Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44012 Topics in Functional Programming Project 2: Minesweeper-o-Matic

Fionn Murphy

January 8, 2025

In this report I will briefly document the design choices of my project solution and how each deliverable was satisfied, as well as reflect on the process of designing my program.

For Documentation on how to run the program and play the game, as well as a discussion on how it all works, refer to the Project README.

1 Design Choices

Given that this project was going to involve extensive use of the Threepenny GUI, I decided to use my Calculator project from the Week 11 exercise as a base for this task. It already had the logic for setting up a User Interface and allowing interaction, so it was mostly a case of adapting and building upon this code to work with a Minesweeper implementation. From the get-go I decided to develop the Minesweeper model and the UI in tandem with each other, and I also spent some time styling my game in a similar manner to the classic version so that it would look and feel good to play. In addition to the basic functionality I also decided to allow the player to reset the game at the click of a button. I implemented the Minesweeper logic in its own module so that I could develop the Auto-Player logic separately, and put the window and game setup logic in the Main Module. Finally, I made sure that my Auto-Player would use basic Minesweeper logic to try and reveal or flag squares before resorting to advanced logic, or failing that using probabilities to select the least dangerous square. I'll go into a more detailed discussion of each of these design choices and how they contributed to satisfying the project deliverables here.

2 Phase 1: Basic Minesweeper

In the project description this deliverable was split into two parts - modelling Minesweeper in Haskell and then adding the UI with Threepenny. However, I decided to develop these two parts together, since I already had a lot of the UI logic from my Week 11 Calculator. My Minesweeper module allows for creating a 10x10 grid with 15 mines randomly distributed throughout the grid. This grid is created automatically when the game is started/reset, and is randomized every time - this means that unlike some more complex implementations the user's first click is not guaranteed to be safe, however this is usually not a requirement for basic Minesweeper.

A Grid of Square objects is passed around every function and kept updated, and each coordinate is linked to a Button in the UI Grid which is kept visually updated in its own separate thread (so that we don't have to pass the buttons into every single function). The buttons have Left and Right-click behaviour defined to allow the user to Left-click on a square to reveal it, and Right-click on a square to flag/unflag it (I have also made it so that flagged squares cannot be revealed, to prevent any mistakes due to misclicks).

The thread that keeps the UI Grid updated also continually checks for the Win/Lose conditions (the function is called every 0.1 seconds). If the user uncovers a mine, the game ends and the player can no longer interact with it. A Game Over message is displayed, and as well every mine in the grid being revealed, any flag that was incorrectly assigned will be highlighted, just like in classic Minesweeper. The game is won and a win message displayed when the user uncovers every square that is *not* a mine (which doesn't require them to have to flag every mine, just that they don't reveal any, which would result in a loss). At this point the user can click the reset button to start a new game (incidentally they can also do this while in the middle of a game, if perhaps they got an undesirable grid layout). There is also a Quality of Life feature that makes it so when a player clicks on a square with no neighbouring mines, all of its neighbours are automatically revealed. This is done recursively, so just like classic Minesweeper it can cause large sections of the grid to open up quickly, saving the user from having to click on each empty square separately!

All of these features come together to deliver a full implementation of Minesweeper as set out in the project deliverables.

3 Phase 2: Automatic Player

Once I had a satisfactory implementation of Minesweeper, it was time to add the Play Move feature. I started off with the basic criteria here: that the program play a move automatically if there is an unambiguously safe move available. The win condition requires that a player reveal every non-mine square, so I made sure my auto-player would prioritize reveal moves first. My program checks for the basic patterns described in the Minesweeper Wiki - squares whose number of flagged neighbours is equal to its clue, meaning any other neighbours can safely be revealed. The only way this program can reveal a mine is if the user incorrectly flags an empty square, otherwise this move is always safe. If the auto-player cannot find any safe squares to reveal, it will then prioritize flagging squares that are known to be mines, again using the basic patterns as described in the wiki. If a square's number of remaining hidden neighbours is equal to its clue (minus any neighbours that have already been flagged), then those remaining hidden neighbours can safely be flagged. Again, assuming that the user hasn't placed an existing flags incorrectly, this move will always be safe.

Even with only these two checks, many games of Minesweeper can be solved entirely by the program, especially if the user opens up a large empty area first. This program will also never fail so long as the user does not make any mistakes, as the program assumes that the current grid layout is fully correct. From here I added a third move option for if no safe flag moves are found, and this involves some of the advanced logic described in the Minesweeper wiki and mentioned in the project description: the 1-2-X pattern. The program will search for this pattern in every position and direction in the grid, and will flag the corresponding mine if it is found. This move also counts as an unambiguously safe move and extends the existing logic to handle more cases.

Finally it was time to make my program fallible, so that I could implement the last deliverable: if no safe move is found, then the program will play the least dangerous move available. My program does this by calculating the local probabilities of each hidden square containing a mine. Each probability is calculated by looking at any revealed neighbours, checking their clue, and dividing it by the number of hidden squares surrounding that neighbour (multiple probabilities for the same square are aggregated together). The squares are sorted lowest to highest based on probability, and the head of that list is selected as the square to reveal. This is not the most optimal way to calculate probabilities in Minesweeper, but for a simple enough implementation it still performs fairly well in most situations, especially since that in my 10x10 grid setup there is unlikely to be any more than a handful of moves that have to be decided by chance, and many of these will be 50/50 anyway. This part of the program mostly comes into play during the opening moves where there is not much information about the grid and some guesses have to made to try and open it up.

