Boxiong Kong (bk2808)
bk2808@columbia.edu

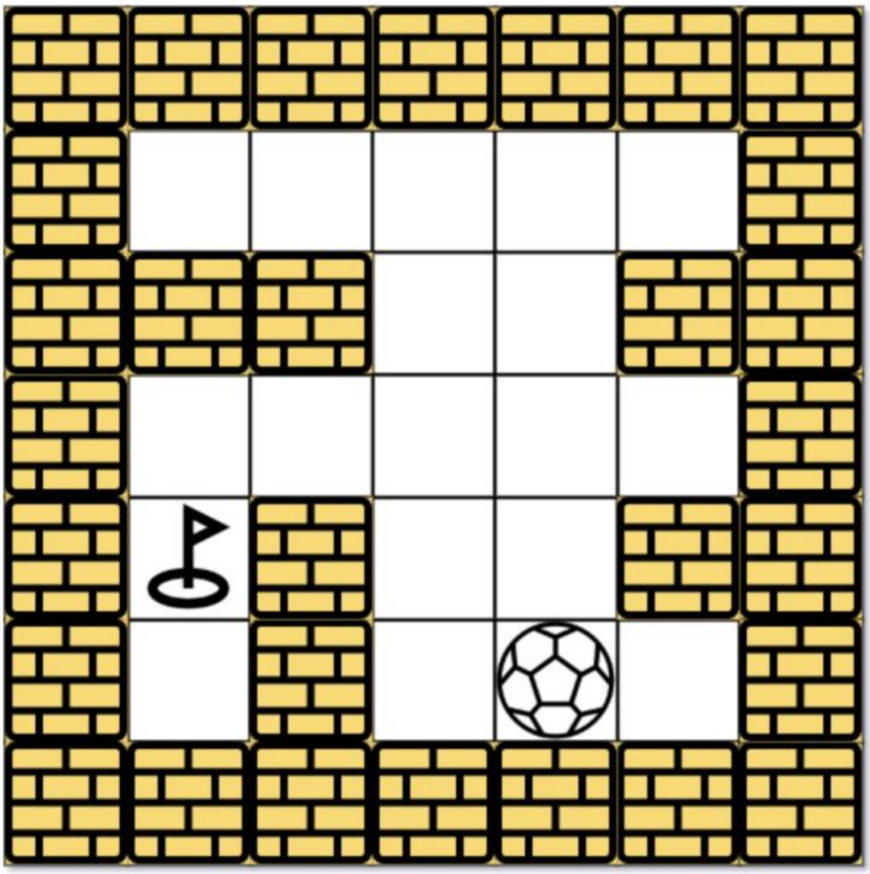
Parallel Functional Programming

Final Project Report: Maze Game

Introduction

In this project, I use Haskell's parallelism to solve a maze game. Given a maze with walls and empty spaces, there is a ball and a hole in it. The ball can move up, down, left and right through the empty spaces and it won't stop until hitting a wall. And the ball will choose the next direction to move if it stops. If the ball goes through the hole, it will drop into the hole.

The initial position of the ball and hole is defined by the player. My program will determine if the ball will drop into the hole after a series of movements. If it is possible for the ball to drop into the hole, the program will generate the instructions that the ball should follow to drop into the hole with the shortest distance. That is, the minimum number of empty spaces the ball has traveled from the start position to the hole.



Sequential implementation

The Pseudocode of the sequential algorithm is as follows.

```
maze_game(maze, ball, hole):
    star_row = ball[0]
    start_col = ball[1]
    heap = [(0, star_row, start_col, "start")]    # steps, row, col, string (direction)
    visited_nodes = set()

    while heap:
        current_distance,    current_row,    current_col,    current_string    =
heappop(heap)
        if (current_row, current_col) not in visited_nodes:
            visited_nodes.add((current_row, current_col))
            if [current_row, current_col] == hole:
                return current_string

            for row_diff, col_diff, direction in [(1, 0, 'down'), (-1, 0, 'up'),
                                                  (0, 1, 'right'), (0, -1, 'left')]:
                row = current_row
                col = current_col
                count = 0

                while 0 <= row + row_diff <= len(maze) - 1
                      and 0 <= col + col_diff <= len(maze[0]) - 1
                      and maze[row + row_diff][col + col_diff] == 0:

                    row += row_diff
                    col += col_diff
                    count += 1

                    if [row, col] == hole:
                        break

                if (row, col) not in visited_nodes:
                    heappush(heap,
                            (current_distance+count,
                            row, col,
                            current_string + direction))

    return 'Impossible to reach the hole!'
```

