Parallel Expectimax: A 2048 Solver

Gabriel Clinger | Matthew Ottomano

Abstract

This project focuses on the game 2048, a fun pastime that you can download on your phone. Several implementations of the game have been implemented in different programming languages, including Haskell, however not many parallel implementations of a solver for the game have been implemented. We focus on parallelizing the minimax algorithm and running metrics for performance.

Introduction

This project aims to test Haskell's parallel programming performance compared to its sequential performance. To do so we took a familiar algorithm we have implemented in python, converted it to haskell and added parallelism to it. We took a well known game; 2048 and used an adversarial tree like search algorithm to win the game. The tree-like nature of the search algorithm gave us an opportunity to parallelise the search and reduce the search time significantly.

2048

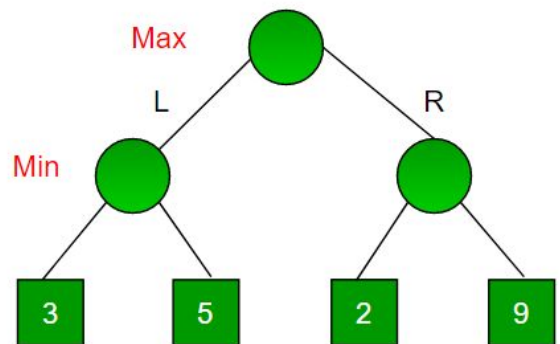
2048 is a single player sliding block puzzle game in which the objective of the game is to slide numbered tiles on a grid to combine them in order to generate a tile with the value of 2048, or even higher.

The game is played on a 4x4 grid with numbered tiles that slide in one of four directions where the tiles slide as far as possible in the chosen direction until

they are stopped by another tile or the edge, and if two of the same numbered tiles collide, they merge into a tile with double the value. If a move causes three consecutive tiles of the same value to slide together only two tiles will combine. If all four spaces in a row or column are the same value the two first and two last will combine. Every turn a new tile will randomly appear in an empty spot on the board with a 90% chance of being a 2 and 10% chance of being a 4. In this version the game is won once a tile reaches a value of 2048.

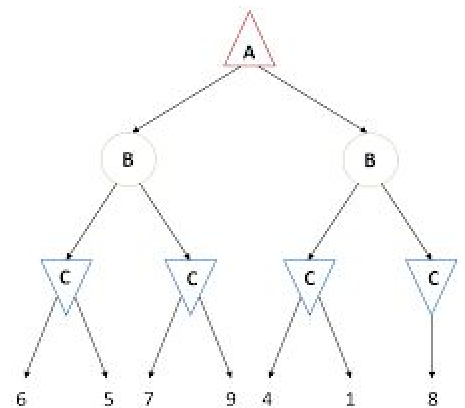
Minimax

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player that assumed the opponent is playing optimally: that is that while the maximizer will chose a move with the most utility, the minimizer/opponent will chose the opposite, the move with the least amount of utility.
Each move is made essentially by anticipating the future moves, with the assumption that the opponent is making the optimal move to win the game. Since 2048 has an element of chance to the game, because the computer move is a randomly inserted tile, it cannot be assumed that the opponent is acting optimally.



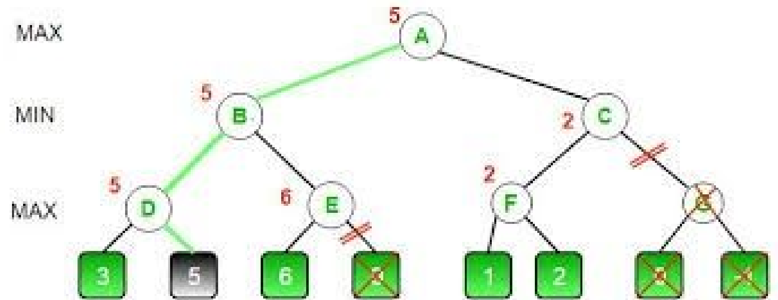
Expectimax

Expectimax search algorithm is a game theory algorithm used to maximize the expected utility instead. While minimax assumes that the opponent plays optimally, expectimax doesn't. Since in 2048 after each move a tile is placed randomly but with a 90% chance of a 2 and 10% chance of a 4 so each move must be made by taking an average of the anticipate expected utility, which is, in this case, 0.9 times the utility of a move with a 2 tile inserted + 0.1 times the utility of a move with a 4 tile inserted.



