yc3858 Yufan Chen

# Introduction

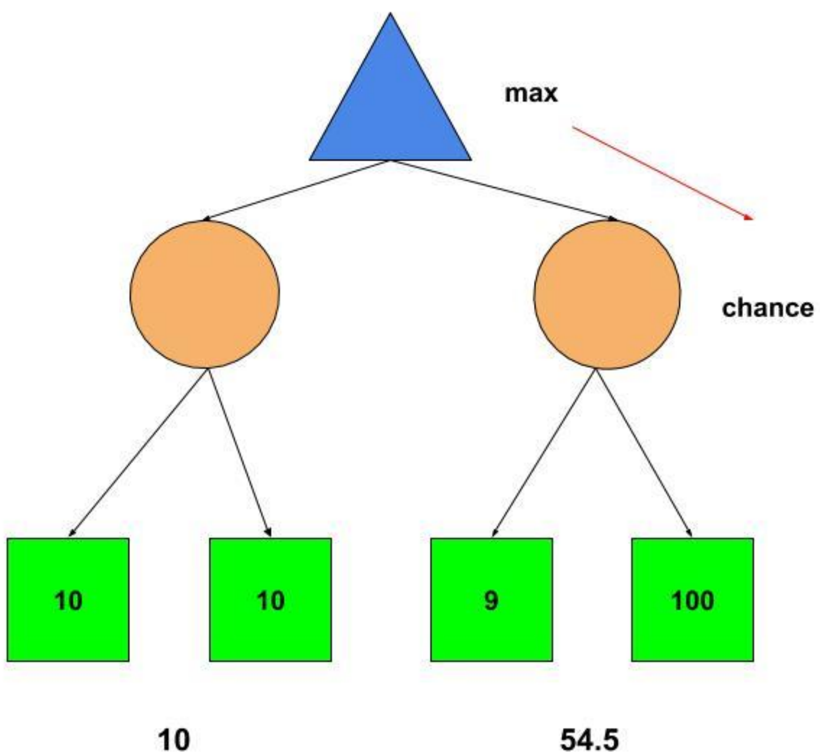
This project uses Expectimax algorithm to implement an AI playing the game 2048. The goal of the AI is to generate an 2048 tile in the board.

## 2048

*2048* is played on a plain 4×4 grid, with numbered tiles that slide when a player moves them using the four arrow keys. Every turn, a new tile appears in an empty spot on the board with a probability of 0.9 to be 2 and 0.1 to be 4. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided.

## Expectimax

The expectimax algorithm is a variation of the Minimax algorithm. While Minimax assumes that the adversary(the minimizer) plays optimally, the Expectimax doesn't. This is useful for modeling environments where adversary agents are not optimal, or their actions are based on chance.



For example, in the search tree above, the minimax agent will choose the left child node because 10 > 9. However, if we are using expectimax, and if we assume that the adversary is not smart at all and can only make the decision randomly, then the expectimax agent will choose the right child node. This is because the expected result value of choosing the left one is $10 * 0.5 + 10 * 0.5 = 10$, while the expected result value of choosing the right one is $9 * 0.5 + 100 * 0.5 = 54.5$.

# Implementation

## Sequential Implementation

We can split the program into several different components:

### Game Implementation

First, we need to make a playable game board, which accepts the move command, move and merge tiles, and randomly spawn a new tile.

#### Moving and Merging the Tiles

For moving towards left, we can simply iterate each row of the board, filter out all the empty slots, and then iterate through each tile in a single row. If the current tile has the same value with the next one, remove both of them and then add the merged one into the original place.

```
moveLeft :: [[Int]] -> [[Int]]
moveLeft board = map moveRow board where
  moveRow :: [Int] -> [Int]
  moveRow row = let merged = merge [x | x <- row, x /= 0] [] in
    reverse $ replicate (4-length merged) 0 ++ merged
  merge :: [Int] -> [Int] -> [Int]
  merge [] acc = acc
  merge [x] acc = x:acc
  merge (f:s:xs) acc
    | f == s = merge xs (f*2:acc)
    | otherwise = merge (s:xs) (f:acc)
```

