Homework I—Cellular Automata

6 November 2026

1 Instructions

Write a framework for simulating cellular automata, together with a terminal application that runs it. A cellular automaton is a grid of cells. In every step, each cell computes its new state from its own state and the states of its neighbours, and all cells are updated at the same time. The best-known example is Conway's Game of Life. The goal of the project is to create a cellular automata framework, which should be generic over the type of cell and over the rule that governs it.

Project deadline: 4 December 2026. Each week of delay reduces the maximum number of points by 5.

Section 2 describes the functionality you must implement. For its full implementation you can receive 10 points. Section 3 describes additional functionality. Next to each item you will find the number of extra points and its identifier. To receive points for additional functionality, you must implement the functionality from section 2 for at least 8 points. Section 4 lists things you must not do in the project.

If the task says that you should implement a struct/trait/function called `X` that is not otherwise defined in detail, its implementation is up to you.

It is completely forbidden to use libraries other than those allowed in the task. Projects using other libraries will not be graded. You may use any elements of the standard library, unless they are forbidden by section 4.

The grade for the project is computed as

$$k = \min(r + a + p, 15),$$

where k is the final grade, r is the number of points from section 2, $a = 0$ if $r < 8$, otherwise a is the number of points from section 3, and p is the number of points from section 4 (zero or negative).

2 Required functionality

1. (2 pts) Core traits.
   (a) A trait `Cell` describing the state of a single cell of the automaton (unrelated to `std::cell::Cell`). A cell state can be cloned and compared, and has a default value: the "empty" state, used for everything outside the grid. It can also be converted to and from the character used in text files and when printing.
   (b) An enum `Neighbourhood` with variants `Moore` and `VonNeumann` (the Moore and von Neumann neighbourhoods), which can list the positions of the neighbours of a cell relative to that cell.
   (c) A trait `Rule<C: Cell>` describing how cells change. Given the current state of a cell and the states of its neighbours, it returns the next state of that cell. It also says which `Neighbourhood` it uses. The order in which the neighbours are passed to the rule is up to you, but it must be documented.
2. (2 pts) The grid.

(a) A generic struct `Grid<C: Cell>` with a width and a height given at runtime and an enum `EdgeMode` with variants `Dead` (cells outside of the grid are `C::default()`) and `Wrap` (the grid is a torus).

(b) Creating a grid, reading a cell and changing a cell. Using coordinates outside of the grid returns an error.

(c) Computing one generation using any `Rule` for the grid's cell type. All new states must be computed from the old generation only.

(d) `Display` for the grid, printing one line of characters per row.

(e) A struct `Simulation` generic over the cell type and the rule, holding a grid, a rule and a generation counter.

3. (2.5 pts) Three rule sets. Each has its own cell type implementing `Cell` and its own struct implementing `Rule`.

   (a) (0.75 pts) **Conway's Game of Life**: two states, Moore neighbourhood. A dead cell with exactly 3 live neighbours becomes alive. A live cell with 2 or 3 live neighbours stays alive. In all other cases the cell is dead in the next generation.

   (b) (0.75 pts) **Brian's Brain**: three states (off, on, dying), Moore neighbourhood. An off cell with exactly 2 *on* neighbours turns on; an on cell becomes dying; a dying cell turns off. Dying cells do not count as on neighbours.

   (c) (1 pt) **Langton's ant**: von Neumann neighbourhood. Every cell has a colour and may hold the ant, facing one of four directions. In one step the ant turns right on a white cell and left on a black one, flips the colour of the cell and moves one cell forward. Because every cell is updated independently, the ant is not moved by the cell it stands on: instead, a cell takes the ant if the ant of one of its neighbours moves onto it. An ant that leaves the grid in the `Dead` mode disappears.

   Use enums for the cell types; do not represent the states with several `bool` fields.

4. (1 pt) Loading patterns.

   (a) Parsing a text into a grid sized to the text: each line is a row and each character is a cell.

   (b) Copying (pasting) a smaller grid onto a larger one at a given position.

   (c) Errors for: an unknown symbol (with line and column), rows of different lengths, an empty pattern and a pattern that does not fit at the given position.

   The example pattern files provided with this task use the symbols below, so your parser has to support at least these:

Life	`.` dead, `#` alive
Brian's Brain	`.` off, `#` on, `o` dying
Langton's ant	`.` white, `#` black, `^`, `>`, `v`, `<` ant on a white cell facing up, right, down, left

   The remaining states of Langton's ant (an ant on a black cell) may use symbols of your choice.

5. (1.5 pts) The application.

