

Goal: -

The goal was to implement a minesweeper robot in Haskell that operates on a 4x4 grid of cells, where each cell can either contain a robot or a mine, the robot can move in any of the four direction and its goal it to locate all the mines and collect them.

Getting started: -

The first thing we did was to add the following type definitions.

A cell that represents a position on the grid by its row and column.

A MyState structure that represents the state of the robot at any given time.

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

The MyState function: -

Takes as input the current state of the robot and returns the state resulting in moving in a certain direction, The function is implemented four times, once for every direction.

```

up:: MyState -> MyState
up (S (x,y) l string state) | x==0 =Null
| otherwise = S (x-1,y) l "up"(S (x,y) l string state)

down:: MyState -> MyState
down (S (x,y) l string state) | x==3 =Null
| otherwise = S (x+1,y) l "down"(S (x,y) l string state)

right:: MyState -> MyState
right (S (x,y) l string state) | y==3 =Null
| otherwise = S (x,y+1) l "right"(S (x,y) l string state)

left:: MyState -> MyState
left (S (x,y) l string state) | y==0 =Null
| otherwise = S (x,y-1) l "left"(S (x,y) l string state)

```

The Collect Function: -

Takes as input a state and returns the state resulting from collecting from the input state. Returns null if the robot is not in the same position as one of the mines.

```

collectHelper:: Cell->[Cell]->[Cell]
collectHelper c l = filter (/=c) l

collect:: MyState -> MyState
collect (S c l string state) | l==collectHelper c l =Null
| otherwise = S c l1 "collect" (S c l string state) where(l1)=(collectHelper c l)

```

The NextMyStates function: -

Takes a state and return a set of states resulting from moving in all four directions or collecting a mine.

```
nextMyStates::MyState->[MyState]
nextMyStates state= filter (/=Null) l1 where(l1)= [up state,down state,left state , right state, collect state]
```

The IsGoal Function: -

Takes a state as input and return true if the robot has no more mines to collect.

```
isGoal::MyState->Bool
isGoal (S c l string state)| l==[]=True
|otherwise=False
```

The Search Function: -

Takes a list of states and checks its contents from head to tail, if the head is a goal state it returns the head, if not, it moves up the list and calls itself recursively with the result of adding the tail of the input list with the resulting next state.

```
search::[MyState]->MyState
search (h:t)| isGoal h ==True =h
|otherwise= search (t++nextMyStates h)
```

The ConstructSolution function: -

Takes a state as input and returns a set of actions that the robot can follow to reach the input state from the initial state.

(The actions are returned in the form of strings).

```
constructSolution :: MyState -> [String]
constructSolution (S c l "" state) = []
constructSolution (S c l string state) = constructSolution state ++ [string]
```

The Solve function: -

Takes as input a cell representing the position of the robot and a set of cells representing the positions of the mines and returns a set of instructions that the robot can follow to collect the mines.

(The actions are returned in the form of strings).

```
solve :: Cell -> [Cell] -> [String]
solve c l = constructSolution (search [state]) where (state) = S c l "" Null
```

Final Results: -

```
WinHugs
File Edit Actions Browse Help
Main> :load "C:\\Users\\OMAR\\Desktop\\Concepts Project .hs"
Main> solve (3,0) [(2,2),(1,2)]
["up","right","right","collect","up","collect"]
Main> solve (0,0) [(0,3),(3,0)]
["down","down","down","collect","up","up","up","right","right","right","collect"]
Main> solve (2,1) [(0,2),(1,3)]
["up","up","right","collect","down","right","collect"]
Main> |
```

Bonus Implementation

The Bonus was about modifying the code so that it works on bigger grids, so to achieve this the following steps were taken:

Modifying MyState: -

MyState was modified to now take two int inputs (i & j) that represent the width and height of a robot or mine.

