representation of Board

The first obstacle in creating our program is representing the hexagonal board in Haskell. To help with this process, the board is internally represented in Haskell as a *GridMap*, which is found in the grid package. A Grid is a data structure which represents the arrangement of tiles on a board (e.g [(0,0), (0, 1), (0,2) ... (11, 11)]. GridMap is a wrapper for Grid and Map; its keys are tile positions (coordinates for the board), and its values represent the data currently occupying that cell.

We chose to use the grid package to help implement the games' functionality because it alleviated the challenge of working in a coordinate system for hexagonal grids, which we had a hard time with. The specifics on the indexing scheme for the hexagonal board used in the package may be found in Red Blob Games' fantastic guide on making use of hexagonal grids.

2.2 Displaying Game State

To display the game state to the user, we make use of the function draw, which is from ChristopherKing42's Hex AI using Monte Carlo Tree Search. We then adapted his implementation for GridMap and Grid. We found this to be the most convenient representation of the board as opposed to drawing hexagonal shapes. In essence, the function takes in the length/width of the board and the board itself, then draws the players' moves by printing the value of each index in the GridMap in an organized fashion.

```
draw :: (M.GridMap gm Char, G.Index (M.BaseGrid gm Char) ~ (Int,
↪  Int)) =>
                Int -> gm Char -> String
draw size board = unlines [line y | y <- [0..size]] where
    line 0 = (' ':) $ take size ['A'..] >>= (:" ") --draw top
    ↪  legend
    line y = replicate y ' ' ++ --white space for formatting
             replicate (length $ show y) '\b' ++  -- delete extra
             ↪  spaces
             show y ++ -- print current number
             concat [[' ', cell x (y - 1)] | x <- [0..size-1]] --
             ↪  line info
    cell x y = case M.lookup (x, y) board of
                    Just a -> a
                    Nothing -> '-'
```

2.3 Heuristics Function

The heuristics function we use to evaluate a game state is called *countConnected*. We found this heuristic for the Hex game in an AI lab from the the Computer Science department at Swarthmore. Given the GridMap (current board state), the representation of the board, and the "color" of the player, countConnected returns the number of pieces for that player which are touching another piece of the same color on the board. The idea behind it is basically that the more "connected" a color is, the more likely it is that a connection leads to a path from one side to the other (with the exception of clusters).

```
countConnected :: (M.GridMap gm v, Ord (G.Index g), Eq v, Num a,
↪   G.Grid g, G.Index (M.BaseGrid gm v)~ G.Index g) => gm v -> g
↪   -> v -> a
countConnected gm board color = sum $ map (\x -> countNeighbor x)
↪   coordinates
        where getCommonColors _ v = v == color -- get kv pairs in
        ↪   grid map that have the same value as color
              coordinates = M.keys $ M.filterWithKey
              ↪   getCommonColors gm -- get list of coordinates
              ↪   belonging to that color (get keys with value v
              ↪   from gm)
              noNeighborIsSameColor = null . flip intersect
              ↪   coordinates . G.neighbours board -- returns
              ↪   boolean on whether a neighbour of the current
              ↪   point is a dot of the same color
              countNeighbor me = if noNeighborIsSameColor me then
              ↪   0 else 1
```

2.4 Minimax Implementation

We are using a variation of minimax known as depth-limited minimax. Furthermore, we will not look at the agents as one being a maximizing one and the other as a minimizing one, but rather as both players attempting to maximize their outcome.

Our depth-limited minimax works in three stages: playGame, minimaxDecision, and maxPx. playGame, as the function mentions, plays the game. This basically means that it will alternate between who puts a tile down, either playerA or playerB, and will call the appropriate function (minimaxDecision on the color 'A' or color 'B'). playGame returns IO (), as it basically prints the boards, and then mentions the winner.

