Parallel Functional Programming Report: MazeSolver

Samya Ahsan (ska2138), Nicole Lin (nsl2126), Alice Wang (aw3271)

Fall 2023

1 Background & Objective

Rat in a Maze is a game set up as a maze on a two-dimensional nxn grid, where open cells have the value 1 (representing True as in the rat can move here) and closed cells (i.e., walls) have the value 0 (representing False as in the rat can't move here). The goal is to find the shortest distance to navigate the 'rat' from the start point (0,0) to the end point (n-1,n-1) in the maze.

The input to the program is a file called maze_examples.txt (generated by maze_generator.py) in which each line represents a maze as a string of 0s and 1s separated by commas. The main Haskell function is designed to take in the maze_examples.txt file and reshape these mazes into 100x100 2D arrays to put into our entry point function astar.

Figure 1 is an example 2d maze input, not one of the 100x100 mazes.

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

Figure 1: Example of 15x15 2d array

The output is a list containing the shortest path found by the maze solver, represented as a list of positions $[(x_0, y_0), (x_1, y_1), ...(x_{n-1}, y_{n-1})]$, where m is the number of steps in the shortest path. If no paths are found, the program will output the line "No path found."

2 Implementations

2.1 Sequential (Base Algorithm)

Our sequential implementation uses the A* algorithm to find the shortest path through the maze, if paths exist. We consider all cells in the maze as either open or closed (i.e. a wall), represented as the values 1 and 0 respectively. Each open cell is represented as a Node having the associated information: position, g-cost, h-cost and parent Node. The position and parent Node are self-explanatory; the g-cost and h-cost combined offer a heuristic for evaluating each Node. The g-cost of a Node is the cost to reach the current Node from the start Node. The h-cost of a Node is the estimated cost to reach the end-point of the maze from the current Node. In our implementation of the A* algorithm, we've used Manhattan distance to approximate each Node's h-cost. Thus, the total estimated cost of a Node $f(n) = g(n) + h(n)$, where $g(n)$ is the g-cost of the current Node n and $h(n)$ is its h-cost. In addition to this specific representation of open cells, the algorithm uses a priority queue as an "open list," keeping track of nodes that are "open" for evaluation and sorting them by total cost (in ascending order). A visited list is also maintained to keep track of nodes that have already been evaluated.

In theory, the A* algorithm begins at the initial node (start position) and adds it to the open list. A main loop then iteratively selects the node with the lowest total cost from the "open list" for evaluation. With the selected node, the algorithm checks whether it's the endpoint of the maze, in which case the algorithm terminates and reconstructs the path from endpoint to start. Otherwise, the node's neighbors are considered, adding them as nodes to the "open list, with updated costs.

Note that our algorithm is a simplified version of A. The A* algorithm can be implemented such that when evaluating the current node's neighbors, each neighbor's g-cost is only updated if its new g-cost (from this current node) is lower than its previous g-cost (from being the neighbor of a previously visited node). In our algorithm, we update cost so that a neighbor node's g-cost is the current node's g-cost plus 1. While this simplification may lead to redundant additions of neighbor nodes to the priority queues, potentially for the same position but with different costs, the priority queue-like structure ensures that nodes with the lowest costs are still prioritized. This adaptation allows for a clear and concise implementation while retaining the core mechanism of selecting nodes with the lowest costs.

Our implementation has the function 'astar' as the entrypoint for the algorithm, taking a maze as input and potentially resulting in the shortest path from start to end (if no path is found, Nothing is returned). The helper function 'go' performs the recursive act of evaluating whether the current node is the end goal, and if not, expanding the search, essentially, to the node's neighbors and updating their costs and associated parent node. Below is our code implementation, sectioned out.

Here are the type synonyms we've used to represent Position and Maze and

our Node data type.

```
Unset
type Position = (Int, Int)
type Maze = [[Int]]

data Node = Node
 { position :: Position
 , gCost :: Int
 , hCost :: Int
 , parent :: Maybe Node
 }

instance Eq Node where
 (==) = (==) `on` position
```

```
instance Ord Node where
 compare = compare `on` (\node -> gCost node + hCost node)
```

Note that in our ‘astar‘ function (shown later), we used a sorted list to maintain the priority queue, rather than importing the priority queue type for simplicity of implementation and practicality. To do so, we declared ‘Node‘ as an instance of the ‘Ord‘ type class (shown above) and implemented the ‘compare‘ function to compare nodes based on total cost (g-cost + h-cost).

Below are our helper functions for the A* algorithm. To summarize:

- ‘initialNode‘ - creates the initial node with position (0, 0), gCost 0, hCost calculated using the heuristic function, and no parent.
- ‘isGoal‘ - checks if a given position is the goal in the maze
- ‘expandNode‘- generates neighboring nodes for a given node, filtering based on valid positions
- ‘updateCosts‘ - updates the cost values for a list of nodes based on the parent’s costs and the heuristic function
- ‘heuristic‘ - computes the heuristic value (Manhattan Distance) for a given position
- ‘isValidPosition‘ - checks if a position is valid in the maze, including whether it’s an open cell (==1)
- ‘extractPath‘ - extracts the path from a goal node back to the initial node, using recursion and nodes’ parent pointers

```
initialNode :: Maze -> Node
initialNode maze = Node {position = (0, 0), gCost = 0, hCost = heuristic maze
(0, 0), parent = Nothing}

isGoal :: Maze -> Position -> Bool
isGoal maze pos = pos == (length maze - 1, length (head maze) - 1)

expandNode :: Maze -> Node -> [Node]
expandNode maze node =
 let (x, y) = position node
     neighbors = filter (\(dx, dy) -> isValidPosition maze (x + dx, y + dy))
[(1, 0), (-1, 0), (0, 1), (0, -1)]
 in map (\(dx, dy) -> Node {position = (x + dx, y + dy), gCost = 0, hCost = 0,
parent = Just node}) neighbors

updateCosts :: Maze -> Node -> [Node] -> [Node]
updateCosts maze parent nodes =
 map
   (\node ->
      let g = gCost parent + 1
          h = heuristic maze (position node)
      in node {gCost = g, hCost = h, parent = Just parent})
   nodes

-- Heuristic function: Manhattan Distance
heuristic :: Maze -> Position -> Int
heuristic maze (x, y) = abs (x - goalX) + abs (y - goalY)
 where
   goalX = length maze - 1
   goalY = length (head maze) - 1

isValidPosition :: Maze -> Position -> Bool
isValidPosition maze (x, y) =
 x >= 0 && y >= 0 && x < length maze && y < length (head maze) && (maze !! x !!
y) == 1

extractPath :: Node -> [Position]
extractPath node = go node []
 where
   go (Node pos _ _ (Just parent)) acc = go parent (pos : acc)
   go _ acc = (0, 0) : acc
```