   (a) The project is split into a library (modules declared in `lib.rs`) and an application (`main.rs`). The application contains only argument parsing, the main loop and printing to the terminal; the library contains everything else.

   (b) Command line options for choosing the rule, the pattern file, the size of the grid, the edge mode, the number of steps and the delay between frames. You can use the *clap* library.

(c) The pattern is placed in the centre of a grid of the given size. Without a pattern file, the grid is filled randomly (you can use the *rand* library).

(d) In each iteration the application clears the terminal, prints the grid and the generation number, and waits for the given delay.

(e) Errors from the library are printed in a readable way and result in a non-zero exit code.

6. (1 pt) An `enum Error` and tests.

   (a) The enum has variants for all categories of errors you encounter, e.g. `OutOfBounds`, `UnknownSymbol`, `IoError(std::io::Error)`, etc. Using the *thiserror* library is recommended. Functions in the library should pass errors on to the application, unless they can handle them themselves.

   (b) Tests in every module you write. Every automaton, both edge modes and every kind of error must be tested.

7. A `README.md` file containing:

   (a) a description of the project structure,

   (b) a description of any functionality not described in the task,

   (c) your favourite module and why it is your favourite.

8. A file `ADDITIONS` containing the list of implemented functionalities from section 3 as a list of their identifiers, e.g.:

   ```
   rle
   tui
   ```

3 Additional functionality

1. (2 pts, `rule_strings`) A rule `LifeLike` configured at runtime by a string in the `B/S` notation, e.g. `B3/S23` (Life), `B36/S23` (HighLife) or `B2/S` (Seeds). `B` lists the numbers of live neighbours that bring a dead cell to life, `S` lists the numbers of live neighbours that keep a live cell alive. The rule can be chosen with `--rule B36/S23`; invalid strings are reported as errors.

2. (2 pts, `rle`) Reading and writing patterns in the RLE format. Selected by the extension of the file, or by an option.

3. (2 pts, `analysis`) Detection of cycles in the evolution of a simulation: a still life or an oscillator, together with its period. An option for writing the number of live cells in every generation to a CSV file.

4. (2 pts, `topology`) An additional grid topology, e.g. a hexagonal grid with 6 neighbours, together with at least one rule that uses it. The grid must also be displayed in the terminal in a way that is as natural as possible for that topology (e.g. for a hexagonal grid, with cells drawn two characters wide and every other row shifted by one character). Adding it should require no changes in the rules that already exist.

5. (1 pt, `colour`) Cell states displayed in the terminal as a symbol together with a colour (e.g. using ANSI escape sequences), for example different colours for the states of Brian's Brain and for the ant. The colours should come from the library, not be hard-coded in the application.

6. (1 pt, `automaton`) Another non-trivial automaton of your own choice, with at least 4 states and a rule that cannot be expressed as a Life-like `B/S` string, e.g. Wireworld. Provide a pattern file demonstrating it and describe it in the `README.md`. Points are awarded for this item only once, no matter how many automata you implement.

7. (3 pts, `tui`) A TUI implemented using *ratatui*, together with *crossterm*. It should allow pausing, single-stepping, changing the speed of the simulation and toggling cells with the cursor.

8. (1–3 pts, `custom`) Any other additional functionality, graded subjectively by the instructor (it is best to consult before the deadline).

4 Prohibited practices

The occurrence of any of the following in the project results in subtracting the number of points specified.

1. Using the methods `unwrap`, `expect` or the macro `panic`. −1 pt for 1 or 2 uses, −2 pts for more. Using them in tests is allowed.

2. Using a struct where an `enum` would be right, i.e. keeping in a struct fields whose value sometimes has meaning and sometimes does not, e.g.

   ```
   struct IP {
    v4: String,
    v6: String,
    is_v4: bool
   }
   ```

   −1 pt for one occurrence, −2 pts for more.

3. Ignoring errors when they should be handled, e.g. calling a function that returns `Result<T, E>` and checking only the `Ok` case with `if let` when the error matters (if it does not matter, this is of course allowed). −1 pt for one or two occurrences, −2 pts for more.

4. Functions longer than 30 lines. −1 pt for 1 or 2, −2 pts for more.

5. Functions that needlessly take ownership of objects when they could only borrow them. −1 pt for 1–3 such functions, −2 pts for more.

6. Using `clone` when it is not needed (i.e. the code compiles without it and the logic is not affected). −1 pt for one or more uses.

7. Using `Rc`, `Arc`, `RefCell`, `Mutex` or `std::cell::Cell`: −2 pts (your own trait `Cell` is of course fine).

8. Using `unsafe`: −2 pts.