Report

Brief description of the implemented functions

(1) Types

- **type Cell = (Int , Int):** defining a type cell which is a pair of 2 integers.
- **data MyState = Null | S Cell [Cell] String MyState deriving (Show,Eq) :** defining a new data type that is either Null or has a constructor S that has 4 parameters.

1) a cell: that is the location of the robot (defined previously).
2) a list of cells: locations of the mines.
3) a String: describing the action performed to reach this location(state).
4) another state: the previous state of type myState (recursive).
deriving show in order for the new type to be visible.
deriving Eq in order to be Checkable for equality.

(2) Actions

• up:: MyState -> MyState:
The function takes as input a state and returns the state resulting from moving up from the input state.

It moves up by subtracting 1 from the x (rows) and setting the string to “up” and setting the initial state to the current input state.

If up will result in going out of the boundaries (x<0) of the grid, Null should be returned.

• down:: MyState -> MyState:
The function takes as input a state and returns the state resulting from moving down from the input state.

It moves down by adding 1 to the x (rows) and setting the string to “down” and setting the initial state to the current input state.

If down will result in going out of the boundaries (x<3) of the grid, Null should be returned.

• left:: MyState -> MyState:
The function takes as input a state and returns the state resulting from moving left from the input state.

It moves left by adding 1 to the y (columns) and setting the string to “left” and setting the initial state to the current input state.

If left will result in going out of the boundaries (y>3) of the grid, Null should be returned.

• right:: MyState -> MyState:

The function takes as input a state and returns the state resulting from moving right from the input state.

It moves right by subtracting 1 from the y (columns) and setting the string to “right” and setting the initial state to the current input state.

.

If right will result in going out of the boundaries (y<0) of the grid, Null should be returned.

• collect:: MyState -> MyState:

The function takes as input a state and returns the state resulting from collecting the mine from the remaining mines in the input state if the location of the robot and the mine are the same. Collecting should not change the position of the robot, but removes the collected mine from the list of mines to be collected. If the robot is not in the same position as one of the mines, Null should be returned.

We are doing so by:

- checking if the location of the robot at this states happens to be in the list of mines using predefined function(elem).
- Changing the string to be “collect”.
- Removing the location of the robot (the collected mine) from the list of mines using the predefined function (filter/=).
- Set the initial state to the current state.

(3) Steps to solve:

• nextMyStates::MyState->[MyState]

The function takes as input a state and returns the set of states resulting from applying up, down, left, right, and collect from the input state. The output set of states should not contain any Null states.

We are doing so by:

- Applying the previously defines action functions on the current input state.
- Putting them in a list.
- Removing any (Null) state resulting from the action functions using the predefined function (filter/=).

• isGoal::MyState->Bool:

The function takes as input a state, returns true if the input state has no more mines to collect (the list of mines is empty), and false otherwise.

We are doing so by checking that the length of the list of mines became empty using the predefined function (length).

• search::[MyState]->MyState:

The function takes as input a list of states. It checks if the head of the input list is a goal state (using the previously defined function (isGoal), if it is a goal, it returns the head. Otherwise, it gets the next states from the state at head of the input list using the previously defined function (nextMyStates), and calls itself recursively with the result of concatenating the tail of the input list with the resulting next states. the order of the concatenation here is important, the next states must be placed by the end of the list that's why we are concatenating the result of nextMyStates to the end of the tail list using (++).

• constructSolution:: MyState ->[String]:

The function takes as input a state and returns a set of strings representing actions that the robot can follow to reach the input state from the initial state. The possible strings in the output list of strings are only "up", "down", "left", "right", and "collect".

we are doing so by adding the strings in a list starting with (the head) the current string of the input state (the last action preformed) followed by the previous actions by calling the function recursively and adding the result to the tail.

Base case is the state with the empty string “ ” which is the very initial state of the robot before taking any actions.