Below is our ‘astar‘ function, which has the ‘go‘ recursive helper function.

```
astar :: Maze -> Maybe [Position]
astar maze = case go [initialNode maze] [] of
 Just node -> Just (extractPath node)
 Nothing   -> Nothing
 where
   go [] _ = Nothing
   go (current:rest) visited
     | isGoal maze (position current) = Just current
     | otherwise =
       let neighbors = filter (\n -> notElem (position n) visited) (expandNode
maze current)
           newNodes = updateCosts maze current neighbors
           sortedNodes = sort (rest ++ newNodes)
       in go sortedNodes (position current : visited)
```

2.2 Parallel

In contrast to other graph search and pathfinding algorithms, parallelizing the A* algorithm poses unique challenges. During each time step, the algorithm assesses the state with the minimum total cost in the priority queue. It proceeds to expand the neighbors of the selected state, calculates the costs associated with these neighbors, and then reinserts them back into the priority queue in a sorted order. The complexities in parallelizing this algorithm stem from the following factors:

- Challenge 1: Working with parallel threads on a single priority queue introduces potential race conditions. Each thread must lock the priority queue to access the most promising state and lock it again to push its neighbors. This locking mechanism hampers concurrency and slows down the algorithm due to contention for queue access.
- Challenge 2: Given the possibility of multiple states having the same cost, the top state in the priority queue may not always result in the optimal path.

To address these challenges, we propose three different parallelization strategies:

2.2.1 ParallelCost

For our first parallelization strategy, we need to make sure that we do not introduce potential race conditions by working with parallel threads on a single priority queue (Challenge 1). Additionally, in the A* algorithm, we need to maintain a global state of visited nodes visible to all parallel evaluations so they don't repeat the same moves that were already evaluated in another thread. Thus, the only operation independent of other states and would not cause race conditions on a single priority queue is the update of each node's cost. rpar, which prioritizes parallel evaluation without a predefined order, is employed instead of rseq, since the nodes will be subsequently sorted based on their costs.

```
astarParallelUpdateCost :: Maze -> Maybe [Position]
astarParallelUpdateCost maze = case go [initialNode maze] [] of
 Just node -> Just (extractPath node)
 Nothing  -> Nothing
 where
  go [] _ = Nothing
  go (current:rest) visited
   | isGoal maze (position current) = Just current
   | otherwise =
    let neighbors = filter (\n -> notElem (position n) visited) (expandNode maze
current)
      newNodes = withStrategy (parList rpar) (updateCosts maze current neighbors)
      sortedNodes = sort (rest ++ newNodes)
    in go sortedNodes (position current : visited)
```

However, since the computation of Manhattan Distance is not resource-intensive, attempting parallelization in this manner is likely to result in significant overhead due to the creation of sparks and threads. Therefore, we explored alternative parallelization strategies below.

2.2.2 ParallelNodes

At each time step, the first node in the priority queue might not necessarily represent the most optimal path since multiple states could have the same cost (Challenge 2). Frequently, the algorithm identifies an optimal path by exploring states ranked lower in the priority queue. In the A* algorithm, we could discontinue exploration of a particular path after realizing that it is not the optimal path, and pivot to exploring the next best path and subsequent alternatives.

With parallelization, a more effective solution would be to create sparks on the expansion of the first k elements of the priority queue, representing the top-k potential path candidates. (k is less than or equal to the number of elements in the priority queue.) Due to the inherent determinism of Haskell and, therefore, the complexity associated with managing a global state of visited nodes across parallel evaluations, we chose to omit this consideration. We also opted to use a single priority queue which could cause race conditions (Challenge 1), but we focused on finding a solution to solve challenge 2 which we believed would yield a reasonable speedup.

```
-- Parallel processing of top 5 nodes in priority queue
astar5 :: Maze -> Maybe [Position]
astar5 maze = case go [initialNode maze] [] of
 Just node -> Just (extractPath node)
 Nothing  -> Nothing
 where
  go [] _ = Nothing
  go (current:rest) visited
   | isGoal maze (position current) = Just current
   | otherwise =
    let neighbors = filter (\n -> notElem (position n) visited) (expandNode maze
current)
      newNodes = updateCosts maze current neighbors
      sortedNodes = sort (rest ++ newNodes)
      top5 = take 5 sortedNodes  -- Select the top 5 nodes
      expandedNodes = parMap rpar (\n -> go [n] (position current : visited)) top5
    in foldr (\result acc -> case result of Just node' -> Just node' ; Nothing -> acc)
Nothing expandedNodes
```

2.2.3 ParallelMaze

Finally, one approach to implementing parallelization involves keeping the sequential A* algorithm unchanged and introducing parallelism through various static partitioning and dynamic partitioning as discussed in class. These implementations are expected to yield substantial speed improvements, as each thread can solve mazes at a pace comparable to the sequential implementation with no dependencies (challenge 1) between different states.

1. Static partitioning:

    - We split mazes to two lists as' and bs' and used rpar and force to parallelly solve two lists of mazes. We can take advantage of two cores.

    ```
    (as, bs) = splitAt (length mazes `div` 2) mazes
       solution = runEval $ do
             as' <- rpar (force (map astar as))
             bs' <- rpar (force (map astar bs))
             rseq as'
             rseq bs'
             return (as'++ bs')
    ```

2. Dynamic partitioning:

    - To assess the parallel solving of mazes utilizing additional cores, we also incorporated dynamic partitioning with the parMap' function.