Since we want to solve the shortest path problem, we design a Dijkstra-based algorithm. We use min-heap as a priority queue and import Data.Heap in Haskell. And we introduce a new data type Heap_item, which contains the moving distance, start row, start column and moving direction of the ball.

```
import Data.Heap

data Heap_item = Heap_item {
            distance :: Int,
            start_row :: Int,
            start_col :: Int,
            direction :: String
      } deriving (Eq, Ord, Show)
```

In the main function, we define the maze (1 represents the wall while 0 represents the empty space) and the location of the ball and the hole. We also initialize a heap and use a set to contain the places that the ball has visited. Then, the main function calls the gameloop function to solve the mazegame problem, which returns a heap. If the heap is empty, then the ball can never drop into the hole. Otherwise, the ball can reach the hole. The first element of the heap gives the instructions that the ball should follow to drop into the hole with the shortest distance.

```
main :: IO ()
main = do
      let maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]]
            ball = (4, 3)
            hole = (0, 1)
            heap_init = Heap_item 0 (fst ball) (snd ball) "start"
            heap = Data.Heap.fromList [heap_init] :: MinHeap Heap_item
            visited_nodes = []
      heap_output <- gameloop heap visited_nodes hole maze
      if isEmpty heap_output then
            putStrLn $ "Impossible to reach the hole!"
      else do
            let heap_head = heaphead heap_output
            putStrLn $ "Instruction: " ++ (direction heap_head) ++ "\nTotal distance: 
" ++ (show $ distance heap_head)
```

The gameloop function is as follows.

```
gameloop :: Monad m => HeapT (Prio MinPolicy Heap_item) () -> [(Int, Int)] -> 
(Int, Int) -> Maze -> m (HeapT (Prio MinPolicy Heap_item) ())
gameloop h visited_nodes hole maze = do
            if isEmpty h
                  then return h
```

```
        else do
            let heap_head = heaphead h
                h_n = Data.Heap.drop 1 h
                current_distance = distance heap_head
                current_row = start_row heap_head
                current_col = start_col heap_head
                current_string = direction heap_head
            if ((current_row, current_col) == hole) then do
                let heap_final = Data.Heap.fromList [heap_head] :: MinHeap Heap_item
                return heap_final
            else do
                let visited_nodes_n = set_insert (current_row, current_col) visited_nodes
                h_d <- helper h_n maze hole visited_nodes_n current_distance current_row current_col current_string 1 0 "down"
                h_u <- helper h_d maze hole visited_nodes_n current_distance current_row current_col current_string (-1) 0 "up"
                h_r <- helper h_u maze hole visited_nodes_n current_distance current_row current_col current_string 0 1 "right"
                h_l <- helper h_r maze hole visited_nodes_n current_distance current_row current_col current_string 0 (-1) "left"
                gameloop h_l visited_nodes_n hole maze
```

The gameloop is based on Dijkstra algorithm. It won’t stop until the ball reached the hole or the heap is empty. In each loop, we pop out the first element in the heap, which is a Heap_item data type. This element indicates the current location of the ball. If this location hasn’t been visited, we add it to the set of visited places, and call a helper function and move function to move the ball in four directions.

The helper function and move function is as follows.

```
helper :: Monad m => HeapT (Prio MinPolicy Heap_item) () -> Maze -> (Int, Int) -> [(Int, Int)] -> Int -> Int -> Int -> String -> Int -> Int -> String -> m (HeapT (Prio MinPolicy Heap_item) ())
helper heap maze hole visited_nodes current_distance current_row current_col current_string row_diff col_diff direction = do
    let result = move maze hole current_row current_col 0 row_diff col_diff
        row_n = first result
        col_n = second result
        count_n = third result
    if not ((row_n, col_n) `elem` visited_nodes) then do
        let heap_item_n = Heap_item (current_distance + count_n) row_n col_n (current_string ++ "->" ++ direction)
```

```
            h_n = Data.Heap.insert heap_item_n heap
        return h_n
    else do
        return heap
```

```
move :: Maze -> (Int, Int) -> Int -> Int -> Int -> Int -> Int -> (Int, Int, Int)
move maze hole row col count row_diff col_diff
  | ((row+row_diff) >= (maze_m maze)) || (row+row_diff) < 0 || ((col+col_diff) >= (maze_n maze)) || (col+col_diff) < 0 || ((maze!!(row+row_diff))!!(col+col_diff)) /= 0 = (row, col, count)
  | (row+row_diff, col+col_diff) == hole = (row+row_diff, col+col_diff, count+1)
  | otherwise = move maze hole (row+row_diff) (col+col_diff) (count+1) row_diff col_diff
```

