An In-depth Analysis of an Elementary Cellular Automaton Implementation

Connor L. Miller, B.S.

2 November 2024

Abstract

Cellular automata (CAs) are mathematical models composed of a grid of cells, each of which evolves through discrete time steps according to a set of rules based on the states of neighboring cells. This paper presents a detailed analysis of an implementation of an elementary cellular automaton in C++. The focus will be on the algorithm's structure, the implications of boundary conditions, and the efficiency of the approach. Space is reserved at the end of the analysis for empirical results, showcasing various generated automata.

1. Introduction

Elementary cellular automata are discrete dynamical systems defined on a one-dimensional array of cells, each capable of being in one of two states (usually represented as `0` and `1`). The evolution of the system occurs through a series of discrete time steps, governed by local rules that determine the state of each cell based on its current state and the states of its neighbors.[1] This research analyzes a C++ implementation of an ECA, highlighting its construction, functionality, and performance metrics. By examining the design choices and computational strategies employed, we aim to provide insights into the broader implications of cellular automata as models of complex systems.

2. Class Definition and Constructor

The implementation of the cellular automaton is encapsulated within the `CellularAutomaton` class. The constructor initializes key attributes essential for the automaton's operation:

```
CellularAutomaton(std::size_t rule, std::size_t steps)
    : rule(rule)
    , steps(steps)
    , size(steps * 2 + 1)
    , currentState(size, 0)
```

Figure 1. Constructor Signature and Initializer List

2.1 Attributes

`std::size_t rule`

An integer that specifies the rule for state transitions, ranging from `0` to `255`, which corresponds to the 256 possible configurations of state transitions for three neighboring cells.

`std::size_t size`

This variable is calculated as `steps * 2 + 1`, providing a symmetric grid around the initial state. The extra two cells account for future states generated in either direction from the center, allowing for a total of `steps` iterations to be processed without boundary constraints.

`std::vector<std::size_t> currentState`

A dynamic array that stores the current state of the automaton. It is initialized with zeros, except for the middle

[1] Weisstein, Eric W. "Elementary Cellular Automaton." From MathWorld--A Wolfram Web Resource.

cell, which is set to `1`, representing the starting condition of the system.

`std::size_t ruleSet[8]`

An array that holds the eight possible transition results for the different combinations of left, center, and right neighbor states. This array is populated by shifting the input rule and using bitwise operations to extract the necessary bits.

2.2 Rule Set Initialization

The constructor populates the `ruleSet` array based on the provided rule. Each entry in this array corresponds to a specific neighborhood configuration:

```
for (std::size_t i = 0; i < 8; ++i)
{
    ruleSet[i] = (rule >> i) & 1;
}
```

Figure 2. Rule Set Initialization Algorithm Implementation

The right shift operator **(>>)** shifts the bits of the `rule` to the right by `i` positions. For instance, if `rule` is `41` (binary: `00101001`), `rule >> 0` results in `00101001`, `rule >> 1` results in `00010100`, and so on.

The bitwise AND operator (`&`) is utilized to isolate the least significant bit of the shifted value. Thus, for `i = 0`, it captures the state of the first neighborhood, and for `i = 1`, it captures the second, and so forth, effectively filling `ruleSet` with binary values representing whether the next state will be "alive" (1) or "dead" (0).

3. State Generation

The `generateNextState` method is responsible for calculating the next configuration of the cellular automaton based on the current state. This function iterates over each cell in the `currentState`, computing the next state for every cell based on the states of its neighbors. The periodic boundary conditions are effectively implemented here to ensure that cells on the edges of the grid wrap around.

```
void generateNextState()
{
    std::vector<std::size_t> newState(size, 0);

    for (std::size_t i = 0; i < size; ++i)
    {
        std::size_t left = currentState[(i == 0) ? size - 1 : i - 1];
        std::size_t center = currentState[i];
        std::size_t right = currentState[(i == size - 1) ? 0 : i + 1];

        std::size_t neighborhood = (left << 2) | (center << 1) | right;

        newState[i] = ruleSet[neighborhood];
    }

    currentState = newState;
}
```