The minimaxDecision function, as the name suggests, will decide where to put a tile. To do this, minimaxDecision first considers all possible moves (or, equivalently, boards with a tile played in a different position), and then selects the board that yields the maximum heuristic value for the player. Subsequently, it returns the heuristic value pair for the board (basically, we return the heuristic values for both player as a tuple, which becomes useful for whoever receives the value being returned).

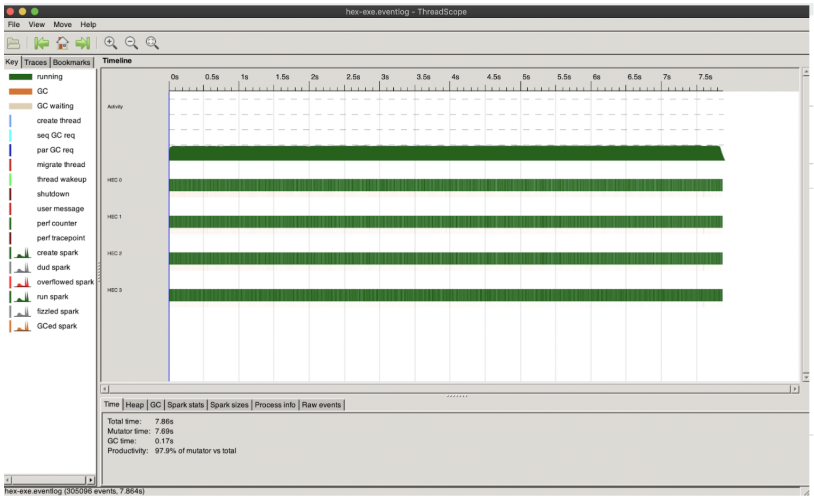
The maxPx function is the core of the minimax algorithm. It iterates over all possible boards, and calls the max function for the opposing player. This function returns a tuple containing the value for the heuristic for the player 'A' in the first position, and the value for the heuristic for player 'B' in the second position.

Please find detailed implementation in the code listing section as it is quite large to put here.

3 Parallelizing

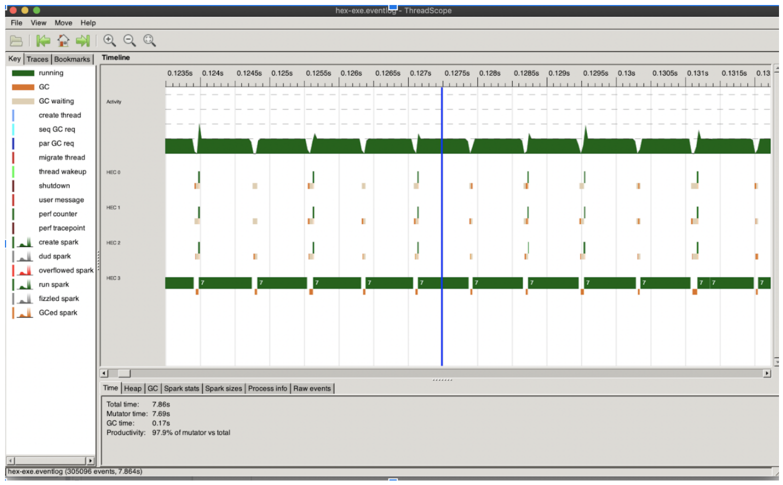
The primary part of our program that we were trying to parallelize were the map operations over lists of "next moves" for a given player using MinMax. Hence, our attempts mainly revolved around 'using' parList and parMap. We tried using rseq as the main strategy. The idea of using rseq was that since the operation after the map operation necessarily requires the "complete picture" for comparison purposes, rseq was sufficient in forcing normal form.