Alpha Beta Pruning

Alpha beta pruning is a method of decreasing the number of nodes that are evaluated by the minimax algorithm in its search tree. Alpha beta pruning decreases the branching factor thus allowing for a faster and deeper search. Alpha beta Pruning stops evaluating a move when there a possibility for a move has been found that is worse than a previously seen move, in which case the move not longer needs to be evaluated. When used with a minimax like search algorithm alpha beta pruning returns the same move but prunes away branches that cannot influence the final move



Alpha beta pruning alone decreased the running time from around 760 seconds to about 95s second sequentially.

Heuristics

Heuristics are a technique used for problem solving more efficiently and quickly. The heuristics we chose were inspired by a stanford research paper, http://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf, which outlines a number of heuristics. We focused on using four primary heuristics which we found to be most useful in achieving a high score with a limited amount of depth search: Weight Matrix, Empty tiles, Monotonicity, and Smoothness.

Weight Matrix

A weighted matrix is simply a grid with higher weighing tiles on one part of the grid, giving more weight to some tiles and less to others. By multiplying our game grid by a weighted matrix it results in giving grids with higher valued tiles in the higher weighted area a much higher utility which results with the algorithm shifting tiles and converging them to one corner of the grid, a strategy which makes it more likely to reach higher valued tiles.

4^6	4^5	4^4	4^3
4^5	4^4	4^3	4^2
4^4	4^3	4^2	4^1
4^3	4^2	4^1	4^0

Empty Tiles

A heuristic of availableCells or empty tiles simply counts the number of available or empty tiles on the grid and multiplies that result by some value, giving more weight, and more utility to grids with more empty tiles. This strategy increases the chance that when possible a merge between tiles will be made and keeping the board unfilled allows for longer gameplay and more chances of reaching a higher score.

Monotonicity

Monotonicity heuristic keeps the tiles in decreasing order by adding utility to grids with tiles in decreasing order. This strategy helps make sure that tiles keep merging and that there are no "isle" tiles keeping other tiles from merging.

Smoothness

Smoothness heuristic keeps tiles of similar value close to one another. Again, this strategy increases chances that tiles merge.

Parallel Implementation

Tools used

The module that our parallel implementation uses is Control.Parallel.Strategies. The reason for using this module is the useful rpar function which sparks its argument in parallel. A list of functions used is below:

```
rpar :: Strategy a
```

rpar sparks its argument (for evaluation in parallel).

```
parList :: Strategy a -> Strategy [a]
```

Evaluate each element of a list in parallel according to given strategy. Equivalent to parTraversable at the list type.

```
runEval :: Eval a -> a
```

Pull the result out of the monad.

Naive Implementation

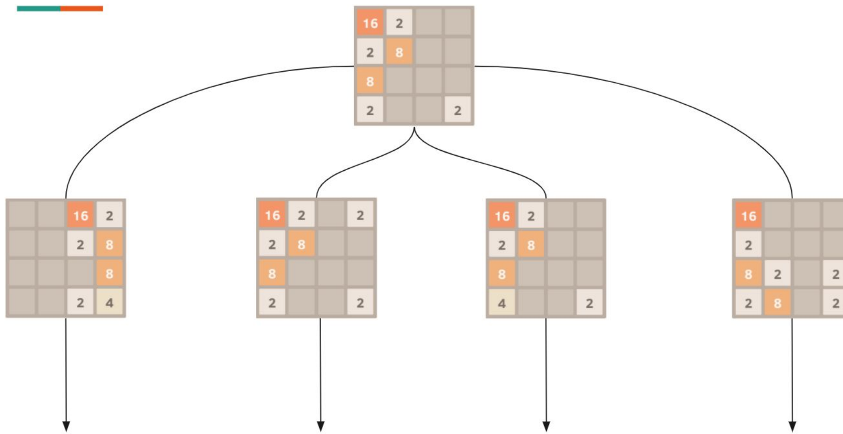
The first attempt we made was a naive injection of rpar into the existing code. We decided to use rpar on every recursive step for the minimizer function. This led to really poor spark management as seen below:

```
Game over
   7,581,186,288 bytes allocated in the heap
      19,993,936 bytes copied during GC
         119,648 bytes maximum residency (3 sample(s))
          52,384 bytes maximum slop
               0 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      7245 colls,  7245 par    0.534s   0.254s     0.0000s    0.0078s
  Gen  1         3 colls,     2 par    0.001s   0.001s     0.0002s    0.0003s

  Parallel GC work balance: 0.71% (serial 0%, perfect 100%)

  TASKS: 8 (1 bound, 7 peak workers (7 total), using -N3)

  SPARKS: 604838 (1130 converted, 0 overflowed, 0 dud, 558581 GC'd, 45127 fizzled)

  INIT    time    0.001s  (  0.004s elapsed)
  MUT     time    4.280s  (  4.362s elapsed)
  GC      time    0.535s  (  0.254s elapsed)
  EXIT    time    0.000s  (  0.009s elapsed)
  Total   time    4.815s  (  4.629s elapsed)

  Alloc rate    1,771,506,778 bytes per MUT second

  Productivity  88.9% of total user, 94.2% of total elapsed
```

