Code Jam 2014 - Qualification Round

Analysis: Minesweeper Master

There are many ways to generate a valid mine configuration. In this analysis, we try to enumerate all possible cases and try to generate a valid configuration for each case (if exists). Later, after having some insight, we provide an easier to implement algorithm to generate a valid mine configuration (if exists).

Enumerating all possible cases

We start by checking the trivial cases:

- If there is only one empty cell, then we can just fill all cells with mines except the cell where you click.
- If R = 1 or C = 1, the mines can be placed from left to right or top to bottom respectively and click on the right-most or the bottom-most cell respectively.

If the board is not in the two trivial cases above, it means the board has at least 2 x 2 size. Then, we can manually check that:

- If the number of empty cells is 2 or 3, it is Impossible to have a valid configuration.
- If R = 2 or C = 2, valid configurations exists only if M is even. For example, if R = 2, C = 7 and M = 5, it is Impossible since M is odd. However, if M = 6, we can place the mines on the left part of the board and click on the bottom right, like this:

```
***....
***...c
```

If the board is not in any of the above case, it means the board is at least 3 x 3 size. In this case, we can always find a valid mine configuration if the number of empty cells is bigger than 9. Here is one way to do it:

- If the number of empty cells is equal or bigger than 3 * C, then the mines can be placed row by row starting from top to bottom. If the number of remaining mines can entirely fill the row or is less than C - 2 then place the mines from left to right in that row. Otherwise, the number of remaining mines is exactly C - 1, place the last mine in the next row. For example:

```
******         ******
*****.         ****..
......   ->    *.....
......         ......
.....c         .....c
```

- If the number of empty cells is less than 3 * C but at least 9, we first fill all rows with mines except the last 3 rows. For the last 3 rows, we fill the remaining mines column by column from the left most column. If the remaining mines on the last column is two, then last mine must be put in the next column. For example:

```
******         ******
**....   ->    ***...
**....         *.....
*....c         *....c
```

Now, we are left with at most 9 empty cells which are located in the 3 x 3 square cells at the bottom right corner. In this case, we can check by hand that if the number of empty cells is 5 or 7, it is Impossible to have a valid mine configuration. Otherwise, we can hard-coded a valid configuration for each number of empty cell in that 3 x 3 square cells.

Sigh... that was a lot of cases to cover! How do we convince ourselves that when we code the solution, we do not miss any corner case?

Brute-force approach

For the small input, the board size is at most 5 x 5. We can check all (25 choose M) possible mine configurations and find one that is valid (i.e., clicking an empty cell in the configuration reveal all other empty cells). To check whether a mine configuration is valid, we can run a flood-fill algorithm (or a simple breath-first search) from the clicked empty cell and verify that all other empty cells are reachable (i.e., they are in one connected component). Note that we should also check all possible click positions. This brute-force approach is fast enough for the small input.

The brute-force approach can be used to check (for small values of R, C, M) whether there is a false-negative in our enumeration strategy above. A false-negative is found when there exist a valid mine configuration, but the enumeration strategy above yields Impossible. Once we are confident that our enumeration strategy does not produce any false-negative, we can use it to solve the large input.

An easier to implement approach

After playing around with several valid mine configurations using the enumeration strategy above, you may notice a pattern: in a valid mine configuration, the number of mines in a particular row is always equal or larger than the number of mines of the rows below it and all the mines are left-aligned in a row. With this insight, we can implement a simpler backtracking algorithm that places mines row by row from top to bottom with non-increasing number of mines as we proceed to fill in the next row and prune if the configuration for the current row is invalid (it can be checked by clicking at the bottom right cell). This backtracking with pruning can handle up to 50 x 50 sized board in reasonable time and is simpler to implement (i.e., no need to enumerate corner / tricky cases).

If the contest time were shorter, we may not have enough time to enumerate all possible cases. In this case, betting on the backtracking algorithm (or any other algorithm that is easier to implement) may be a good idea. Finding such algorithms is an art :).