First, we tried to add 'using' parList rseq to all map operations. This yielded in the fastest version of the program, but there were nearly no differences between the sequential version of the program and the parallel version. Upon further examination of threadscope, we found that only one core was really active at a certain time.



```
Total time:    7.86s
Mutator time:  7.69s
GC time:       0.17s
Productivity:  97.9% of mutator vs total
```

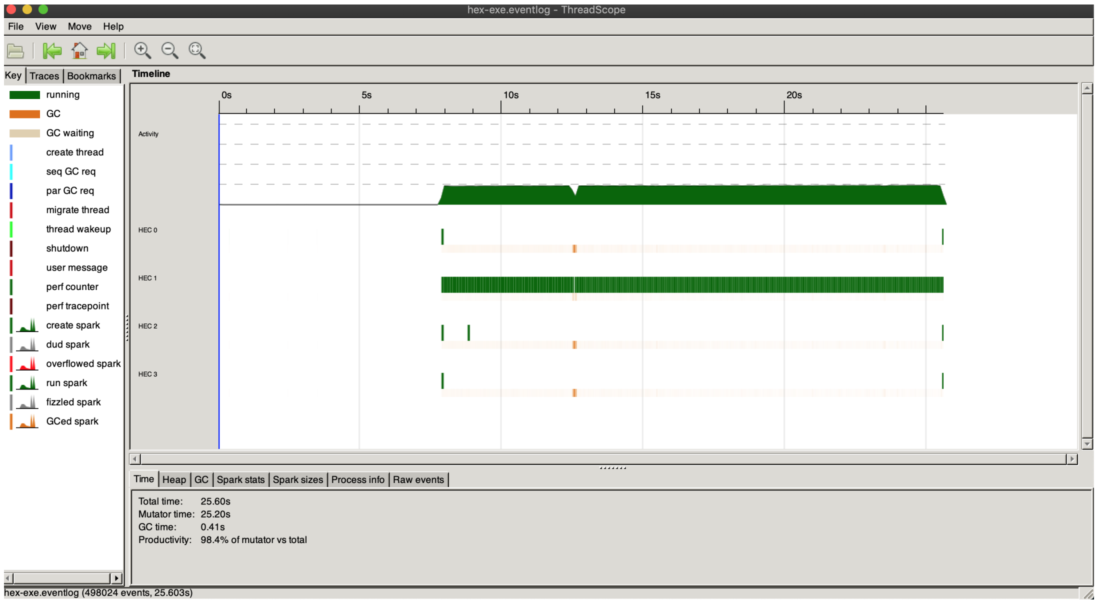
```
hex-exe.eventlog (305096 events, 7.864s)
```



```
Total time:    7.86s
Mutator time:  7.69s
GC time:       0.17s
Productivity:  97.9% of mutator vs total
```

```
hex-exe.eventlog (305096 events, 7.864s)
```

To try and remedy this problem, we tried to instead distribute the map operations with rpar, but evaluate more immediate operations with rseq. The idea was to continue to the next level in WHNM until gathered for evaluation. Furthermore, we were much more conservative with where to introduce parallelism. This is the version of the program that we handed in, but it is in fact slower than the former version. However, the threadscope analysis shows a different picture.



```
Total time:    25.60s
Mutator time:  25.20s
GC time:       0.41s
Productivity:  98.4% of mutator vs total
```

hex-exe.eventlog (498024 events, 25.603s)

Hence, we may assume that the former version is superior and that true parallelism for our agent hence requires further examination. Please note all tests are done without printing the board at each step (print slows down everything dramatically).

4 Notes

Further optimizations of our agent involve better heuristics and more importantly alpha beta pruning. The search space of our program grows exponentially at every level, and hence it is difficult for us to run tests with non-trivial depths on our machines when the board size is bigger.

