Project Title: Elementary Cellular Automaton

Student Name: Ricardo Garcia

Course Title: Intermediate Programming ECE 231L 008

Instructor Name: Andrew Fierro

Due: 7 May 2024, 2:30 PM

---

1. Introduction and Overview

Coding a cellular automaton enhances programming abilities. Working with arrays and/or matrices, having data that is searched, sorted, and analyzed. The project aims to implement a basic cellular automaton, an Elementary Cellular Automaton, that utilizes a one-dimensional grid, simple binary states, and a small local neighborhood. The rules will use the binary of a number (0-255) that will dictate the rules. For example, for the number 30:

30 when converted to binary results in 00011110

This creates a ruleset as follows

Neighborhood	111	110	101	100	011	010	001	000
Result	0	0	0	1	1	1	1	0

Starting with a single central cell results in the following

```
00000000000000100000000000000
00000000000001110000000000000
00000000000011001000000000000
00000000000110111100000000000
00000000001100100010000000000
00000000011011110111000000000
00000000110010000100100000000
00000001101111001111110000000
```

2. Objectives

Objective 1: Take an int (0-255) to define the ruleset, an initial state (using 0 and 1), and the number of steps as an argument. Ex. 110 000000000010000000000 11 results in rule 110, center is at 0, for 11 steps.

Objective 2: Convert the int to a binary number and store it as a ruleset

Objective 3: Calculate the next generation and save it to a vector or similar data structure.

Objective 4: Neatly display all the generations chronologically to the console.

---

3. Classes and Functions

This program contains the class *ElementaryCA*, the constructor that creates a rule set and stores the initial state; the resulting states are not made until the *run(n)* function is called. The *display()* function displays the CA to the console.

A. *createRuleset(int rule) -> std::map<std::string, int>*

This function generates a ruleset for the cellular automaton based on a provided rule number. The rule number is converted to an 8-bit binary representation, where each bit of this binary string represents a specific outcome (0 or 1) for each possible neighborhood configuration (111, 110, 101, 100, 011, 010, 001, 000) the cell to the left, center and right. The function returns a hash map of each of the eight possible three-bit combinations to their corresponding outputs, as the binary string dictates.

B. *nextState(std::string state) -> std::string*

Given the current state and the ruleset, this function generates the next state of the automaton. It iterates over each cell in the state, determines its neighborhood configuration by considering the current cell and its immediate neighbors to the left and right, and wraps around to handle the edge cells. Using the neighborhood configuration, it looks up the resulting new state for that neighborhood in the ruleset and returns the next generation of the automaton.

C. *generate(int steps) -> std::vector<std::string>*

This function runs the cellular automaton simulation for a given number of steps starting from an initial state. It utilizes the *nextState* function to compute the state of the automaton for each step and stores each state in a vector, which is returned.

D. *makePretty(std::string state, char on, char off) -> std::string*

This function converts a state, as a string, into a more visually appealing format for display, using specified characters for ‘on’ (1) and ‘off’ (0) states. It iterates through the state string and replaces each ‘0’ with the ‘off’ character and each ‘1’ with the on character, constructing and returning a new formatted string that is more neat for visualization.

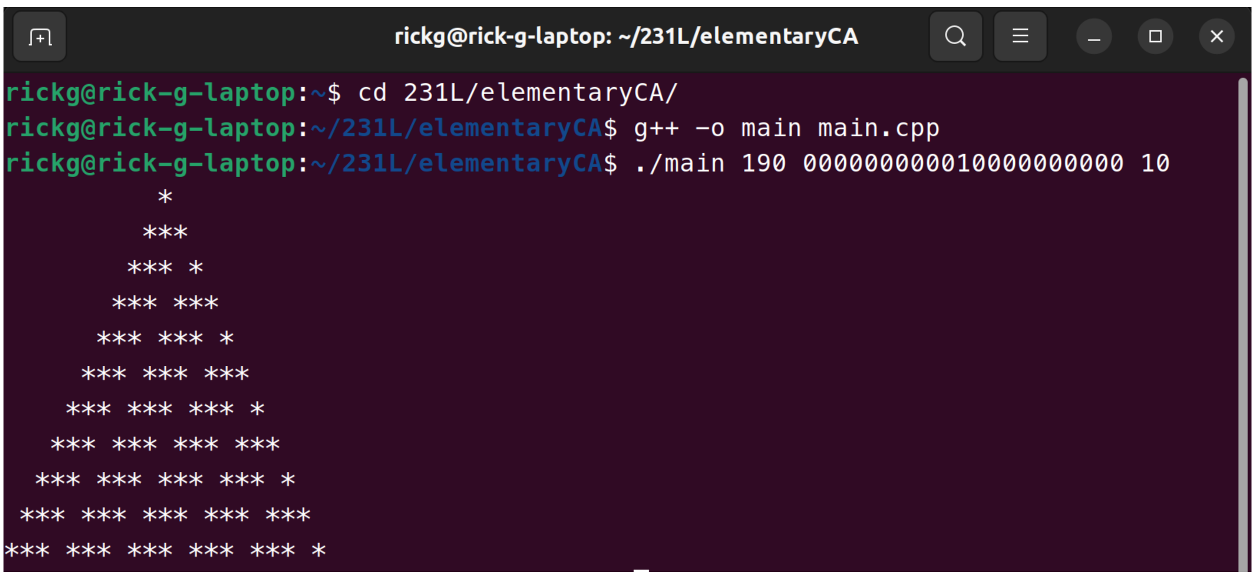
---

4. Running the Program

A. Open the console and navigate to the ‘main.cpp’ file for Elementary CA

B. Run ‘g++ -o main main.cpp’, no other libraries or header files are needed.

C. Run ‘./main <rule> <initial state> <n of steps>’. Ex. ‘./main 190 000000000010000000000 10, ‘190’ is the rule number, ‘000000000010000000000’ is the initial state, and ‘10’ is the number of steps the simulation will execute.



---

5. Troubleshooting

A. Git Version Errors

Upon beginning the project, I debated whether to store the states as a *std::string* or a char[]. Attempting to make a new branch with git to try to use *char[]* instead of *std::string*, I lost both the original branch and the new one.

B. Wraps-Around to Handle the Edge Cells

Initially, to calculate the cell to the left, I had: *prev = state[(i - 1) % len].* This problem resulted in a negative integer, leading to improper generation on automatons that approached the edge. This was easily fixed by adding the length of state: *prev = state[(i - 1 + len) % len],* meaning that the prev is not always positive.

---

6. GitHub Repository

https://github.com/rickwgarcia/elementaryCA

---

7. Sources

https://mathworld.wolfram.com/ElementaryCellularAutomaton.html

https://en.wikipedia.org/wiki/Cellular_automaton