    ```
    parMap' :: (a -> b) -> [a] -> Eval [b]
    parMap' _ [] = return []
    parMap' f (a:as) = do
      b <- rpar (f a)
      bs <- parMap' f as
      return (b:bs)

    -- within main function
    solution = runEval(parMap' astar mazes)
    ```

3 Evaluation and Results

3.1 ParallelCost

The initial effort to concurrently update the costs of the neighboring states revealed subpar performance improvements. This lack of success is attributed to the inherent limitations of the maze setup, where each maze has a maximum of 4 possible next states, and, therefore, a maximum of 4 costs to update. Moreover, the simplicity of Manhattan Distance used in the updateCost operation, requires minimal calculation. Attempting to update these costs in parallel introduces excessive overhead compared to the computational workload. We speculate that with a more complex cost function, there may be potential performance gains through parallel evaluation.

Table 1: Performance Comparison for ParallelCost

	Time (s)	Speed Up
Sequential	12.961	1.00x
Parallel-Cost (1-core)	12.909	1.00x
Parallel-Cost (2-core)	12.614	1.02x
Parallel-Cost (3-core)	12.737	1.02x
Parallel-Cost (4-core)	13.043	0.99x
Parallel-Cost (5-core)	12.854	1.01x

3.2 ParallelNodes

ParallelNodes demonstrates effective performance by providing a substantial speedup compared to its sequential counterpart that only looks at the node with the lowest cost. This observation challenges the notion that the state with the lowest cost in the priority queue is always the optimal solution. In our experiments, we maintain a fixed number of expanded nodes at 5.

Table 2: Performance Comparison for ParallelNodes

	Time (s)	Speed Up
Sequential	12.961	1.00x
Parallel-Nodes (1-core)	6.602	1.96x
Parallel-Nodes (2-core)	5.547	2.33x
Parallel-Nodes (3-core)	5.010	2.58x
Parallel-Nodes (4-core)	4.115	3.15x
Parallel-Cost (5-core)	3.254	3.98x

In Table 2 above, even when utilizing just one core, ParallelNodes demonstrates a speedup of nearly 2x, whereas in ParallelCost, the speedup was

approximately 1x. This observation suggests that the locking mechanism that slows down the algorithm due to contention for queue access may be outweighed by the efficiency gained in discovering the shortest path, potentially residing within one of the top 5 elements.

In cases where the number of cores does not match k(=5), the ParallelNodes algorithm resorts to simultaneously executing multiple expansion computations on a single core. As the number of cores increases, each core gains the ability to manage distinct expansions which yields a notable performance boost — roughly 3.98x times faster than the sequential implementation. While we see later that ParallelNodes' performance just falls short as compared to the performance of ParallelMaze, the speedup is quite impressive.

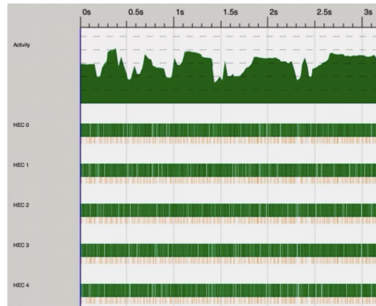

Figure 2: ParallelNodes; core=5

As illustrated in the figure above, there are no idle cores, and the distribution of tasks appears to be relatively efficient, provided the number of cores employed exceeds k. The garbage collection is pretty consistent during the entire runtime.

Table 3: Performance Comparison for ParallelNodes - Increasing Cores Used

	Time (s)	Speed Up
Parallel-Nodes (k=5, 8-core)	3.704	3.50x
Parallel-Nodes (k=5, 10-core)	3.935	3.29x

Table 4: Performance Comparison for ParallelNodes where k=10

	Time (s)	Speed Up
Parallel-Nodes (k=10, 1-core)	6.199	2.09x
Parallel-Nodes (k=10, 2-core)	4.986	2.60x
Parallel-Nodes (k=10, 4-core)	4.396	2.95x
Parallel-Nodes (k=10, 8-core)	3.859	3.35x
Parallel-Nodes (k=10, 10-core)	3.400	3.81x

From tables 3 and 4, we also found that increasing both the number of cores and the number of expanded nodes, k, to a larger value did not result in a significant improvement in the final outcome as compared to that of ParallelNodes (k=5; 5-core), and this is attributed to two primary reasons. Firstly, Amdahl's law posits that the speedup in a parallel algorithm is constrained by the extent of the sequential fraction of the task. Secondly, by increasing k, the additional threads created may explore states that are progressively less likely to represent the optimal path, given their higher associated cost f(n) = g(n) + h(n). Consequently, any potential speedup is deemed less probable.

3.3 ParallelMaze

Tables 5 and 6 show the speedup using parallelization over k-mazes.

Table 5: Performance Comparison for ParallelMaze: Static Partitioning

	Time (s)	Speed Up
Sequential	12.961	1.00x
Parallel-Maze (1-core)	11.376	1.14x
Parallel-Maze (2-core)	6.696	1.94x

The static partitioning approach employing 2 cores demonstrated commendable performance, achieving a speedup of 1.94. The program uses both cores effectively for the majority of the execution; however, core 1 becomes idle toward the end of the program, while core 2 continues its execution (Figure 3). This is likely because the first 50 example mazes might be easier to solve sequentially than the second 50 example mazes, and therefore, core 1 becomes idle while it waits for core 2 to finish executing the second half of the example mazes.

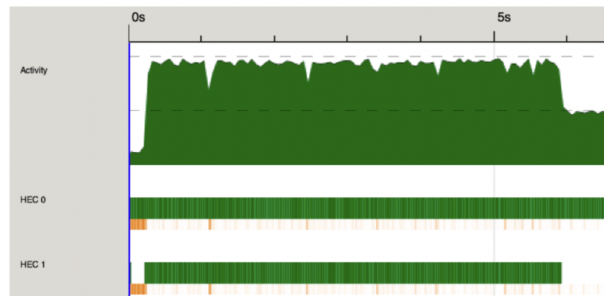

Figure 3: ParallelMaze - Static; core = 2

The parallel GC work balance was 44.98% (serial 0%, perfect 100%). There were 2 sparks created (1 converted, 1 fizzled). The fizzed spark

is likely explained by the main computation having already performed the work, thus causing it to fail to turn into a real thread.

