

Making Minesweeper in a Single Regex

2026-09-19 • 8 minute read

Here's a video of me playing **Minesweeper** in a text editor, only having to click 'Replace' to open a cell.



In this blog post, I'll go into detail how this was made, and how the magic powering this works.

Pushing Regex Substitution

Recently, while experimenting with regex for a side project, I discovered a very interesting property.

As you might know, when replacing something with regex, you can use a special pattern in the replacement string to insert a group's value into the output (using `$1`, `$2`, etc)

So, I got curious how would this work with the **alternation operator** (`|`)...

Every single group in a regex gets its own unique index in order of appearance, meaning you can reference groups from specific branches. But with the alternation operator, only one branch will ever succeed—all groups from all other branches will be unmatched.

As it turns out, when you refer to an unmatched group in the replacement pattern, it simply evaluates to an empty string.

This made me realize that you can treat the alternation operator as an **if-statement**: if a branch matches, output these groups; if another branch matches, output something else.

Conditional String Replacement

As a very simple example, let's say we have two separate "operations" that we might want to perform:

- `repeat( )(\w+)` should repeat a word twice (`repeat hello` → `hello hello`)
- `swap (\w+)( )(\w+)` should swap both words (`swap hello world` → `world hello`)

In both cases, we capture a space to include in the output. The first case can be replaced with `$2$1$2` (word, space, word) to duplicate the word. The second case can be replaced with `$3$2$1` (2nd word, space, 1st word) to swap both words.

To combine this, we use the alternation operator,

```
^repeat( )(\w+)|swap (\w+)( )(\w+)$
```

and combine both replacement patterns, offsetting group counts, like so: `$2$1$2$5$4$3`

- In the first case, groups 3-5 are empty, so `$5$4$3` is empty. `$2$1$2` repeats the word twice.
- In the second case, groups 1-2 are empty, so `$2$1$2` is empty. `$5$4$3` swaps both words.

So what do we do with this? Inspired by my last blog post, where I made Minesweeper in only 247 bytes of JavaScript code, I thought to myself: Can I make Minesweeper in Regex?

The Limit of State Machines

What we have right now is a Finite-State Machine: the text is the current state, each branch in the regex is a transition rule, and replacing once steps the machine to its next state.

Technically, we can achieve full turing completeness and generate the board dynamically, but only if the player is willing to spam the 'replace' button dozens of times just to open a single cell...

I decided to hardcode the board into the regex itself, as I wanted to focus on playability. This allows the game to only require one click per move like I wanted, and would also solve some other problems down the line.

I started by making a script to generate a complete Minesweeper board, which would then be used to generate the regex and the replacement pattern dynamically.

The Main Logic

The main gameplay idea is that the player starts with a grid of underscores representing unopened cells, and to open a cell, they replace an underscore with `x` and click "Replace".

So how do we actually reveal a cell? Our replacement pattern cannot contain any characters other than group references, because we need to output text conditionally, and hardcoded numbers will always appear in the output.

A solution I came up with is to add a "character bank" (like `0l2345678*`) to the end of the text. When a branch succeeds, we capture the exact character we need from the bank *inside* of the branch, and output that. If a branch doesn't match, the capture group with that character is also empty, outputting nothing.

Each cell could have its own branch checking for `x` at its coordinates, and if it matches, the branch takes a specific digit from the bank and replaces the `x` with it, while keeping everything else the same.

Matching Cells

We can match a specific cell by using `[\s\S]{idx}`, where `idx` is the index of the cell. `[\s\S]` matches any character, including newlines (unlike `.`). This expression skips `idx` characters, leaving the next character as our target cell.

Suppose we have a **`10x10` board**, and the cell at `(x=9, y=3)` has **one neighboring mine**.

We can use `y * (width + 1) + x` to convert a cell's coordinate to an index, in this case, `(9, 3)` becomes `3 * (10 + 1) + 9` or `42`.

Keep in mind that the board should be treated as having one extra hidden column, as newlines are counted, which is why we add `1` to the actual width.

We can add a branch for revealing this cell like so:

```
([\s\S]{42})x(?=.*(1))
```

Replacing that with `$1$2` keeps everything before the target cell, and replaces `x` with `1`, the amount of neighbors `(9, 3)` has.

I used lookaheads (`(?=…)`) here, as they let us search ahead of the text without moving the cursor or capturing anything. They can still contain capture groups, which lets us peek at the character bank and grab the needed digit without touching the rest of the text.

Repeat this for every cell in the board, and we've got a playable Minesweeper!

There's a better way...

This worked fine as a very first version, but that results in far too many capture groups for the entire board. Apparently, this replacement pattern syntax only allows you to refer to groups from `$1` to `$99`, meaning we actually have a group limit!

I thought that I could fix this using **named groups**. You can define them with `(?<name>…)` and refer to them with `$<name>`. But as it turns out, named groups in replacement patterns are not widely supported! At least not in VSCode, the text editor I use.

