AI for Minesweeper Using Backtracking

Shobhit Saxena

Branch : *Computer Engineering*
Batch : *B.Tech. 2nd Year*
National Institute of Technology, Kurukshetra

What is Minesweeper

The goal of the game is to uncover all the squares that do not contain mines
(with the left mouse button) without being "destroyed" by clicking on a mine.

Versions of Minesweeper are frequently bundled with operating systems and GUIs,
including Minesweeper in Windows, KMines in KDE (Unix-like OSes), Gnomine
in GNOME

Rules To play Minesweeper

➢ The location of the mines is discovered by a process of logic.

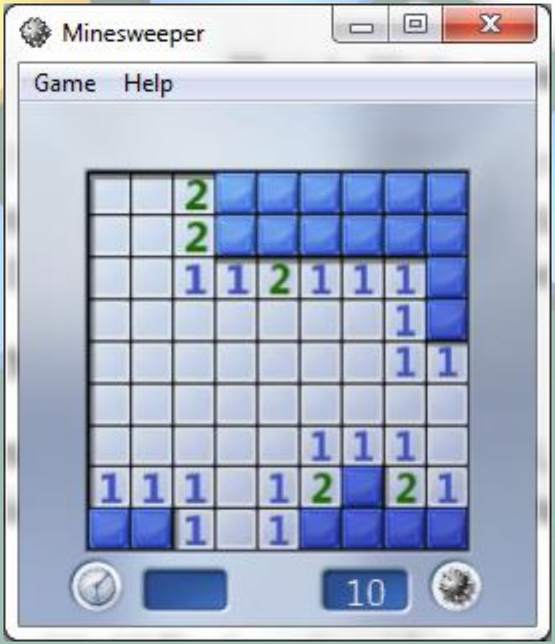
➢Clicking on the game board will reveal what is hidden underneath the chosen square or squares (a large number of blank squares may be revealed in one go if they are adjacent to each other).

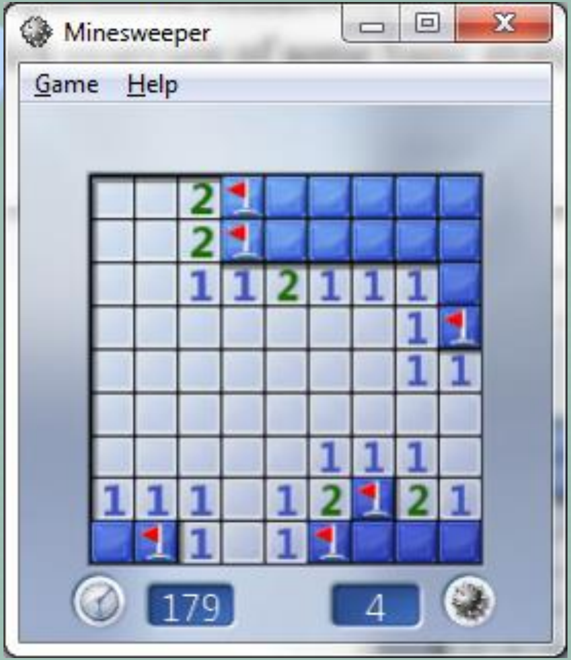
➢Some squares are blank but some contain numbers (1 to 8), each number being the number of mines adjacent to the uncovered square.

➢To help avoid hitting a mine, the location of a suspected mine can be marked by flagging it with the right mouse button.

Some Basic steps followed while playing minesweeper

1.We start with a 10×10 Beginner's grid, and click on a square in the middle:



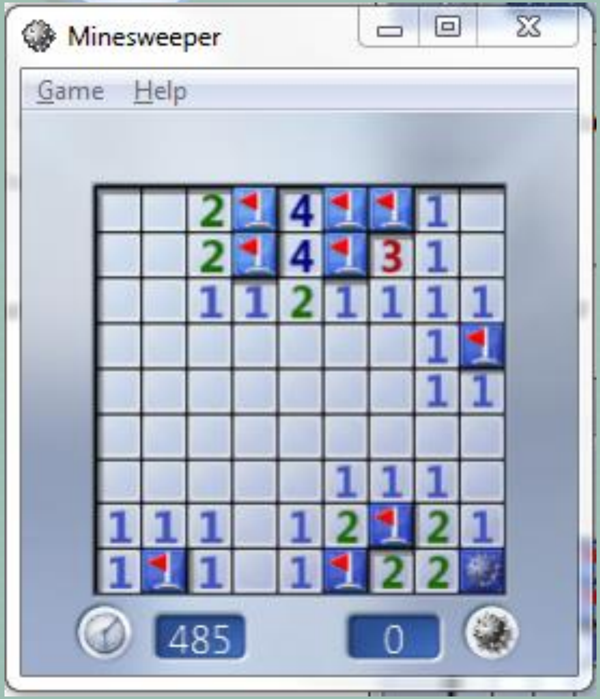


<u>After marking the mines</u>

We can quickly identify some of the mines. When the number 1 has exactly one empty square around it, then we know there's a mine there.