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

Resizing the grid: -

The way the new size of the grid is determined is by getting the biggest X position of all the mines and getting the biggest Y position of all the mines and setting the grid size to one plus these values. This is done using the newly implemented getX and getY functions.

```
getX :: [Cell] -> Int
getX [(x,y)] = x
getX ((x,y):(x1,y1):t) | x > x1 = getX ((x,y):t)
                        | otherwise = getX ((x1,y1):t)

getY :: [Cell] -> Int
getY [(x,y)] = y
getY ((x,y):(x1,y1):t) | y > y1 = getY ((x,y):t)
                        | otherwise = getY ((x1,y1):t)
```

The solve function is then modified to call getX and getY.

```
solve :: Cell -> [Cell] -> [String]
solve c l = constructSolution (search [state]) where (state) = S (getX (c:l)) (getY (c:l)) c l "" Null
```

To Avoid Stack Overflow: -

To avoid getting a stack overflow an algorithm was constructed to filter the states, where it checks which of the states doesn't lead to a winning scenario and removes it from the list.

```
filtersearch:: [MyState]->[MyState]
filtersearch []=[]
filtersearch l = filter (helperSearch (minimumLength l)) l

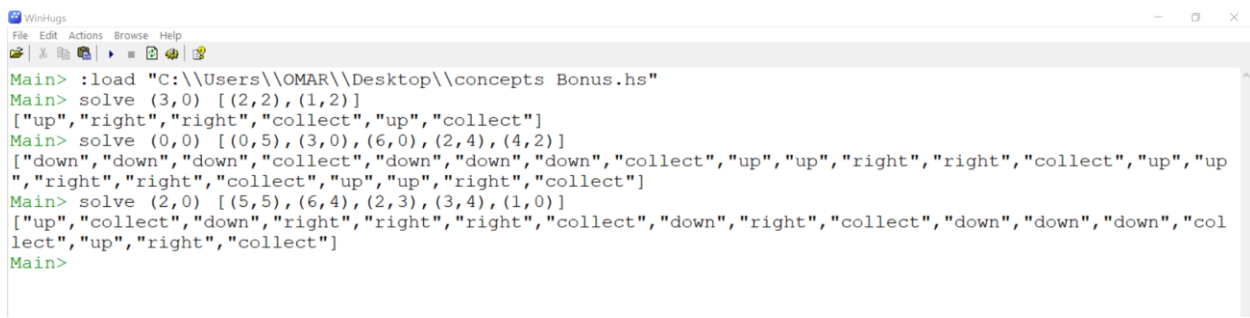
helperSearch:: Int->MyState->Bool
helperSearch x (S c l string state)| x>=length l= True
| otherwise=False

minimumLength::[MyState]->Int
minimumLength [(S c l string state)] =length l
minimumLength ((S c l string state):(S c1 l1 string1 state1):t)| (length l)< (length l1) =minimumLength ((S c l string state):t)
| otherwise=minimumLength ((S c1 l1 string1 state1):t)
```

Finally, search was modified to call filterSearch on the tail of the list.

```
search:: [MyState]->MyState
search (h:t)| isGoal h ==True =h
| otherwise= search (filtersearch(t)++nextMyStates h)
```

Final Results: -



```
WinHugs
File Edit Actions Browse Help
Main> :load "C:\\Users\\OMAR\\Desktop\\concepts Bonus.hs"
Main> solve (3,0) [(2,2), (1,2)]
["up", "right", "right", "collect", "up", "collect"]
Main> solve (0,0) [(0,5), (3,0), (6,0), (2,4), (4,2)]
["down", "down", "down", "collect", "down", "down", "down", "collect", "up", "up", "right", "right", "collect", "up", "up", "right", "right", "collect", "up", "up", "right", "collect", "up", "up", "right", "collect"]
Main> solve (2,0) [(5,5), (6,4), (2,3), (3,4), (1,0)]
["up", "collect", "down", "right", "right", "right", "collect", "down", "right", "collect", "down", "down", "down", "collect", "up", "right", "collect"]
Main>
```