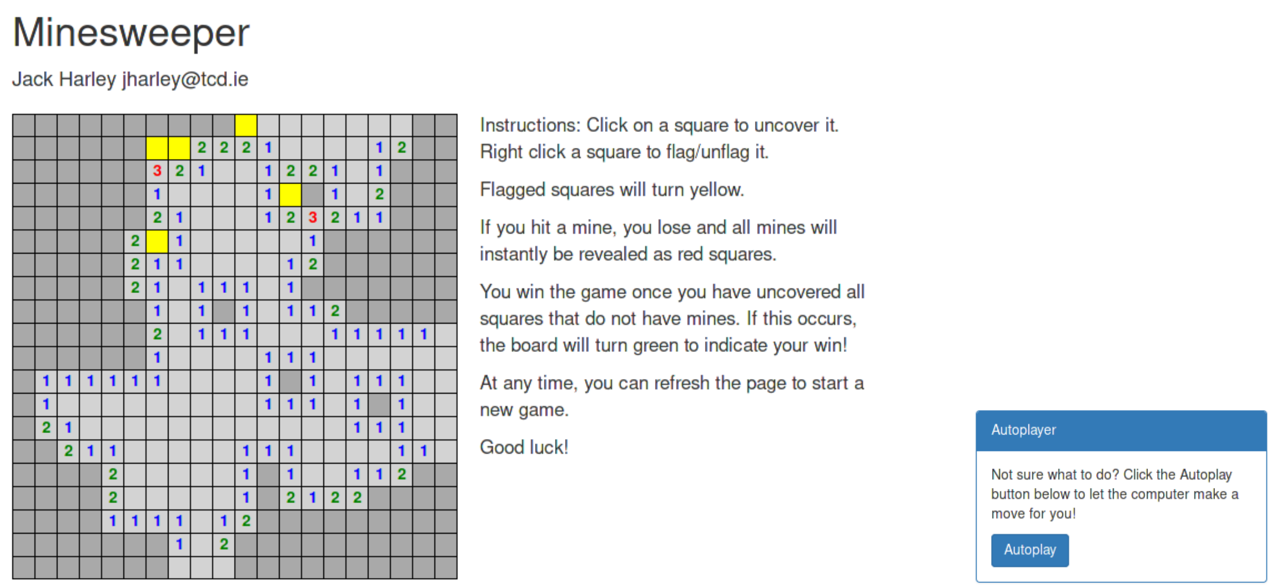
CSU44012 Topics in Functional Programming
Assignment #2
Minesweeper

Jack Harley jharley@tcd.ie — Student No. 16317123

January 2021



Contents

1 Introduction

I have implemented a fully functional Minesweeper game in Haskell with the Threepenny GUI serving the interface, and also an autosolver/autoplayer which can perform a move by clicking an "Autoplay" button.

The code is well commented and I have also documented and explained key parts of it in this PDF.

Stack successfully builds and executes the solution binary, serving the GUI at `http://localhost:8023`.

I attempted to integrate it into an Electron application, so that the interface would launch automatically into an Electron window (embedded Chrome, see `https://www.electronjs.org/`) but unfortunately realized partway through that the effort required to get it working was likely to be disproportionate to the improvement in functionality.

2 Design and Implementation

I will cover a few of the more important functions in some detail in this section. The comments included in the source files should be sufficient to explain the simpler function.

2.1 Basic Minesweeper Model

The model for the game is implemented in Minesweeper.hs.

I modelled the game board as an ADT with 4 fields:

```
data Board = Board { size :: Int, mines :: Grid, uncovered :: Grid, flagged :: Grid }
```

size defines the horizontal and vertical length in squares of the game grid (all grids are squares). **mines**, **uncovered** and **flagged** hold data structures that respectively indicate the squares that have mines, have been uncovered by the user and have been flagged by the user.

