Minesweeper Game - Written Report

Serzhan Kenesbek

December 2, 2024

Task Overview

The task was to implement a Minesweeper game with the following features:

- Console (or other user interface) for gameplay.
- Representation and initialization of the game grid with mines.
- Game logic for revealing cells and handling win/loss conditions.
- Error reporting for invalid moves or inputs.
- Unit and property-based tests.

The game logic is implemented in Haskell, and the focus was to ensure that the game is functional, user-friendly, and thoroughly tested.

Architecture of the Solution

The architecture of the Minesweeper game is modular and follows principles of functional programming, focusing on separation of concerns and high cohesion within each module. The game was divided into several components:

Modules

Cell Module

This module defines the `Cell` data type, which represents each cell in the grid. Each cell can either be:

- A mine.

- A revealed cell with a number indicating the count of adjacent mines.
- A flagged cell.
- An unrevealed cell.

The `emptyCell` function initializes a cell with default values.

Grid Module

This module is responsible for the representation and initialization of the game grid. The grid is implemented using `Vector` from the `Data.Vector` module for efficient access and modification. The main responsibilities of this module include:

- Generating a grid with mines placed randomly.
- Rendering the grid for the user to see.
- Handling operations on the grid such as revealing cells, flagging/unflagging cells, and checking for out-of-bounds errors.

Game Module

The `Game` module contains the core logic for the Minesweeper game. This includes:

- Handling user inputs for revealing and flagging cells.
- Checking for win/loss conditions.
- Cascading reveal of cells when a cell with no adjacent mines is revealed.
- Managing the first move to ensure it is never a mine.

Main Module

The entry point of the program, where user input is processed, and the game loop is executed. The main module uses the functions from the `Game` and `Grid` modules to manage the gameplay.

Why Certain Architectural Decisions Were Made

Modular Design

The decision to divide the program into modules (e.g., `Cell`, `Grid`, `Game`, `Main`) was made to ensure high maintainability and modularity. Each module is responsible for a specific part of the game's functionality, which:

- Increases code readability and understanding.
- Simplifies testing by allowing individual components to be tested separately.
- Makes the code easier to maintain and extend, as each module is decoupled from others.

Recursive Logic for Revealing Cells

The recursive logic for revealing cells (in the `revealCell` function) was chosen because it reflects the depth-first traversal of the grid. This approach is particularly suitable for the cascading reveal of adjacent cells when an empty cell (with no adjacent mines) is revealed.

Use of `Vector` for Grid Representation

The initial implementation used lists for the grid, but performance concerns led to switching to `Vector`. `Vector` provides efficient random access (O(1) time complexity) and in-place updates compared to lists, which have O(n) time complexity for these operations. This improves the performance of grid operations, especially when dealing with larger grids.

Use of `Either String` for Error Handling

The game logic uses `Either String` to handle errors. This choice allows errors to be explicitly propagated through the game's functions, such as invalid moves or attempts to flag a revealed cell. By using `Left` to return an error message and `Right` to return the valid result, the code remains easy to read and understand.

Why Certain Libraries Were Chosen

`Data.Vector`

`Data.Vector` was chosen for its efficient handling of random access and updates. Since Minesweeper involves frequent operations on specific cells of the grid, `Vector` allows for more efficient performance compared to lists, especially when the grid is large.

`Hedgehog` and `Test.Tasty`

`Hedgehog` was chosen for property-based testing due to its ability to generate a wide range of test cases automatically, helping to ensure that the game logic handles various edge cases. `Test.Tasty` was chosen as the testing framework because it integrates well with both `HUnit` and `Hedgehog`, allowing for both unit tests and property-based tests to be executed in a unified testing environment.

`System.Random`

This library was used to generate random numbers for placing mines on the grid. It provides simple and effective tools for generating random indices, ensuring that mines are placed randomly across the grid.

Investigation of the Performance

Grid Representation with `Vector`

The initial use of lists for representing the grid caused performance problems, especially when performing grid operations on larger dimensions. Switching to `Vector` greatly improved the performance, as it allows for constant-time random access and modifications. This change reduced the time complexity of accessing and updating cells from O(n) to O(1).

Handling Small Grids

While testing small grids, an issue arose where the game would hang due to an infinite loop when attempting to place mines. This was due to the first move excluding neighboring cells, leaving no valid cells for mine placement. The problem was fixed by calculating the valid cells before attempting to

place mines and ensuring the number of mines doesn't exceed the number of valid cells.

Recursive Functions and Stack Overflow

The recursive approach for revealing cells worked well for smaller grids, but it could cause stack overflow issues for very large grids. In future improvements, I plan to optimize this using a breadth-first approach or a more iterative solution, which could help with large grids.

Optimization with Iteration

The recursive nature of the reveal logic is elegant, but may not be the most efficient for larger grids. An iterative approach using a queue or stack might be more suitable for handling large grids without risking stack overflow errors.

Conclusion

The Minesweeper game was successfully implemented using Haskell, applying functional programming principles and maintaining modularity throughout the project. The grid representation was optimized using `Vector`, and recursive logic was used to handle cascading cell reveals. Error handling was integrated using the `Either` type, and both unit and property-based tests were implemented to ensure the correctness of the game logic.

The project demonstrates the application of Haskell's powerful features, such as immutability, algebraic data types, and higher-order functions. While the current implementation works well for smaller grids, further optimizations will be made for handling larger grids more efficiently.