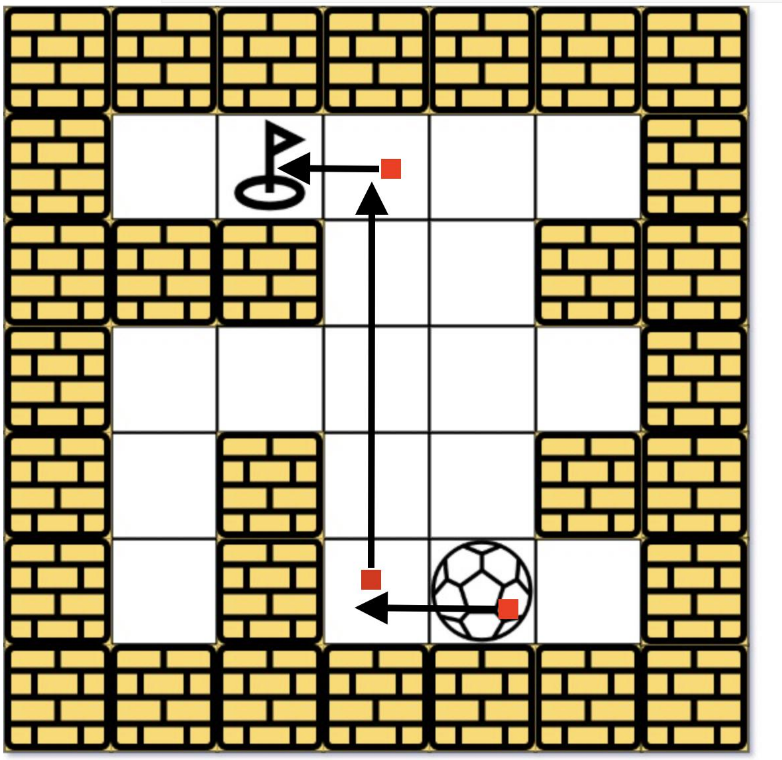
The move function moves the ball through the empty spaces until hitting a wall or reaching the hole. The helper function determines if the current position of the ball after the move has visited before. If not, we use current location, total distance and direction to create a new Heap_item, and insert it to the heap.

Now we test the correctness of this algorithm.

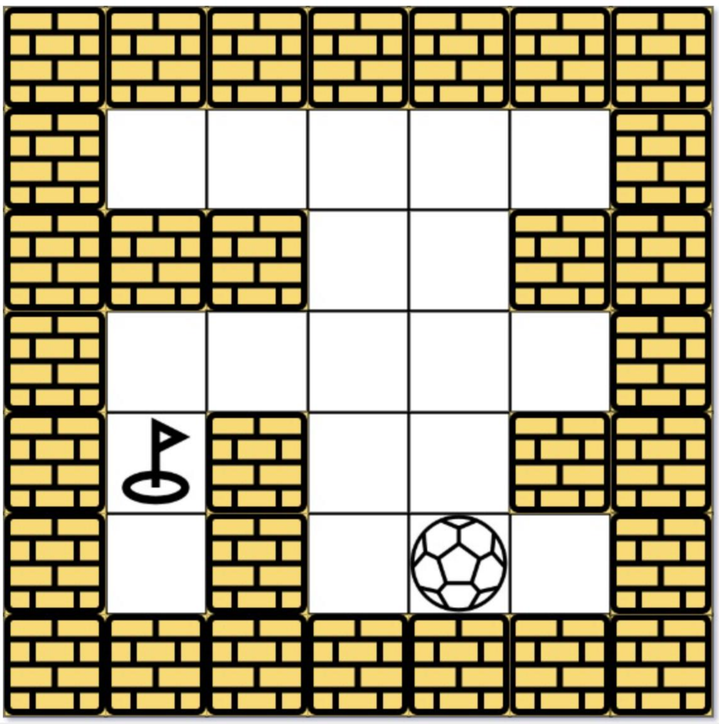
Given a maze [[0, 0, 0, 0, 0],
[1, 1, 0, 0, 1],
[0, 0, 0, 0, 0],
[0, 1, 0, 0, 1],
[0, 1, 0, 0, 0]],
the initial location of the ball (4, 3) and the location of the hole (0, 1). The output of the sequential algorithm is as follows.

```
*Main Lib Paths_mzgame> :l mazegame_sequential
[1 of 1] Compiling Main             ( mazegame_sequential.hs, interpreted )
Ok, one module loaded.
*Main> main
Instruction: start->left->up->left
Total distance: 6
```

It gives the instructions that the ball should follow to drop into the hole with the shortest distance, that is, start -> left -> up -> left. It also gives the value of the shortest distance, which is 6.