It is also important to note the extremely low total time. This run was an instance where our solver failed very quickly; this means that a full run to 2048 using this implementation would generate a nightmarish amount of sparks that are barely converted.

A Better Approach to Parallelizing

After our first iteration, we decided to sit down and understand this from a bigger picture. From a fundamental standpoint, the solver has up to four moves to choose from at any turn. The main goal is to get the utility for each move using minimax and then choosing the move with the largest utility. As seen in class, to reduce the number of sparks generated from running fibonacci in parallel, we stopped parallelizing at a certain depth. Due to the efficiency of alpha-beta pruning, we decided that we only need to run rpar once on each of the resulting grids from the four possible moves at every step. This looks like the diagram below:



This diagram shows that we run rpar on each of the different moves and then calculate the rest sequentially. In the sequential version, these utilities were calculated one at a time, but this implementation allows the solver to get the needed utilities simultaneously. Due to the limit we put on parallelism, we hoped that far fewer sparks would be created and the conversion-to-creation ratio was far greater.

We use parList to map rpar to a list which includes the function that starts the minimax algorithm four times with each of the possible grids as input. The code is shown below:

```
parUtility :: [Grid] -> Integer -> Integer -> Int -> Eval [Integer]
parUtility grids a b maxDepth = parList rpar [chance c a b (maxDepth - 1) | c <- grids]
```

As one may notice, the parUtility function returns an Eval [Integer], which we can't work with alone. Therefore, after finishing the computation to get the utilities, we simply use runEval to turn the Eval [Integer] into a list of integers that we can use:

```
where utilities = runEval (parUtility (grids) a b maxDepth)
      maximum = maxUtility utilities
      grids = getChildren grid
```

<u>Results</u>

<u>Sequential Performance</u>

Before adding alpha-beta pruning, we decided to see how our solver ran. When running the solver with a depth of 6, it took on average 760 seconds to achieve a score of 2048.

After adding the alpha-beta pruning to our minimax implementation, the solver managed to achieve a score of 2048 in **42** seconds on average. Admittedly, we probably could have stopped here since our solver was able to achieve the meaning of life, the universe and everything, however we decided to parallelize it anyway.

