Minesweeper Report

Miguel Arrieta 20332427

All the project deliverables were satisfactorily completed.

Introduction

This assignment required us to develop an implementation of Minesweeper, and also an auto player that will choose the best move based on some tactics. This report includes my reasoning for choices that I made designing the game and my reflection on the project.

High level design decisions

Board

The very first thing I had to think about was; what did I want to model a minesweeper board like? So the first thing I decided was that I would have two different models, one which would be used internally and one which would be exported. The reasoning behind this was that I didn't want agents to be able to cheat and just straight up be able to look at whether a cell had a mine or not. I will dive into the types of this exported board later on.
The very first thing I did was create a type for a cell, this data type stores whether it has a mine, and the state it is in (**CellState**) - Flagged, Revealed, Unrevealed. Next I created a type for a board (the one to not be exported), which has the board state (Won, Lost or Playing), the total number of mines and a grid of cells.
It is also worth noting that I decided that all boards would have the same height as width.
For the grid I chose to use a 2D Vector, a Vector of Vectors of Cells, since O(1) indexing was a priority.
I needed the user to be able to view the board somehow, for this I created the type **VisualBoard**, which is the same as Board except its grid is of type [[((Int, Int), VisualState)]].
To go along with it I created a type called **VisualState**, which described the state of a cell as the user can see it, that being covered, uncovered, number of mines, flagged or exploded.
The thought process behind this was that users could use this to implement their own interface in which a player can view the game.
I created an API that users could use to interact with the board and play minesweeper.
The main type that was exported was **VisualBoard**, since that's what users would use to see the state of the board. The main functions were **toggleFlag**, **revealCell**, **generateBoard** and **getBoardVisuals**.
When implementing **revealCell** I chose to make it so that if a user clicked on a mine on their first move, I would move that mine elsewhere and let them play the move, since I thought that was fairer.

Agent

Although I implemented most of the GUI next, I think it makes more sense to talk about the AI agent that I built first. The first thing I did was make a data type to abstract moves, those being Uncover (Int, Int) and Flag !(Int, Int).
I then created a function for the agent to uncover a random square, since I knew I would most likely need this unless I made my AI extremely good.
Next was the actual AI function **makeMove**, I split this up into two strategies which are quite obvious when you think about them:

1. If the current square has the same number of surrounding mines and flags then the rest of the squares must be safe to mine
2. If the current square has the same number of surrounding mines as covered cells + flagged cells then they are all mines, we can flag them

I ran these strategies for each uncovered square in the board and if this resulted in no obvious moves I returned Nothing. Upon failure the user was free to deal with this as they saw fit, I chose to then call **uncoverRandom** in my GUI, but if the assignment was different I could have also displayed some sort of message saying that there was no obvious move or whatever. The point is that I didn't make one function that would do it all, I left it up to the user of my Agent module to decide what to do on failure.
After doing this I decided to implement a sort of simple but advanced tactic, simple in terms of implementation, but advanced in terms of relying on strategies that the best minesweeper players use. I at least did not know about this strategy before taking on this project, and I play minesweeper like 4 times a year!!
I decided to make it so that if a user called **uncoverRandom** on their first move, we would uncover the top right corner. Any of the corners except the top left corner would have resulted in the same result. The reason why we choose a corner is because corners are the hardest places to clear, and in games you can often get stuck at the end, having to make a random choice in a corner (like a 50/50). Corners also have the best chance at finding an opening as stated here:
https://www.minesweeper.info/wiki/Strategy#:~:text=66/33%20situations-,First%20Click,-The%20first%20click
The reason why I didn't choose the left corner is basically because of insider knowledge, but also the original minesweeper was built like this, or so I read.
Let me explain: If we were to click a square at the start and that happened to be a mine, we can be sure that there is a mine in the top left corner, since if there wasn't a mine there before, that's the first square that minesweeper will populate with a mine, and if there was a mine there before, then it still has a mine. Of course we don't know if we clicked on a mine on our first move or not, but we can use this to our advantage and click on a different corner to have possibly more information about the top left corner.

Main - The GUI

I chose to have three different difficulty levels going off of the minesweeper game that google hosts, easy (10x10 10 mines), medium (18x18 40 mines) and hard (24x24 99 mines).
The user could then click one of these buttons where they would be able to play minesweeper. From this screen the user has:

- A home button which they can use to go back to the main page

- A reset button which they can use to restart the game with a new board on the current difficulty
- A mode button which allows the user to switch from mining and flagging mode
- An auto move button which allows the user to get the AI to play a move (the AI first tries to play an obvious move and if it fails it plays a random move)

The user can also just mine and flag squares themselves. The current mode is visible as a picture on the mode button.
If they are on mining mode they can left click to mine and right click to flag, and if they are on flagging mode they can left click to flag and right click to mine. This was built with touch screen users in mind, as they cannot right click as easily as other users.

Reflection

I am extremely happy with how the project turned out. Not only was I able to satisfactorily provide all the deliverables, but I think the program is visually appealing and very responsive.

How suitable was Haskell for this project?

For everything but the GUI, I loved using Haskell, especially due to how easy it is to compose functions and abstract types using ADTs. Haskell makes it extremely easy to compose functions and break functions down into more manageable and reusable functions. The fact that haskell is lazy proved to be extremely useful for designing things like generating a random board as I could do stuff like: take x $ nub randoms where randoms is an infinite list, and as long as the random number generator was fair, this wasn't too inefficient for small x's.
I however really disliked using threepenny for this. Not for the reasons I expected but because it does not seem to mark elements for garbage collection very efficiently. It does not hold your hand at all. I had to heavily optimise my program so that it would not crash every two seconds. You are basically required to mark elements for deletion if you wish for threepenny to not freak out and bring you back to the landing page. It also does not seem to cache images; if you load image X 200 times, it will load it 200 times, even though it is the same image. This caused huge slow downs in my program and would frequently crash it. However, my biggest gripe isn't with the above comments, it is the fact that this isn't documented anywhere, and my biggest gripe of all: The fact that threepenny will swallow up errors that are thrown and go back to whatever function you gave it in startGUI, without even letting you know in the console what caused the error in the first place. This is very bad design in my opinion, especially coming from a Functional language, where being transparent is kind of the whole point, swallowing up errors and not being able to even enter some sort of debug mode where you can see the error causes is, in my eyes, one of the worst decisions in the library.
I realise that I have strayed off topic, and this was more of a "what my issues with threepenny are" but it was a very notable moment in my journey building this project as I spent 2 days just figuring all this out. However I enjoyed using Haskell for a very real application and was surprised to see just how easy it is to build real programs with it.
My love for functional lives on.

References:

I got the minesweeper assets from here: https://gooseninja.itch.io/pixel-minesweeper