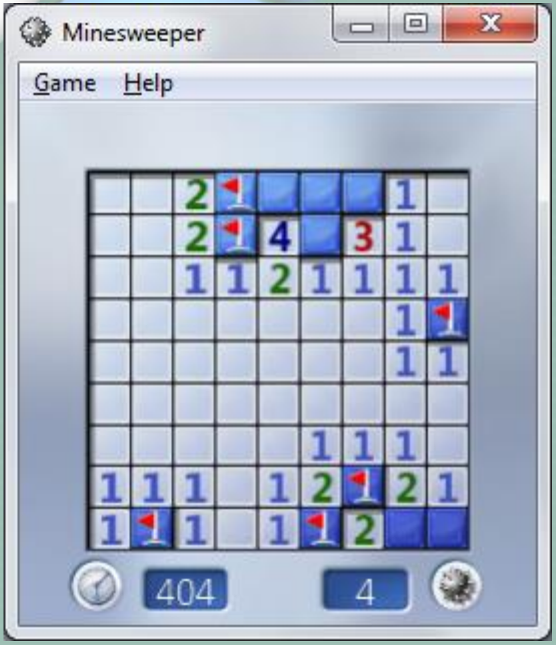
When You won the Game

The game is won once all blank squares have been uncovered without hitting a mine, any remaining mines not identified by flags being automatically flagged by the computer.



Straightforward Algorithm

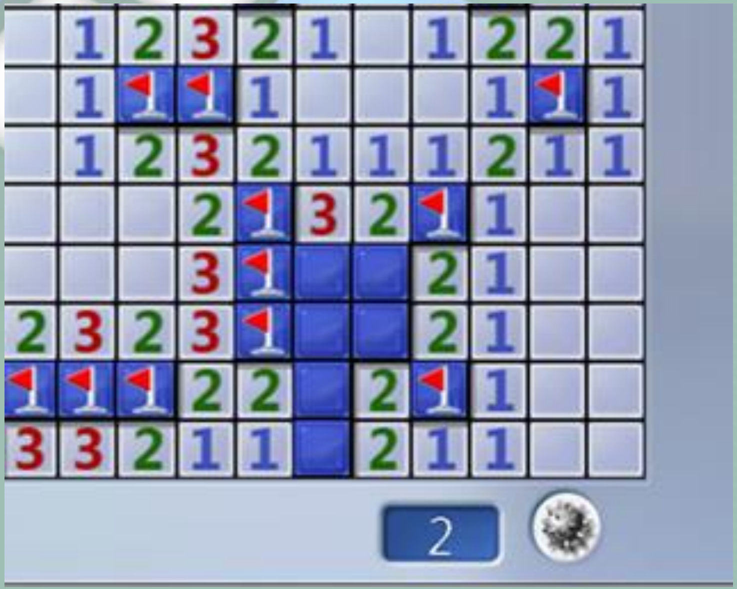
If a 1 has a mine around it, then we know that all the other squares around the 1 *cannot* be mines.
So let’s go ahead and click on the squares that we know are not mines:



Keep doing this. In this case, it turns out that these two simple strategies are enough to solve the Beginner’s grid:

Roadmap to an AI

Consider the following scenario:



Using the straightforward method, we seem to be stuck.
Up until now, whenever we mark a square as having a mine or safe, we've only had to look at a single 3×3 chunk at a time. This strategy fails us here: the trick is to employ a **multisquare** algorithm – look at multiple different squares at once.