Table 6: Performance Comparison for ParallelMaze: Dynamic Partitioning

	Time (s)	Speed Up
Sequential	12.961	1.00x
Parallel-Maze (1-core)	13.450	0.96x
Parallel-Maze (2-core)	7.193	1.80x
Parallel-Maze (3-core)	4.773	2.72x
Parallel-Maze (4-core)	3.810	3.40x
Parallel-Maze (5-core)	3.237	4.00x

By contrast, dynamic partitioning with 2 cores has a lower speedup (1.80x) than that of static partitioning with 2 cores (1.94x). However, when we increase the number of cores linearly to 5 cores, the speedup increases almost linearly to 4.00x for 5 cores. This speedup is the fastest we have seen across all parallelization strategies and hyperparameters.

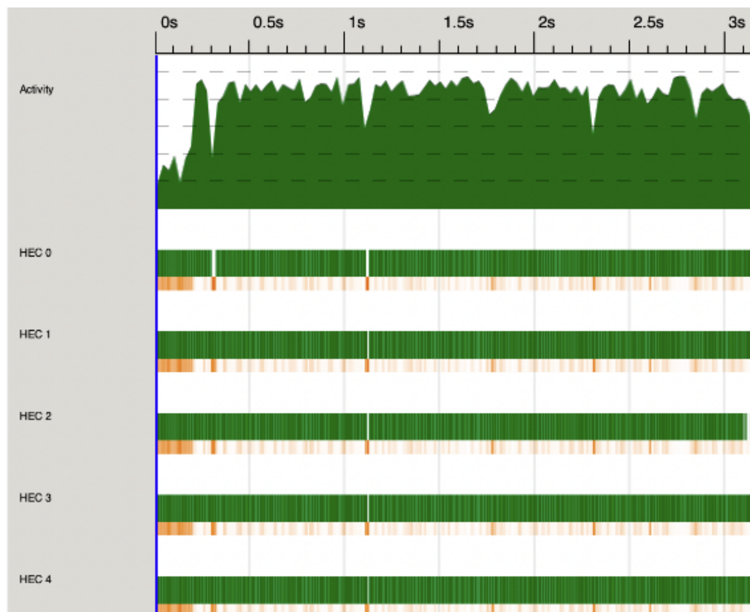

Figure 4: ParallelMaze - Dynamic; core = 5

From Threadscope, we saw that 100 sparks were created, and 100 were converted. This is excellent, as it suggests that parallelism was fully utilized in this approach. There were no wasted sparks and none were lost due to garbage collection or fizzling out/failing to turn into a real thread. At any given time stamp, almost all cores were used.

4 Conclusion

Our investigation highlights that parallelism can improve the efficiency of finding the shortest path in a maze. Nevertheless, not every aspect of the A* algorithm lends itself to parallelization for notable performance gains. Take, for instance, the updateCost function (calculation of Manhattan Distance), which is not particularly resource-intensive; hence, parallelizing such calculations does not yield substantial benefits. Among the three tested parallelization methods, the most effective was parMaze, specifically dynamic partitioning utilizing 5 cores. Its speedup was 4.00x.

5 Future Work

We hope to enhance the efficiency of our current implementation by adopting a more optimized data structure for storing nodes, considering the integration of PQueue instead of relying on a list that necessitates subsequent sorting. Additionally, we want to explore the possibility of introducing two global states to keep track of visited nodes, reducing redundant computations during the traversal process, and a priority queue, mitigating contention issues and enhancing the synchronization of parallel threads accessing and modifying the priority queue. Collectively, these refinements are anticipated to contribute to a better performance.

6 Code

6.1 maze_generator.py

```
from tkinter import *
from random import randint

def generate_maze(ms):
    visited_cells = []
    walls = []
    revisited_cells = []

    maze = [[0 for _ in range(ms)] for _ in range(ms)]

    def check_neighbours(ccr, ccc):
        neighbours = [
            [ccr, ccc - 1, ccr - 1, ccc - 2,
            ccr, ccc - 2, ccr + 1, ccc - 2,
            ccr - 1, ccc - 1, ccr + 1, ccc - 1],  # left
            [ccr, ccc + 1, ccr - 1, ccc + 2,
```

```
            ccr, ccc + 2, ccr + 1, ccc + 2,
            ccr - 1, ccc + 1, ccr + 1, ccc + 1],  # right
            [ccr - 1, ccc, ccr - 2, ccc - 1,
            ccr - 2, ccc, ccr - 2, ccc + 1,
            ccr - 1, ccc - 1, ccr - 1, ccc + 1],  # top
            [ccr + 1, ccc, ccr + 2, ccc - 1,
            ccr + 2, ccc, ccr + 2, ccc + 1, ccr + 1,
            ccc - 1, ccr + 1, ccc + 1]]  # bottom
        visitable_neighbours = []
        for i in neighbours:
            if 0 < i[0] < (ms - 1) and 0 < i[1] < (ms - 1):
                if maze[i[2]][i[3]] or maze[i[4]][i[5]] or
                maze[i[6]][i[7]] or maze[i[8]][i[9]] or
                maze[i[10]][i[11]]:
                    walls.append(i[0:2])
                else:
                    visitable_neighbours.append(i[0:2])
        return visitable_neighbours

    scr = randint(1, ms - 2)
    scc = ms - 2
    # start_color = 'Green'
    ccr, ccc = scr, scc

    maze[ccr][ccc] = 1
    finished = False
    while not finished:
        visitable_neighbours = check_neighbours(ccr, ccc)
        if len(visitable_neighbours) != 0:
            d = randint(1, len(visitable_neighbours)) - 1
            ncr, ncc = visitable_neighbours[d]
            maze[ncr][ncc] = 1
            visited_cells.append([ncr, ncc])
            ccr, ccc = ncr, ncc
        if len(visitable_neighbours) == 0:
            try:
                ccr, ccc = visited_cells.pop()
                revisited_cells.append([ccr, ccc])

            except:
                finished = True

    maze_str = ""
    flat_maze = [item for sublist in maze[1:-1] for item in sublist[1:-1]]
    output = ','.join(map(str, map(int, flat_maze)))
```

```
    return output

# Write the flattened maze to a text file
with open('maze_examples.txt', 'w') as file:
    for i in range(100):
        maze = generate_maze(102)
        file.write(maze + '\n')
```

