The functions:

1-up: this function takes an input of type MyState and returns also MyState, it makes the robot move one cell upwards and checks if it is going out of the board by comparing its row -1 if it is less than 0 it returns null else it returns its new state.

2-down: this function takes an input of type MyState and returns also MyState, it makes the robot move one cell downwards and checks if it is going out of the board by comparing its row+1 if it is more than 3 (the grid width -1) it returns null else it returns its new state.

3- left: this function takes an input of type MyState which has and returns also MyState, it makes the robot move one cell to the left and checks if it is going out of the board by comparing its column -1 if it is less than 0 it returns null else it returns its new state.

4- right: this function takes an input of type MyState which has and returns also MyState, it makes the robot move one cell to the right and checks if it is going out of the board by comparing its column+1 if it is more than 3 (the grid length -1) it returns null else it returns its new state.

5- collect: this function takes an input of type MyState which has and returns also MyState, it helps the robot to collect the mines and removing the mine's cell from the list of mines by using helper method called **(collectHelper) :**which takes as an input a cell (the robot's position) and a list of cells (the mines' positions) and returns Boolean, it checks whether the robot is in the same position with any mine or not, also by using another helper method called :(**collectHelper2):** which takes as input a cell and a list of cells ,it removes the cell in which the robot has collected the mine in it from the list of mines and return the new list .

6 - nextMyStates: this function takes an input of type MyState which has and returns a list of MyState representing all possible actions a robot can take, by concatenating all helpers representing all actions in project

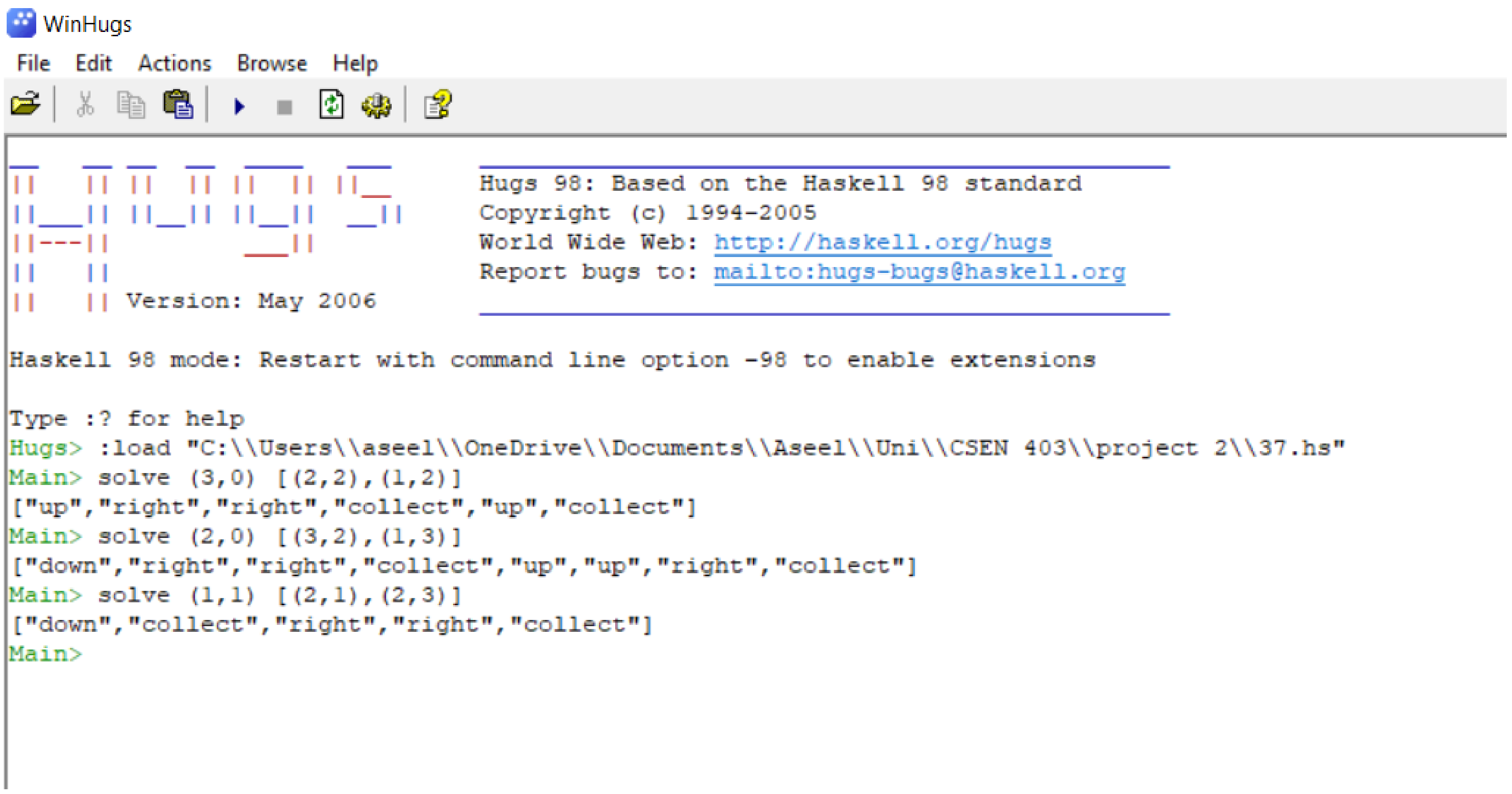
- **a-** checkUp: which checks if moving up would result in an out of board movement(null) then it returns an empty list (nothing to be concatenated) else it returns a list of the function up
- **b-** checkDown: which checks if moving down would result in an out of board movement(null) then it returns an empty list (nothing to be concatenated) else it returns a list of the function down
- **c-** checkLeft: which checks if moving left would result in an out of board movement(null) then it returns an empty list (nothing to be concatenated) else it returns a list of the function left
- **d-** checkRight: which checks if moving right would result in an out of board movement(null) then it returns an empty list (nothing to be concatenated) else it returns a list of the function right
- **e-** checkCollect: which checks if there isn't a mine to collect where the robot stands(null) then it returns an empty list (nothing to be concatenated) else it returns a list of the function collect

