Haskell 2048 game

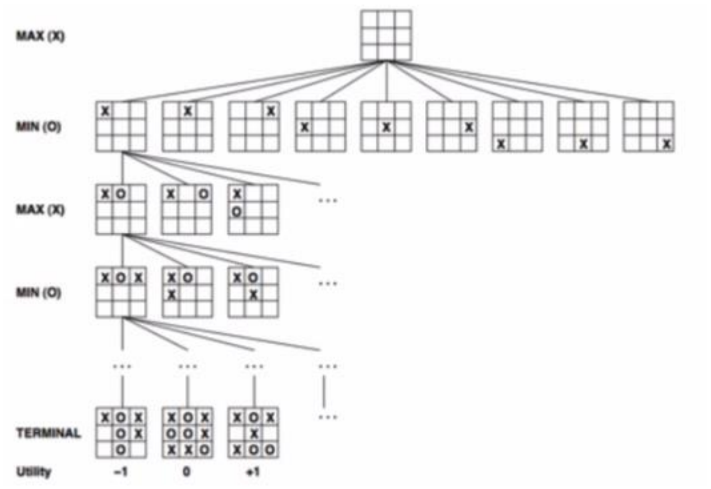
Name: Tri Minh Do
UNI: tmd2142

1. Background

Implement an intelligent agent to automatically play 2048 game and achieve 2048 consistently.

2. Algorithm

We will implement Minimax algorithm with custom heuristics function. We will use search depth of 6 for this project.



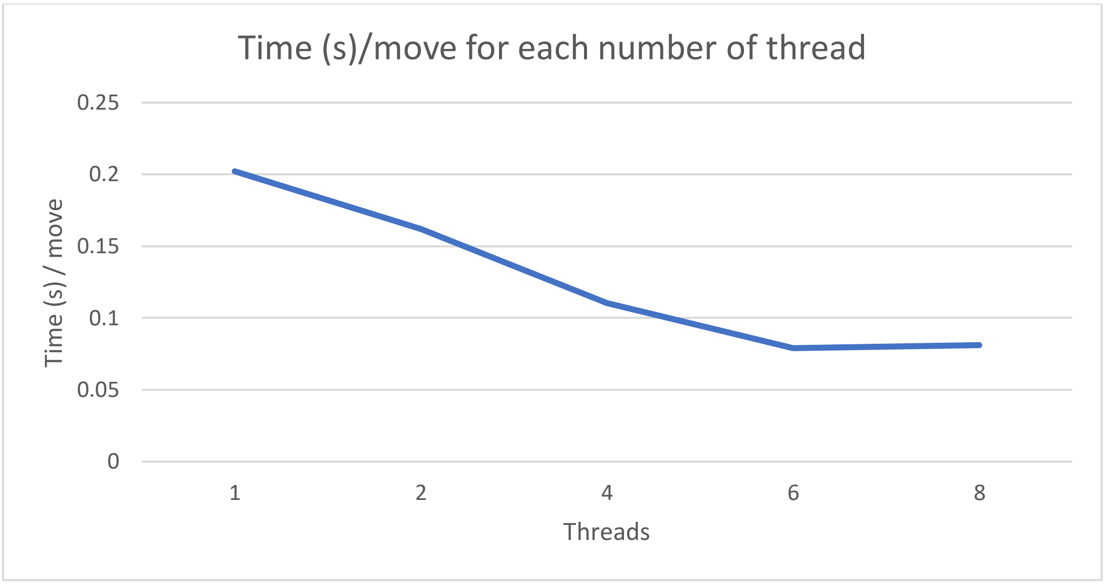
3. Speed up with Parallelism

We will use parMap to run the search in parallel where each thread will search 1 branch in the search tree and pick the highest utility move. We will only start new thread for each tree at the 1st level.