6.2 sequential.hs

```
import Data.List (sort)
import Data.Function (on)
import Data.List.Split (splitOn)
import Data.List.Split (chunksOf)

type Position = (Int, Int)
type Maze = [[Int]]

data Node = Node
  { position :: Position
  , gCost :: Int
  , hCost :: Int
  , parent :: Maybe Node
  }

instance Eq Node where
  (==) = (==) `on` position

instance Ord Node where
  compare = compare `on` (\node -> gCost node + hCost node)

initialNode :: Maze -> Node
initialNode maze = Node {position = (0, 0), gCost = 0, hCost =
heuristic maze (0, 0), parent = Nothing}

isGoal :: Maze -> Position -> Bool
isGoal maze pos = pos == (length maze - 1, length (head maze) - 1)

expandNode :: Maze -> Node -> [Node]
expandNode maze node =
  let (x, y) = position node
      neighbors = filter (\(dx, dy) -> isValidPosition maze
      (x + dx, y + dy)) [(1, 0), (-1, 0), (0, 1), (0, -1)]
  in map (\(dx, dy) -> Node {position = (x + dx, y + dy),
  gCost = 0, hCost = 0, parent = Just node}) neighbors
```

```
updateCosts :: Maze -> Node -> [Node] -> [Node]
updateCosts maze parent nodes =
  map
    (\node ->
       let g = gCost parent + 1
           h = heuristic maze (position node)
       in node {gCost = g, hCost = h, parent = Just parent})
    nodes

-- Heuristic function: Manhattan Distance
heuristic :: Maze -> Position -> Int
heuristic maze (x, y) = abs (x - goalX) + abs (y - goalY)
  where
    goalX = length maze - 1
    goalY = length (head maze) - 1

isValidPosition :: Maze -> Position -> Bool
isValidPosition maze (x, y) =
  x >= 0 && y >= 0 && x < length maze && y < length (head maze)
  && (maze !! x !! y) == 1

extractPath :: Node -> [Position]
extractPath node = go node []
  where
    go (Node pos _ _ (Just parent)) acc = go parent (pos : acc)
    go _ acc = (0, 0) : acc

-- Sequential A*
astar :: Maze -> Maybe [Position]
astar maze = case go [initialNode maze] [] of
  Just node -> Just (extractPath node)
  Nothing   -> Nothing
  where
    go [] _ = Nothing
    go (current:rest) visited
      | isGoal maze (position current) = Just current
      | otherwise =
        let neighbors = filter (\n -> notElem (position n) visited)
        (expandNode maze current)
            newNodes = updateCosts maze current neighbors
            sortedNodes = sort (rest ++ newNodes)
        in go sortedNodes (position current : visited)

divideIntoArray :: String -> [[Int]]
divideIntoArray mazeString =
```

```
chunksOf 100 (map read (splitOn "," mazeString))

printPath :: Maybe [Position] -> IO ()
printPath Nothing = putStrLn "No path found."
printPath (Just path) = putStrLn $ "Path: " ++ show path
```

```
main :: IO ()
main = do
  content <- readFile "maze_examples.txt"
  let mazeStrings = lines content
      mazes = map divideIntoArray mazeStrings
      solution = map astar mazes

  mapM_ printPath solution
```

6.3 ParallelCost.hs

```
import Data.List (sort)
import Data.Function (on)
import Control.Parallel.Strategies (parList, rseq, withStrategy)
import Data.List.Split (splitOn)
import Data.List.Split (chunksOf)

type Position = (Int, Int)
type Maze = [[Int]]

data Node = Node
  { position :: Position
  , gCost :: Int
  , hCost :: Int
  , parent :: Maybe Node
  }

instance Eq Node where
  (==) = (==) `on` position

instance Ord Node where
  compare = compare `on` (\node -> gCost node + hCost node)

initialNode :: Maze -> Node
initialNode maze = Node {position = (0, 0), gCost = 0, hCost =
heuristic maze (0, 0), parent = Nothing}

isGoal :: Maze -> Position -> Bool
isGoal maze pos = pos == (length maze - 1, length (head maze) - 1)
```

```haskell
expandNode :: Maze -> Node -> [Node]
expandNode maze node =
  let (x, y) = position node
      neighbors =
      filter (\(dx, dy) -> isValidPosition maze (x + dx, y + dy))
      [(1, 0), (-1, 0), (0, 1), (0, -1)]
  in map (\(dx, dy) -> Node {position = (x + dx, y + dy),
  gCost = 0, hCost = 0, parent = Just node}) neighbors

updateCosts :: Maze -> Node -> [Node] -> [Node]
updateCosts maze parent nodes =
  map
    (\node ->
       let g = gCost parent + 1
           h = heuristic maze (position node)
       in node {gCost = g, hCost = h, parent = Just parent})
    nodes

-- Heuristic function: Manhattan Distance
heuristic :: Maze -> Position -> Int
heuristic maze (x, y) = abs (x - goalX) + abs (y - goalY)
  where
    goalX = length maze - 1
    goalY = length (head maze) - 1

isValidPosition :: Maze -> Position -> Bool
isValidPosition maze (x, y) =
  x >= 0 && y >= 0 && x < length maze && y < length (head maze)
  && (maze !! x !! y) == 1

extractPath :: Node -> [Position]
extractPath node = go node []
  where
    go (Node pos _ _ (Just parent)) acc = go parent (pos : acc)
    go _ acc = (0, 0) : acc

astarParallelUpdateCost :: Maze -> Maybe [Position]
astarParallelUpdateCost maze = case go [initialNode maze] [] of
  Just node -> Just (extractPath node)
  Nothing   -> Nothing
  where
    go [] _ = Nothing
    go (current:rest) visited
      | isGoal maze (position current) = Just current
      | otherwise =
```

```
        let neighbors =
        filter (\n -> notElem (position n) visited)
        (expandNode maze current)
            newNodes = withStrategy (parList rseq)
            (updateCosts maze current neighbors)
            sortedNodes = sort (rest ++ newNodes)
        in go sortedNodes (position current : visited)

divideIntoArray :: String -> [[Int]]
divideIntoArray mazeString =
chunksOf 100 (map read (splitOn "," mazeString))

printPath :: Maybe [Position] -> IO ()
printPath Nothing = putStrLn "No path found."
printPath (Just path) = putStrLn $ "Path: " ++ show path
```

```
main :: IO ()
main = do
  content <- readFile "maze_examples.txt"
  let mazeStrings = lines content
      mazes = map divideIntoArray mazeStrings
      solution = map astarParallelUpdateCost mazes

  mapM_ printPath solution
```