As shown in the picture above, the ball can reach the hole at the shortest distance of 6, following the given instructions.



As shown in the picture above. If the initial location of the ball (4, 3) and the location of the hole (3, 0). The output of the sequential algorithm is as follows.

```
*Main> :l mazegame_sequential
[1 of 1] Compiling Main             ( mazegame_sequential.hs, interpreted )
Ok, one module loaded.
*Main> main
Impossible to reach the hole!
```

It indicates that the ball cannot reach the hole, which is correct.

Parallel implementation

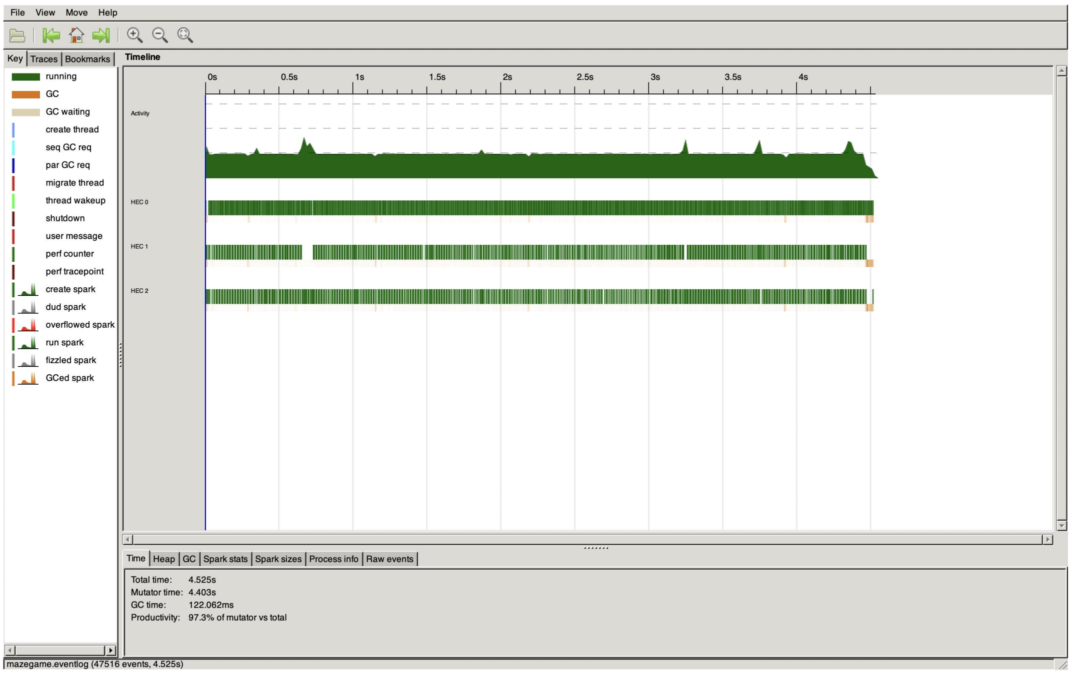
In each gameloop, we need to check four directions of the ball movement. These four movements are relatively independent, which can be divided into four word problems. Using Control.Parallel.Strategies, we can map the parameters of the four movements into the move function, and use parList and rpar to call the move function in parallel. The key code is as follows.

ins = map (move maze hole current_row current_col 0) [(1,0), (-1,0), (0,1), (0,-1)] `using` parList rpar

To test the program, we use a loop-shaped maze with the following structure.

```
[ [1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [0, 0, 1, 0, 0, 0, 1, 0, 0, 0]]
```

We extended the maze of this structure to 1,000 dimensions. Running the program with three cores, the results on ThreadScope is shown as follows.



As shown above, the program runs in parallel on three cores.