For moving towards other directions, just rotate the board so that we can reuse the moveLeft function above, and then rotate the board back.

```haskell
transpose :: [[Int]] -> [[Int]]
transpose [r1, r2, r3, r4] = map (\(x1,x2,x3,x4) -> [x1,x2,x3,x4]) $ zip4 r1 r2 r3 r4
transpose _ = error "can not transpose a non 4x4 matrix"

moveDown :: [[Int]] -> [[Int]]
moveDown board = reverse $ moveUp $ reverse board

moveUp :: [[Int]] -> [[Int]]
moveUp board = transpose $ moveLeft $ transpose board

moveRight :: [[Int]] -> [[Int]]
moveRight board = map reverse $ moveLeft $ map reverse board
```

#### Spawning a New Tile

According to the probability model, it has 90% percent to generate a 2 and 10% to generate 4. The position is chosen randomly among all empty slots.

Therefore, we can first collect indices of all empty slots, and randomly choose one slot among them, and then choose the value for that new tile under the probability model.

```haskell
fill :: [[Int]] -> Int -> Int -> Int -> [[Int]]
fill board x y v = prev ++ (newRow : next) where
  (prev, row:next) = splitAt x board
  newRow = take y row ++ v : drop (y+1) row

spawn :: [[Int]] -> IO [[Int]]
spawn board = do
  let slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row, val == 0]
  case length slots of
    0 -> pure board
    _ -> do
      val <- randomRIO (1, 10::Int) >>= pure . (\x -> if x == 1 then 4 else 2)
      (xpos, ypos) <- randomRIO (0, length slots-1) >>= pure . (slots !!)
      return $ fill board xpos ypos val

```

## Agent Implementation

To implement the expectimax agent, we need a heuristic policy and a search function.

#### Heuristic Function

I used the heuristic function from [https://stackoverflow.com/a/28824788](https://stackoverflow.com/a/28824788).

Intuitively, we want to keep the largest tile at the corner, and organize tiles descendingly in a sort of snake. For example, this is an ideal situation:

```
512        256        4    4
1024        128        8    2
2048        64         8    2
4096         16        16    0
```

The head of the "snake" is at bottom-left, and the tail is at bottom-right.

Therefore, under the implementation, we set the top-left slot to have the largest weight. The weights descend exponentially from the snake head to the snake tail.

```haskell
snake = map fromIntegral $ concat $ map (\(i, row) -> if i `mod` (2::Int) == 0 then reverse row else row) $ zip [0..] $ transpose board
snakeSumHeu = foldl (\acc (i, x) -> acc + x/10**i) 0 $ zip [0..] snake
```

To fix the snake head to the bottom-left corner, here we minus a penalty value to the final heuristic value if the bottom-left tile is not the largest tile in the board:

```haskell
snakeMax = maximum snake
snakeHeadHeu = if head snake == snakeMax then 0 else (abs $ head snake - snakeMax) ** 2
```

Therefore, the final heuristic value is:

```haskell
snakeSumHeu - snakeHeadHeu
```

**Search Function**

The search function accepts the board, the current search depth, and whether it is now on the player's move.

If the search depth is 0, simply evaluates the current board and returns the heuristic value.

If currently it is on the player's move, search through all valid actions (moveLeft, moveUp, moveDown, moveRIght), returns the best one.