From the lower 2, we know that one of the two circled squares has a mine, while the other doesn't. We just don't know which one has the mine(fig 1)

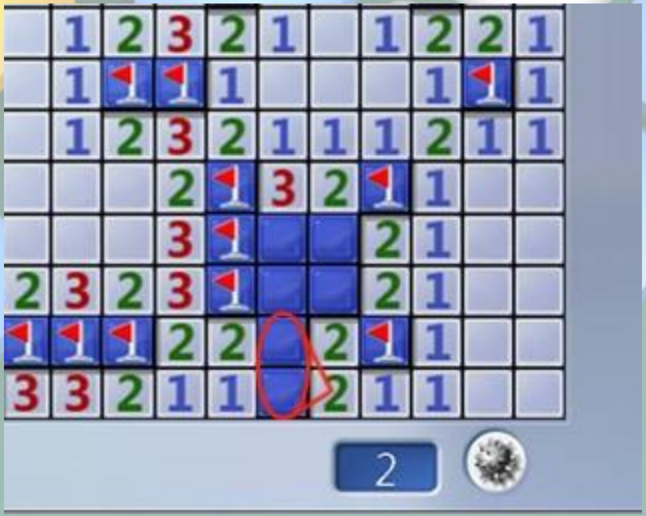


Fig 1

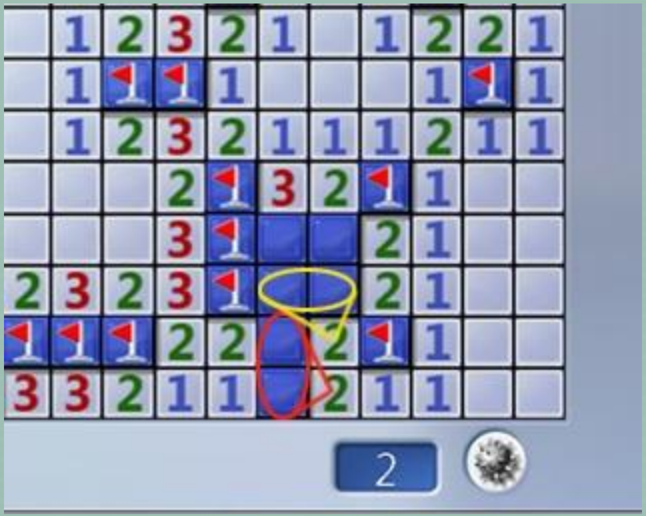


Fig 2

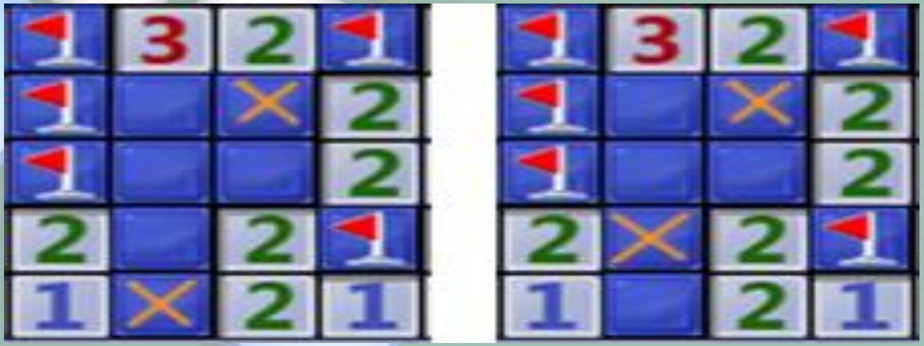
Although this doesn't tell us anything right now, we can combine this information with the next 2: we can deduce that the two yellowed squares are empty(fig 2)

The Tank Solver Algorithm

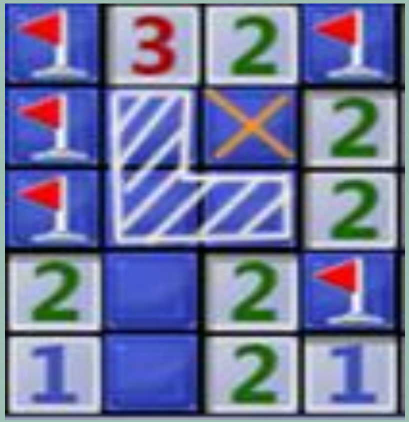
It’s difficult to make the computer think deductively like we just did. But there is a way to achieve the same results, without deductive thinking.

The idea for the Tank algorithm is to **enumerate all possible**configurations of mines for a position, and see what’s in common between these configurations.