Figure 3. State Evolution Algorithm Implementation

3.1 Neighborhood Calculation with Periodic Boundary Handling

Each cell's state is derived from its neighboring cells. The three neighbors are left, right, and center, allowing for a local interaction that is characteristic of cellular automata.

```
left = currentState[(i == 0) ? size - 1 : i - 1];
center = currentState[i];
right = currentState[(i == size - 1) ? 0 : i + 1];
```

Figure 4. Periodic Boundary Handling Implementation

For the leftmost cell (index `0`), the left neighbor wraps around to the last cell of the array (`size - 1`), while for all other cells, it is simply the previous index (`i - 1`). Similarly, for the rightmost cell (index `size - 1`), the right neighbor wraps around to the first cell (`0`), while for other cells, it is the next index (`i + 1`). The center is directly taken from the `currentState` array.

The choice of periodic boundary wrapping is significant for several reasons. Firstly, it allows the cellular automaton to simulate a toroidal (doughnut-shaped) space, which models situations where interactions extend beyond finite boundaries, akin to infinite systems in physics. By treating edges as continuous, periodic boundary conditions eliminate artificial behaviors that may arise from fixed boundaries, thereby facilitating a more natural evolution of patterns. Additionally, this approach enhances symmetry across the grid, ensuring that each cell interacts uniformly with its neighbors, which contributes to the emergent behavior of the automaton. Furthermore, periodic boundaries facilitate exploratory studies in complex systems and emergent behavior, allowing researchers to

investigate phenomena without the constraints of edge effects that could bias their results.[2]

3.2 Bitwise Neighborhood Encoding

The left shift operator **(<<)** shifts the bits of the `left` and `center` cells to their respective positions in a three-bit binary number, allowing for the creation of a unique representation for each combination of cell states.

```
neighborhood = (left << 2) | (center << 1) | right;
```

Figure 5. Neighborhood Encoding Implementation

For instance:

If `left = 1`, `center = 0`, `right = 1`, the binary representation of `neighborhood` would be `101` (which corresponds to `5` in decimal). This binary encoding allows us to directly index into the `ruleSet` array to determine the next state.

3.3 Rule Application

Once the neighborhood configuration is determined, it serves as an index into the `ruleSet` array to obtain the new state for the current cell:

```
newState[i] = ruleSet[neighborhood];
```

Figure 6. New State Assignment Implementation

This line of code efficiently applies the local transition rules defined by the user, enabling the automaton to evolve based on its previous configuration.

4. Displaying the Current State

The method `displayCurrentState` visualizes the current configuration of the cellular automaton:

```
void displayCurrentState() const
{
    for (std::size_t cell : currentState)
    {
        std::cout << (cell ? "██" : "  ");
    }

    std::cout << '\n';
}
```

Figure 7. Model Visualization Algorithm Implementation

The display employs a simple yet effective method to convey the state of each cell. Cells in the `1` state are represented by a block character ("██"), while cells in the `0` state are represented by two spaces (" "). This binary representation provides a clear visual indication of the evolving patterns in the automaton, allowing users to easily discern changes over iterations.

5. Execution of the Automaton

The `run` method orchestrates the overall operation of the automaton.

```
void run()
{
    for (std::size_t i = 0; i < steps; ++i)
    {
        displayCurrentState();
        generateNextState();
    }
}
```

Figure 8. High-Level Model Evolution Algorithm Implementation

This method loops through the specified number of `steps`, displaying the current state and generating the next state iteratively. It allows observers to see the progression of the cellular automaton in real-time.

[2] B. J. LuValle, "The Effects of Boundary Conditions on Cellular Automata," Complex Systems, 28(1), 2019 pp. 97–124.

6. Main Function and User Interaction

The main function serves as the entry point of the application, providing the interface through which users can interact with the cellular automaton.

```cpp
int main()
{
    std::size_t rule;
    std::size_t steps;

    std::cout << "Enter rule number (0-255): ";
    std::cin >> rule;
    std::cout << "Enter number of steps: ";
    std::cin >> steps;

    CellularAutomaton automaton(rule, steps);
    automaton.run();

    return 0;
}
```

Figure 9. Main Function Implementation

The program begins by prompting the user for a rule number and the number of simulation steps. The user is asked to enter a rule number between 0 and 255, as cellular automata rules are commonly defined within this range. The subsequent prompt requests the number of steps, which dictates how many iterations of the automaton will be executed.

