Advent of Code 2025 Day 03

Deval Deliwala

Part 1

Given a grid of rolls of paper @:

```
..@@.@@@@.
@@@.@.@.@@
@@@@@.@.@@
@.@@@@..@.
@@.@@@@.@@
.@@@@@@@.@
.@.@.@.@@@
@.@@@.@@@@
.@@@@@@@@.
@.@.@@@.@.
```

find the number of rolls of paper that have *fewer than four* rolls of paper in the eight adjacent positions:

```
x x x
x @ x
x x x
```

Naive Implementation

The input file containing `@` and `.` is shaped as a matrix. We build a matrix containing 1s and 0s, where 0 refers to an empty `.` and 1 refers to an roll of paper `@`. Then we just have to iterate through each entry of a matrix, and if it is a `1`, we look at the eight adjacent positions and count how many are also `1`.

```
// build matrix
let matrix = build_matrix(file_input);
// count number of rolls with < 4 adjacent rolls
let num_rolls = 0;
for entry in matrix
    if entry == ‘1’ and num_adjacent_ones < 4
        num_rolls += 1
return num_rolls
```

At this point problems are becoming complex enough to demand writing clean and organized code. Given we are working as a matrix, we'll make a `Matrix` struct with associated functions to help solve the problem. This `Matrix` will be stored row-major, and have a `n_rows` and `n_cols` fields to allow us to prevent going out of bounds.

This algorithm will be divided into two steps:
1. Building the full 1s and 0s matrix from the input text file of `@`s and `.`s.
2. Iterating over every matrix entry in contiguous rows and accumulating the number of `1`s with fewer than 4 adjacent `1`s.

```rust
use std::io::{BufRead, BufReader, Error, ErrorKind};
use std::fs::File;
use crate::y2025::DAY4_FILE;
use std::time::Instant;


/// Binary (1s nad 0s) matrix to store rolls of papers on a grid.
///
/// Arguments:
/// - `n_rows`: [usize]   - number of rows
/// - `n_cols`: [usize]   - number of cols
/// - `data`  : [Vec<u8>] - row-major data storage
struct Matrix {
    n_rows: usize,
    n_cols: usize,
    // row-major layout
    data: Vec<u8>
}

impl Matrix {
    fn new(n_rows: usize, n_cols: usize, data: Vec<u8>) -> Self {
        Self { n_rows, n_cols, data }
    }

    /// Returns a contiguous slice of a row from self
    fn get_row(&self, row_idx: usize) -> Option<&[u8]> {

        if row_idx >= self.n_rows { return None; }

        let row_start = row_idx * self.n_cols;
        let row_end   = row_start + self.n_cols;

        Some(&self.data[row_start .. row_end])
    }


    /// Returns the entry of a given (row, col) entry in `[Self]`
    /// - Some([u8]) - if (row, col) entry is valid
    /// - None       - otherwise
    fn get_entry(&self, row_idx: isize, col_idx: isize) -> Option<u8> {

        if row_idx < 0 || col_idx < 0 {
            return None;
        }

        let (row_idx, col_idx) = (row_idx as usize, col_idx as usize);
        if row_idx >= self.n_rows || col_idx >= self.n_cols {
            return None;
        }

        Some(self.data[row_idx * self.n_cols + col_idx])
    }

    /// For a given row entry, calculates number of adjacent `1`s
    fn sum_adjacent(&self, row_idx: usize, col_idx: usize) -> usize {
        let mut adj_count = 0;
        let row_idx = row_idx as isize;
        let col_idx = col_idx as isize;

        for dc in -1..=1 {
            for dr in -1..=1 {
                if dc == 0 && dr == 0 {
                    continue;
                }

                match self.get_entry(row_idx + dr, col_idx + dc) {
                    // matrix is only full of 1s and 0s.
                    // so we can just add every entry -> # 1s
                    Some(v) => { adj_count += v as usize },
                    None    => { continue; },
                }
            }
        }

        adj_count
    }
}


/// Appends a row to a given `data` [Vec] based on entries in provided
/// &[str] buffer. Used by [build_matrix] to build full [Matrix] struct.
fn build_matrix_row(buffer: &str, data: &mut Vec<u8>) -> Result<(), Error> {
    for &entry in buffer.as_bytes().iter() {
        if entry == b'.' {
            data.push(0)
        } else if entry == b'@' {
            data.push(1)
        } else {
            return Err(
              Error::new(
                ErrorKind::InvalidInput,
                "Expected grid containing @ and . only"
              )
            );
        }
    }

    Ok(())
}


/// Builds a [Matrix] struct from the given input `file`
/// containing a grid of "@"s and "."s
fn build_matrix(file: &str) -> Result<Matrix, Error> {
    let mut buf_reader = BufReader::new(File::open(file)?);
    let mut string_buf = String::new();
    let mut data: Vec<u8> = Vec::new();

    // first row to initialize `n_cols`
    buf_reader.read_line(&mut string_buf)?;
    let buffer = string_buf.trim();
    let n_cols = buffer.len();
    build_matrix_row(buffer, &mut data)?;
    string_buf.clear();

    let mut n_rows = 1;
    while buf_reader.read_line(&mut string_buf)? != 0 {
        let buffer = string_buf.trim();
        if buffer.is_empty() {
            continue;
        }

        if buffer.len() != n_cols {
            return Err(
              Error::new(
                ErrorKind::InvalidInput,
                "Expected grid containing equal length rows"
              )
            );
        }

        n_rows += 1;
        build_matrix_row(buffer, &mut data)?;
        string_buf.clear();
    }

    Ok( Matrix::new(n_rows, n_cols, data) )
}


/// Main driver for part 1.
fn run_pt1() -> Result<usize, Error> {
    let mut roll_count = 0;
    let matrix = build_matrix(DAY4_FILE)?;

    for row_idx in 0..matrix.n_rows {
        if let Some(row_slice) = matrix.get_row(row_idx) {
            for (col_idx, &entry) in row_slice.iter().enumerate() {
                if entry == 1 && matrix.sum_adjacent(row_idx, col_idx) < 4 {
                        roll_count += 1;
                }
            }
        }
    }

    // final answer
    Ok(roll_count)
}
```

The rest is in progress.