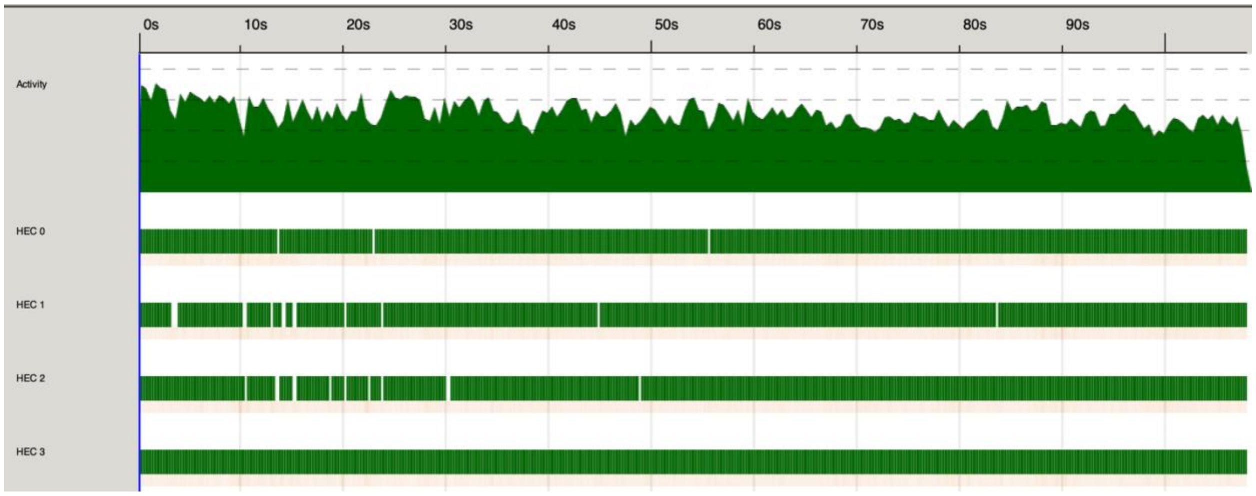
4. Performance Improvement

We will run the game with different number of threads. We will run through the whole game and keep track of the number of moves taken and the overall running time. We then compare the average time it takes to calculate 1 move between different search depth and number of threads.

Score	Time	Moves	Threads	Time (s)/move
2048	206	1019	1	0.202158979
2048	156	963	2	0.161993769
2048	107	971	4	0.110195675
2048	79	1000	6	0.079
2048	121	1493	8	0.081044876



Threadscope with 4 threads (very balanced work on all threads):