If currently it is on the system's move, search through all the possible ways to spawn a new tile, then returns the expected heuristic value according to the probability model.

```haskell
search :: [[Int]] -> Int -> Bool -> Double
search board depth onMove
  | depth == 0 || (onMove && not (canMove board)) = heuristic board
  | onMove = maximum $ heuristic board:map (\action -> search (action board) (depth-1) False) actions
  | otherwise = sum $ map fillOne choices
  where
    fillOne (x,y,(v, p)) = p * (search (fill board x y v) (depth-1) True) / fromIntegral (length slots)
    choices = [(x, y, vp) | (x, y) <- slots, vp <- [(2, 0.9), (4, 0.1)]::[(Int, Double)] ]
    slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row, val == 0]
    actions = [moveUp, moveLeft, moveRight, moveDown]

```

## Main Function

Finally, we need a main function to generate the initial board, use the agent to do search, take the move action and then output the current game board. We end this game either when 2048 is generated or there is no way to move the tiles.

```haskell
play :: [[Int]] -> IO ()
play board
  | elem 2048 (concat board) = printBoard board >> putStrLn "Success."
  | canPlay board = do
    printBoard board
    case elem 0 (concat nextBoard) of
      False -> putStrLn "Lost."
      _ ->
        spawn nextBoard >>= play
  | otherwise = printBoard board >> putStrLn "Lost."
  where
    nextBoard = snd $ maximumBy (compare `on` fst) $ map dbfunc actions where
      dbfunc = \action -> helper action
      helper = \action -> let next = action board in (search next <maximum search depth> False, next)
    actions = [moveUp, moveLeft, moveRight, moveDown]

main :: IO ()
main = pure [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]] >>= spawn >>= spawn >>= play

```

### Maximum Search Depth

Now we need to pick a good maximum search death to get the best tradeoff between success rate and running time. In order to find that, we run the program 1,000 times under different maximum search depth, and record the corresponding running time and success rate:

(Note: only successful runs are counted into average running time )

| Maximum Search Depth | Success Rate | Average Running Time |
|---|---|---|
| 1 | 0 | 0.08s |
| 2 | 85.7% | 0.7s |
| 3 | 87.9% | 4.5s |
| 4 | 96.4% | 29s |
| 5 | 99.3% | 287s |

According to the result, we picked 4 to be the maximum search depth because it gives a very good success rate as well as an acceptable running time. The rest parts of this report will use 4 as the maximum search depth.

# Parallel Implementation

# Attempt 1

Parallelize every search steps by using `parMap rpar` instead of `map` to transfer to the next level of the search tree.

```
search :: [[Int]] -> Int -> Bool -> Double
search board depth onMove
  | depth == 0 || (onMove && not (canMove board)) = heuristic board
  | onMove = maximum $ heuristic board:(parMap rpar) (\action -> search (action board) (depth-1) False) actions
  | otherwise = sum $ parMap rpar $ fillOne choices
  where
    fillOne (x,y,(v, p)) = p * (search (fill board x y v) (depth-1) True) / fromIntegral (length slots)
    choices = [(x, y, vp) | (x, y) <- slots, vp <- [(2, 0.9), (4, 0.1)]::[(Int, Double)] ]
    slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row, val == 0]
    actions = [moveUp, moveLeft, moveRight, moveDown]

play :: [[Int]] -> IO ()
play board
  | elem 2048 (concat board) = printBoard board >> putStrLn "Success."
  | canPlay board = do
    printBoard board
    case elem 0 (concat nextBoard) of
      False -> putStrLn "Lost."
      _ ->
        spawn nextBoard >>= play
  | otherwise = printBoard board >> putStrLn "Lost."
  where
    nextBoard = snd $ maximumBy (compare `on` fst) $ parMap rpar helper actions where
      helper = \action -> let next = action board in (search next 4 False, next)
    actions = [moveUp, moveLeft, moveRight, moveDown]
```

The running result of sparks:

```
SPARKS: 37093731 (835503 converted, 0 overflowed, 0 dud, 20232203 GC'd, 16026025 fizzled)
```

From this result we can see that most of the sparks are garbage collected. Therefore we can conclude the tasks are too fine-grained and we need to make it more coarse-grained.