In the example, there are two possible configurations:



You can check for yourself that no other configuration could work here. We’ve deduced that the one square with a cross *must* contain a mine, and the three squares shaded white below *must not* contain a mine:

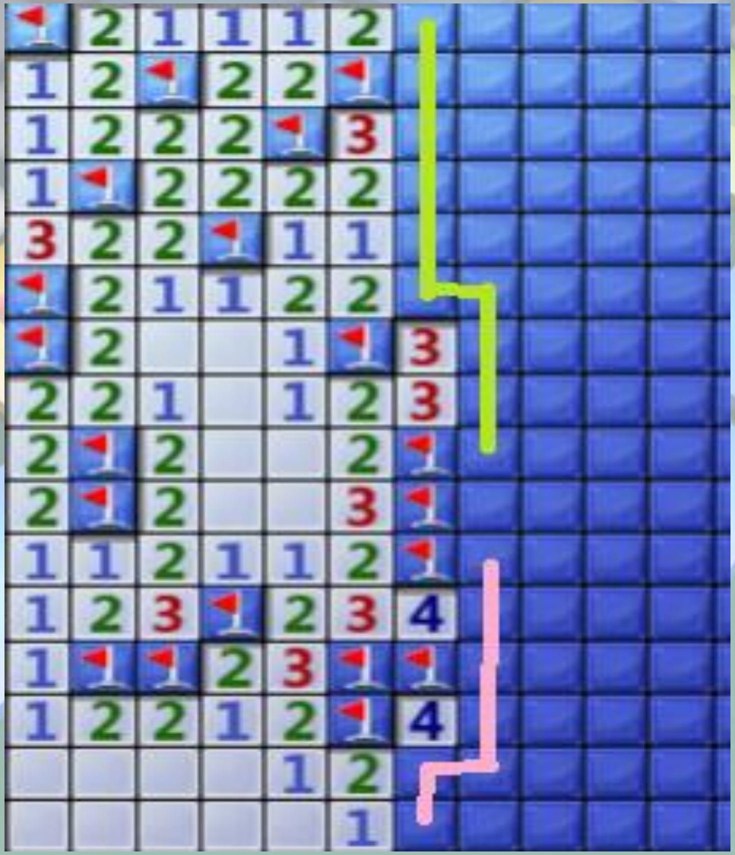


Implementation of A Optimized Algorithm

To implement this algorithm, we first make a list of **border tiles**: all the tiles
we aren’t sure about but have some partial information.

Now we have a list of   border tiles. If we’re considering every possible configuration,
 there are several disjoints regions of them. This number is cut down enough for this algorithm , but we can make one important optimization.

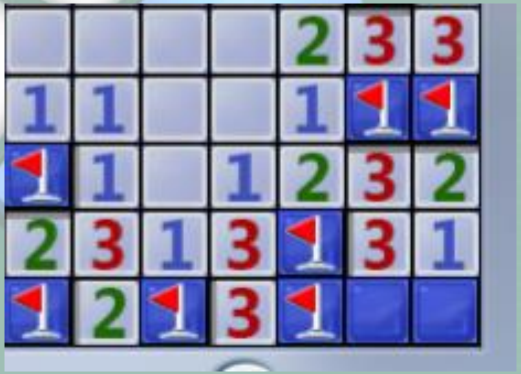
The optimization is **segregating** the border tiles into several disjoint regions



- If you look carefully, whatever happens in the green area has no effect on what happens in the pink area – we can effectively consider them separately.
- How much of a speedup do we get? In this case, the green region has 10 tiles, the pink has 7. Taken together, we need to search through combinations. With segregation, we only have : about a 100x speedup.
- Practically, the optimization brought the algorithm from stopping for several seconds (sometimes minutes) to think, to giving the solution instantly.

Probability: Making the Best Guess

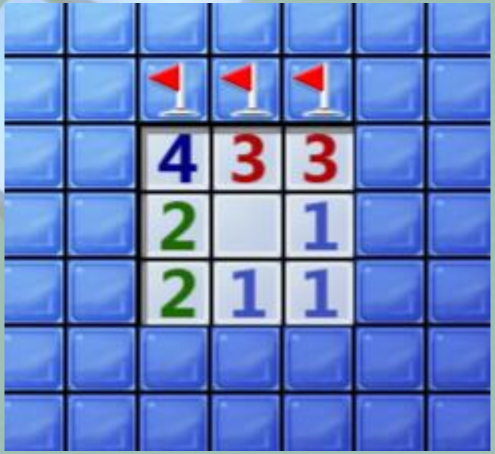
- Are we done now? Can our AI dutifully solve any minesweeper grid we throw at it, with 100% accuracy?
- Unsurprisingly, no:



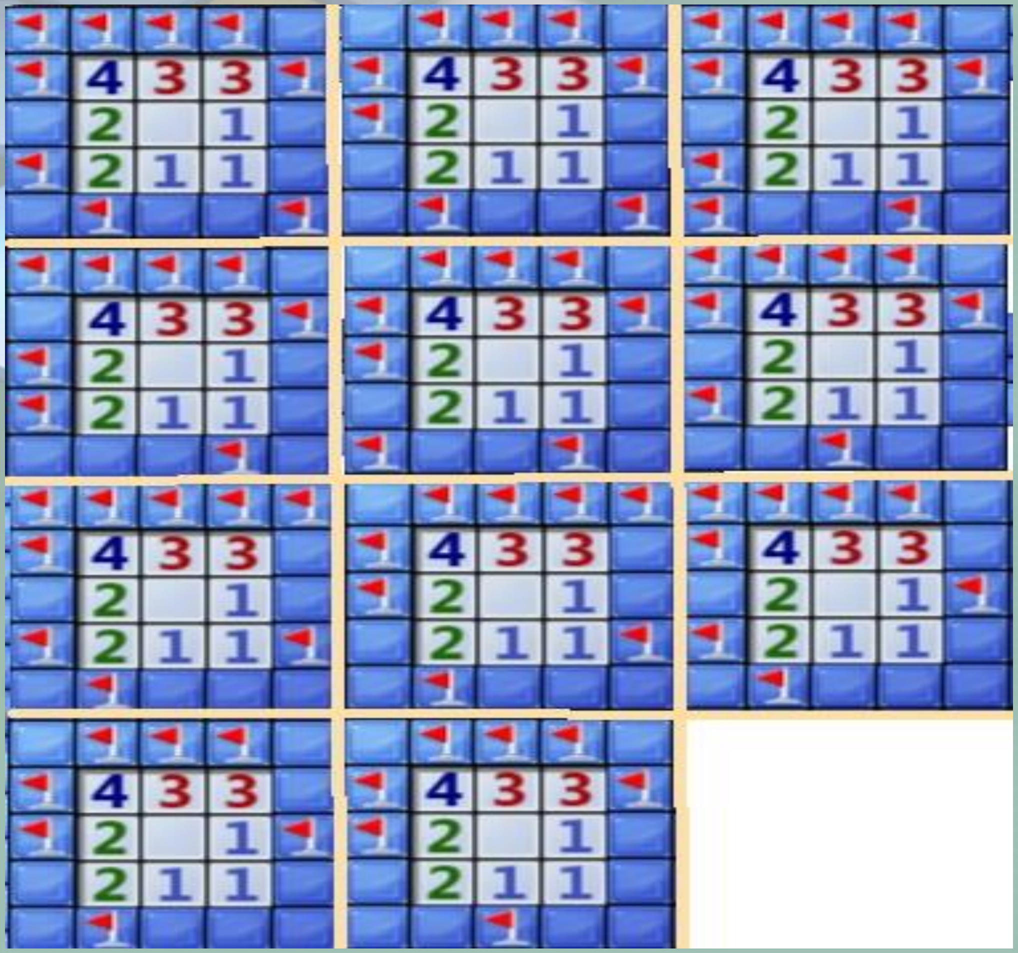
- One of the two squares has a mine. It could be in either, with equal probability. No matter how cleverly we program our AI, we can't do better than a 50-50 guess. Sorry.
- The Tank solver fails here, no surprise. Under exactly what circumstances does the Tank algorithm fail?
- If it failed, it means that for **every** border tile, there exists **some**configuration that this tile has a mine, and **some** configuration that this tile is empty. Otherwise the Tank solver would have 'solved' this particular tile.

Making the best guess

In other words, if it failed, we are forced to guess. But before we put in a random guess, we can do some more analysis, just to make sure that we're making the **best guess** we could make.

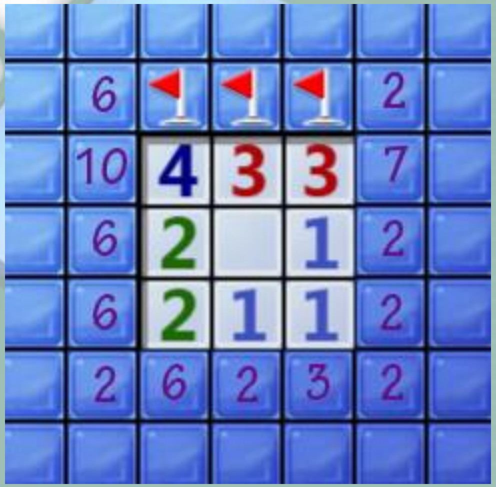


- From the 3 in the middle, we know that three of them are mines, as marked. But marking mines doesn't give us any *new information* about the grid: in order to gain information, we have to uncover some square. Out of the 13 possible squares to uncover, it's not at all clear which one is the best.
- The Tank solver finds 11 possible configurations.