Program listing

mazegame_sequential.hs

```
import Data.Heap
import System.Exit(die)
import Control.Monad
import Control.Parallel.Strategies

type Maze = [[Int]]

data Heap_item = Heap_item {
            distance :: Int,
            start_row :: Int,
            start_col :: Int,
            direction :: String
      } deriving (Eq, Ord, Show)


set_insert :: Eq a => a -> [a] -> [a]
set_insert x xs
      | not (x `elem` xs) = x:xs
      | otherwise              = xs


heaphead :: HeapT (Prio MinPolicy Heap_item) () -> Heap_item
heaphead heap = head (Data.Heap.take 1 heap)


main :: IO ()
main = do
      let maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]]
            ball = (4, 3)
            hole = (3, 0)
            heap_init = Heap_item 0 (fst ball) (snd ball) "start"
            heap = Data.Heap.fromList [heap_init] :: MinHeap Heap_item
            visited_nodes = []
      heap_output <- gameloop heap visited_nodes hole maze
      if isEmpty heap_output then
            putStrLn $ "Impossible to reach the hole!"
      else do
            let heap_head = heaphead heap_output
            putStrLn $ "Instruction: " ++ (direction heap_head) ++ "\nTotal distance: "
++ (show $ distance heap_head)
```

```
gameloop :: Monad m => HeapT (Prio MinPolicy Heap_item) () -> [(Int, Int)] -> (Int, Int) -> Maze -> m (HeapT (Prio MinPolicy Heap_item) ())
gameloop h visited_nodes hole maze = do
        if isEmpty h
            then return h
        else do
            let heap_head = heaphead h
                h_n = Data.Heap.drop 1 h
                current_distance = distance heap_head
                current_row = start_row heap_head
                current_col = start_col heap_head
                current_string = direction heap_head
            if ((current_row, current_col) == hole) then do
                let heap_final = Data.Heap.fromList [heap_head] :: MinHeap Heap_item
                return heap_final
            else do
                let visited_nodes_n = set_insert (current_row, current_col) visited_nodes
                h_d <- helper h_n maze hole visited_nodes_n current_distance current_row current_col current_string 1 0 "down"
                h_u <- helper h_d maze hole visited_nodes_n current_distance current_row current_col current_string (-1) 0 "up"
                h_r <- helper h_u maze hole visited_nodes_n current_distance current_row current_col current_string 0 1 "right"
                h_l <- helper h_r maze hole visited_nodes_n current_distance current_row current_col current_string 0 (-1) "left"
                gameloop h_l visited_nodes_n hole maze


helper :: Monad m => HeapT (Prio MinPolicy Heap_item) () -> Maze -> (Int, Int) -> [(Int, Int)] -> Int -> Int -> Int -> String -> Int -> Int -> String -> m (HeapT (Prio MinPolicy Heap_item) ())
helper heap maze hole visited_nodes current_distance current_row current_col current_string row_diff col_diff direction = do
    let result = move maze hole current_row current_col 0 row_diff col_diff
        row_n = first result
        col_n = second result
        count_n = third result
    if not ((row_n, col_n) `elem` visited_nodes) then do
        let heap_item_n = Heap_item (current_distance + count_n) row_n col_n (current_string ++ "->" ++ direction)
```

```
                h_n = Data.Heap.insert heap_item_n heap
            return h_n
        else do
            return heap


move :: Maze -> (Int, Int) -> Int -> Int -> Int -> Int -> Int -> (Int, Int, Int)
move maze hole row col count row_diff col_diff
   | ((row+row_diff) >= (maze_m maze)) || (row+row_diff) < 0 || ((col+col_diff) >=
(maze_n maze)) || (col+col_diff) < 0 || ((maze!!(row+row_diff))!!(col+col_diff)) /= 0
= (row, col, count)
   | (row+row_diff, col+col_diff) == hole = (row+row_diff, col+col_diff, count+1)
   | otherwise = move maze hole (row+row_diff) (col+col_diff) (count+1) row_diff
col_diff


first :: (a, b, c) -> a
first (a,_,_) = a

second :: (a, b, c) -> b
second (_,b,_) = b

third :: (a, b, c) -> c
third (_,_,c) = c

maze_m :: Maze -> Int
maze_m maze = length maze

maze_n :: Maze -> Int
maze_n maze = length $ head maze
```