## Attempt 2

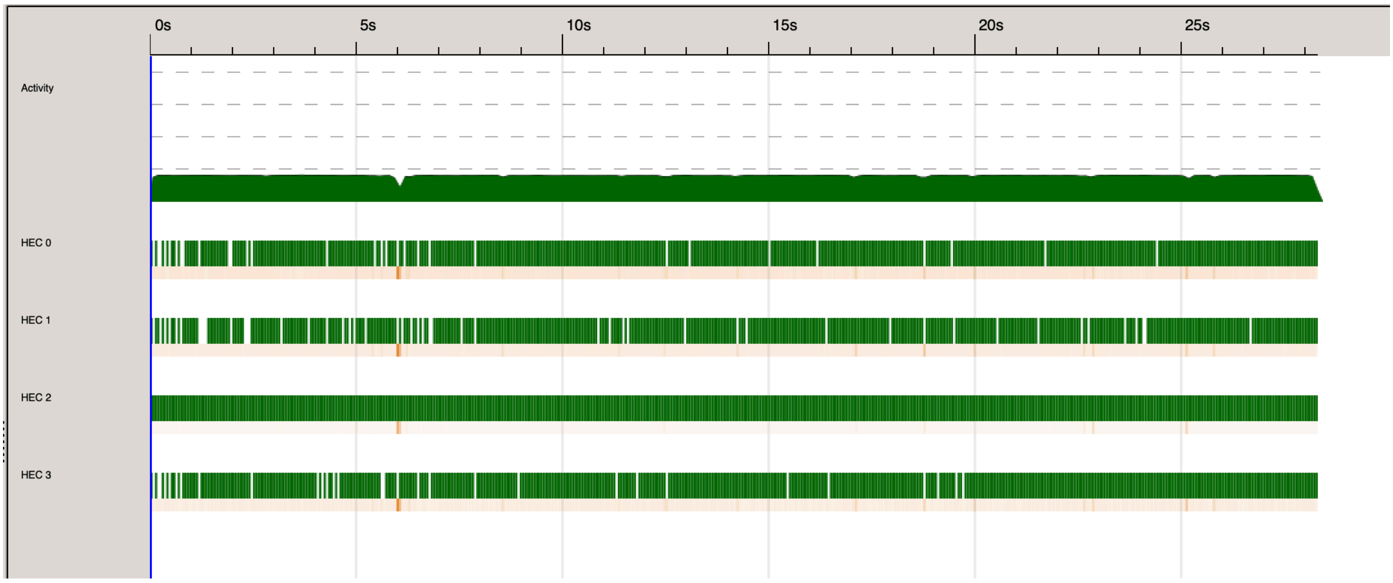
Only parallelize the top search step:

```haskell
play :: [[Int]] -> IO ()
play board
  | elem 2048 (concat board) = printBoard board >> putStrLn "Success."
  | canPlay board = do
    printBoard board
    case elem 0 (concat nextBoard) of
      False -> putStrLn "Lost."
      _ ->
        spawn nextBoard >>= play
  | otherwise = printBoard board >> putStrLn "Lost."
  where
    nextBoard = snd $ maximumBy (compare `on` fst) $ parMap rpar helper actions where
      helper = \action -> let next = action board in (search next 4 False, next)
    actions = [moveUp, moveLeft, moveRight, moveDown]
```

The spark result is:

```
SPARKS: 3816 (2861 converted, 0 overflowed, 0 dud, 949 GC'd, 6 fizzled)
```

We can see the sparks are generated normally. However from threadscope:




The workload is very uneven. From the activity row we can see that there is only one core working almost at any time.

