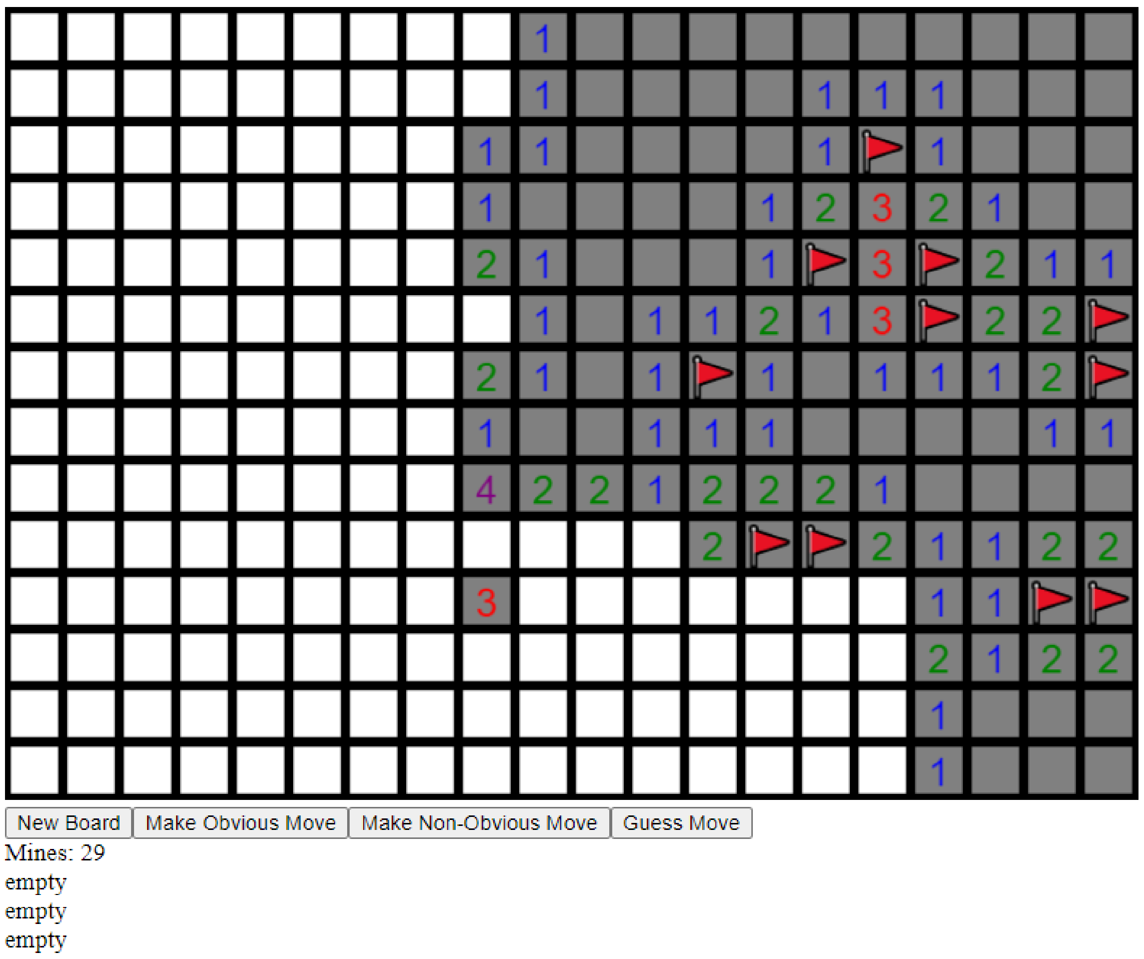
Minesweeper In Haskell

Tom Moran - 17328981

Overview



```
Mines: 29
empty
empty
empty
```

Three files make up this minesweeper game and minesweeper solver: minesweeper.hs, Main.hs and AIPlayer.hs. The game is fully implemented with all the features you would expect from a regular minesweeper clone. The solver performs well and implements a naive probability algorithm when it is required to guess. This works quite well and is able to solve boards relatively efficiently.

