Minesweeper - game and solver implementation

Vlad Vasilescu
CEN 1.3B (Ragnar)

Year 2017 - 2018
Semester II

Problem Statement:

Minesweeper is a single-player puzzle video game. The objective of the game is to clear a rectangular board containing hidden "mines" or bombs without detonating any of them, with help from clues about number of neighboring mines in each cell.

What I did was implement both the game itself aswell as a solver for it. The player can chose between playing the game or running the solver.

Application Design:

The game uses a number of functions that have been placed into libraries that represent the process they are used for:

- Functions used in generating the board.
- Functions used while playing the game.
- Functions used by the solver.

Generating the board:

The first set of numbers the player has to give as input is the board length, height and the number of bombs, and also the coordinates for the first square to be opened. This is to prevent generating an unsolvable board.

The board as a data type:

The board is defined as a custom data type which stores board length, height, number of bombs, number of open squares, whether the bomb has been hit or not and 2 matrices which store the bomb positions and the visiblity of the cells.

Pseudocode algorithm:
Data: The board and starting coordinates.
Result: The generated board.
while *there are still bombs to place* **do**
  generate random set of coordinates for bomb;
  if *bomb is adjecant or on starting coordinates* **then**
    generate new set of coordinates for bomb;
  else
    place bomb and increase the value of all adjacent cells by one;
  end
end

Algorithm 1: Setting up the board.

Playing the game:

Playing the game consists of 3 functions that are ran inside a loop until a cell containing a bomb has been opened or when all cells except those containing bombs have been opened:

- Function that prints the board.
  Simply takes the board as input and prints a cell value to standard output only if it's visible.
- Function that reveals a square.
  Takes the board and 2 coordinates as input and reveals the square on that position. If the value on cell is "0" then the function calls itself for all adjacent cells. If the cell opened is a bomb it stores that a bomb has been hit so the game can stop.
- Function that flags a square.
  Just like the "reveal" function, takes as input the board and 2 coordinates but this time it places/removes a flag from the given position. If there is no flag it places one otherwise it removes the already placed flag.

Pseudecode algorithms for the three functions:

Data: The board
Result: Prints the board to standard output
forall *cells in board* **do**
  if *cell is visible* **then**
    print cell value;
  else
    print a hidden cell;
  end
end

Algorithm 2: Printing the board.

```
Data: The board and coordinates for the cell to be opened
Result: Changes the visibility of the cell
if cell is not visible then
    make the cell visible;
    if cell is bomb then
        store that bomb has been hit;
    else
        if cell is "0" then
            call this function for all adjance cells;
        else
            return
        end
    end
else
    return
end
```

Algorithm 3: Reveals a cell.

```
Data: The board and cooridnates for the cell to be flaged
Result: Changes the visility of the cell
if cell is not visible & cell is not flagged then
    flag the cell;
else
    remove flag from the cell;
end
```

Algorithm 4: Places/Removes flag from a cell.

Solving the game:

The solver uses 2 methods each with it's own library of functions all called from the main solver function.

The "solver helper" as data type:

The "solver helper" is custom data type that contains the number of squares that have been marked on each step aswell as two matrices: one for storing the marked cells and one for the chances.

The "marking" method:

The marking method is the main method for solving as you can always be sure that the cell it opens is guaranteed to not be a bomb. The way this works is that it first finds every cell that has the same number of adjancet hidden squares as the number on the cell. On the second pass it checks if any cell is "satisfied" (has the same number of marked cells around it as the number on cell) and

opens all un-marked hidden cells around it.
There are three functions used by this method:

- The first one returns the number of adjacent marked cells.
- The second function marks the squares on the board.
- The third function opens up all the unmarked cells around a satisfied one.

```
Data: The board and the solver helper
Result: Marks all eligible squares on the board
forall cells in board do
    if adjacent hidden squares = number on the cell then
        mark all adjacent hidden squares;
    end
end
```

Algorithm 5: Function that marks cells.

```
Data: The board and the solver helper
Result: Opens the unmarked hidden cells around a satisfied square
forall cells in board do
    if adjacent marked cells = number on the cell then
        open all unmarked hidden adjacent cells;
    end
end
```

Algorithm 6: Opens all the cells know to not be bombs.

The "chances" method:

This method is only used when marking one hasn't managed to mark any cell which means it's not 100% reliable. The way it works is by finding each cell that is visible and has at least one hidden cell adjacent to it and adding the number on the cell divided by the number of hidden cells around it to each hidden cell (each hiddden cell starts at 0).

```
Data: The board and the solver helper
Result: Set the chances matrix
forall cells in board do
    if number of adjacent hidden cells ≠ 0 then
        forall adjacent hidden cells do
            adjacent hidden cell := cell / number of adjacent hidden cells;
        end
    end
end
```

Algorithm 7: Setting the chances for the entire board.

The second function just reveals the cell with the lowest chance.

Generating test data:

Generating test data for the solver is really easy becauase the solver uses the board generated by the same functions that are used for making a playing a board. The player just selects whether he wants to play or use the solver on the board after the first set of input data is given. As for generating the board no matter how big the board is it's only gonna require 5 numbers for the size, the nr. of bombs and the first set of coordinates.

Conclusions:

Summary of achievements:

My main two goals that I managed to achieve while working on this project are: First of all creating a playable and solvabale board where the player cant hit a bomb or a number cell on the first try. This was done by generating the board after the first set of coorinates is given by the player and not allowing bombs to be placed on or adjacent to that posititon.

And secondly implementing a semi-reliable solving algorithm. This was by far the hardest task because I used no outsde sources and came up with the 2 methods used myself. The solver is not guaranteed to work reliably because of the way the game works. When the number of bombs is close to the number of total cells on the board the chances of encountering a 33% or even a 50% probability of a cell being a bomb go up.

Future of the project:

As of now I plan on making the playing side of the game as user friendly as possible by adding some more key features and increasing the visibily of the board on the screen when dealing with a large number of lines and columns.

Bibliography

[1] Minesweeper (Video Game)
https://goo.gl/QUjRbT

[2] Stack Overflow
https://stackoverflow.com/

[3] Microsoft Minesweeper
https://goo.gl/75w3oT