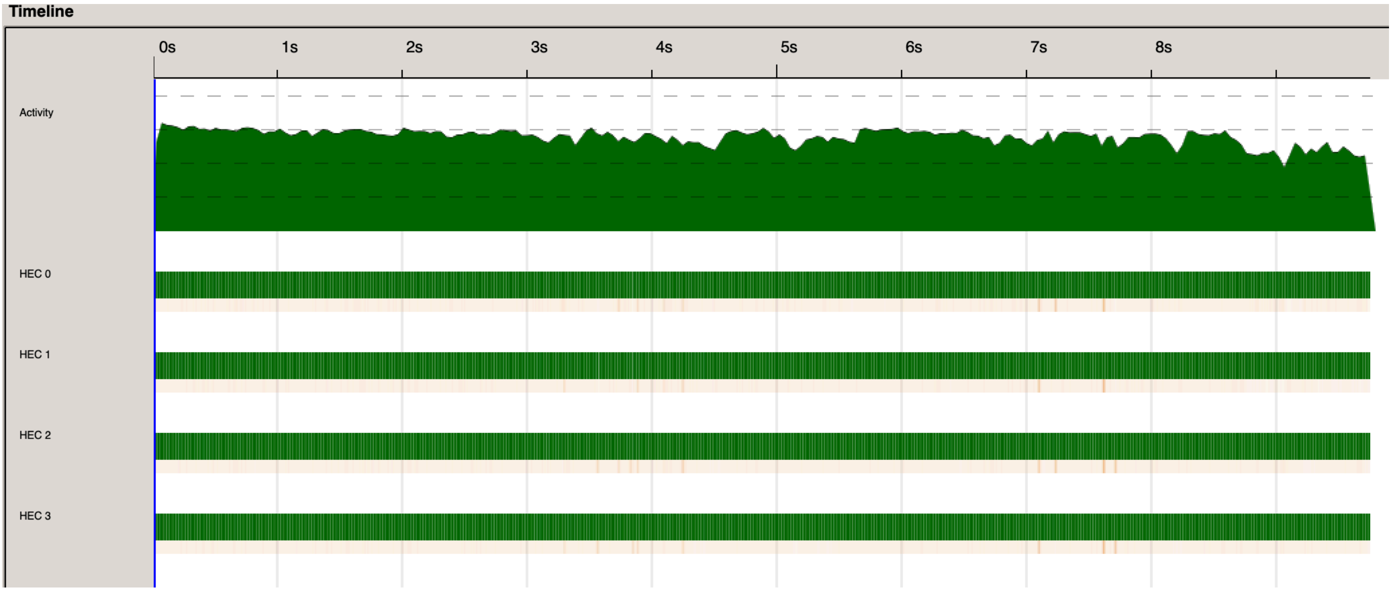
## Attempt 3

Only parallelize the search steps of which onMove is false and the depth is greater than 3:

```haskell
search :: [[Int]] -> Int -> Bool -> Double
search board depth onMove
  | depth == 0 || (onMove && not (canMove board)) = heuristic board
  | onMove = maximum $ heuristic board:map (\action -> search (action board) (depth-1) False) actions
  | otherwise = sum $ mapF fillOne choices
  where
    mapF = if depth > 3 then parMap rpar else map
    fillOne (x,y,(v, p)) = p * (search (fill board x y v) (depth-1) True) / fromIntegral (length slots)
    choices = [(x, y, vp) | (x, y) <- slots, vp <- [(2, 0.9), (4, 0.1)]::[(Int, Double)] ]
    slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row, val == 0]
    actions = [moveUp, moveLeft, moveRight, moveDown]
```

The sparks look great:

```
SPARKS: 106512 (97231 converted, 0 overflowed, 0 dud, 5226 GC'd, 4055 fizzled)
```




From threadscope, on average the speedup is about 3x. The workload is distributed evenly.

## Other Attempts

I also tried other attempts to adjust the parallel components, including moving the board in parallel, calculating the heuristic function in parallel, adjust the depth for allowing parallel and so on. However the best result I got is the result from attempt 3.

# Performance Evaluation

We evaluate the performance via two dimensions: success rate and running time.

To make accurate result, we run the program 1,00 times and get the average running time.

Note: only successful runs are counted into the average running time.