The Grid type is a 2D list of Booleans with the outer list denoting rows and the inner list denoting columns:

```
type Grid = [[Bool]]
```

With this structure, you can determine whether the 2nd row down, 4th column across has a mine with the following simple expression: (N.B. rows and columns are 0-indexed)

```
(mines !! 1) !! 3
```

And indeed this is how the hasMine function is implemented in my code:

```
hasMine :: Board -> Square -> Bool
hasMine b (r,c) | validSquare b (r,c) = (mines b !! r) !! c
```

Throughout my implementation, particular squares are referred to with a 2-tuple defining first the row and then the column as 0-indexed integers, with (0,0) being the square at the top left of the board:

```
type Square = (Int, Int)
```

2.2 Creating a Game

A fresh game board is initialised with the createBoard function:

```
createBoard :: Int -> Float -> StdGen -> Board
createBoard size mineRatio rng = Board size (seedGrid rng mineRatio (createGrid False size))
                                            (createGrid False size)
                                            (createGrid False size)
```

The function requires a size (number of squares in both horizontal and vertical directions), a "mine ratio" and a random number generator instance. It then produces a Board instance with three initialised grids. The uncovered and flagged grids are initialised with all False values (since the user will not have uncovered or flagged any squares yet).

The mines grid is initialised with all False values, but then the seedGrid function is used to randomly seed mines into the grid by making a random decision with probability of the provided mineRatio for every square on the grid.

For example, with a mine ratio of 0.1, every square will have a one in ten chance of having a mine, and after the decisions have been made for every square, roughly one tenth of the grid will have mines.

seedGrid works by splitting the random number generator repeatedly, one instance for each row of the grid, and then calling seedList on each row. The seeded rows are then joined back together at the end of the recursion. The full source for seedGrid, seedList and seedList' can be found in the appendix and project files.

2.3 Handling Game Moves

2.3.1 Uncover

Uncover is triggered in the UI by left clicking on a square. It triggers the following function:

```
uncover :: Board -> Square -> Board
uncover b s | not $ validSquare b s = b
            | isUncovered b s = b
            | hasMine b s = Board (size b) (mines b) (createGrid True (size b)) (flagged b)
            | otherwise = uncoverRecurse
                  (Board (size b) (mines b) (modSquare (uncovered b) s True) (flagged b)) s
```

The first guard handles cases where the provided Square is not valid (lies outside the edge of the Board), in this case, the Board is returned unchanged.

The second guard handles cases where the provided Square is already uncovered, and again the board is returned unchanged.

The third guard handles cases where a user clicks on a mine. In this case the game has ended and the player has lost, therefore the function simply replaces the uncovered Grid with an all True Grid. The user will therefore immediately see the entire grid, including all of the mines.

The final guard handles normal cases where the Square clicked is safe. It reconstructs the uncovered Grid, replacing the Square at s with a True status using the modSquare function. It then also calls the uncoverRecurse function on the modified Board:

```haskell
uncoverRecurse :: Board -> Square -> Board
uncoverRecurse b s | adjacentMines b s == 0 = uncoverAll b $ adjacentSquares b s
                   | otherwise = b
```

uncoverRecurse checks if the newly uncovered Square has 0 adjacent mines, and if so, uncovers all of them. This can trigger recursion where large parts of the Board will be uncovered.

2.3.2 Flag

Flag is triggered in the UI by right clicking on a square. It is intended to be used when a user wants to mark a square they think has a mine. It functions as a toggle, so a user can also unflag a square. It triggers the following function:

```haskell
flag :: Board -> Square -> Board
flag b s | not $ validSquare b s = b
         | isUncovered b s = b
         | isFlagged b s = Board (size b) (mines b) (uncovered b) (modSquare (flagged b) s False)
         | otherwise = Board (size b) (mines b) (uncovered b) (modSquare (flagged b) s True)
```

Flag works similarly to uncover.