Parallel Performance

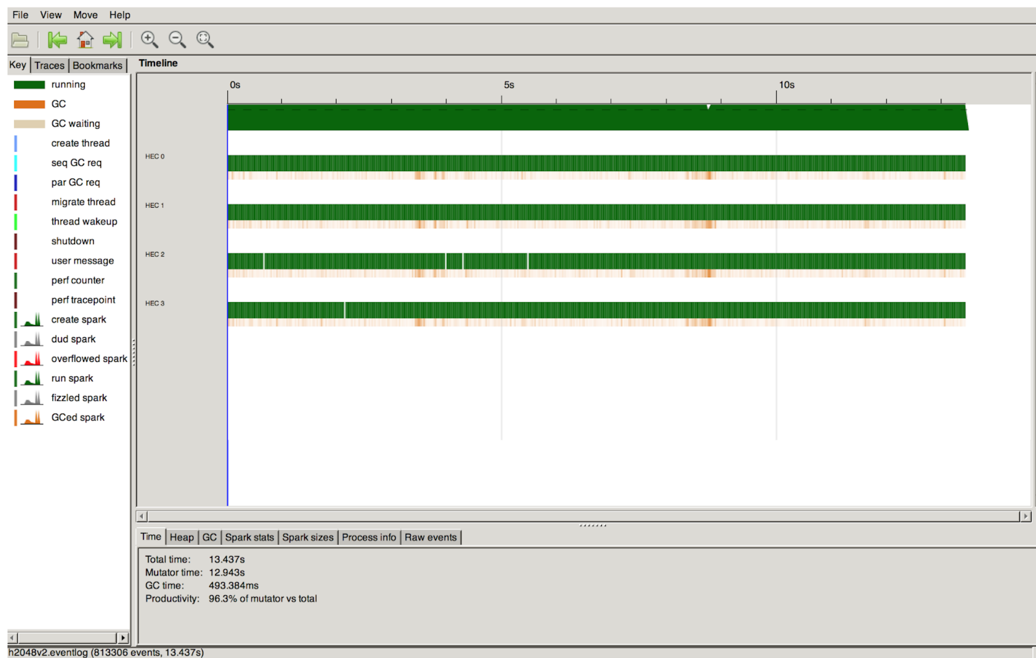
Using our multi-core parallel implementation, we received the numbers below:

Cores	Time(s)	Moves	Time(s)/ Moves	Sparks Created	Sparks Converted	Sparks GC'd	Sparks Fizzled
4	15.414	1082	0.01424584	7686	5524	1061	1101
3	16.193	991	0.01634006	7184	5202	971	1011
2	16.641	970	0.01715567	6976	5028	953	995
1	41.831	960	0.04357396	0	0	0	0

The important metric to consider is the Time(s)/Moves. This is because due to the random component of 2048, some runs may take more moves to reach 2048 than others. Therefore, the more telling metric is how much time did the solver take per move, which is one iteration of the game. As the table shows, we get a stark decrease from the sequential version to the parallel version. The Time(s)/Moves metric then decreases slightly with each core added.

Another important observation is how our spark management is doing. Compared to our naive implementation of parallelism, the number of sparks is far lower and the conversion-to-creation rate is significantly higher.

We also analyzed the load balancing of using 4 cores with our parallel implementation using ThreadScope and received a very balanced visualization, as seen below:



Conclusion

The utilization of Haskell's parallelism drastically increased the performance of minimax with alpha-beta pruning in the game of 2048. While the most dramatic difference in time was seen when incorporating alpha-beta pruning into the minimax algorithm, we were able to more than half the average sequential time of the solver using parallelism. This results in the ability to run the solver at higher depths than can normally be achieved on the sequential version.

Future Work

The solver can be extended in multiple directions. One direction that can be iterated on is figuring out the optimal depth to stop parallelizing. Our implementation stops at the first level, simply calculating the utilities of the four available moves in parallel. This can be done at multiple levels however which will result in faster performance. The questions to be answered is how fast is this difference, how is spark management when increasing depth, and again, what is the optimal depth to stop.

Another improvement to the solver is tuning the weights of the heuristics used in the solver. This process was done by trial and error on our end, however this process can be automated by running multiple solvers in parallel to figure out the best weights to use.