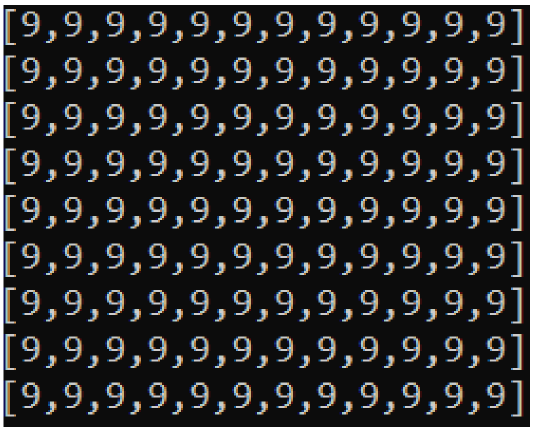
Minesweeper.hs

The game logic is defined in this file. Two boards are used to represent the game:

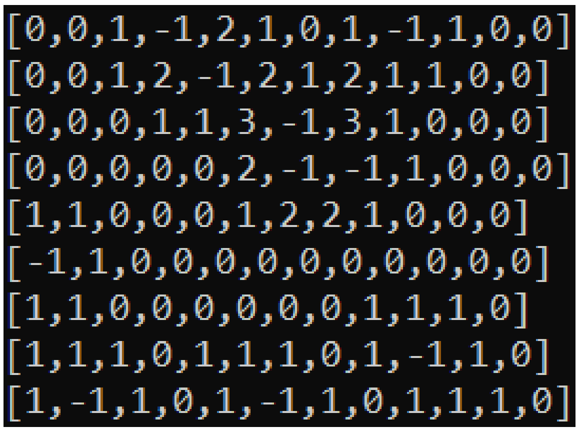
1. An **ApparentGrid** - This is what the player sees so it contains unopened squares, opened squares containing the number of adjacent mines and flags (and mines if the player loses).
2. An **ActualGrid** - This contains the underlying information the player should not see until uncovered so where the mines are located and the number of adjacent mines to each square.

Examples of both and what the numbers represent are shown below.

ApparentGrid ***ActualGrid***



```
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
[9,9,9,9,9,9,9,9,9,9,9,9]
```



```
[0,0,1,-1,2,1,0,1,-1,1,0,0]
[0,0,1,2,-1,2,1,2,1,1,0,0]
[0,0,0,1,1,3,-1,3,1,0,0,0]
[0,0,0,0,0,2,-1,-1,1,0,0,0]
[1,1,0,0,0,1,2,2,1,0,0,0]
[-1,1,0,0,0,0,0,0,0,0,0,0]
[1,1,0,0,0,0,0,1,1,1,0]
[1,1,1,0,1,1,1,0,1,-1,1,0]
[1,-1,1,0,1,-1,1,0,1,1,1,0]
```

ApparentGrid

-	9	= unopened
-	10	= flagged
-	0-8	= opened (number of adj mines)

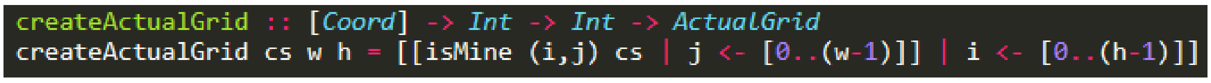
ActualGrid

-	-1	= mine
-	0-8	= number of adj mines

To initialise the game a random number, the dimensions of the board and the number of mines is supplied which is used to generate the two boards above.

To help with these data structures a coordinate data type (Coord) is defined as `(Int, Int)`. A 2D indexing function is also defined which takes a Coord and a grid and outputs the value at that coordinate. Its type signature is `(@!!) :: Coord -> [[a]] -> a`.