6.4 ParallelNodes.hs

```
import Data.List (sort)
import Data.Function (on)
import Control.Parallel.Strategies (parMap, rpar)
import Data.List.Split (splitOn)
import Data.List.Split (chunksOf)

type Position = (Int, Int)
type Maze = [[Int]]

data Node = Node
  { position :: Position
  , gCost :: Int
  , hCost :: Int
  , parent :: Maybe Node
  }

instance Eq Node where
  (==) = (==) `on` position
```

```
instance Ord Node where
  compare = compare `on` (\node -> gCost node + hCost node)

initialNode :: Maze -> Node
initialNode maze = Node {position = (0, 0), gCost = 0,
hCost = heuristic maze (0, 0), parent = Nothing}

isGoal :: Maze -> Position -> Bool
isGoal maze pos = pos == (length maze - 1, length (head maze) - 1)

expandNode :: Maze -> Node -> [Node]
expandNode maze node =
  let (x, y) = position node
      neighbors = filter (\(dx, dy) -> isValidPosition maze
      (x + dx, y + dy)) [(1, 0), (-1, 0), (0, 1), (0, -1)]
  in map (\(dx, dy) -> Node {position = (x + dx, y + dy),
  gCost = 0, hCost = 0, parent = Just node}) neighbors

updateCosts :: Maze -> Node -> [Node] -> [Node]
updateCosts maze parent nodes =
  map
    (\node ->
       let g = gCost parent + 1
           h = heuristic maze (position node)
       in node {gCost = g, hCost = h, parent = Just parent})
    nodes

-- Heuristic function: Manhattan Distance
heuristic :: Maze -> Position -> Int
heuristic maze (x, y) = abs (x - goalX) + abs (y - goalY)
  where
    goalX = length maze - 1
    goalY = length (head maze) - 1

isValidPosition :: Maze -> Position -> Bool
isValidPosition maze (x, y) =
  x >= 0 && y >= 0 && x < length maze && y < length (head maze)
  && (maze !! x !! y) == 1

extractPath :: Node -> [Position]
extractPath node = go node []
  where
    go (Node pos _ _ (Just parent)) acc = go parent (pos : acc)
    go _ acc = (0, 0) : acc
```

```
-- Parallel processing of top 5 nodes in priority queue
astar5 :: Maze -> Maybe [Position]
astar5 maze = case go [initialNode maze] [] of
  Just node -> Just (extractPath node)
  Nothing   -> Nothing
  where
    go [] _ = Nothing
    go (current:rest) visited
      | isGoal maze (position current) = Just current
      | otherwise =
        let neighbors = filter
        (\n -> notElem (position n) visited)
        (expandNode maze current)
            newNodes = updateCosts maze current neighbors
            sortedNodes = sort (rest ++ newNodes)
            top5 = take 5 sortedNodes  -- Select the top 5 nodes
            expandedNodes = parMap rpar (\n -> go [n]
            (position current : visited)) top5
        in foldr (\result acc -> case result of Just node' ->
        Just node'; Nothing -> acc) Nothing expandedNodes

divideIntoArray :: String -> [[Int]]
divideIntoArray mazeString =
chunksOf 100 (map read (splitOn "," mazeString))

printPath :: Maybe [Position] -> IO ()
printPath Nothing = putStrLn "No path found."
printPath (Just path) = putStrLn $ "Path: " ++ show path
```

```
main :: IO ()
main = do
  content <- readFile "maze_examples.txt"
  let mazeStrings = lines content
      mazes = map divideIntoArray mazeStrings
      solution = map astar5 mazes

  mapM_ printPath solution
```

