German University in Cairo
الجامعة الألمانية بالقاهرة

CSEN 403: Concepts of Programming Languages

16th of June, 2022

Minesweeper Robot

Authors

Team #99

Omar Hesham, 52-8724, T10

Nardy Mechael, 52-8695, T08

Omar Tamer, 52-11870, T15

Mohammed Khaled, 52-21246, T05

Introduction

The following report describes the technical implementation of a program to solve any 4x4 minesweeper board, implemented using Haskell.

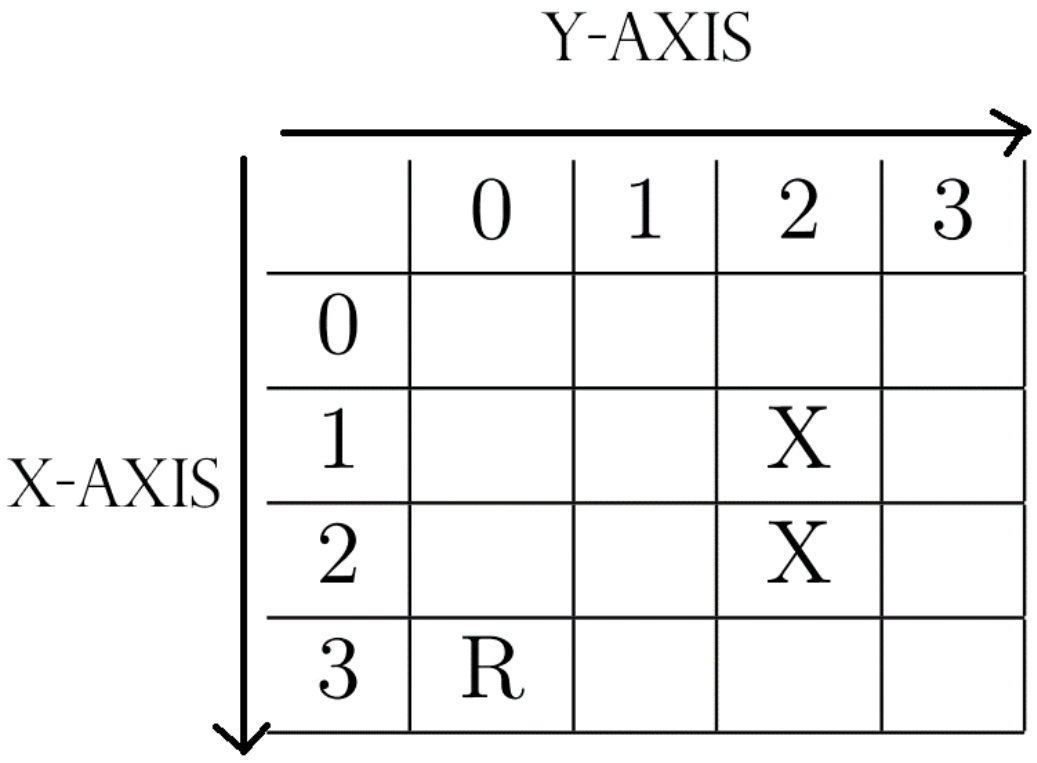

Figure (1): The board, showing one possible configuration to solve where R represents the Robot and X are mines to collect.

Implementation

This section describes the technical implementation of the robot in Haskell. It describes each function used in the implementation and what it does.

Note: Function names are written in dark red to highlight them.

Data Structures

```
type Cell = (Int, Int)
```

The representation of a cell on the board as a pair of x and y coordinates.

```
data MyState = Null | S Cell [Cell] String MyState deriving (Show, Eq)
```

The representation of any state of the board. A state can either be null, or is otherwise represented using the constructor S as:

Cell: The current cell on which the robot is located

[Cell]: The list of remaining cells the robot must visit (i.e the cells containing mines)

String: The move the robot has taken to reach the current state from the previous state

MyState: The previous state.

Note that this is a recursive data structure.

Helper Functions

Only two helper functions were used for the implementation of the robot. Their functionality will become evident as you read the rest of the report. Consequently, you may want to revisit this section during or after your reading of the report.

delete :: Cell -> [Cell] -> [Cell]

Takes as an input a cell and a list of cells and returns a list of cells from which the input cell has been removed.

removeNulls :: [MyState] -> [MyState]

Takes as an input a list of states, removes any Null states, and returns the resulting list after the removals.

Main Functions

up :: MyState -> MyState

down :: MyState -> MyState

left :: MyState -> MyState

right :: MyState -> MyState

Represent the robot making a move on the board in each respective direction. It takes as an input a state (the current state of the board) and returns the resulting state after the robot has made its move.

Due to the board's representation within the program (refer to the Introduction page), making a move modifies the coordinate of the robot's cell in the following way (assuming the robot is currently located at cell (x, y)):

Up: (x - 1, y)

Down (x + 1, y)

Left: (x, y – 1)

Right: (x,y + 1)

collect :: MyState -> MyState

Takes as an input a state. If the input state is such that the robot is currently located in a cell that contains a mine, the function returns the resulting state from collecting that mine. Otherwise, it returns Null.

nextMyStates :: MyState -> [MyState]

Returns all possible states that can result from the current state by taking any action (that is, moving in any direction or collecting)

isGoal :: MyState -> Bool

Returns true if the current state is a goal state and false otherwise.

The current state is considered a goal state if there are no more mines left to collect.

search :: [MyState] -> MyState

Takes as an input a list of states and searches for a goal state that can be obtained through any of them.

If the first state in the list is a goal, the function returns true. Otherwise, it searches the remaining list of states concatenated with the nextMyState of the first state.

constructSolution :: MyState -> [String]

The function takes as input a state and returns a list of strings representing actions that the robot can follow to reach the input state from the initial state.

solve :: Cell->[Cell]->[String]

The function takes as input a cell representing the starting position of the robot, a list of cells representing the positions of the mines the robot must collect, and returns a set of strings representing actions that the robot can follow to reach a goal state from the initial state.

The solving algorithm works in the following way:

```
solve cell listCells = constructSolution (search [(S cell listCells "" Null)])
```

It finds a solution by first searching for the goal state, then finding the sequence of moves that can be performed to reach that goal state from the input state.

Note that since the function takes as an input a cell and a list of cells, a state has to be constructed using this information in order to be searched (where the resulting state's String is an empty literal ("") and its previous state is Null)

Demonstration

This section shows two runs of the program to solve two random 4x4 boards (the program was run using the WinHugs compiler.)

```
Main> solve (2, 0) [(1, 2), (3, 3)]
["up","right","right","collect","down","down","right","collect"]
```

Figure (2): Run 1

```
Main> solve (2, 2) [(0, 1), (1, 3)]
["up","right","collect","up","left","left","collect"]
```

Figure (3): Run 2