## Testing Environment

- MacBook Pro Mid 2015
- CPU: 2.2 GHz Quad-Core Intel Core i7
- Memory: 16 GB 1600 MHz DDR3

## Parallel Evaluation

### Testing Parameter

- parallel implementation with using 1, 2, 4, 8, 16 threads
- haskell sequential implementation
- python sequential implementation

| Program | Threads | Average Running Time | Speedup |
|---|---|---|---|
| Parallel Implementation | 1 | 25.9s | 1.01 |
| | 2 | 14.4s | 1.82 |
| | 4 | 9.5s | 2.76 |
| | 8 | 9.3s | 2.82 |
| | 16 | 9.9s | 2.65 |
| | 32 | 9.4s | 2.79 |
| Sequential Implementation | | 26.3s | |
| Python Sequential Implementation | | 385s | |

## Algorithm Evaluation

To evaluate whether expectimax is better than minimax with alpha-beta pruning with regard to 2048 game, we use the minimax with alpha-beta pruning implementation from 2048-puzzle1 and 2048-puzzle2.

In order to do a fair comparison, I modified the heuristic function of the implementations above to make it the same as the expectimax one. The results of using the same heuristic function and using the original heuristic function are both recorded.

The commands for running them are as follow:

```
# 2048-puzzle1
./Haskell2048 +RTS -ls -N4
# 2048-puzzle2
stack exec pf2048-exe <depth> mixed +RTS -N4 -s
```

| Source | Heuristic Function | Max Depth | Average Running Time | Success Rate |
|---|---|---|---|---|
| 2048-puzzle1 | Same | 4 | 4.5s | 15% |
| | | 5 | 39.9s | 39% |
| | Original | 4 | 5.6s | 27% |
| | | 5 | 40.3s | 74% |
| 2048-puzzle2 | Same | 5 | 2.5s | 27% |
| | | 6 | 11.61s | 17% |
| | Original | 5 | 7.7s | 25% |
| | | 6 | 64.3s | 36% |

From the result, we can see that under the same running time, the success rate of using expectimax is much better than minimax ones. Under the same success rate, the running time of expectimax is much better than the minimax ones. We conclude that expectimax has a better performance compared to minimax in the game 2048.

## Reference


1. http://www.cs.columbia.edu/~sedwards/classes/2019/4995-fall/reports/2048-puzzle1.pdf
2. http://www.cs.columbia.edu/~sedwards/classes/2019/4995-fall/reports/2048-puzzle2.pdf
3. https://github.com/gjdanis/2048
4. https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22498940#22498940