7- isGoal: this function takes a state as input and checks if the list of cells (list of mines to collect) of the state is empty return true else return false.

8- search: this function takes a list of states as input and return the state that achieves the goal (of collecting all mines) by checking if the first state of the list if it achieved the goal (isGoal) it will return it else it will get all possible moves(states) from this state with (nextMyStates) and concatenate the states to the tail of the list and keep recursing till finding the winning state

9-constructSolution : this function takes a state and return a string representing all moves taken to achieve this state from input state by concatenating a list of the string in a state (representing the move) with the recursion on the parent state

10- solve: this function takes a cell representing the position of the robot and a list of cells representing the positions of the mines and return a string representing the actions that could be taken to collect all mines by calling the function constructSolution on the output of the function search which takes the initial state as an input



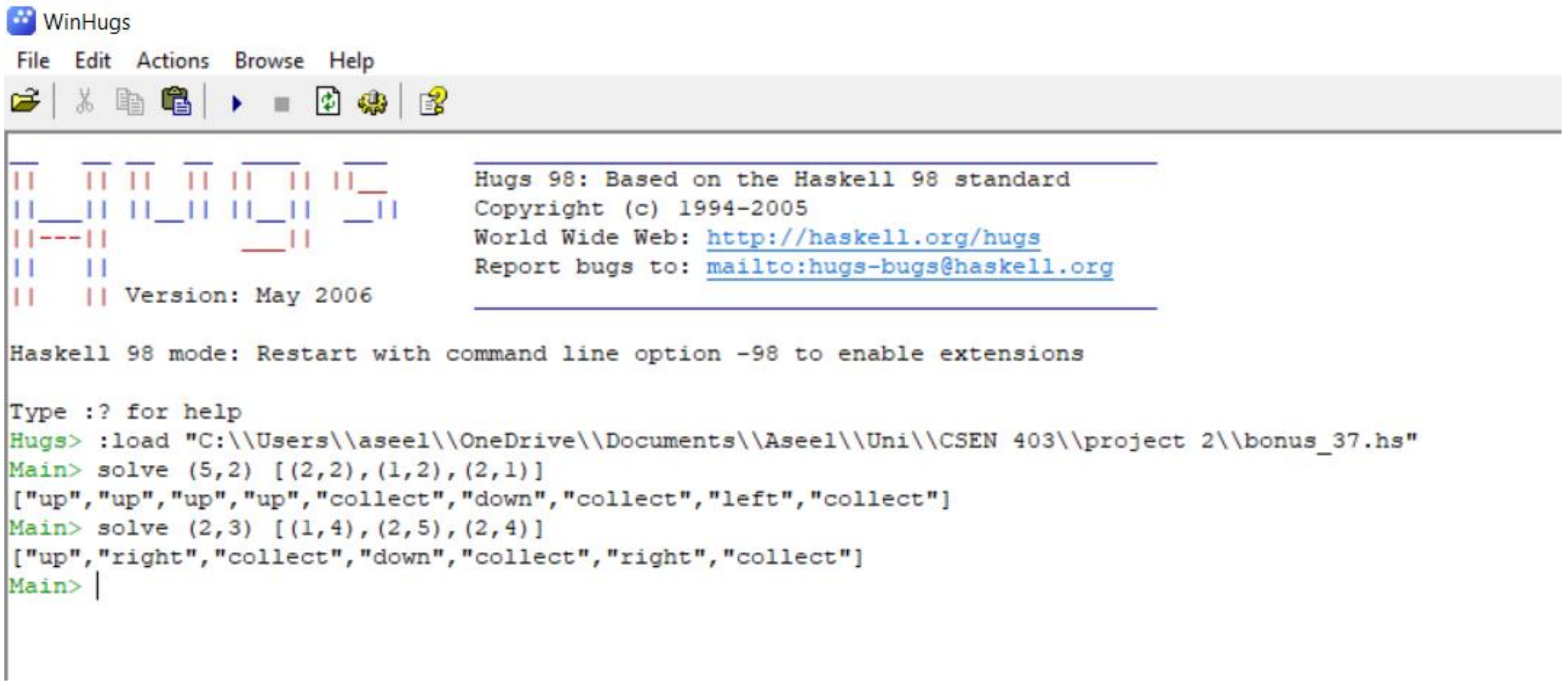
```
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: mailto:hugs-bugs@haskell.org
||   || Version: May 2006       _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "C:\\Users\\aseel\\OneDrive\\Documents\\Aseel\\Uni\\CSEN 403\\project 2\\37.hs"
Main> solve (3,0) [(2,2),(1,2)]
["up","right","right","collect","up","collect"]
Main> solve (2,0) [(3,2),(1,3)]
["down","right","right","collect","up","up","right","collect"]
Main> solve (1,1) [(2,1),(2,3)]
["down","collect","right","right","collect"]
Main>
```