Code

```
import Prelude hiding (Left, Right)
--import Data.Char (toLower)
```

```
import Data.List
import System.IO
import System.Random
import Text.Printf
import Control.Parallel.Strategies


data Move = Up | Down | Left | Right
type Grid = [[Int]]

start :: IO Grid
start = do grid'  <- addTile $ replicate 4 [0, 0, 0, 0]
           addTile grid'

merge :: [Int] -> [Int]
merge xs = merged ++ padding
    where padding = replicate (length xs - length merged) 0
          merged  = combine $ filter (/= 0) xs
          combine (x:y:ys) | x == y = x * 2 : combine ys
                           | otherwise = x  : combine (y:ys)
          combine x = x

move :: Move -> Grid -> Grid
move Left  = map merge
move Right = map (reverse . merge . reverse)
move Up    = transpose . move Left  . transpose
move Down  = transpose . move Right . transpose

getZeroes :: Grid -> [(Int, Int)]
getZeroes grid = filter (\(row, col) -> (grid!!row)!!col == 0) coordinates
    where singleRow n = zip (replicate 4 n) [0..3]
          coordinates = concatMap singleRow [0..3]

setSquare :: Grid -> (Int, Int) -> Int -> Grid
setSquare grid (row, col) val = pre ++ [mid] ++ post
    where pre  = take row grid
          mid  = take col (grid!!row) ++ [val] ++ drop (col + 1) (grid!!row)
          post = drop (row + 1) grid

isMoveLeft :: Grid -> Bool
```

```
isMoveLeft grid = sum allChoices > 0
    where allChoices = map (length . getZeroes . flip move grid) directions
          directions = [Left, Right, Up, Down]

getChildren :: Grid -> [Grid]
getChildren grid = filter (\x -> x /= grid) [move d grid | d <- directions]
    where directions = [Left, Right, Up, Down]


printGrid :: Grid -> IO ()
printGrid grid = do
    putStrLn ""
    mapM_ (putStrLn . showRow) grid

showRow :: [Int] -> String
showRow = concatMap (printf "%5d")

check2048 :: Grid -> Bool
check2048 grid = [] /= filter (== 2048) (concat grid)

addTile :: Grid -> IO Grid
addTile grid = do
    let candidates = getZeroes grid
    pick <- choose candidates
    val  <- choose [2,2,2,2,2,2,2,2,2,4]
    let new_grid = setSquare grid pick val
    return new_grid

choose :: [a] -> IO a
choose xs = do
    i <- randomRIO (0, length xs-1)
    return (xs !! i)

sumOfTiles :: Grid -> Integer
sumOfTiles grid = toInteger $ sum $ map sum grid


weightMatrix :: Grid -> Integer
weightMatrix grid = sumOfTiles $ zipWith (zipWith (*)) matrix grid
```

```
    --where matrix = [[1073741824, 268435456, 67108864,
16777216],[65536,262144,1048576,4194304],[16384,4096,1024,256],[1,4,16,64]]
    --where matrix = [[1073741824, 268435456, 67108864,
16777216],[4194304,1048576,262144,65536],[16384,4096,1024,256],[64,16,4,1]]
    --where matrix = [[7,6,5,4],[6,5,4,3],[5,4,3,2],[4,3,2,1]]
    where matrix = if maxTile grid <= 512 then [[21,8,3,3],[9,5,2],[4,3]] else
[[19,9,5,3],[8,4,2],[3]]
    --where matrix = [[26000,,22,20],[12,14,16,18],[10,8,6,4],[1,2,3,4]]

monotonicity :: Grid -> Int -> Integer
monotonicity [] _ = 0
monotonicity (x:xs) currentValue = fromIntegral (monotonicityHelper x currentValue) +
monotonicity xs currentValue

monotonicityHelper :: [Int] -> Int -> Int
monotonicityHelper [] _ = 0
monotonicityHelper (x:xs) currentValue
            | x < currentValue = 1 + monotonicityHelper xs x
            | otherwise = monotonicityHelper xs x

smoothness :: Grid -> Integer
smoothness [] = 0
smoothness (x:xs) = smoothnessHelper x + smoothness xs

smoothnessHelper :: [Int] -> Integer
smoothnessHelper [] = 0
smoothnessHelper (a:b:c:d:_) = fromIntegral $ ((abs (a - b)) + (abs (b - c)) + (abs (c - d))) * 5
smoothnessHelper [_] = error "wrong number of elements"
smoothnessHelper [_,_] = error "wrong number of elements"
smoothnessHelper [_,_,_] = error "wrong number of elements"



availableCells :: Grid -> Integer
availableCells grid = toInteger $ sum $ map zeros grid
    where zeros l = length $ filter (\x -> x == 0) l

weWon :: Grid -> Integer
weWon grid
    | maxTile grid == 2048 = 999999
```

```
    | otherwise = 0


utility :: Grid -> Integer