5. Code Listing

```
import Data.List     (elemIndices, intercalate, transpose)
import System.Random (randomRIO)
import Data.List (sortOn)
import Control.Parallel.Strategies

-- Each inner list is a row, starting with the top row
-- A 0 is an empty tile
type Board = [[Int]]

data Direction = North | East | South | West

slideLeft :: Board -> Board
slideLeft = map slideRow
  where slideRow [ ] = [ ]
        slideRow [x] = [x]
        slideRow (x:y:zs)
          | x == 0 = slideRow (y : zs) ++ [0]
          | y == 0 = slideRow (x : zs) ++ [0] -- So that things will combine when 0's are between them
          | x == y = (x + y) : slideRow zs ++ [0]
          | otherwise = x : slideRow (y : zs)

slide :: Direction -> Board -> Board
slide North = transpose . slideLeft . transpose
slide East  = map reverse . slideLeft . map reverse
slide South = transpose . map reverse . slideLeft . map reverse . transpose
slide West  = slideLeft

-- Tells us if the player won the game by getting a 2048 tile
completed :: Board -> Bool
completed b = any (elem 2048) b

-- Tells us if the game is over because there are no valid moves left
stalled :: Board -> Bool
stalled b = all stalled' b && all stalled' (transpose b)
  where stalled' row = notElem 0 row && noNeighbors row
        noNeighbors [ ] = True
        noNeighbors [_] = True
        noNeighbors (x:y:zs)
          | x == y    = False
          | otherwise = noNeighbors (y:zs)
```

```
-- Returns tuples of the indices of all of the empty tiles
emptyTiles :: Board -> [(Int, Int)]
emptyTiles = concatMap (uncurry search) . zip [0..3]
  where search n = zip (replicate 4 n) . elemIndices 0

-- Given a point, update replaces the value at the point on the board with the given value
updateTile :: (Int, Int) -> Int -> Board -> Board
updateTile (rowI, columnI) value = updateIndex (updateIndex (const value) columnI) rowI
  where updateIndex fn i list = take i list ++ fn (head $ drop i list) : tail (drop i list)

-- Adds a tile to a random empty spot.
-- 90% of the time the tile is a 2, 10% of the time it is a 4
addTile :: Board -> IO Board
addTile b = do
  let tiles = emptyTiles b
  newPoint <- randomRIO (0, length tiles - 1) >>= return . (tiles !!)
  newValue <- randomRIO (1, 10 :: Int) >>= return . \x -> if x == 1 then 4 else 2
  return $ updateTile newPoint newValue b

-- Our main game loop
gameloop :: Int -> Board -> IO ()
gameloop count b = do
    putStrLn "--------------------"
    putStrLn $ boardToString b
    putStrLn $ "Max tile: " ++ (show $ maxTile b)
    putStrLn $ "Move count: " ++ (show $ count)
    putStrLn "--------------------"
    if stalled b
        then putStrLn "Game over."
        else if completed b
            then putStrLn "You won!"
            else do
             b1 <- getMove b
             addTile b1 >>= gameloop (count + 1)


-- Board pretty printing
boardToString :: Board -> String
boardToString = init . unlines . map (vertical . map (pad . showSpecial))
  where vertical = ('|' :) . (++ "|") . intercalate "|"
        showSpecial 0 = ""
        showSpecial n = show n
        pad s = replicate (4 - length padTemp) ' ' ++ padTemp
```

```haskell
        where padTemp = s ++ if length s < 3 then " " else ""

-- Available Move
getAvailableMoves :: Board -> [Board]
getAvailableMoves b = filter (\x -> x /= b) [slide d b | d <- [North, West, South, East]]

getRandomizeBoard:: Board -> Int -> [(Board, Int)]
getRandomizeBoard b val = sortOn (\(_,d) -> -d) [(updateTile (x,y) val b, diff b x y) | (x,y) <-
emptyTiles b]

--Max Tile
maxTile :: Board -> Int
maxTile b = maximum $ map maximum b

-- Get next move by using MiniMax algorithm
getMove :: Board -> IO Board
getMove b = do
  let res = parMap rpar id [minimize m m 1 | m <- getAvailableMoves b]
      (board,_) = getBestMove b (-999999) res
  return board

getBestMove :: Board -> Int -> [(Board, Board, Int)] -> (Board,Int)
getBestMove maxBoard maxUtility [] = (maxBoard,maxUtility)
getBestMove maxBoard maxUtility ((_,b,u):xs)
  | u > maxUtility = getBestMove b u xs
  | otherwise = getBestMove maxBoard maxUtility xs

maximize :: Board -> Int -> (Board, Int)
maximize b depth
  | length moves == 0 = (b,0)
  | depth > 7 = (b,eval b)
  | otherwise = getBestMove maxChild maxUtility parResult
    where moves = getAvailableMoves b
          parResult = [minimize m m (depth + 1) | m <- moves]
          maxChild = b
          maxUtility = -999999


minimize :: Board -> Board -> Int -> (Board, Board, Int)
minimize b org depth
  | length moves == 0 || depth > 7 = (b, org, eval b)
  | otherwise = subMinimize moves minChild org minUtility depth
    where moves = moves_2 ++ moves_4
          moves_2 = [fst x | x <- rm_2]
          moves_4 = [fst x | x <- rm_4]
          rm_2 = getRandomizeBoard b 2
          rm_4 = getRandomizeBoard b 4
          minChild = b
          minUtility = 999999


subMinimize :: [Board] -> Board -> Board -> Int -> Int -> (Board, Board, Int)
subMinimize [] minChild org minUtility _ = (minChild, org, minUtility)
subMinimize (m:ms) minChild org minUtility depth
  | utility < minUtility = subMinimize ms m org utility depth
  | otherwise = subMinimize ms minChild org minUtility depth
    where utility = snd $ maximize m (depth + 1)


-- heuristic function
eval :: Board -> Int
eval b
  | (maxTile b) <= 512 = sum $ map (\(x,y) -> sum $ zipWith (*) x y) c
  | otherwise = sum $ map (\(x,y) -> sum $ zipWith (*) x y) d
    where
      c = zip [[21,8,3,3],[9,5,2],[4,3]] b
      d = zip [[19,9,5,3],[8,4,2],[3]] b

diff :: Board -> Int -> Int -> Int
diff b i j =  a + c
  where a = (diffup b i j) + (diffdown b i j)
        c = (diffleft b i j) + (diffright b i j)

diffup :: Board -> Int -> Int -> Int
diffup b i j
  | i > 0 = abs $ ((b !! i) !! j) - ((b !! (i-1)) !! j)
  | otherwise = 0
diffdown :: Board -> Int -> Int -> Int
diffdown b i j
  | i < 3 = abs $ ((b !! i) !! j) - ((b !! (i+1)) !! j)
  | otherwise = 0
diffleft :: Board -> Int -> Int -> Int
diffleft b i j
  | j > 0 = abs $ ((b !! i) !! j) - ((b !! i) !! (j-1))
  | otherwise = 0
diffright :: Board -> Int -> Int -> Int
diffright b i j
  | j < 3 = abs $ ((b !! i) !! j) - ((b !! i) !! (j+1))
```

```haskell
  | otherwise = 0


main :: IO ()
main = do
   let board = replicate 4 (replicate 4 0)
   b1 <- addTile board >>= addTile -- Add two tiles randomly and start the game!
   gameloop 0 b1
```