With all these, I have an automatic Minesweeper player that performs quite well overall. It can identify and prioritize unambiguously safe moves using both basic and advanced logic, and can use probability calculations to perform the least dangerous move if all else fails, satisfying the project deliverables for this phase.

4 Reflection

I would say that most of my time spent on this project was getting the Minesweeper implementation to work with Threepenny GUI. I think that Haskell was actually quite useful for designing the basic game and the auto-play function - it was easy to pass around references and define simple data types that could be easily pattern matched. However I will not deny that it was at times painful trying to get the User Interface to link up with the basic Haskell code - one example was when I was implementing the feature to recursively reveal the neighbours of empty squares and trying to get the Buttons in the UI to reflect this change - I eventually had to resort to concurrency to make this work! I think I would freely admit that Haskell would not be my favourite language in which to design a User Interface, though the purity of Haskell was certainly a somewhat welcome change to what I am convinced is the non-deterministic behaviour of JavaScript, for instance.

Debugging problems was somewhat easier than other languages - it's usually to do with types and you're always told exactly what was expected and what was encountered. It was very useful to be able to use pattern matching to perform logic with the squares, and using IORefs to keep track of states made it easy to pass everything around to the relevant functions. I even got a bit of concurrency in there, which is quite simple to do in Haskell.

I would say that overall I enjoyed working on this project and getting each deliverable implemented. Had I a bit more time I would have liked to add pattern-reduction to my 1-2-X check, and also added support for the 1-1-X check. I also would have liked to take a stab at the more complex but more optimal method of calculating probabilities using grid sections and permutations. However, I'm happy with everything I managed to implement and that I have satisfied each of the project deliverables.

Overleaf are a couple of screenshots of the game. A more in depth discussion of how everything works can be found in the README of the submitted project, or on the project GitHub here.

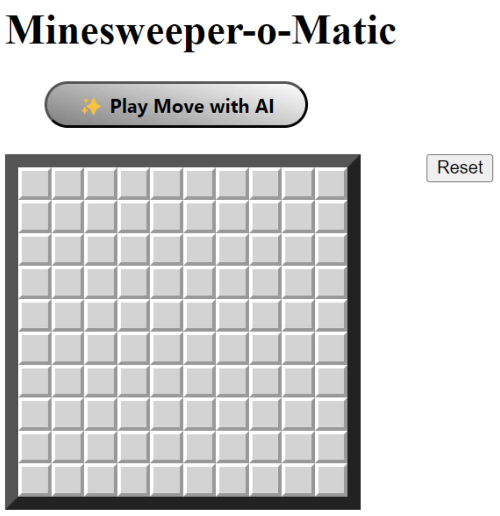

Figure 1: A New Game

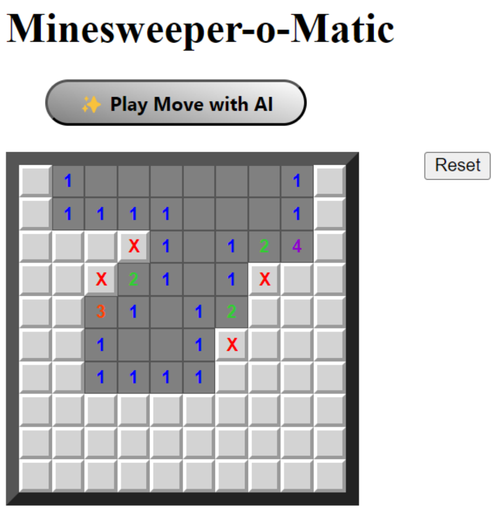

Figure 2: A Game in Progress

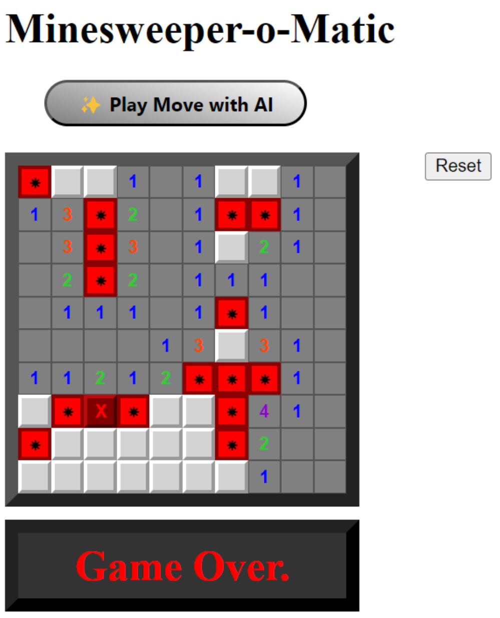

Figure 3: Game Over!

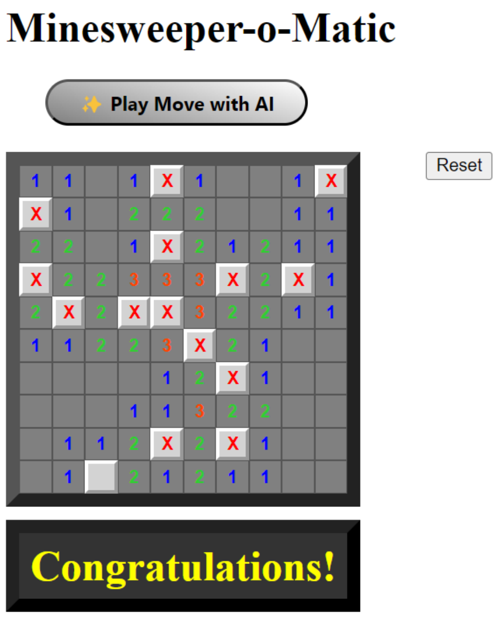

Figure 4: A Solved Game