utility grid = weightMatrix grid + (100 * availableCells grid) + (15 * monotonicity grid 9999) +
(15 * monotonicity (transpose grid) 9999) - (smoothness grid) - (smoothness (transpose grid)) +
weWon grid
--utility grid = fromIntegral $ (3 * monotonicity grid 9999) + (3 * monotonicity (transpose grid)
9999) - (smoothness grid) - (smoothness (transpose grid)) + (maxTile grid) + (fromIntegral $ 3 *
availableCells grid) + (fromIntegral $ sumOfTiles grid)


parMaximize :: Grid -> Integer -> Integer -> Int -> Grid
parMaximize grid a b maxDepth
  | maxDepth == 0 || not (isMoveLeft grid) = grid
  | otherwise = chooseGrid (chooseGridIndex utilities maximumm 0) (grids)
    where utilities = runEval (parUtility (grids) a b maxDepth)
          maximumm = maxU utilities
          grids = getChildren grid


parUtility :: [Grid] -> Integer -> Integer -> Int -> Eval [Integer]
parUtility grids a b maxDepth = parList rpar [chance c a b (maxDepth - 1) | c <- grids]

maxU :: [Integer] -> Integer
maxU utilities = maximum utilities

chooseGridIndex :: [Integer] -> Integer -> Int -> Int
chooseGridIndex [] _ _ = error "wrong number of elements"
chooseGridIndex (x:xs) maxUtil starter
                | x == maxUtil = starter
                | otherwise = chooseGridIndex xs maxUtil (starter + 1)

chooseGrid :: Int -> [Grid] -> Grid
chooseGrid index grids = grids !! index

maximize :: Grid -> Integer -> Integer -> Int -> (Grid, Integer)
maximize grid a b maxDepth
  | maxDepth == 0 || not (isMoveLeft grid) = (grid, utility grid)
  | otherwise = maxHelper (getChildren grid) grid a b (-999999999999) maxDepth
```

```
maxHelper :: [Grid] -> Grid -> Integer -> Integer -> Integer -> Int -> (Grid, Integer)
maxHelper [] maxChild _ _ maxUtility _ = (maxChild, maxUtility)
maxHelper (c:children) maxChild a b maxUtility maxDepth
  | util > maxUtility = maxHelper (c:children) c a b util maxDepth
  | maxUtility >= b = maxHelper children maxChild a b maxUtility maxDepth
  | maxUtility > a = maxHelper (c:children) maxChild maxUtility b maxUtility maxDepth
  | otherwise = maxHelper children maxChild a b maxUtility maxDepth
    --where utility = runEval $ rpar (chance c a b (maxDepth - 1))
    where util = chance c a b (maxDepth - 1)

chance :: Grid -> Integer -> Integer -> Int -> Integer
chance grid a b maxDepth
  | maxDepth == 0 = utility grid
  | otherwise = toInteger (round ((9/10) * (realToFrac (minimize grid a b 2 (maxDepth - 1))))) +
toInteger (round ((1/10) * (realToFrac (minimize grid a b 4 (maxDepth - 1)))))

minimize :: Grid -> Integer -> Integer -> Int -> Int -> Integer
minimize grid a b tileVal maxDepth
  | maxDepth == 0 || availableCells grid == 0 = utility grid
  | otherwise = minHelper grid (getZeroes grid) tileVal a b 999999999999 maxDepth

minHelper :: Grid -> [(Int, Int)] -> Int -> Integer -> Integer -> Integer -> Int -> Integer
minHelper _ [] _ _ _ minUtility _ = minUtility
minHelper grid (c:cells) tVal a b minUtility maxDepth
  | util < minUtility = minHelper grid (c:cells) tVal a b util maxDepth
  | minUtility <= a = minHelper grid cells tVal a b minUtility maxDepth
  | minUtility < b = minHelper grid (c:cells) tVal a minUtility minUtility maxDepth
  | otherwise = minHelper grid cells tVal a b minUtility maxDepth
    where (_, util) = maximize (setSquare grid c tVal) a b (maxDepth - 1)


maxTile :: Grid -> Int
maxTile b = maximum $ map maximum b


gameLoop :: Grid -> Int -> IO ()
gameLoop grid movess
    | isMoveLeft grid = do
        printGrid grid
```

```
        if check2048 grid
        then print movess
        else do --new_grid <- newGrid grid
                let newGrid = parMaximize grid (-999999999999) 999999999999 6
                if grid /= newGrid
                then do new <- addTile newGrid
                        gameLoop new (movess+1)
                else gameLoop grid (movess+1)
    | otherwise = do
        printGrid grid
        putStrLn "Game over"
        print movess

main :: IO ()
main = do
    hSetBuffering stdin NoBuffering
    grid <- start
    gameLoop grid 0
```