After a bit of thinking, I came up with a very good way to reduce group count: **reverse the logic**.

Suppose cells at index 3, 7 and 31 all have exactly four neighboring mines. Instead of replacing `x` with `4` for all these cells, we can use a single capturing group checking for *all* target cells:

```
(
  [\s\S]{3}
  |[\s\S]{7}
  |[\s\S]{31}
  |…
)
x
(?=.*(4))
```

It's important to note that the alternation operator still checks other possible branches if the outer part failed to match, as the regex engine will keep

backtracking until it has checked every possible state.

This reduced the group count from **200** down to just **16** on a `10x10` board! Since minesweeper boards rarely see digits over 4 or 5, we can omit digits that are never encountered, giving the final number of **6 to 12** groups on average.

But what about recursion?

The next thing I wanted to do is **open empty cells recursively**.

The first version was annoying to play because whenever you encountered `0` you had to replace all neighbors with `x` manually and click "Replace" over and over—something that a Minesweeper game typically does for you.

It was quite difficult to figure out how to make this better. My first idea was to make the regex replace underscores next to zeroes with `x`, so that you at least just had to spam "Replace". As I was trying to implement that, I suddenly got a very good idea...

You see, we already know the entire board when generating the regex...

What if we pre-calculate all regions of zeroes?

Baking in Flood Fill

We can generate a list of all related regions of zeroes using this algorithm:

```
for all cells that have zero neighboring mines {
  if the cell is already visited, skip.

  create new region with
    zeroes = []
    to_open = []

  explore all neighbors recursively {
    add the cell to 'to_open' if its not in the list

    if the cell is zero and not visited {
      mark as visited
      add to 'zeroes'
      recurse
```

```
      }
    }

    save region
  }
```

This produces a list of all separate regions of zeroes on the board.

The idea is very simple: If the player opens a cell in any of a region's `zeroes`, we reveal all cells from that region's `to_open`.

Revealing a Region

We already have a very good way to check there's an `x` in a set of coordinates, so we can re-use that here using **lookaheads** instead.

```
(?=
  (
    [\s\S]{7}
    |[\s\S]{8}
    |[\s\S]{9}
    |…
  )
  x
)
```

Each region will have its own branch with this "condition" at the start. If the lookahead matches, that means the player wants to open one of the zeroes from a region, so we proceed to reveal the entire region.

We first grab all the digits that we need for this region from the character bank:

```
(?=.*(0)(1)(2)…)
```

The cursor is still at the start of the board, because we used lookaheads.

We have to go from the start of the board to the end of the region, keeping everything outside of the region, and inserting digits for all cells from the region's `to_open`.

As a very simple example, this can look something like:

```
(.{5}).{3}(.{5})
```

Here, we would output the first group (keeping the first 5 characters), then output three digits (discarding what was there previously, as these cells should get revealed now), and then output the second group to keep the next 5 characters.

This works perfectly!!!

Empty cells now get revealed recursively, all in a single replace.

It felt so satisfying getting this to work. Bit of an illusion, though, but I think that makes it even cooler.

Final Touches

The last remaining items on my feature list were **win and loss detection**. I wanted to show a special message whenever the player won or lost, and reveal the entire board if they opened a mine.

We actually already have everything needed for this!

- For **loss detection**, we check if the player tried to reveal a mine using a lookahead. We can then consume the entire board using `.+` and reconstruct the actual board in the replacement pattern, using characters from the bank.
- For **win detection**, we can match the entire board character by character, using `[_Ff]` for mines (either empty or a flag). This ensures the player has uncovered all cells correctly, and did not cheat, as there's only one possible completed state if you ignore the flags.

To show a message, we can simply extend the character bank with some letters. `You won!` and `You lost!` only needs 10 characters total.

Then we simply match these letters in a branch, and reconstruct the message in the output. The message will only appear if the branch matches, i.e. if the player won or lost!

The final playable game ends up looking like this:

```
--------
--------
```

```
--------
--------
--------
*012345678 !Youwnlst
```

Hmm, bit ugly, isn't it?

Making it Fancier

This is quite obvious once you realize it, but I think it still adds a bit of magic to the game.

Look at this text:

```
RegexSweeper by yui.dev

--------
--------
--------
--------
--------

To open a cell, replace it with 'x' and run the
regex.
Revealed cells will be displayed like this:
012345678

There are 5*!
```

The tutorial *is* the character bank! And it has almost all letters of the english alphabet for us to use.

Of course, mentioning how revealed cells are displayed is a bit silly for a Minesweeper game, but I bet you didn't think much of that at first, did you?

> If you are enjoying this post, consider supporting my work, or reach out if you'd like to work together—I am currently looking for remote opportunities!

Try it out!

You can play Minesweeper in Regex on this interactive page! Source code is available on GitHub.

Thank you for reading! Leave a comment below if you have any questions or feedback.

1,867 words • 77 paragraphs

Discussion on GitHub

1 comment – *powered by giscus*

Oldest Newest

Write Preview Aa

Sign in to comment