If the square is not valid or already uncovered then the board is returned unchanged. If the square is already flagged then we toggle it to unflagged, and if it doesn't match any of the guards then we toggle it to flagged.

2.4 Program Startup

The entry point for the program is the main function. It calls startGUI with the setup function as a parameter. ThreePenny then initialises using the setup function.

Setup registers two stylesheets: the minified bootstrap.min.css I use in all of my web related projects and a minesweeper.css file I wrote to give some styling to the page and the game board.

Setup then creates a new pseudo random number generator and initialises a new game board. It then stores the game board state into a global IORef. This IORef will be modified when updating state due to a user action, and read from during a re-render of the board. The unwrapped Board instance (b) is also maintained and passed to the rendering functions for the initial render:

```haskell
rng <- liftIO newStdGen
let b = createBoard 20 0.08 rng :: Board
iob <- liftIO $ newIORef b
```

The body of the page is then rendered, one part to note is the custom JS I include at the bottom of the body:

```haskell
mkElement "script" # set (attr "src") "/static/custom.js"]
```

This custom.js script is a one-liner which prevents right clicks from opening a menu when playing the game:

```javascript
document.addEventListener('contextmenu', event => event.preventDefault());
```

2.5 Rendering the Game Board

The board itself is rendered as a table with ID ”table” in a div with ID ”gameCont”:

```
UI.div # set UI.id_ "gameCont" #+ [mkElement "table" # set UI.id_ "table" #+ rows iob b 0]
```

This calls the rows function with an IORef Board and an already unwrapped copy of the Board. Rows then recursively calls cells to render each row of cells, and cells calls the cell function to render the individual cells:

```
rows iob b r | r < size b = (mkElement "tr" #+ cells iob b r 0) : rows iob b (r+1)
             | otherwise = []

cells iob b r c | c < size b = cell iob b (r,c) : cells iob b r (c+1)
                | otherwise = []
```

The cell function receives an (r,c) pair and renders that specific cell, calling functions from Minesweeper.hs to determine the desired background colour, text colour and text to display in each cell.

Two event handlers are also attached on to the cell to handle left and right clicks, triggering an update of the IORef Board by uncovering or flagging a square respectively.

```
cell iob b (r,c) = do
    cell <- mkElement "td" #. (squareBgColour b (r,c) ++ " " ++ squareTextColour b (r,c))
                           #+ [string $ squareAscii b (r,c)]

    on UI.click cell $ \_ -> do
        liftIO $ modifyIORef' iob $ \oldB -> uncover oldB (r,c)
        refresh iob

    on UI.contextmenu cell $ \_ -> do
        liftIO $ modifyIORef' iob $ \oldB -> flag oldB (r,c)
        refresh iob

    return cell
```

You will note that after an update to the IORef Board due to a player move, the refresh function is called. This function reads an up to date copy of the Board state from the IORef Board and re-renders the game board, replacing the old copy:

```
refresh iob = do
    b <- liftIO $ readIORef iob

    table <- getElementById w "table"

    cont <- getElementById w "gameCont"
    let cont' = return $ fromJust cont
    cont' #+ [mkElement "table" # set UI.id_ "table" #+ rows iob b 0]
    when (isJust table) $ delete (fromJust table)

    -- workaround
    newTable <- getElementById w "table"
    when (isNothing newTable) $ do
    liftIO $ putStrLn "Render failed, triggering repeat"
    refresh iob
```

The final section of this refresh function is a workaround for an issue I had with ThreePenny. ThreePenny occasionally (approx 1 in every 20 moves or so) fails to re-render the table when asked, instead producing an empty container. Despite extensive debugging I have been unable to pinpoint the cause. I suspect there is a bug in ThreePenny itself at this point, though I cannot rule out a mistake of my own. The workaround tests if the new table was created, and if not, repeats the refresh (this always fixes the problem). Theoretically I suspect the issue could occur twice in a row (though I have not seen this happen), in the case that it does the function should recursively run until a successful render occurs.