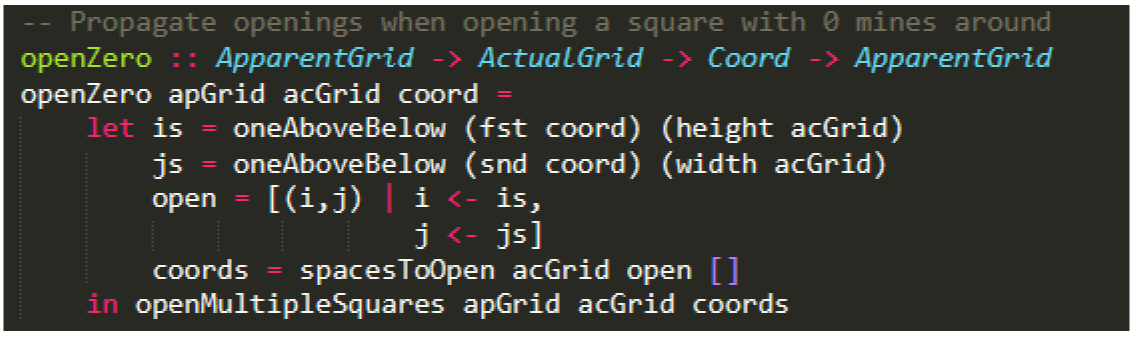
List comprehensions were used extensively to manipulate the grids. A lot of the functions in this file follow a similar pattern to ‘createActualGrid’ which creates the ActualGrid given a list of the generated mine locations and the grid dimensions. ‘isMine’ checks if (i,j) is in the given list of mine coordinates.



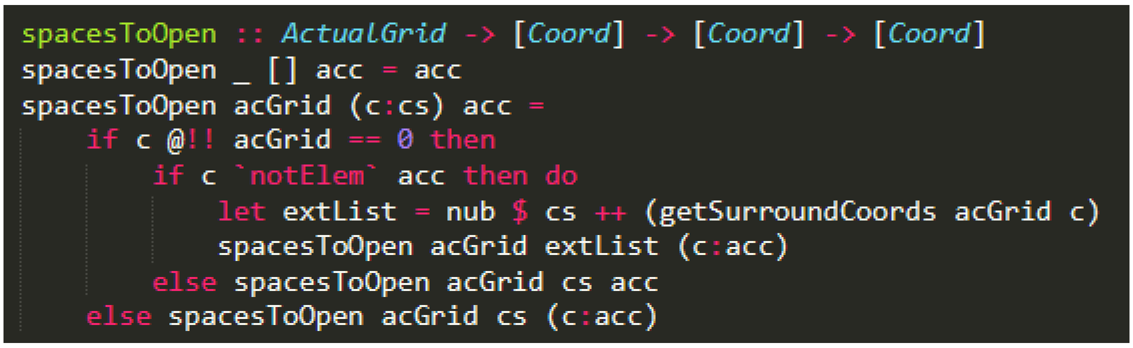
```haskell
createActualGrid :: [Coord] -> Int -> Int -> ActualGrid
createActualGrid cs w h = [[isMine (i,j) cs | j <- [0..(w-1)]] | i <- [0..(h-1)]]
```

Propagation of openings is another important feature of minesweeper. If a square is opened with no adjacent mines then its neighbours should be automatically opened. If any of these also have no adjacent mines then all of its neighbours should be opened and so on.

The code below gathers all the coordinates that should be opened once a square with no adjacent mines is opened and opens them all.