6.5 ParallelMazeStatic.hs

```
    import Data.List (sort)
import Data.Function (on)
import Control.Parallel.Strategies (rpar, rseq, runEval)
import Data.List.Split (splitOn)
```

```haskell
import Data.List.Split (chunksOf)
import Control.DeepSeq (force)

type Position = (Int, Int)
type Maze = [[Int]]

data Node = Node
  { position :: Position
  , gCost :: Int
  , hCost :: Int
  , parent :: Maybe Node
  }

instance Eq Node where
  (==) = (==) `on` position

instance Ord Node where
  compare = compare `on` (\node -> gCost node + hCost node)

initialNode :: Maze -> Node
initialNode maze = Node {position = (0, 0),
gCost = 0, hCost = heuristic maze (0, 0), parent = Nothing}

isGoal :: Maze -> Position -> Bool
isGoal maze pos = pos == (length maze - 1, length (head maze) - 1)

expandNode :: Maze -> Node -> [Node]
expandNode maze node =
  let (x, y) = position node
      neighbors =
      filter (\(dx, dy) -> isValidPosition maze (x + dx, y + dy))
      [(1, 0), (-1, 0), (0, 1), (0, -1)]
  in map (\(dx, dy) -> Node {position = (x + dx, y + dy),
  gCost = 0, hCost = 0, parent = Just node}) neighbors

updateCosts :: Maze -> Node -> [Node] -> [Node]
updateCosts maze parent nodes =
  map
    (\node ->
       let g = gCost parent + 1
           h = heuristic maze (position node)
       in node {gCost = g, hCost = h, parent = Just parent})
    nodes

-- Heuristic function: Manhattan Distance
heuristic :: Maze -> Position -> Int
```

```haskell
heuristic maze (x, y) = abs (x - goalX) + abs (y - goalY)
  where
    goalX = length maze - 1
    goalY = length (head maze) - 1

isValidPosition :: Maze -> Position -> Bool
isValidPosition maze (x, y) =
  x >= 0 && y >= 0 && x < length maze && y < length (head maze)
  && (maze !! x !! y) == 1

extractPath :: Node -> [Position]
extractPath node = go node []
  where
    go (Node pos _ _ (Just parent)) acc = go parent (pos : acc)
    go _ acc = (0, 0) : acc

-- Sequential A*
astar :: Maze -> Maybe [Position]
astar maze = case go [initialNode maze] [] of
  Just node -> Just (extractPath node)
  Nothing   -> Nothing
  where
    go [] _ = Nothing
    go (current:rest) visited
      | isGoal maze (position current) = Just current
      | otherwise =
        let neighbors =
        filter (\n -> notElem (position n) visited)
        (expandNode maze current)
            newNodes = updateCosts maze current neighbors
            sortedNodes = sort (rest ++ newNodes)
        in go sortedNodes (position current : visited)

divideIntoArray :: String -> [[Int]]
divideIntoArray mazeString =
chunksOf 100 (map read (splitOn "," mazeString))

printPath :: Maybe [Position] -> IO ()
printPath Nothing = putStrLn "No path found."
printPath (Just path) = putStrLn $ "Path: " ++ show path
```

```haskell
main :: IO ()
main = do
  content <- readFile "maze_examples.txt"
  let mazeStrings = lines content
```

```haskell
    mazes = map divideIntoArray mazeStrings
    ------------------- static partitioning ------------------------
    (as, bs) = splitAt (length mazes `div` 2) mazes
    solution = runEval $ do
                 as' <- rpar (force (map astar as))
                 bs' <- rpar (force (map astar bs))
                 _ <- rseq as'
                 _ <- rseq bs'
                 return (as'++ bs')

  mapM_ printPath solution
```

6.6 ParallelMazeDynamic.hs

```haskell
import Data.List (sort)
import Data.Function (on)
import Control.Parallel.Strategies (rpar, runEval, Eval)
import Data.List.Split (splitOn)
import Data.List.Split (chunksOf)

type Position = (Int, Int)
type Maze = [[Int]]

data Node = Node
  { position :: Position
  , gCost :: Int
  , hCost :: Int
  , parent :: Maybe Node
  }

instance Eq Node where
  (==) = (==) `on` position

instance Ord Node where
  compare = compare `on` (\node -> gCost node + hCost node)

initialNode :: Maze -> Node
initialNode maze = Node {position = (0, 0),
gCost = 0, hCost = heuristic maze (0, 0), parent = Nothing}

isGoal :: Maze -> Position -> Bool
isGoal maze pos = pos == (length maze - 1, length (head maze) - 1)

expandNode :: Maze -> Node -> [Node]
expandNode maze node =
```

```
  let (x, y) = position node
      neighbors = filter (\(dx, dy) ->
      isValidPosition maze (x + dx, y + dy))
      [(1, 0), (-1, 0), (0, 1), (0, -1)]
  in map (\(dx, dy) -> Node {position =
  (x + dx, y + dy), gCost = 0, hCost = 0, parent =
  Just node}) neighbors

updateCosts :: Maze -> Node -> [Node] -> [Node]
updateCosts maze parent nodes =
  map
    (\node ->
       let g = gCost parent + 1
           h = heuristic maze (position node)
       in node {gCost = g, hCost = h, parent = Just parent})
    nodes

-- Heuristic function: Manhattan Distance
heuristic :: Maze -> Position -> Int
heuristic maze (x, y) = abs (x - goalX) + abs (y - goalY)
  where
    goalX = length maze - 1
    goalY = length (head maze) - 1

isValidPosition :: Maze -> Position -> Bool
isValidPosition maze (x, y) =
  x >= 0 && y >= 0 && x < length maze &&
  y < length (head maze) && (maze !! x !! y) == 1

extractPath :: Node -> [Position]
extractPath node = go node []
  where
    go (Node pos _ _ (Just parent)) acc = go parent (pos : acc)
    go _ acc = (0, 0) : acc

-- Sequential A*
astar :: Maze -> Maybe [Position]
astar maze = case go [initialNode maze] [] of
  Just node -> Just (extractPath node)
  Nothing   -> Nothing
  where
    go [] _ = Nothing
    go (current:rest) visited
      | isGoal maze (position current) = Just current
      | otherwise =
        let neighbors = filter
```

```
        (\n -> notElem (position n) visited)
        (expandNode maze current)
            newNodes = updateCosts maze current neighbors
            sortedNodes = sort (rest ++ newNodes)
        in go sortedNodes (position current : visited)

parMap' :: (a -> b) -> [a] -> Eval [b]
parMap' _ [] = return []
parMap' f (a:as) = do
  b <- rpar (f a)
  bs <- parMap' f as
  return (b:bs)

divideIntoArray :: String -> [[Int]]
divideIntoArray mazeString = chunksOf 100
(map read (splitOn "," mazeString))

printPath :: Maybe [Position] -> IO ()
printPath Nothing = putStrLn "No path found."
printPath (Just path) = putStrLn $ "Path: " ++ show path
```

```
main :: IO ()
main = do
  content <- readFile "maze_examples.txt"
  let mazeStrings = lines content
      mazes = map divideIntoArray mazeStrings
      ----------------- dynamic partitioning -----------------
      solution = runEval(parMap' astar mazes)

  mapM_ printPath solution
```