For the bonus implementation:

down , right, nextMyStates , checkDown , checkRight, search: was modified to take 2 integer inputs representing the dimensions of the grid the robot will walk in that will be obtained from the solve function with using two helper functions

a- **getSize:** that takes a list of cells representing mines' positions concatenated to robot's position and integer zero representing maximum number so far and return an integer representing the actual maximum integer by comparing the row of the first cell with the maximum so far till the list is empty to obtain the maximum dimension of the rows

b- **getSize2:** this function does the same as the previous one but on the columns



```
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: mailto:hugs-bugs@haskell.org
||   || Version: May 2006       _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "C:\\Users\\aseel\\OneDrive\\Documents\\Aseel\\Uni\\CSEN 403\\project 2\\bonus_37.hs"
Main> solve (5,2) [(2,2),(1,2),(2,1)]
["up","up","up","up","collect","down","collect","left","collect"]
Main> solve (2,3) [(1,4),(2,5),(2,4)]
["up","right","collect","down","collect","right","collect"]
Main> 
```

Code

```
type Cell = (Int,Int) --(row,column)
data MyState = Null | S Cell [Cell] String MyState deriving (Show,Eq) -- s(robotPosition [minesPositions] lastAction parentStateBeforelastAction)

up :: MyState -> MyState
up (S (a,b) h j k) = if a-1 <0 then Null else S (((a-1),b)) h "up" (S(a,b) h j k)

down :: MyState -> MyState
down (S (a,b) h j k) = if a+1 >3 then Null else S (((a+1),b)) h "down" (S(a,b) h j k)

left :: MyState -> MyState
left (S (a,b) h j k) = if b-1 <0 then Null else S ((a,(b-1))) h "left" (S(a,b) h j k)

right :: MyState -> MyState
right (S (a,b) h j k) = if b+1 >3 then Null else S ((a,(b+1))) h "right" (S(a,b) h j k)

collect:: MyState -> MyState
collect (S p c j k ) = if collectHelper p c == False then Null else S p (collectHelper2 p c ) "collect" (S p c j k)

collectHelper :: Cell -> [Cell] -> Bool
collectHelper _ []  = False
collectHelper (a,b) ((x,y):t)  = if a==x && b==y then True else collectHelper (a,b) t

collectHelper2 :: Cell -> [Cell] -> [Cell]
collectHelper2 _ [] = []
collectHelper2 (a,b) ((x,y):t)  = if a==x && b==y then t else (x,y): collectHelper2 (a,b) t

nextMyStates :: MyState -> [MyState]
nextMyStates s = checkUp s ++ checkDown s ++ checkLeft s ++ checkRight s ++ checkCollect s ++ []

checkUp :: MyState -> [MyState]
checkUp s = if up s == Null then [] else [up s]

checkDown :: MyState -> [MyState]
checkDown s = if down s == Null then [] else [down s]

checkLeft :: MyState -> [MyState]
checkLeft s = if left s == Null then [] else [left s]

checkRight :: MyState -> [MyState]
checkRight s = if right s == Null then [] else [right s]

checkCollect  :: MyState -> [MyState]
checkCollect s = if collect s == Null then [] else [collect s]
```

```haskell
isGoal::MyState->Bool
isGoal (S p c j k ) = if c == [] then True else False

search::[MyState]->MyState
search (h:t) = if isGoal h then h else search (t ++ nextMyStates h)

constructSolution :: MyState -> [String]
constructSolution Null = []
constructSolution (S _ _ "" _) = []
constructSolution (S p c j k )= constructSolution k  ++ [j]

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