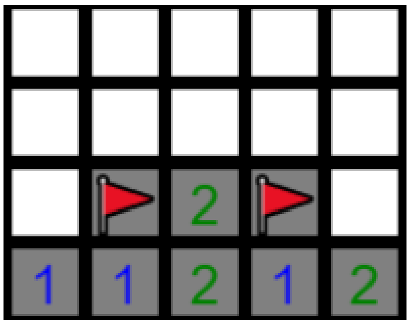


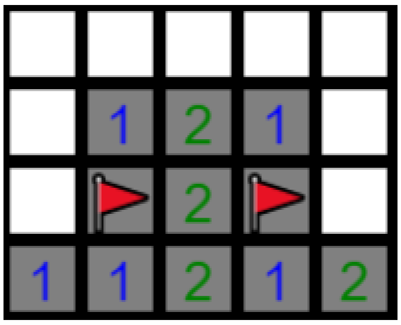
```haskell
-- Propagate openings when opening a square with 0 mines around
openZero :: ApparentGrid -> ActualGrid -> Coord -> ApparentGrid
openZero apGrid acGrid coord =
    let is = oneAboveBelow (fst coord) (height acGrid)
        js = oneAboveBelow (snd coord) (width acGrid)
        open = [(i,j) | i <- is,
                        j <- js]
        coords = spacesToOpen acGrid open []
    in openMultipleSquares apGrid acGrid coords
```



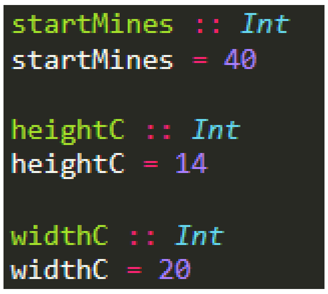
```haskell
spacesToOpen :: ActualGrid -> [Coord] -> [Coord] -> [Coord]
spacesToOpen _ [] acc = acc
spacesToOpen acGrid (c:cs) acc =
    if c @!! acGrid == 0 then
        if c `notElem` acc then do
            let extList = nub $ cs ++ (getSurroundCoords acGrid c)
            spacesToOpen acGrid extList (c:acc)
        else spacesToOpen acGrid cs acc
    else spacesToOpen acGrid cs (c:acc)
```

This is also how ‘chording’ is implemented. Looking at the ‘2’ in between the 2 flags below it is clear that the three squares above are clear. Chording means clicking on the already uncovered two will automatically open the rest of the squares. Implementing ‘`openZero`’ when the number of adjacent mines == number of flags and not just when the adjacent mines == 0 allows chording.





Main.hs

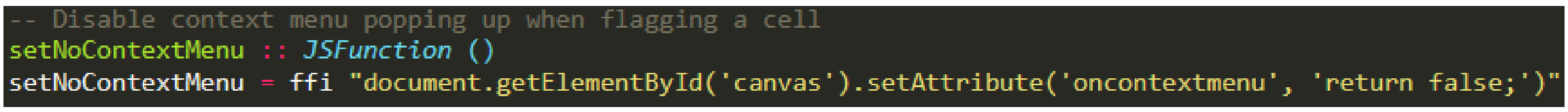


```haskell
startMines :: Int
startMines = 40

heightC :: Int
heightC = 14

widthC :: Int
widthC = 20
```

Main implements the GUI using Threepenny. The grid is implemented using a canvas. Clicking on the canvas converts the screen coordinates into grid coordinates using the dimensions of the grid: ‘`convCoord :: (Double, Double) -> (Int, Int)`’. So instead of having (h*w) buttons we just change the size of the grid. The cell sizes are constant so all the user needs to change to adjust the difficulty are the constants heightC, widthC and numMines.