6.7 tests.hs

```
    import Data.List (sort)
import Data.Function (on)
import Data.List.Split (splitOn)
import Data.List.Split (chunksOf)
import qualified Test.HUnit as HUnit

type Position = (Int, Int)
type Maze = [[Int]]

data Node = Node
  { position :: Position
```

```haskell
  , gCost :: Int
  , hCost :: Int
  , parent :: Maybe Node
  }

instance Eq Node where
  (==) = (==) `on` position

instance Ord Node where
  compare = compare `on` (\node -> gCost node + hCost node)

initialNode :: Maze -> Node
initialNode maze = Node {position = (0, 0),
gCost = 0, hCost = heuristic maze (0, 0), parent = Nothing}

isGoal :: Maze -> Position -> Bool
isGoal maze pos = pos == (length maze - 1, length (head maze) - 1)

expandNode :: Maze -> Node -> [Node]
expandNode maze node =
  let (x, y) = position node
      neighbors = filter (\(dx, dy) ->
      isValidPosition maze (x + dx, y + dy))
      [(1, 0), (-1, 0), (0, 1), (0, -1)]
  in map (\(dx, dy) -> Node {position =
  (x + dx, y + dy), gCost = 0, hCost = 0, parent = Just node})
  neighbors

updateCosts :: Maze -> Node -> [Node] -> [Node]
updateCosts maze parent nodes =
  map
    (\node ->
       let g = gCost parent + 1
           h = heuristic maze (position node)
       in node {gCost = g, hCost = h, parent = Just parent})
    nodes

-- Heuristic function: Manhattan Distance
heuristic :: Maze -> Position -> Int
heuristic maze (x, y) = abs (x - goalX) + abs (y - goalY)
  where
    goalX = length maze - 1
    goalY = length (head maze) - 1

isValidPosition :: Maze -> Position -> Bool
isValidPosition maze (x, y) =
```

```
  x >= 0 && y >= 0 && x < length maze &&
  y < length (head maze) && (maze !! x !! y) == 1

extractPath :: Node -> [Position]
extractPath node = go node []
  where
    go (Node pos _ _ (Just parent)) acc = go parent (pos : acc)
    go _ acc = (0, 0) : acc

-- Sequential A*
astar :: Maze -> Maybe [Position]
astar maze = case go [initialNode maze] [] of
  Just node -> Just (extractPath node)
  Nothing   -> Nothing
  where
    go [] _ = Nothing
    go (current:rest) visited
      | isGoal maze (position current) = Just current
      | otherwise =
        let neighbors = filter
        (\n -> notElem (position n) visited)
        (expandNode maze current)
            newNodes = updateCosts maze current neighbors
            sortedNodes = sort (rest ++ newNodes)
        in go sortedNodes (position current : visited)

------------------------ Sample mazes for testing ------------------------
-- Example 1: see if A* can find the SHORTEST path
sampleMaze1 :: Maze
sampleMaze1 =
  [ [1, 1, 1, 1, 1],
    [1, 0, 1, 0, 1],
    [1, 0, 1, 0, 1],
    [1, 0, 1, 0, 1],
    [1, 1, 1, 0, 1]
  ]
-- Solution 1
sampleSolution1 :: [Position]
sampleSolution1 = [(0,0),(0,1),(0,2),(0,3),(0,4),(1,4),
(2,4),(3,4),(4,4)]
-- another path but not the shortest
-- (0,0),(1,0),(2,0),(3,0),(4,0),(4,1),(4,2),(3,2),
(2,2),(1,2),(0,2),(0,3),(0,4),(1,4),(2,4),(3,4),(4,4)

-- Example 2
sampleMaze2 :: Maze
```

```
sampleMaze2 =
  [ [1, 0, 1, 1, 1],
    [1, 0, 1, 0, 1],
    [1, 0, 1, 0, 1],
    [1, 1, 1, 0, 1],
    [1, 0, 1, 0, 1]
  ]
-- Solution 2
sampleSolution2 :: [Position]
sampleSolution2 = [(0,0),(1,0),(2,0),(3,0),(3,1),(3,2),
(2,2),(1,2),(0,2),(0,3),(0,4),(1,4),(2,4),(3,4),(4,4)]

-- Example 3: see if A* can find a path in a more complicated maze, and move
sampleMaze3 :: Maze
sampleMaze3 =
  [ [1, 1, 1, 1, 1, 1, 0, 1, 1, 0],
    [0, 0, 1, 1, 0, 1, 1, 0, 1, 0],
    [1, 1, 1, 0, 1, 0, 1, 1, 0, 1],
    [1, 1, 0, 1, 1, 1, 1, 1, 1, 1],
    [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 1, 1, 0, 1, 0],
    [0, 1, 1, 1, 0, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 1, 1, 1, 1, 1, 0],
    [0, 1, 1, 1, 1, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 1, 1, 1, 1, 1, 1] ]
-- Solution 3
sampleSolution3 :: [Position]
sampleSolution3 = [(0,0),(0,1),(0,2),(0,3),(0,4),
(0,5),(1,5),(1,6),(2,6),(3,6),(3,5),(4,5),(5,5),
(5,6),(6,6),(7,6),(7,5),(7,4),(8,4),(9,4),(9,5),
(9,6),(9,7),(9,8),(9,9)]

-- Example 4: see the edge case where there are no paths in a maze
sampleMaze4 :: Maze
sampleMaze4 =
  [ [1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0],
    [1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0],
    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1],
    [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0],
    [1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1],
    [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0],
    [1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1],
    [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0],
    [1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1],
    [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0],
    [1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1],
```

```
     [0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0],
     [1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1],
     [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1],
     [1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1]]
-- Solution 4 is Nothing

---------------------------------------------------------------------------

-- Define test cases with an expected solution
testCase1 :: HUnit.Test
testCase1 = HUnit.TestCase $ HUnit.assertEqual
  "Test Case 1"
  (Just sampleSolution1)
  (astar sampleMaze1)

testCase2 :: HUnit.Test
testCase2 = HUnit.TestCase $ HUnit.assertEqual
  "Test Case 2"
  (Just sampleSolution2)
  (astar sampleMaze2)

testCase3 :: HUnit.Test
testCase3 = HUnit.TestCase $ HUnit.assertEqual
  "Test Case 3"
  (Just sampleSolution3)
  (astar sampleMaze3)

testCase4 :: HUnit.Test
testCase4 = HUnit.TestCase $ HUnit.assertEqual
  "Test Case 4"
  Nothing
  (astar sampleMaze4)



main :: IO ()
main = do
  -- Add your test cases here
  let testSuite = HUnit.TestList
  [testCase1, testCase2, testCase3, testCase4]

  -- Run the tests
  HUnit.runTestTT testSuite >>= print
```