Each of these 11 configurations should be equally likely to be the actual position – so we can assign each square a **probability** that it contains a mine, by counting how many (of the 11) configurations does it contain a mine:

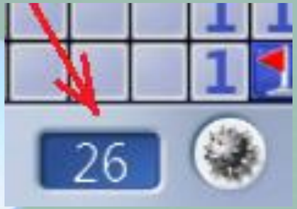
Making the best guess



Our best guess would be to click on any of the squares marked ‘2’: in all these cases, we stand an 82% chance of being correct!

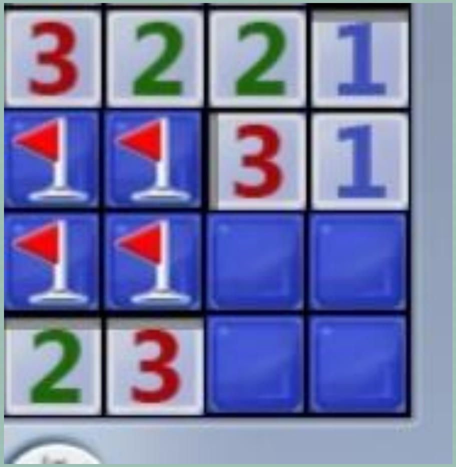
Endgame Tactics

Up until now, we haven't utilized this:



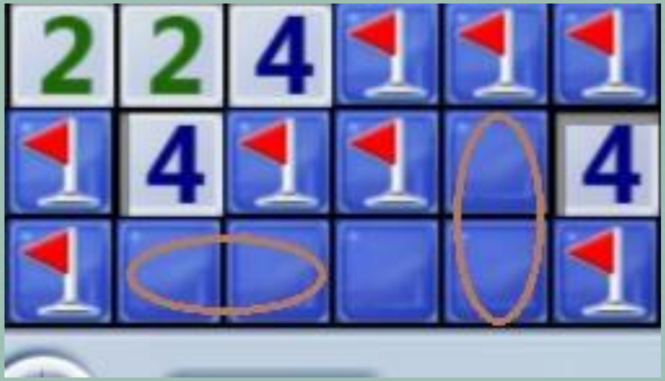
The mine counter. Normally, this information isn't of too much use for us, but in many endgame cases it saves us from guessing.

For example:

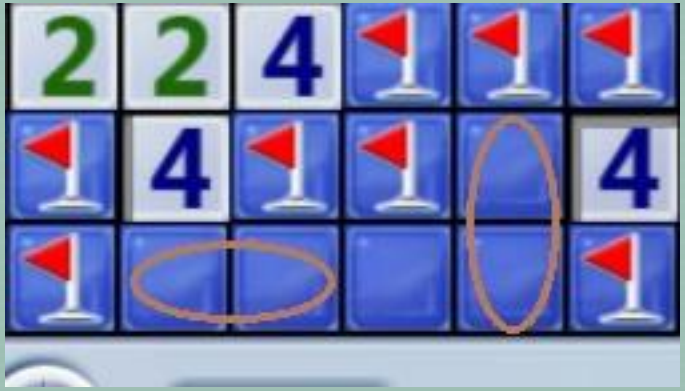


- Here, we would have a 50-50 guess, where two possibilities are equally likely.

- But what if the mine counter reads 1? The 2-mine configuration is eliminated, leaving just one possibility left. We can safely open the three tiles on the perimeter.
- So far we have assumed that we only have information on a tile if there's a number next to it. For the most part, that's true. If you pick a tile in some distant unexplored corner, who knows if there's a mine there?



- The mine counter reads 2. Each of the two circled regions gives us a 50-50 chance – and the Tank algorithm stops here.
- Of course, the middle square is safe!
- To modify the algorithm to solve these cases, when there aren't that many tiles left, do the recursion on *all* the remaining tiles, not just the border tiles.
- The two tricks here have the shared property that they rely on the mine counter. Reading the mine counter, however, is a non-trivial task that I won't attempt; instead, the program is coded in with the total number of mines in the grid, and keeps track of the mines left internally.



Conclusion

•The naïve algorithm could not solve it, unless we get very lucky.

•Tank Solver with probabilistic guessing solves it about 20% of the time.

•Adding the two endgame tricks bumps it up to a 50% success rate.