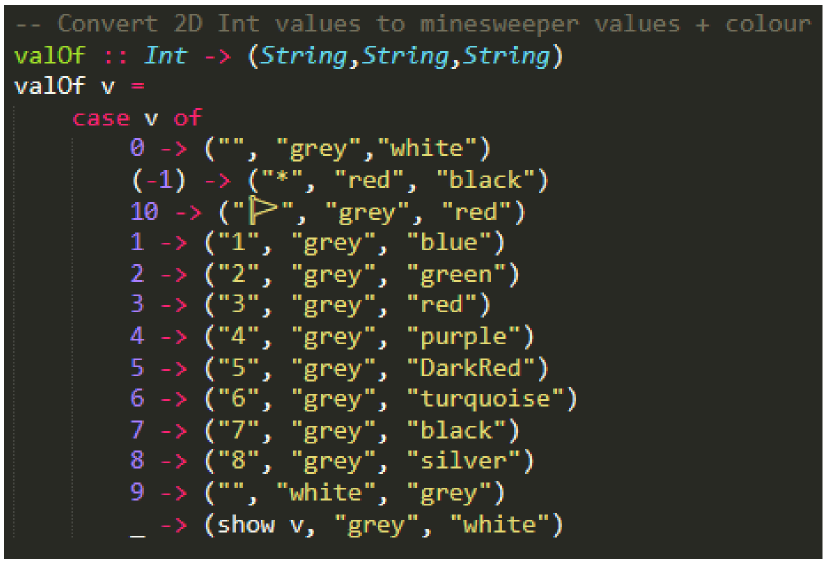
On load ‘`initGame`’ from minesweeper.hs is called giving the current time as the random number. This will be different each time the project is executed. The boards are stored in IORefs, the number of mines are displayed in the mine counter, the canvas is set to the given height and width and the game is drawn.

The JavaScript Foreign Function Interface (FFI) is used to disable the context menu appearing on right clicks inside the canvas. This allows easy flagging of squares.



```haskell
-- Disable context menu popping up when flagging a cell
setNoContextMenu :: JSFunction ()
setNoContextMenu = ffi "document.getElementById('canvas').setAttribute('oncontextmenu', 'return false;')"
```

Each button was given a handler and the canvas got a right and left click handler. The canvas mouse coordinates also needed to be saved in an IORef as `‘UI.click’` meant left click only but did not supply the mouse coordinates.



```haskell
-- Convert 2D Int values to minesweeper values + colour
valOf :: Int -> (String,String,String)
valOf v =
    case v of
        0 -> ("", "grey","white")
        (-1) -> ("*", "red", "black")
        10 -> ("🏳", "grey", "red")
        1 -> ("1", "grey", "blue")
        2 -> ("2", "grey", "green")
        3 -> ("3", "grey", "red")
        4 -> ("4", "grey", "purple")
        5 -> ("5", "grey", "DarkRed")
        6 -> ("6", "grey", "turquoise")
        7 -> ("7", "grey", "black")
        8 -> ("8", "grey", "silver")
        9 -> ("", "white", "grey")
        _ -> (show v, "grey", "white")
```

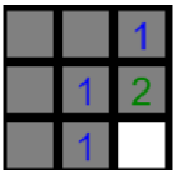
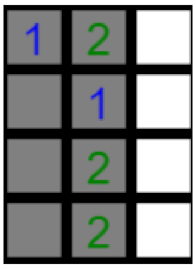
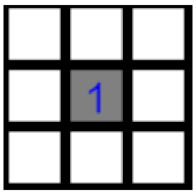
The squares were drawn as filled rectangles. The ApparentGrid (from minesweeper.hs) was translated from the 2D Int array into actual minesweeper values using the function on the left. Given a value, the string representation, text colour and background colour were returned.

Eg.

- A mine (-1) was a black asterix with a red background.
- A flag was a ref flag emoji on a grey background.
- Unopened (9) was no text on a white background

AIPlayer.hs

The minesweeper solver is implemented here. There are three levels to the solver:

1. Obvious moves - the scenarios where there are n mines left to place around a square and there are n spaces. Eg. We can tell there must a mine in the bottom left just by looking at the ‘1’ in the middle	
2. Non-Obvious moves - scenarios where we have enough information to make a move but we have to look at multiple squares to make the decision. This is implemented through matrices in reduced row echelon form. Eg. looking at the ‘1’ and ‘2’ in the two middle squares we can tell there is a mine at the bottom right and no mine in the top right.	
3. Guesses - where we do not have enough information and must guess based on probability. Eg. the starting moves to a game. If we get a single ‘1’ off our best bet is to try again adjacent to the one.	

Obvious Moves

Obvious moves are quite straightforward. `'aiObviousMove'` is given the ApparentGrid, ActualGrid and number of mines remaining and returns a newGrid. It goes through a flagging phase and an opening phase.

