Documentation Cellular Automats (Assignment 15)

Mario Neuner, Christoph Bichlmeier, Jakob Guentner
Leander Decristoforo

June 21, 2025

Contents

1 Part A: One Dimensional Cellular Automats

1.1 Task

The task involves implementing a one-dimensional cellular automaton system that evolves through discrete time steps based on predefined rules. The automaton consists of a linear array of cells, each in state 0 or 1, where the next state is determined by a cell's current state and its immediate neighbors. The system must support four specific rules (22, 106, 187, 214) and handle two initialization modes: a determined start with a 1 in the middle and zeros around and a random configuration. The program takes two command-line arguments, N for grid size and M for the number of time iterations and outputs the automaton's state after each iteration. These results can then be visualized using a separate plotting tool.

1.2 Idea

This project is built around a modular and rule-driven approach to simulating cellular automata. At the heart of the system is a structure called "cellauto", which holds the key components of the simulation. It holds the current state of the grid, stored as a character array, the rule being applied encoded as an 8-bit pattern and parameters like the grid size N and number of iterations M.

The rules themselves are defined using arrays of eight characters, where each position corresponds to one of the possible neighborhood cell configurations (e.g., the pattern "111" maps to index "0"). During the simulation, the system updates the grid in steps. For each cell, considering its left, center, and right neighbors, wrapping around at the edges, determining the corresponding rule index, and updating the cell's state based on the rule.

There are two modes of initialization implemented, one is determined, starting with a single 1 in the center and zeros around, and one random. The results of each simulation are saved in a format designed for easy visualization. Each line in the output file represents the complete state of the cells at a given time step.

1.3 Implementation

Data Structures and Rule Encoding

At the core of the system is the `cellauto`-struct, defined in `structs.h`. This struct houses all the parameters and data needed to run a simulation, including the current state of the grid, the active rule, the simulation data, and the mode of initialization:
The rule definitions themselves are declared as global constants in `structs.h` and initialized in `structs.c`. Each rule is represented as an array of eight characters `(e.g., RULE_22)`, corresponding to the eight possible arrangement of three cells. These arrays are indexed from 0 to 7, where each index corresponds to a specific neighborhood pattern. For instance, `Rule_22 = {0,0,0,1,0,1,1,0}` defines the rule's response to configurations ranging from 111 with index 0 to 000 with index 7.

State Management and Rule Application

The evolution of the automaton is made by functions implemented in `cell.c`. Where two initialization modes are used. One Deterministic via `reset(cellauto *c)`, which sets a single active cell (1) at the center of the grid, and the other one Random via `randomize(cellauto *c)`, which assigns each cell a 0 or 1 at random, using `srand(time(NULL))`.
The key function that does the state transitions is `apply_rule(cellauto *c, char *next_state)`. For each cell in the current state array, the function considers its left, center, and right neighbors. These three bits form a 3-digit binary number, which is converted into a decimal index between 0 and 7. This index is then used to look up the new state in the rule array. For example, the neighborhood 0 1 0 corresponds to binary 010, or decimal 2. Applying `Rule_22`, the next state is retrieved via `RULE_22[2]`, which equals to 0.

Simulation Flow and Execution

The entry point of the program is `main()` in `1d_states.c`. It expects two command-line arguments: the grid size `N` and the number of iterations `M`. It starts with the input handling, where the program verifies the validity of user input and allocates memory for a `cellauto` instance and its state array. It returns an error message if memory allocation fails or if the inputs are invalid.
First it runs the deterministic Initialization where the grid is initialized using `reset()`. The simulation runs for each of the four predefined rules (`RULE_22`, `RULE_106`, `RULE_187`, `RULE_214`), updating the `rule` pointer and `rule_name`

along the way. For each rule, the `steps()` function is called to run the system for the given number of iteration steps. At each step, the full state is saved in a file named `1d_states/1d_rule_<Regel>.txt`, where `<Regel>` is the rule number.
For Random Initialization Phase the random number generator initializes a new starting grid. For each rule, the state is randomized in `randomize()`, and the simulation is repeated as explained previously. Each rule's results under random initial conditions are also saved to the file named `1d_states/1d_rule_<Regel>_random.txt` for comparison.

1.4 Output

The program generates two types of output:

First a `Text File` for each rule (e.g., `1d_rule_22.txt`) recording the grid state per iteration, space-separated (e.g., `0 0 1 1 0 0 0`). These files are saved in the `1d_states` directory, where they are created automatically.
There is also a `Visualization` created with the `plot_1d` tool reading the created files and producing PNG images (e.g., `1d_plots/1d_rule_22.png`) using `gnuplot`. Each image depicts the automaton's evolution over time, with rows representing iterations and black/white pixels for 1/0 states.

i. To use the system, you have to compile by using the `Makefile` with the command `"make"`.

ii. To run the simulation, use the command: `"./1d_states N M"` (e.g., `201 100`).

iii. To generate the plots, use the command: `"./plot_1d 22"` (e.g., for `Rule_22`).

Examples: Here with initial state of 1 in the middle.