mazegame_parallel.hs

```
import Data.Heap
import Control.Monad
import Control.DeepSeq
import Control.Parallel.Strategies

type Maze = [[Int]]

data Heap_item = Heap_item {
            distance :: Int,
            start_row :: Int,
```

```
        start_col :: Int,
        direction :: String
    } deriving (Eq, Ord, Show)


set_insert :: Eq a => a -> [a] -> [a]
set_insert x xs
    | not (x `elem` xs) = x:xs
    | otherwise         = xs


heaphead :: HeapT (Prio MinPolicy Heap_item) () -> Heap_item
heaphead heap = head (Data.Heap.take 1 heap)

maze_constructor :: Int -> Maze
maze_constructor n = ((p n 1 []) : (replicate (n-2) (odd_to_1 n 1 []))) ++ [q n 1 []]
  where
    odd_to_1 n i result
      | i > n = result
      | mod i 2 == 1 = odd_to_1 n (i+1) (result++[1])
      | otherwise = odd_to_1 n (i+1) (result++[0])
    p n i result
      | i > n = result
      | mod i 4 == 1 = p n (i+1) (result++[1])
      | otherwise = p n (i+1) (result++[0])
    q n i result
      | i > n = result
      | mod i 4 == 3 = q n (i+1) (result++[1])
      | otherwise = q n (i+1) (result++[0])

main :: IO ()
main = do
    let maze = maze_constructor 10
        ball = (9, 9)
        hole = (9, 0)
        heap_init = Heap_item 0 (fst ball) (snd ball) "start"
        heap = Data.Heap.fromList [heap_init] :: MinHeap Heap_item
        visited_nodes = []
    heap_output <- gameloop heap visited_nodes hole maze
    if isEmpty heap_output then
        putStrLn $ "Impossible to reach the hole!"
    else do
        let heap_head = heaphead heap_output
        putStrLn $ "Instruction: " ++ (direction heap_head) ++ "\nTotal distance: "
```

```
++ (show $ distance heap_head)


gameloop :: Monad m => HeapT (Prio MinPolicy Heap_item) () -> [(Int, Int)] -> (Int,
Int) -> Maze -> m (HeapT (Prio MinPolicy Heap_item) ())
gameloop h visited_nodes hole maze = do
            if isEmpty h
                  then return h
            else do
                  let heap_head = heaphead h
                        h_n = Data.Heap.drop 1 h
                        current_distance = distance heap_head
                        current_row = start_row heap_head
                        current_col = start_col heap_head
                        current_string = direction heap_head
                  if ((current_row, current_col) == hole) then do
                        let heap_final = Data.Heap.fromList [heap_head] :: MinHeap
Heap_item
                        return heap_final
                  else do
                        let visited_nodes_n = set_insert (current_row, current_col)
visited_nodes
                              ins = map (move maze hole current_row current_col 0)
[(1,0), (-1,0), (0,1), (0,-1)] `using` parList rpar
                        h_l <- helper h_n visited_nodes_n current_distance current_string
["down", "up", "right", "left"] ins
                        gameloop h_l visited_nodes_n hole maze


helper :: Monad m => HeapT (Prio MinPolicy Heap_item) () -> [(Int, Int)] -> Int ->
String -> [String] ->    [(Int, Int, Int)] -> m (HeapT (Prio MinPolicy Heap_item) ())
helper heap visited_nodes current_distance current_string direction instruction = do
      if Prelude.null instruction
            then return heap
      else do
            let i = head instruction
                  row_n = first i
                  col_n = second i
                  count_n = third i
                  d = head direction
            if not ((row_n, col_n) `elem` visited_nodes) then do
                  let heap_item_n = Heap_item (current_distance + count_n) row_n
col_n (current_string ++ "->" ++ d)
                        h_n = Data.Heap.insert heap_item_n heap
```

```
                    helper h_n visited_nodes current_distance current_string (Prelude.drop 1 direction) (Prelude.drop 1 instruction)
              else do
                    helper heap visited_nodes current_distance current_string (Prelude.drop 1 direction) (Prelude.drop 1 instruction)


move :: Maze -> (Int, Int) -> Int -> Int -> Int -> (Int, Int) -> (Int, Int, Int)
move maze hole row col count (row_diff, col_diff)
   | ((row+row_diff) >= (maze_m maze)) || (row+row_diff) < 0 || ((col+col_diff) >= (maze_n maze)) || (col+col_diff) < 0 || ((maze!!(row+row_diff))!!(col+col_diff)) /= 0 = (row, col, count)
   | (row+row_diff, col+col_diff) == hole = (row+row_diff, col+col_diff, count+1)
   | otherwise = move maze hole (row+row_diff) (col+col_diff) (count+1) (row_diff, col_diff)


first :: (a, b, c) -> a
first (a,_,_) = a

second :: (a, b, c) -> b
second (_,b,_) = b

third :: (a, b, c) -> c
third (_,_,c) = c

maze_m :: Maze -> Int
maze_m maze = length maze

maze_n :: Maze -> Int
maze_n maze = length $ head maze
```