Programming Project 1 - Minesweeper Solver

Please submit your solution via email to the instructor with CC to ly603@nyu.edu. The deadline for the project is May 18.

The goal of this project is to implement a minesweeper game together with a *perfect minesweeper solver*. The game is played on a two-dimensional grid of cells that may contain mines. The player wins the game if all cells containing mines have been marked. To obtain information about which cells may contain mines, the player can reveal cells. If a cell with a mine is revealed, the player loses. If the revealed cell does not contain a mine, it shows the number of neighboring cells that contain mines. This information can be used to deduce which of the unrevealed cells contains a mine and which does not.

A perfect minesweeper solver plays the minesweeper game without guessing which cells can be safely revealed, unless the current board configuration does not allow to deduce whether some unrevealed cell contains a mine or not. In particular, a perfect solver can be used to decide whether a given board configuration can be solved without guessing. Solving minesweeper games is closely related to the *Minesweeper Consistency Problem*, which asks whether, given a board configuration, there exists an assignment of mines to unrevealed cells that is consistent with the configuration. For more information about this problem see Richard Kaye's Minesweeper Page[1].

Part 1 Alloy Model of Minesweeper Game and Solver

In the first part of this project you will develop an Alloy Model of a minesweeper game and solver.

(a) Develop an Alloy model of minesweeper board configurations for *abstract* minesweeper games. In an abstract minesweeper game the board is an undirected graph with nodes representing cells and edges indicating which cells are neighbors. Other than that, abstract minesweeper games are played exactly like normal minesweeper games. Think about what properties you want the field encoding the neighbor relation to hold and add appropriate facts to your model. Simulate some board configurations.

(b) Think about invariants of board configurations that can occur while playing a minesweeper game and add appropriate predicates to your model. For instance, during a minesweeper game, marked cells are never revealed. Another important property is that the number of marked cells never exceeds the number of mines on the board. There are more such invariants that you want to specify. Simulate some board configurations both satisfying and violating your specified invariants.

(c) Specify the operations on board configurations that you need for playing the game. These are: (1) mark a cell on the board, (2) unmark a cell on the board, and (3) reveal a cell on the board. You will also need to write a function that computes, for a given

[1] `http://web.mat.bham.ac.uk/R.W.Kaye/minesw/`

cell, the number of neighboring cells that contain a mine. Make sure that revealing a cell is propagated, i.e., if a cell with no neighboring mines is revealed then all neighboring cells are also revealed.

(d) Write assertions checking whether your operations preserve all the invariants of board configurations that you have specified. If some invariant is violated by an operation, add an appropriate precondition to the operation.

(e) Write a predicate that holds true for board configurations that are safe, i.e. do not contain a revealed mine. Write another predicate that holds true for configurations that are won, i.e., on which all unrevealed mines are marked. Use these predicates and your operations on minesweeper boards to simulate some minesweeper plays, both winning and losing.

(f) Write a predicate that holds true for consistent board configuration according to the Minesweeper Consistency Problem. Simulate some consistent and inconsistent boards. Use your consistency predicate to write a perfect minesweeper solver and use your solver to solve some minesweeper boards. Make sure that your solver does not cheat by directly accessing the fields that encode which cells contain mines. You can do this by declaring these fields *private* and putting your consistency predicate and the solver into a separate module.

Part 2 Minesweeper Game and Solver in Java

In Part 2 of the project you will implement the minesweeper game and solver in Java. We will restrict ourselves to the classic mine sweeper game played on a two-dimensional grid.

(a) Implement the actual minesweeper game with the board and all board operations such as revealing cells, marking cells, and determining the number of neighboring mines of revealed cells. The board and its operations should be encapsulated in a class that provides all the functionality for a client (such as the solver) to play the game. Make sure that your class does not expose information to clients that should not be directly accessible by a player of mine sweeper, such as which unrevealed cells actually contain mines. Also it should not be possible for a client to undo a reveal operation. The board class is allowed to expose the total number of hidden mines, though.

(b) Implement a simple parser for board configurations and add code to create instances of your board class from the parsed input files. The input file format is a text file with one line for each row of the board and each line consisting of a (white) space separated list of entries, one entry for each cell in the row. Each entry is one of the characters ’H’, ’M’, and ’R” where ’H’ specifies that the cell is not revealed and does not contain a mine, ’M’ specifies that the cell is not revealed and contains a mine, and ’R’ specifies that the cell is revealed and does not contain a mine. Figure 1 shows two examples of boards encoded in the input format.

(c) Transfer the invariants on board configurations, as well as the pre- and post-conditions of the board operations that you discovered in Phase 1 from Alloy to JML. Add these

```
H H M M
H H M H
H H H H
```

```
H H M M H H
H R R R R H
M R R R R M
M R R R R M
H R R R R H
H H M M H H
```

Figure 1: Two boards encoded in the input file format. The left-hand side shows the encoding of a board consisting of three rows and four columns in which all cells are hidden. The right-hand side shows the encoding of a board with six rows and six columns in which some of the cells are already revealed.

specifications to your board class and test them using runtime assertion checking. You may use JMLUnit for generating unit tests, but this is not mandatory.

(d) Implement a perfect minesweeper solver as a client of your board class. The solver should print all the solving steps to standard out. If the cell in the i-th column and j-th row is revealed, it should print `reveal i j`. Similarly, if the cell in the i-th column and j-th row is marked, it should print `mark i j`. Note that rows and columns should be indexed starting from 0. If a solving step was guessed, then this should also be recorded, by printing `guess reveal i j`, respectively, `guess mark i j`. The final outcome of the game should be recorded by printing either `game lost` or `game won`. A possible output of your solver generated for solving the board on the left-hand side of Figure 1 is:

```
guess reveal 0 0
mark 2 0
mark 2 1
reveal 2 2
reveal 3 2
reveal 3 1
mark 3 0
game won
```

Another possible output for the same board is:

```
guess reveal 3 0
game lost
```

A possible output for the board on the right-hand side of Figure 1 is:

```
reveal 0 0
reveal 0 5
reveal 5 0
reveal 5 5
mark 0 2
mark 0 3
mark 2 0
mark 2 5
mark 3 0
mark 3 5
mark 5 2
mark 5 3
game won
```

Note that the board on the right-hand side can be solved without guessing, so your solver should **always** win this board and **never** guess any steps while solving this board.

(e) Write a main program that (1) parses an input board from a file whose name is provided as a command line parameter, (2) solves the board with your solver, and (3) outputs the solution steps and outcome of the game on standard out. The main program should not print anything else on standard out except for the specified output of the solver. When the solver loses a game for which guessing cannot be avoided, your program should **not** attempt to repeatedly solve the game until it has found a winning sequence of solving steps.

(f) The course web site provides a zip file with sample board configurations in various sizes. You can use these boards for testing your solver. Some of the boards can only be solved with guessing, others can be solved without guessing. Make sure that your solver does not guess on any of the boards that can be solved without guessing.