Another thing to note is that we realized our state evaluation function was wrong very late in our review process. The game winning condition required only for a complete path to exist for either player from their chosen side, while we thought all squares needed to be filled prior to comparison. We did not have time to replace our existing state evaluation function, but I will show the extent to which we have worked on the replacement. The idea is at each step, to check if there are any tiles that are at a boundary "belonging" to the player. If there are, then we will perform DFS and recursively attempt to find a path between the starting tile to a tile of the opposing side. The code compiles but it requires changing our MinMax function which we did not have time to do.

```
--boundary tiles for each of the sides
bot g = (chunksOf 4 (G.boundary g)) !! 0
top g = (chunksOf 4 (G.boundary g)) !! 2
left g = (chunksOf 4 (G.boundary g)) !! 1
right g = (chunksOf 4 (G.boundary g)) !! 3

-- get list of coordinates belonging to that color (get keys with
↪  value v from gm)
coordinates gm color = M.keys $ M.filterWithKey (const (color
↪  ==)) gm

gameOver gm board color
  | color == 'A' && hasBoundary topTiles = maximum $ map
  ↪  (\candidateTile -> findPath (M.delete candidateTile gm)
  ↪  board color botTiles candidateTile) (intersect myTiles
  ↪  topTiles)
  | color == 'A' && hasBoundary botTiles = maximum $ map
  ↪  (\candidateTile -> findPath (M.delete candidateTile gm)
  ↪  board color topTiles candidateTile) (intersect myTiles
  ↪  botTiles)
  | color == 'B' && hasBoundary leftTiles = maximum $ map
  ↪  (\candidateTile -> findPath (M.delete candidateTile gm)
  ↪  board color rightTiles candidateTile) (intersect myTiles
  ↪  leftTiles)
  | color == 'B' && hasBoundary rightTiles = maximum $ map
  ↪  (\candidateTile -> findPath (M.delete candidateTile gm)
  ↪  board color leftTiles candidateTile) (intersect myTiles
  ↪  rightTiles)
  | otherwise = 0
  where hasBoundary dir = not $ null $ intersect myTiles dir
        myTiles = coordinates gm color
        topTiles = top board
        botTiles = bot board
        leftTiles = left board
        rightTiles = right board
        findPath gm board color desiredDirection candidateTile
            | desiredDirection `G.contains` candidateTile = 1
            | null (intersect (G.neighbours board candidateTile)
            ↪  gm) = 0
```

```
| otherwise = maximum $ map (\candidateTile ->
↪  findPath (M.delete candidateTile gm) board color
↪  desiredDirection candidateTile) (intersect
↪  (coordinates gm color) (G.neighbours board
↪  candidateTile))
```

To run our program, please follow standard stack syntax. In particular, stack install will generate an executable for threadscope analysis, stack run will run a convenient instance of the project.