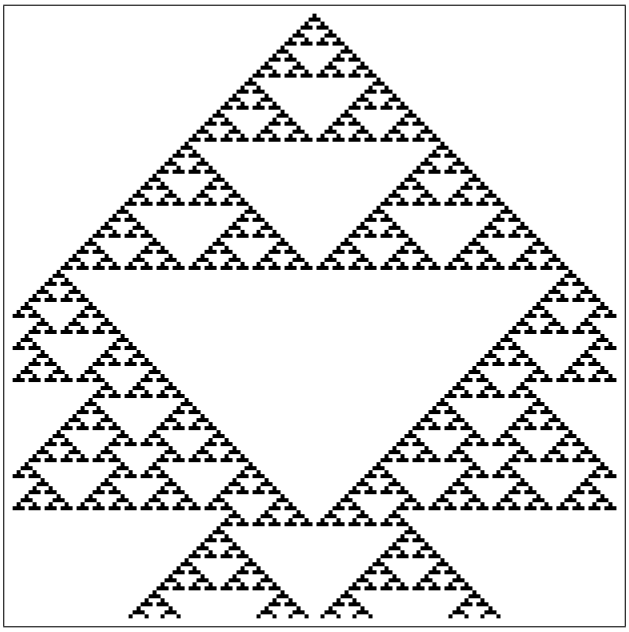

Figure 1: Rule_22; N=151, M=151

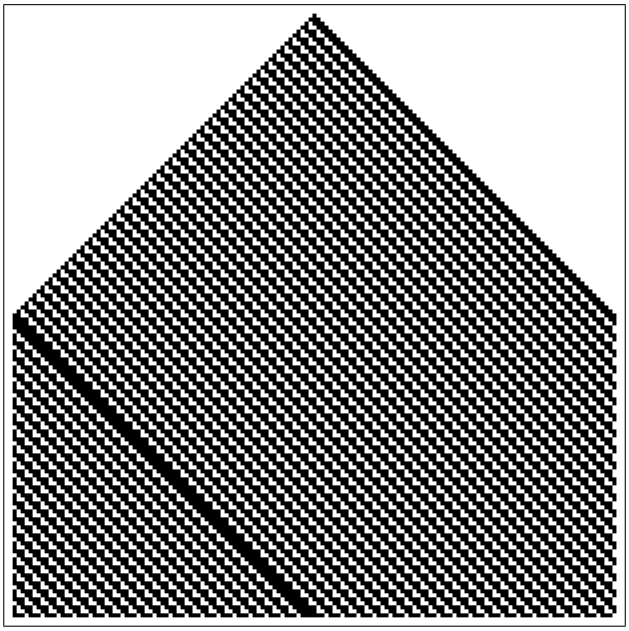

Figure 2: Rule_214; N=151, M=151

2 Part B

2.1 Idea

2.2 Implementation

2.3 Output

`The program compiles with "gcc -Wall -Wextra -Werror -Wpedantic -std=c18 rakete.c -o rakete".` When running `./rakete <masse_leer> <masse_treibstoff> <geschwindigkeit_treibstoff> <massenverlustrate_treibstoff>` you get the output:

3 Appendix: Code

3.1 Part A: Main

Listing 1: Main (1d_states.c)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <time.h>
5
6   // Own headers with function declarations, structs etc.
7   #include "structs.h"
8   #include "cell.h"
9   #include "stepcom.h"
10
11
12  int main (int argc, char *argv[]) {
13      if (argc != 3) {
14          fprintf(stderr, "Usage: %s <n> <m>\n", argv[0]);
15          return 1;
16      }
17
18      // User input for size and iterations
19      int size = atoi(argv[1]);
20      if (size <= 0) {
21          fprintf(stderr, "Error: Size must be a positive integer.\n");
22          return EXIT_FAILURE;
23      }
24      int iterations = atoi(argv[2]);
25
26      // Initialize state
27      cellauto *cell = malloc(sizeof(cellauto));
28      if (!cell) {
29          fprintf(stderr, "Error: Memory allocation failed.\n");
30          return EXIT_FAILURE;
31      }
```

```
        cell->state = malloc(size * sizeof(char));
        if (!cell->state) {
            fprintf(stderr, "Error: Memory allocation for state failed.\n");
            free(cell);
            return EXIT_FAILURE;
        }
        cell->rule = NULL;
        cell->rule_name = 0;
        cell->rand = false;
        cell->iterations = iterations;
        cell->size = size;


        reset(cell); // Initialize state with a single '1' in the middle
        cell->rule = RULE_22;
        cell->rule_name = 22;

        // Compute steps for not random initial condition
        steps(cell);
        reset(cell);
        cell->rule = RULE_106;
        cell->rule_name = 106;

        steps(cell);
        reset(cell);
        cell->rule = RULE_187;
        cell->rule_name = 187;

        steps(cell);
        reset(cell);
        cell->rule = RULE_214;
        cell->rule_name = 214;

        steps(cell);
        reset(cell);


        // Now random states
        cell->rand = true;

        // Set random initial state
        srand(time(NULL)); // Seed for random number generation
        randomize(cell);
        cell->rule = RULE_22;
        cell->rule_name = 22;

        // Compute steps for random initial condition
        steps(cell);
        randomize(cell);
        cell->rule = RULE_106;
        cell->rule_name = 106;

        steps(cell);
        randomize(cell);
        cell->rule = RULE_187;
        cell->rule_name = 187;

        steps(cell);
        randomize(cell);
        cell->rule = RULE_214;
        cell->rule_name = 214;

        steps(cell);

        // Free allocated memory
        free(cell->state);
        free(cell);

        return EXIT_SUCCESS;
    }
```