# Appendix

## Sequential Implementation Code

```
import Data.List(zip4, maximumBy)
import System.Random(randomRIO)
import System.Console.ANSI(clearScreen)
import Data.Function(on)

transpose :: [[Int]] -> [[Int]]
transpose [r1, r2, r3, r4] = map (\(x1,x2,x3,x4) -> [x1,x2,x3,x4]) $ zip4 r1 r2 r3 r4
transpose _ = error "can not transpose a non 4x4 matrix"

moveDown :: [[Int]] -> [[Int]]
moveDown board = reverse $ moveUp $ reverse board

moveUp :: [[Int]] -> [[Int]]
```

```haskell
moveUp board = transpose $ moveLeft $ transpose board

moveRight :: [[Int]] -> [[Int]]
moveRight board = map reverse $ moveLeft $ map reverse board

moveLeft :: [[Int]] -> [[Int]]
moveLeft board = map moveRow board where
  moveRow :: [Int] -> [Int]
  moveRow row = let merged = merge [x | x <- row, x /= 0] [] in
    reverse $ replicate (4-length merged) 0 ++ merged
  merge :: [Int] -> [Int] -> [Int]
  merge [] acc = acc
  merge [x] acc = x:acc
  merge (f:s:xs) acc
    | f == s = merge xs (f*2:acc)
    | otherwise = merge (s:xs) (f:acc)

canMove :: [[Int]] -> Bool
canMove board = any checkRow board where
  checkRow :: [Int] -> Bool
  checkRow row = any (\(a, b) -> a == b || a == 0 || b == 0) $ zip row (tail row)


fill :: [[Int]] -> Int -> Int -> Int -> [[Int]]
fill board x y v = prev ++ (newRow : next) where
  (prev, row:next) = splitAt x board
  newRow = take y row ++ v : drop (y+1) row

spawn :: [[Int]] -> IO [[Int]]
spawn board = do
  let slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row,
val == 0]
  case length slots of
    0 -> pure board
    _ -> do
      val <- randomRIO (1, 10::Int) >>= pure . (\x -> if x == 1 then 4 else 2)
      (xpos, ypos) <- randomRIO (0, length slots-1) >>= pure . (slots !!)
      return $ fill board xpos ypos val

printBoard :: [[Int]] -> IO ()
printBoard board = clearScreen >> mapM_ printRow board >> putStrLn "" where
    printRow row = putStrLn $ tail $ foldr printNum "" $ map show row
    printNum num out = (replicate (5 - (length num)) ' ')++num++out


heuristic :: [[Int]] -> Double
heuristic board
  | canMove board = snakeSumHeu - snakeHeadHeu
  | otherwise = read "-Infinity" where
```

```haskell
    snakeHeadHeu = if head snake == snakeMax then 0 else (abs $ head snake - snakeMax) ** 2
    snakeSumHeu = foldl (\acc (i, x) -> acc + x/10**i) 0 $ zip [0..] snake
    snakeMax = maximum snake
    snake = map fromIntegral $ concat $ map (\(i, row) -> if i `mod` (2::Int) == 0 then reverse row else row) $ zip [0..] $ transpose board

search :: [[Int]] -> Int -> Bool -> Double
search board depth onMove
  | depth == 0 || (onMove && not (canMove board)) = heuristic board
  | onMove = maximum $ heuristic board:map (\action -> search (action board) (depth-1) False) actions
  | otherwise = sum $ map fillOne choices
  where
    fillOne (x,y,(v, p)) = p * (search (fill board x y v) (depth-1) True) / fromIntegral (length slots)
    choices = [(x, y, vp) | (x, y) <- slots, vp <- [(2, 0.9), (4, 0.1)]::[(Int, Double)] ]
    slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row, val == 0]
    actions = [moveUp, moveLeft, moveRight, moveDown]

canPlay :: [[Int]] -> Bool
canPlay board = (canMove board) || (canMove $ transpose board)

play :: [[Int]] -> IO ()
play board
  | elem 2048 (concat board) = printBoard board >> putStrLn "Success."
  | canPlay board = do
    printBoard board
    case elem 0 (concat nextBoard) of
      False -> putStrLn "Lost."
      _ ->
        spawn nextBoard >>= play
  | otherwise = printBoard board >> putStrLn "Lost."
  where
    nextBoard = snd $ maximumBy (compare `on` fst) $ map helper actions where
      helper = \action -> let next = action board in (search next 4 False, next)
    actions = [moveUp, moveLeft, moveRight, moveDown]

main :: IO ()
main = do
  pure [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]] >>= spawn >>= spawn >>= play
```