5 Code Listing

```
{-# LANGUAGE GADTs, FlexibleContexts #-}

module Main where
import Math.Geometry.Grid.Hexagonal ( paraHexGrid )
import qualified Math.Geometry.Grid as G
import qualified Math.Geometry.GridMap as M
import Math.Geometry.GridMap.Lazy ( lazyGridMap )
import Text.Read
import Data.List (intersect, maximumBy )
import Control.Parallel.Strategies

computeValidMoves :: M.GridMap gm Char => gm Char -> gm Char
computeValidMoves gm = M.filter (\v -> v /= 'A' && v /= 'B' ) gm

maxPA :: (M.GridMap gm Char, Ord (G.Index (M.BaseGrid gm Char)),
                Ord b) =>
               gm Char -> t -> Int -> Int -> (gm Char -> t ->
                ↪  Char -> b) -> (b, b)
maxPA gm board max_depth curr_depth heur_func = do
        let valid_moves = computeValidMoves gm

        if curr_depth >= max_depth || length (M.keys valid_moves)
        ↪  < max_depth then
            ( heur_func gm board 'A', heur_func gm board 'B')
        else
            do

                -- valid_boards is of type [GridMap]
                let valid_boards = map (\k -> M.insert k 'A' gm)
                ↪  (M.keys valid_moves)


                -- board_values is of "type" [(GridMap,
                ↪  (Heuristic for A, Heuristic for B) )]
```

```
                    let board_values = map ( \grid-> ( grid, ( maxPB
                    ↪  (grid) (board) ( max_depth ) ( curr_depth + 1
                    ↪  ) ( heur_func ) ) ) ) valid_boards
                    -- computes max, since comparisons are required
                    ↪  here rseq is enough since it will need to
                    ↪  evaluate to normal form here
                    let board_max_value_A = maximumBy (
                    ↪  \(_,(heur_a_1,_)) (_,(heur_a_2,_)) -> compare
                    ↪  heur_a_1 heur_a_2) board_values


                    (snd board_max_value_A)



maxPB :: (M.GridMap gm Char, Ord (G.Index (M.BaseGrid gm Char)),
                 Ord b) =>
                gm Char -> t -> Int -> Int -> (gm Char -> t ->
                ↪  Char -> b) -> (b, b)
maxPB gm board max_depth curr_depth heur_func = do
        let valid_moves = computeValidMoves gm

        if curr_depth >= max_depth || length (M.keys valid_moves)
        ↪  < max_depth then
            (heur_func gm board 'A', heur_func gm board 'B')
        else
            do
                -- valid_boards is of type [GridMap]
                let valid_boards = map (\k -> M.insert k 'B' gm)
                ↪  (M.keys valid_moves)

                -- board_values is of "type" [(GridMap,
                ↪  (Heuristic for A, Heuristic for B) )]
                let board_values = map ( \grid-> ( grid, ( maxPA
                ↪  (grid) (board) ( max_depth ) ( curr_depth + 1
                ↪  ) ( heur_func ) ) ) ) valid_boards

                let board_max_value_B = maximumBy ( \(_,
                ↪  (_,heur_b_1) ) (_, (_,heur_b_2 ) ) -> compare
                ↪  heur_b_1 heur_b_2  ) board_values
                (snd board_max_value_B)

minimax_decision :: (M.GridMap gm Char,
                          Ord (G.Index (M.BaseGrid gm Char)),
                          ↪  Ord a) =>
                         gm Char
```

```
                              -> t
                              -> Char
                              -> (gm Char -> t -> Char -> a)
                              -> Int
                              -> (gm Char, (a, a))
minimax_decision gm board color heuristic max_depth = do

    let valid_moves = computeValidMoves gm

    if color == 'A' then
        do
            let valid_boards = map (\k -> M.insert k 'A' gm)
            ↪  (M.keys valid_moves)

            let board_values = map ( \grid-> ( grid, ( maxPB
            ↪  (grid) (board) ( max_depth ) ( 1 ) ( heuristic )
            ↪  ) ) ) valid_boards

            let board_max_value_A = maximumBy ( \(_,(heur_a_1,_))
            ↪  (_,(heur_a_2,_)) -> compare heur_a_1 heur_a_2)
            ↪  board_values

            board_max_value_A

    else
        do
            let valid_boards = map (\k -> M.insert k 'B' gm)
            ↪  (M.keys valid_moves)


            let board_values = map ( \grid-> ( grid, ( maxPA
            ↪  (grid) (board) ( max_depth ) ( 1 ) ( heuristic )
            ↪  ) ) ) valid_boards
            let board_max_value_B = maximumBy ( \(_, (_,heur_b_1)
            ↪  ) (_, (_,heur_b_2 ) ) -> compare heur_b_1
