Minesweeper Mace4

Virghileanu Teodor, Garleanu Alexandru

December 7, 2021

Teodor - Group: 30431
Alexandru - Group: 30434
Professor: Anca Nicoleta Marginean

Contents

1 The Problem

1.1 Introduction

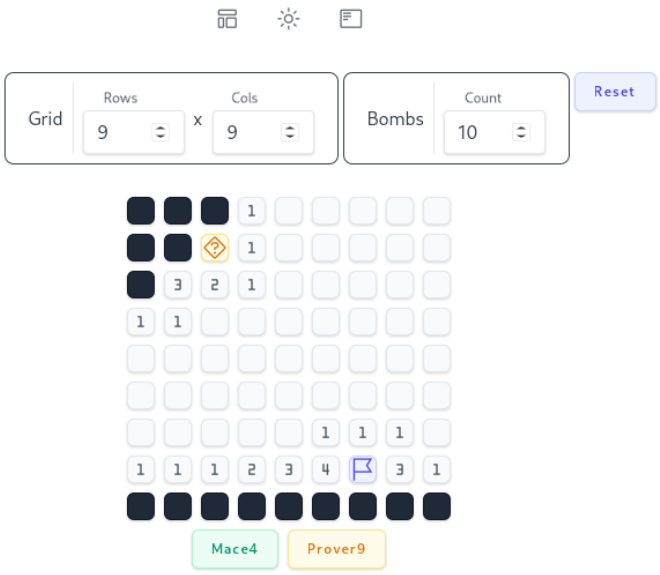
This project aims to solve the minesweeper puzzle in a user-friendly way, as if the computer is showing the user how to solve it, rather than showing the solution as a whole, with text proof.

In order to achieve this, I'll make use of both mace4 and prover9 for different purposes, however the current state of the project only makes use of mace4.

1.2 Look and feel

This is a simple configurable minesweeper game that you can play. The controls available are the number of rows, columns and the number of bombs.

These controls are used to initialise the game board, after which the user is free to ***break the ice*** and play, or ask ***mace4*** to solve one step, or the entire board in multiple steps.



More screenshots will be found at the end, including a demonstrative video

1.3 Terminology

Before continuing with the explanation of the project, it is mandatory to note a few words and the meaning behind them.

1.3.1 Board elements

The grid which makes up the game will be a *Minesweeper Board*. As such, each element of the matrix will be a *Minesweeper Cell*.

1.3.2 Hints

User hints consist of any persistent input from the user other than revealing a cell, such as the *flag cell* and *unknown cell*.

1.3.3 Frontier

The list of cells which provide useful information on how to solve the next iteration of the game will be reffered as the frontier.

This list can also be regarded as the most recently revealed cells which have neighboring unrevealed cells.

2 The Architecture

2.1 Frontend

The frontend solution is developed in typescript using VueJs 3.0 and styled via WindiCSS

Using these two powerful tools, a small frontend library is created that contains buttons and inputs, with dark and white theme support.

Although impressive enough, no code regarding the frontend will be shared, or described, since it far outweighs the purpose of this project

2.2 Backend

The frontend is backed up by Tauri which wraps a WebView and exposes a safe and controllable API in rustlang to communicate with the overlying WebView.

A fair amount of code will be described and explained, since it the direct power source of the mace4 input file

Although impressive enough, no other details will be described about the architecture and architectural decisions. This can quickly go out of hand, and there is no need for talk aside from the purpose of the project.

3 Prover9 - Mace4

As far as *Prover9 and Mace4* are concerned, they only need to look up an input file generated by the backend and generate the model list with all the possible outcomes available of the current frontier, or try to prove a user hint.

The results will be *further processed* by the backend to extract and display mutations on the board accordingly.

3.1 Decisions

As this highly detailed paper states, it is close to impossible to solve this puzzle using prover9 *(or first order propositional logic in general)* because of the huge amount of statements we need to work with.