Flagging: For a given square if the number of adjacent flags < adjacent mines and the number of unopened adjacent == (adjacent mines - adjacent flags) then we can safely flag the remaining unopened squares.

Opening: If the global mine counter equals 0 and there are still unopened squares we can safely open them. Otherwise, for a given square, if the number of adjacent flags == number of mines then we can safely open the adjacent unopened squares. This is also the condition needed to try `'openZero'` which attempts to propagate the changes.

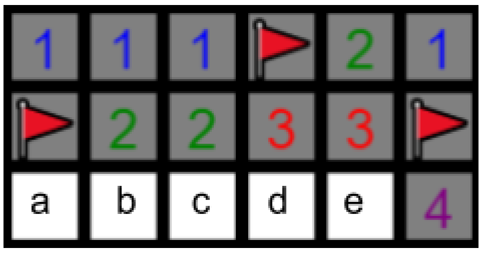
Non-Obvious Moves

The approach for finding more sophisticated moves came from an article - Solving Minesweeper with Matrices by Robert Massaioli (https://massaioli.wordpress.com/2013/01/12/solving-minesweeper-with-matricies/). And the `rref` function which converts a matrix into reduced row echelon form was adapted from https://rosettacode.org/wiki/Reduced_row_echelon_form#Haskell

If we consider the frontier of the board (the unopened squares which touch opened ones and opened squares which touch unopened squares) we can form a set of equations which when converted into reduced row echelon form, as if to solve the set of equations, can indicate which squares are safe and which squares have mines.

Taking this scenario below as an example. Labelling the unopened squares a-e we can see that there is 1 mine in [a,b,c]. Similarly there is 1 in [b,c,d]. There are 2 in [c,d,e]. Continuing on we can construct equations in the form:

sum of adjacent unopened squares = numMines - numFlags



gives

$$a+b+c=2-1=1$$
$$b+c+d=2-1=1$$
$$c+d+e=3-1=2$$
$$d+e=3-2=1$$

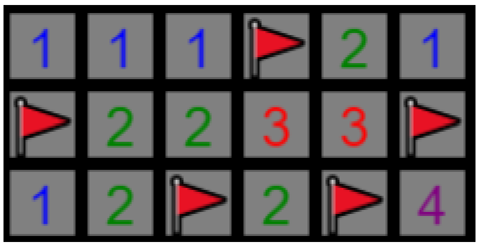
Converting these equations into matrix form gives:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Converting to reduced row echelon form gives:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

We haven't solved the full set of equations, but we didn't need to. In the third equation the only possibility is that c = 1 meaning c must be a mine. However there's other information we can use due to the binary nature of each variable; each variable is either a mine or not a mine. Row two also gives us some information - it says $b - e = -1$. This can only be true if e is a mine (1) and b is not a mine (0). This information allows us to uncover b, flag c and flag e. Continuing on leaves us with the solved row.



The reduced row echelon form of the matrix is returned and displayed under the canvas when the 'Non-Obvious Move' button is pressed.

So we can see the rules flip based on the sign of the last element of each row. The generalised rules for decoding the rows of the converted matrix are as follows.

```
for each row:
      max_bound, min_bound = 0
      for each element in row (excluding the augmented column element):
          if num is positive: add to max_bound
          else: add to min_bound

      if augmented column value == max_bound and max_bound != 0:
           The negative numbers in the row are not mines
           The positive numbers in the row are mines

      if augmented column value == min_bound:
           The negative numbers in the row are mines
           The positive numbers in the row are not mines
```

Guesses

Sometimes there is not enough information on the board to be sure of a mine's location. In this case for a solver to work it needs to guess. For this I mainly focused on the start game scenarios and implemented a naive probability algorithm that works some of the time.