2.6 Autosolver

My autosolving code is implemented in Autosolver.hs. When a user clicks the Autoplay button it fires a call to playAutoMove, which calls nextMove to determine a move and then plays it, returning the modified board:

```
playAutoMove :: Board -> Board
playAutoMove b | fst (nextMove b) == Uncover = uncover b $ snd (nextMove b)
               | fst (nextMove b) == Flag = flag b $ snd (nextMove b)
               | otherwise = b

nextMove :: Board -> (MoveType, Square)
nextMove b | not . null $ uncoverStrat b = (Uncover, head $ uncoverStrat b)
           | not . null $ flagStrat b = (Flag, head $ flagStrat b)
           | not . null $ uncoverStratFallback b = (Uncover, head $ uncoverStratFallback b)
           | otherwise = (None, (0,0))
```

The prioritisation for strategies is shown in nextMove, first it will attempt a move from the uncoverStrat, if none are available then it will try a move from flagStrat, and finally it will fallback to the uncoverStratFallback, which simply uncovers the first covered square on the board.

The strategies work by returning a list of possible moves that could be made in the form of 2-tuples with the first element being either Uncover, Flag or None, and the second the square to perform the move on.

uncoverStrat looks for an uncovered square with at least 1 covered square adjacent to it, and any possible mines have already been accounted for by flagging. It then uncovers all adjacent squares (since they must be safe).

flagStrat looks for an uncovered square with at least 1 covered square adjacent to it, where the number of adjacent squares is equal to the number of adjacent mines. It can then flag all of those squares (since they are guaranteed to be mines).

Together these two strategies consistently solve any game with a relatively low mine ratio.

As an example of the code, here is the uncoverStrat:

```
uncoverStrat :: Board -> [Square]
uncoverStrat b =
    filter (not. isFlagged b) $
    filter (isCovered b) $
    concatMap (adjacentSquares b) $
    filter (\s -> adjacentCovereds b s > adjacentFlags b s) $
    filter (\s -> adjacentFlags b s == adjacentMines b s) $
    filter (\s -> adjacentMines b s > 0) $
    uncoveredSquares b
```

As you can see it works quite elegantly (in my opinion) by progressively filtering and transforming sets of squares to a safe set to uncover. The logic for each step of the processes are detailed in textual form in comments above the strategies.

3 Reflection

This was an interesting project and I have learned a lot from it. I think Haskell worked excellently for the project, it continues to feel slightly magical to me after I've finished producing a set of functions and they all neatly slot into each other in such an intuitive way.

On reflection I'm not hugely happy with the core data structures I chose to represent the game state and believe I fell into the trap of thinking like an imperative programmer when designing them. 2D lists initially seemed like an intuitive and efficient way to store the state. However, when it came to writing the uncover and flag functions I realised it was not a particularly optimal choice. Modification of a single element required splitting the lists up two levels deep which made for some overly complex code. I also suspect that this approach is not particularly efficient for data access or modification.

If I was to entirely redesign the project I would try implementing it with a different core data structure. Possible options include a 2D Data.Array structure that would function somewhat similarly to the current approach, or possible an association list or Data.Map based structure (which would be indexed by the (row,col) tuples).

I found working with ThreePenny difficult initially, there are limited examples on the web for usage and it took me quite a bit of time wrestling with it before I had some code with a reasonable structure for the main interface setup section. I would've liked to see a method of including static HTML into a page without having to write all of it in the ThreePenny eDSL and an ability to prevent the default action triggered by a browser event occurring (in my case right click opening a context menu) without having to embed a custom script.

In general, I'm also not a huge fan of user interfaces in HTML using a browser/embedded Chromium (Electron) to execute them. I think these apps are quite wasteful in terms of memory usage. If I produce a GUI app in the future with Haskell I think I'll look into GTK/Qt bindings, particularly since I have experience using them with other languages.