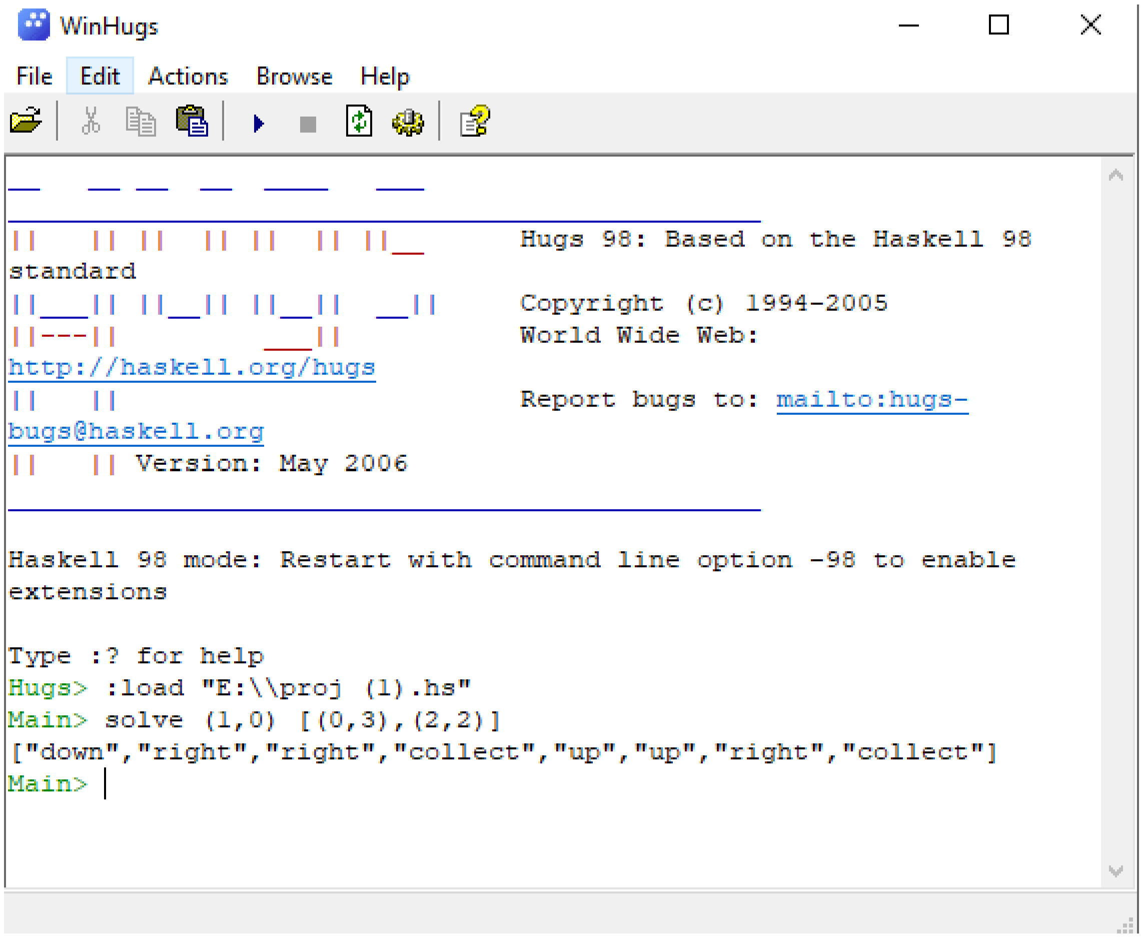
• solve :: Cell->[Cell]->[String]: which is the last step

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

We are doing so by:

- Find all possible actions that the robot could take starting from the initial input state using the previously defined function (nextMyStates).
- Searching for a solution in that resulting list using the previously defined function(search).
- Constructing a solution (a list of string of the possible actions taken) from the state resulting from the search function, using the previously defined function (constructSolution).
- Reversing the result to display the steps of action from initial step to last step, using the predefined function (reverse).