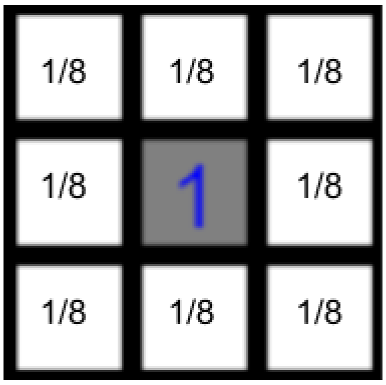
The basic idea involves taking the same frontier of squares used in the previous section, considering them square by square and adding their individual probabilities together and averaging over number of contributing squares then choosing the lowest value to uncover.

This works well in the early game and less well in the post game but still is better than completely random guessing. Some examples will show best how this works:

Suppose we're working with a 14x20 board with 40 mines and our first move reveals a single 1. There is no obvious safe move but we're not completely blind. With 1 mine in the 8 adjacent squares we get a probability ⅛ in each surrounding square.

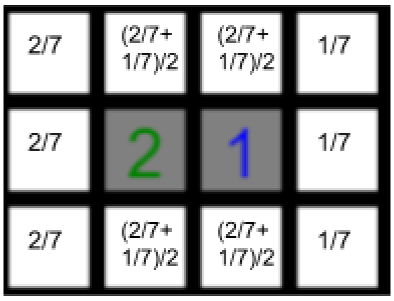
We also know that there are 40 mines in the other (14x20)-1 squares which is 0.143. So our best bet is to pick one of the 1-adjacent squares as 0.125 < 0.143

If a single '2' had been revealed then we'd be better off choosing randomly away from the '2' as 0.25 > 0.143.



Say uncovering the square to the left of 1 opens a 2. Calculating the probability for each adjacent square to the 1 and 2 gives 1/7 and 2/7. Averaging these where they overlap gives us the image to the right.

Here we would choose one of the squares in the rightmost column.



The combination of these three approaches is powerful and solves a large number of boards. Over a small sample size of 30 boards these were the results using purely the AISolver:

Beginner (9x9,10)	**Intermediate** (16x16,40)	**Advanced** (16x30,99)
24/30 = 80%	18/30 = 60%	11/30 = 37%

While some of these losses were unavoidable the results are basically a reflection of the level of the guesser as the two obvious approaches are unable to cause losses. While naive it is at least fast as opposed to a potential enumeration of all possible boards when no obvious move was available. If it was not near an endgame there could be millions of possible boards which was the reason for not implementing that approach.

The three buttons are kept separate purely to illustrate what each button does. If I meant for other people to actually play this then there would just be one general 'Make Move' button.

Thoughts / Reflection

I felt Haskell was an appropriate language for describing the rules and game logic of minesweeper. Deciding that the base functions would take a grid and output a grid and then being able to map out the type signatures helped to structure the program. This might have led to too many helper functions being defined but I think that's mainly a drawback of working with 2D list structures in Haskell. Arrays could have been used instead of lists due to the amount of list accesses / alterations being done which would have improved efficiency with larger boards.

Despite the clear structuring, bugs still appeared and were sometimes difficult to pin down without the imperative programming execution flow. I introduced a bug early on that broke non square boards which took a bit of fiddling to figure out. It was also not clear how to implement some of the probability algorithms in the functional programming style which is why I eventually settled on the naive guesser. The lack of conventional global variables also made it difficult to implement letting the user select board size and difficulty. The generation of random numbers in a functionally pure language also caused problems leading to the mine coordinates being generated before the user clicks which isn't ideal. In most minesweeper games the first click is guaranteed to be safe meaning the mines are placed after the first click but structurally this proved difficult.

Minesweeper lends itself to recursive programming which meant it was a good fit for a functional approach. Threepenny GUI was fine once you figured out what you were doing. The foreign function interface was also useful as Threepenny does not implement absolutely everything you might need. As far as the actual development goes, it was not the most streamlined process. Writing the code involved editing some source files, stopping the game if it was running, compiling (which took not an insignificant amount of time) then executing the game and repeating.

A video demonstration is attached: https://youtu.be/jD3inIBqUu8