This problem is mitigated by the use of the frontier instead of the whole board, and the use of a convenient condition that removes redundant cells from statements, reducing the overall size of the file by **a lot**

3.2 Statements

There are two key statements that we work with: the cell combinations in which bombs can appear, and the according mutual exclusion implications

Example of statement that implies the existence of a bomb on cell[0][2]

```
m02.
```

Example of statement which contains combinations of possible bomb placements

```
(m67&m87)|(m67&m88)|(m87&m88).
```

And their according mutual exclusion implication statements

```
(m67&m87) -> -((m87&m88)|(m67&m88)).
(m67&m88) -> -((m67&m87)|(m87&m88)).
(m87&m88) -> -((m67&m87)|(m67&m88)).
(m71&m80) -> -((m80&m81)|(m71&m81)).
(m71&m81) -> -((m71&m80)|(m80&m81)).
(m80&m81) -> -((m71&m80)|(m71&m81)).
```

3.3 Tree shaking

Under normal circumstances, for a cell that has **B** bomb neighbors we need to generate a statement containing:

$$\binom{8}{B}$$

OR expressions, each containing **N** **AND** expressions. On top of that we also need the other **B** statements for mutual exclusion

This quickly goes out of hand even for smaller board, since for one **B**=4 we have a total of **70 + 4** statements.

Thankfully this is mitigated by using a filter which adjusts B and removes redundant cells from the statements. This phenomenon is similar to tree shaking. While there is no tree, there is no doubt about shaking being involved.

3.4 Input

The input file template is simple

```
assign(max_seconds, 30).

formulas(assumptions).
% Magic here
end_of_list.

formulas(goals).
end_of_list.
```

Goals are set only when requesting to prove a user's hint

3.5 Generating assumptions

The point is already made, a statement is generated for each cell on the frontier. The result is then deduplicated and fed into the mutual exclusion implications generator.

This yields a minimal input size

3.5.1 Types

Code is less readable without the knowledge of several types

```
pub struct Cell {
/*cell state fields*/
}
pub struct Board {
  rows: i32,
  cols: i32,
  cells: Vec<Vec<Cell>>
}
pub type IndexPair = (i32, i32);
type StatementSource = Vec<Vec<IndexPair>>;
type Statement = String;
type Statements = Vec<String>;
pub type IndexedCell = (IndexPair, Cell);
```

The use of these type and type aliases will make it easier to read and understand the code that follows.

3.5.2 The frontier

Getting the frontier is almost a trivial task. It is a fill algorithm highly dependant on unvisited neighbors and the unvisited neighbors' visited neighbors.

The fill is performed on unvisited neighbors, however, their visited neighbors are extracted.

```
pub fn get_frontier(board: &Board) -> IndexVector {
  /*init data structures*/
  for i in 0..rows {
    for j in 0..cols {
      if visited[i][j] == false {
        stack.push((i, j));
        while !stack.is_empty() {
          if let Some(current) = stack.pop() {
            visited[current.0][current.1] = true;
            let neighbors: Vec<IndexPair> =
                get_neighbors(current, rows, cols)
                  .filter(|&(x, y)|
                    board.cells[x][y].isRevealed == false
                  );
            let subjects: Vec<IndexPair> = (&neighbors)
              .map(|pair: IndexPair| -> Vec<IndexPair> {
                get_neighbors(pair, rows, cols)
                  .filter(|&(x, y)|
                    match board.cells[x][y] {
                      Cell {
                        isRevealed: true,
                        adjacentBombs,
                        isBomb: false,
                        ..
                      } => adjacentBombs > 0,
                      Cell {..} => false
                    }
                  )
                }
              )
              .flatten()
              .collect();

            /*push unique from subjects in frontier*/
            /*push unique from neighbors in stack*/
          }}}}}
  frontier
}
```

From this frontier, the StatementSources are generated, which are then converted to strings