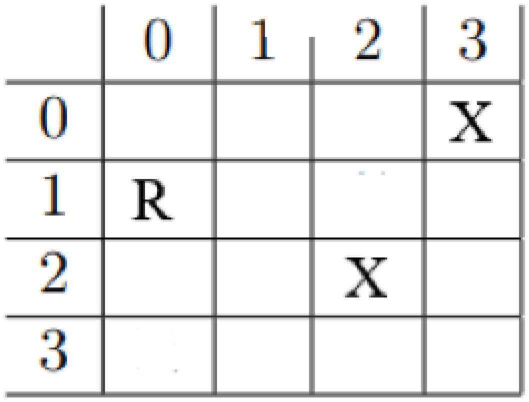
two different grid configurations



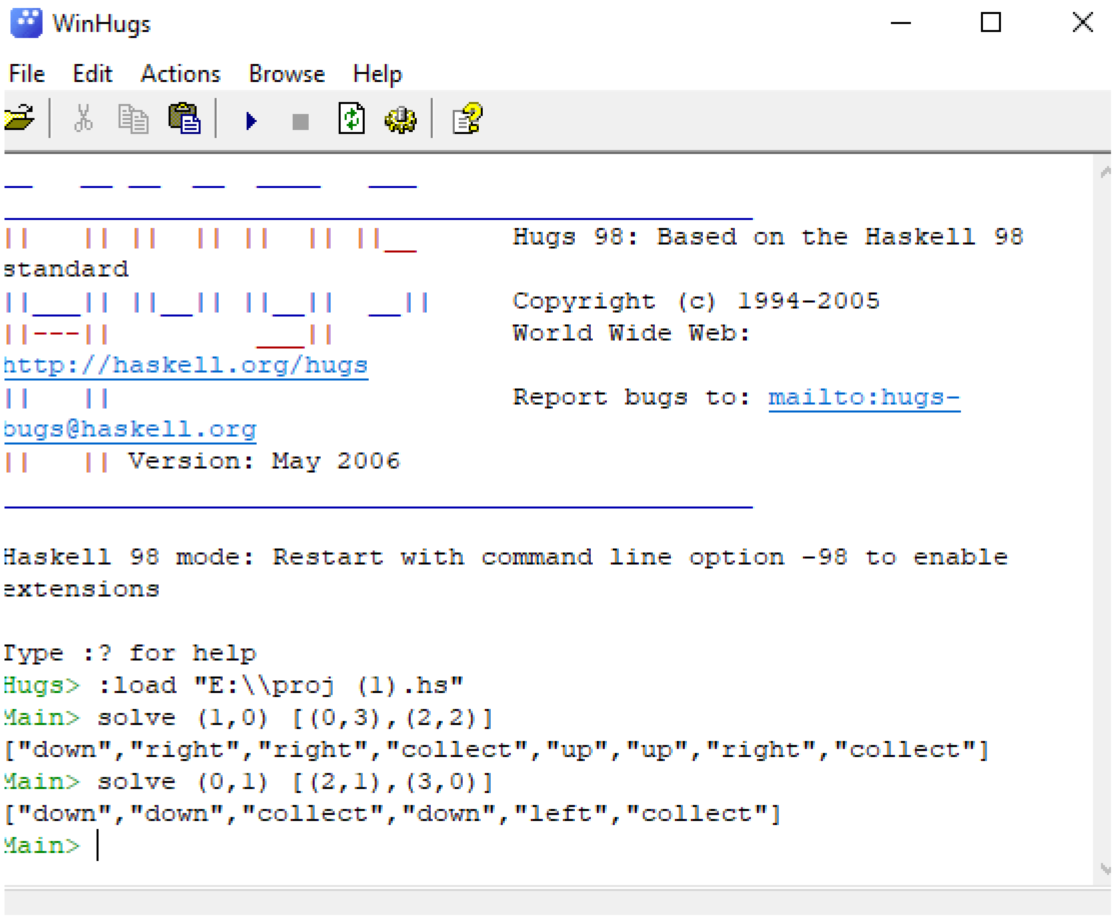
```
__   __ __  __  ____   ___         _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         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 "E:\\proj (1).hs"
Main> solve (1,0) [(0,3),(2,2)]
["down","right","right","collect","up","up","right","collect"]
Main>
```



	0	1	2	3
0				X
1	R			
2			X	
3				



```
__   __ __  __  ____   ___
||   || ||  || ||  || ||__      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 "E:\\proj (1).hs"
Main> solve (1,0) [(0,3),(2,2)]
["down","right","right","collect","up","up","right","collect"]
Main> solve (0,1) [(2,1),(3,0)]
["down","down","collect","down","left","collect"]
Main> 
```

	0	1	2	3
0		R		
1				
2		X		
3	X			