            ↪  heur_b_2  ) board_values

            board_max_value_B


par_minimax_decision :: (M.GridMap gm Char,
                                Ord (G.Index (M.BaseGrid gm
                                 ↪  Char)), Ord a) =>
                              gm Char
                              -> t
                              -> Char
```

```haskell
                              -> (gm Char -> t -> Char -> a)
                              -> Int
                              -> (gm Char, (a, a))

par_minimax_decision gm board color heuristic max_depth = do

    let valid_moves = computeValidMoves gm

    if color == 'A' then
        do
            let valid_boards = map (\k -> M.insert k 'A' gm)
            ↪  (M.keys valid_moves) `using` parList rseq

            let board_values = parMap rpar ( \grid-> ( grid, (
            ↪  maxPB  (grid) (board) ( max_depth ) ( 1 ) (
            ↪  heuristic ) ) ) ) valid_boards  --`using` parList
            ↪  rpar

            let board_max_value_A = maximumBy ( \(_,(heur_a_1,_))
            ↪  (_,(heur_a_2,_)) -> compare heur_a_1 heur_a_2)
            ↪  board_values

            board_max_value_A

    else
        do
            let valid_boards = map (\k -> M.insert k 'B' gm)
            ↪  (M.keys valid_moves)  `using` parList rseq


            let board_values = parMap rpar ( \grid-> ( grid, (
            ↪  maxPA  (grid) (board) ( max_depth ) ( 1 ) (
            ↪  heuristic ) ) ) ) valid_boards  --`using` parList
            ↪  rpar

            let board_max_value_B = maximumBy ( \(_, (_,heur_b_1)
            ↪  ) (_, (_,heur_b_2 ) ) -> compare heur_b_1
            ↪  heur_b_2  ) board_values

            board_max_value_B

par_playGame :: (Num a1, M.GridMap gm Char,
                       Ord (G.Index (M.BaseGrid gm Char)), Ord
                       ↪  a2, Eq a1,
                       G.Index (M.BaseGrid gm Char) ~ (Int, Int))
                       ↪  =>
```

```
                  Int
                  -> gm Char
                  -> t
                  -> Char
                  -> (gm Char -> t -> Char -> a2)
                  -> (gm Char -> t -> (gm Char -> t -> Char
                  ↪  -> a2) -> a1)
                  -> Int
                  -> IO ()

par_playGame b_size gm board color heur_fn go_fn max_depth = do

    let game_over = go_fn gm board heur_fn

    case game_over of
        0 ->  if color == 'A' then
                do
                    let decision_gm_val = par_minimax_decision gm
                    ↪  board color heur_fn max_depth
                    let decision_gm = fst decision_gm_val
                    putStrLn $ draw b_size decision_gm

                    playGame b_size decision_gm board 'B' heur_fn
                    ↪  go_fn max_depth
              else
                do
                    let decision_gm_val = par_minimax_decision gm
                    ↪  board color heur_fn max_depth
                    let decision_gm = fst decision_gm_val
                    putStrLn $ draw b_size decision_gm

                    playGame b_size decision_gm board 'A' heur_fn
                    ↪  go_fn max_depth
        1 -> putStrLn "A wins"
        2 -> putStrLn "B wins"
        _ -> error "Error in game processing"


playGame :: (Num a1, M.GridMap gm Char,
                  Ord (G.Index (M.BaseGrid gm Char)), Ord a2, Eq
                  ↪  a1,
                  G.Index (M.BaseGrid gm Char) ~ (Int, Int)) =>
                 Int
                 -> gm Char
                 -> t
                 -> Char
```

```
                  -> (gm Char -> t -> Char -> a2)
                  -> (gm Char -> t -> (gm Char -> t -> Char ->
                  ↪  a2) -> a1)
                  -> Int
                  -> IO ()
playGame b_size gm board color heur_fn go_fn max_depth = do

    let game_over = go_fn gm board heur_fn

    case game_over of
        0 ->  if color == 'A' then
                do
                    let decision_gm_val = minimax_decision gm
                    ↪  board color heur_fn max_depth
                    let decision_gm = fst decision_gm_val
                    putStrLn $ draw b_size decision_gm

                    playGame b_size decision_gm board 'B' heur_fn
                    ↪  go_fn max_depth
              else
                do
                    let decision_gm_val = minimax_decision gm
                    ↪  board color heur_fn max_depth