3.5.3 Statement Sources

```
pub fn get_statement_source(board: &Board)
    -> Vec<StatementSource> {
  /*init local vars*/
  get_frontier(board)
    .map(|index|
      (
        // the leading hint
        index,
        // its unknown neighbors
        get_neighbors(index, rows, cols)
         .filter(|&(x, y)| board.cells[x][y].isRevealed==false)
      )
    )
    .map(|(source, neighbors)| -> StatementSource {
      let variants = board.cells
        [source.0][source.1].adjacentBombs;
      if neighbors.len() <= variants {
        vec![neighbors]
      }
      else {
        Combinations::new(
        // candidates for combinations to take from
          neighbors,
        // number of possible bombs at a time
          board.cells[source.0][source.1].adjacentBombs
        )
      }
    })
}
```

3.6 Parsing Output

A great deal of work is done by Mace4 whose output command is

$$mace4 - c - m - 1 - n2 - fminesweeper.in$$

The output is piped into 2 sequential regex matches. The first regex match (i) is responsible for identifying each model, while the second regex (r) is responsible for extracting captured groups from each individual relation. Due to formatting

issues, the model detector will contain a r replacement instead of the uncaptured relation regex expression

```
interpretation\( \d+, \[number=(\d+), seconds=\d+], \[r+)\s+\]\)\

((?:relation\(m(\d+), \[ (\d+) \]\),?)+)
```

Each *relation* will be converted into a cell state and get pushed into list of possible states for each unique matrix index. Each state list will be reduced to a single cell which can be a *flagged cell*, a *revealed cell* or an *unknown cell*. This result is then mapped into a list of IndexedCells containing the matrix index and cell state for each cell that requires to be mutated on the board.

```
pub fn reduce_cell_state_variation(states: Vec<Cell>) -> Cell {
  let mut is_bomb = true;
  let mut is_not_bomb = true;
  for state in states {
    if state.isBomb == false {
    // all states should be bombs for this to be 100% a bomb
      is_bomb = false;
    }
    if state.isBomb == true {
    // all states should be safe for this to be 100% safe
      is_not_bomb = false;
    }
  }
  let is_unknown = !(is_bomb ^ is_not_bomb);

  Cell {
    isBomb: is_bomb,
    isRevealed: is_not_bomb,
    isFlagged: is_bomb,
    isUnknown: is_unknown,
    adjacentBombs: 0,
    isHint: false, // this field name is missleading, ignore
  }
}
```

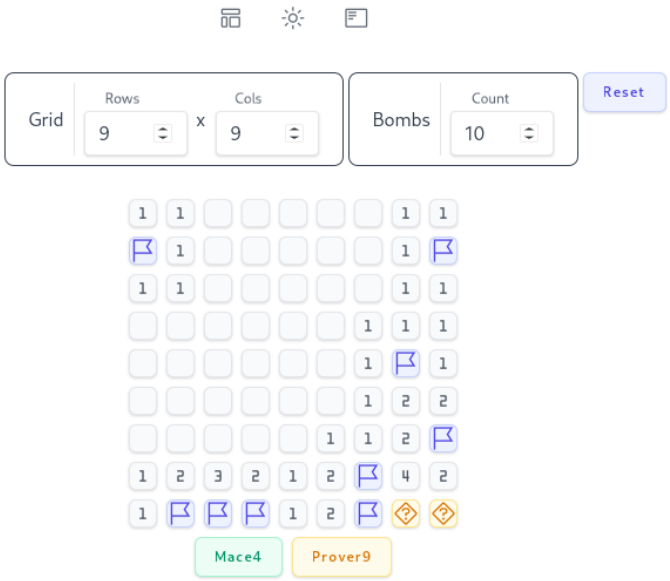
4 Conclusions

This project has proven success, although not polished at the level i want. The current implementation is not finished, but it is up to par with everything discussed on this paper. It is perfectly capable of solving the minesweeper board, in sequential steps, with an entertaining animation to look at, and enough visual feedback to dictate basic decision making. Even when cells are not deterministic, the board will display it as an unknown cell. Very convenient.

4.1 Future improvements

Mace4 and Prover9 provide a lot of useful information in the given output. It can be processed and displayed as tooltips on the cells containing decoded boolean conditions as means of reasoning to the user.

4.2 More screenshots



unsolvable board example

4.2.1 Demonstration

A demonstration is here