## Parallel Implementation Code

```
import Data.List(zip4, maximumBy)
import System.Random(randomRIO)
import System.Console.ANSI(clearScreen)
import Data.Function(on)
import Control.Parallel.Strategies

transpose :: [[Int]] -> [[Int]]
transpose [r1, r2, r3, r4] = map (\(x1,x2,x3,x4) -> [x1,x2,x3,x4]) $ zip4 r1 r2 r3
r4
transpose _ = error "can not transpose a non 4x4 matrix"

moveDown :: [[Int]] -> [[Int]]
moveDown board = reverse $ moveUp $ reverse board

moveUp :: [[Int]] -> [[Int]]
moveUp board = transpose $ moveLeft $ transpose board

moveRight :: [[Int]] -> [[Int]]
moveRight board = map reverse $ moveLeft $ map reverse board

moveLeft :: [[Int]] -> [[Int]]
moveLeft board = map moveRow board where
  moveRow :: [Int] -> [Int]
  moveRow row = let merged = merge [x | x <- row, x /= 0] [] in
    reverse $ replicate (4-length merged) 0 ++ merged
  merge :: [Int] -> [Int] -> [Int]
  merge [] acc = acc
  merge [x] acc = x:acc
  merge (f:s:xs) acc
    | f == s = merge xs (f*2:acc)
    | otherwise = merge (s:xs) (f:acc)

canMove :: [[Int]] -> Bool
canMove board = any checkRow board where
  checkRow :: [Int] -> Bool
  checkRow row = any (\(a, b) -> a == b || a == 0 || b == 0) $ zip row (tail row)


fill :: [[Int]] -> Int -> Int -> Int -> [[Int]]
fill board x y v = prev ++ (newRow : next) where
  (prev, row:next) = splitAt x board
  newRow = take y row ++ v : drop (y+1) row

spawn :: [[Int]] -> IO [[Int]]
spawn board = do
  let slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row,
val == 0]
```

```haskell
  case length slots of
    0 -> pure board
    _ -> do
      val <- randomRIO (1, 10::Int) >>= pure . (\x -> if x == 1 then 4 else 2)
      (xpos, ypos) <- randomRIO (0, length slots-1) >>= pure . (slots !!)
      return $ fill board xpos ypos val

printBoard :: [[Int]] -> IO ()
printBoard board = clearScreen >> mapM_ printRow board >> putStrLn "" where
    printRow row = putStrLn $ tail $ foldr printNum "" $ map show row
    printNum num out = (replicate (5 - (length num)) ' ')++num++out


heuristic :: [[Int]] -> Double
heuristic board
  | canMove board = snakeSumHeu - snakeHeadHeu
  | otherwise = read "-Infinity" where
    snakeHeadHeu = if head snake == snakeMax then 0 else (abs $ head snake - snakeMax) ** 2
    snakeSumHeu = foldl (\acc (i, x) -> acc + x/10**i) 0 $ zip [0..] snake
    snakeMax = maximum snake
    snake = map fromIntegral $ concat $ map (\(i, row) -> if i `mod` (2::Int) == 0 then reverse row else row) $ zip [0..] $ transpose board

search :: [[Int]] -> Int -> Bool -> Double
search board depth onMove
  | depth == 0 || (onMove && not (canMove board)) = heuristic board
  | onMove = maximum $ heuristic board:map (\action -> search (action board) (depth-1) False) actions
  | otherwise = sum $ mapF fillOne choices
  where
    mapF = if depth > 3 then parMap rpar else map
    fillOne (x,y,(v, p)) = p * (search (fill board x y v) (depth-1) True) / fromIntegral (length slots)
    choices = [(x, y, vp) | (x, y) <- slots, vp <- [(2, 0.9), (4, 0.1)]::[(Int, Double)] ]
    slots = [ (x, y) | (x, row) <- zip [0..] board, (y, val) <- zip [0..] row, val == 0]
    actions = [moveUp, moveLeft, moveRight, moveDown]

canPlay :: [[Int]] -> Bool
canPlay board = (canMove board) || (canMove $ transpose board)

play :: [[Int]] -> IO ()
play board
  | elem 2048 (concat board) = printBoard board >> putStrLn "Success."
  | canPlay board = do
    printBoard board
    case elem 0 (concat nextBoard) of
```

```haskell
      False -> putStrLn "Lost."
      _ ->
        spawn nextBoard >>= play
  | otherwise = printBoard board >> putStrLn "Lost."
  where
    nextBoard = snd $ maximumBy (compare `on` fst) $ map helper actions where
      helper = \action -> let next = action board in (search next 4 False, next)
    actions = [moveUp, moveLeft, moveRight, moveDown]

main :: IO ()
main = do
  pure [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]] >>= spawn >>= spawn >>= play
```