                    let decision_gm = fst decision_gm_val
                    putStrLn $ draw b_size decision_gm

                    playGame b_size decision_gm board 'A' heur_fn
                    ↪  go_fn max_depth
        1 -> putStrLn "A wins"
        2 -> putStrLn "B wins"
        _ -> error "Error in game processing"


basicGameOver :: (M.GridMap gm Char, Ord a, Num p) => gm Char ->
↪  t -> (gm Char -> t -> Char -> a) -> p
basicGameOver gm board heur_fn = case (M.toList
↪  (computeValidMoves gm)) of
                              [] -> if heur_fn gm board 'A' >=
                              ↪  heur_fn gm board 'B' then 1 else
                              ↪  2
                              _  -> 0



-- get keys with value v from gm
-- getKeys gm v = filter (==v) gm

-- countConnected: find the number of pieces that are touching
↪  another piece of the same color on the board
↪  https://www.cs.swarthmore.edu/~bryce/cs63/s16/labs/hex.html
-- for all pieces, +1 if they are touching another piece
-- a piece is touching another piece if a coordinate belonging to
↪  a color is a neighbour of that piece
countConnected :: (M.GridMap gm v, Ord (G.Index g), Eq v, Num a,
↪  G.Grid g, G.Index (M.BaseGrid gm v) ~ G.Index g) => gm v -> g
↪  -> v -> a
countConnected gm board color = sum $ map (\x -> countNeighbor x)
↪  coordinates
        where getCommonColors _ v = v == color -- get kv pairs in
         ↪  grid map that have the same value as color
              coordinates = M.keys $ M.filterWithKey
               ↪  getCommonColors gm -- get list of coordinates
               ↪  belonging to that color (get keys with value v
               ↪  from gm)
              noNeighborIsSameColor = null . flip intersect
               ↪  coordinates . G.neighbours board -- returns
               ↪  boolean on whether a neighbour of the current
               ↪  point is a dot of the same color
              countNeighbor me = if noNeighborIsSameColor me then
               ↪  0 else 1


-- ask user for how much time AI should spend
askTime :: IO Integer
askTime = do
    putStrLn "How deep should the AI have to compute each move?
     ↪  (Default is 2)"
    response <- getLine
    case response of
        "" -> return 2
        s -> case readMaybe s of --parse input
            Just n -> return n --if number, return that number
            Nothing -> askTime --if it aint, ask again

-- ask user for how big the board should be
askSize :: IO Integer
askSize = do
    putStrLn "How big should the Hex board be? (Default is 11)"
    response <- getLine
    case response of
        "" -> return 11
        s -> case readMaybe s of --parse input and make sure that
        ↪  it's int
            Just n -> return n
            Nothing -> askSize

askParallel :: IO Integer
askParallel = do
    putStrLn "Should we run it parallel? (0=no, 1=yes)"
    response <- getLine
    case response of
        "" -> return 0
        s -> case readMaybe s of
                Just n -> return n
                Nothing -> askParallel

main :: IO ()
main = do
    pre_depth <- askTime
    let max_depth = fromIntegral pre_depth
    pre_size <- askSize
    is_parallel <- askParallel
    let size = fromIntegral pre_size
    let hex_b = paraHexGrid size size
    let hex_grid = lazyGridMap hex_b (take (size*size) (repeat
    ↪  '-'))

    if is_parallel == 1 then
        do
            par_playGame size hex_grid hex_b 'A' (countConnected)
            ↪  (basicGameOver) max_depth
            return ()

    else
        do
            playGame size hex_grid hex_b 'A' (countConnected)
            ↪  (basicGameOver) max_depth
            return ()



draw :: (M.GridMap gm Char, G.Index (M.BaseGrid gm Char) ~ (Int,
↪  Int)) => Int -> gm Char -> String
draw size board = unlines [line y | y <- [0..size]] where
    line 0 = (' ':) $ take size ['A'..] >>= (:" ") --draw top
    ↪  legend
```

```
line y = replicate y ' ' ++ --white space for formatting
         replicate (length $ show y) '\b' ++  -- delete extra
         ↪  spaces
         show y ++ -- print current number
         concat [[' ', cell x (y - 1)] | x <- [0..size-1]] --
         ↪  actual line info
cell x y = case M.lookup (x, y) board of
                Just a -> a -- change this w actual data type
                ↪  used by value (this one needs
                ↪  FlexibleContexts)
                Nothing -> '-'
```