After collecting the inputs, a `CellularAutomaton` object named `automaton` is instantiated using the provided rule and steps. The `run` method is then called on the `automaton` object to start the simulation. The interaction flow is linear and intuitive, guiding the user through the essential steps required to initiate a simulation.

6. Time Complexity

The time complexity of the `generateNextState` function is $O(n)$, where n represents the size of the state vector. This linear time complexity arises from the fact that the algorithm processes each cell in the grid in a single pass to compute the new states based on the values of their neighboring cells. Specifically, the function iterates through the entire `currentState` vector, which contains the states of all the cells. For each cell, it retrieves the values of the left, center, and right neighbors to form a neighborhood representation, which is then used to determine the new state based on the predefined rules stored in the `ruleSet` array.

To elaborate, the computational steps involved in processing each cell include accessing its value and the values of its neighbors, performing bitwise operations to construct a neighborhood value, and assigning a new state to that cell based on the corresponding rule. Given that all these operations are constant time $O(1)$, and since the algorithm must perform these operations for all n cells in the state vector, the overall time complexity remains linear. This efficiency is crucial, particularly for simulations that involve many steps or complex cellular automata, as it ensures that the computational load grows proportionately with the size of the grid rather than exponentially.

7. Space Complexity

The space complexity of the implementation is also $O(n)$ due to the utilization of two vectors: `currentState` and `newState`. Each vector requires storage that is directly proportional to the number of cells in the grid, which is dictated by the number of steps specified by the user, as the grid size is calculated as:

$$size = steps \times 2 + 1$$

In more detail, the `currentState` vector stores the current configuration of the cellular automaton, representing the state of each cell at any given moment. The `newState` vector is used to store the updated state of each cell after the computation for the next generation is complete. The need for both vectors simultaneously is what contributes to the linear space complexity.

8. Conclusion

The implementation of an elementary cellular automaton in C++ presented herein serves as a robust framework for exploring the fundamental principles underlying cellular automata. The model showcases how computational efficiency can be achieved while maintaining clarity in code structure by employing efficient data structures, such as vectors for dynamic state representation and arrays for rule sets. The choice of periodic boundary conditions enhances the model's realism, allowing for a continuous

simulation that mimics natural phenomena without the artificial constraints imposed by fixed boundaries.

9. References

Weisstein, Eric W. "Elementary Cellular Automaton." From MathWorld--A Wolfram Web Resource. https://mathworld.wolfram.com/ElementaryCellularAutomaton.html

B. J. LuValle, “The Effects of Boundary Conditions on Cellular Automata,” *Complex Systems*, **28**(1), 2019 pp. 97–124.

10. Generated Automata

The following images are elementary cellular automatons generated using the model presented above, demonstrating its ability to generate cellular automatons across the entire range of rules accurately.

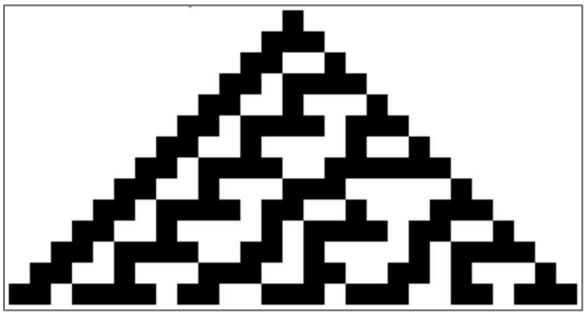

Figure 10. Rule 30 Generation

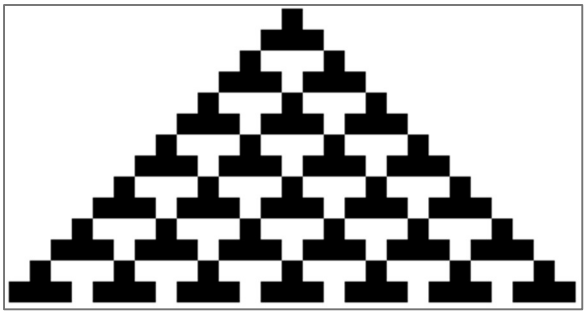

Figure 11. Rule 54 Generation

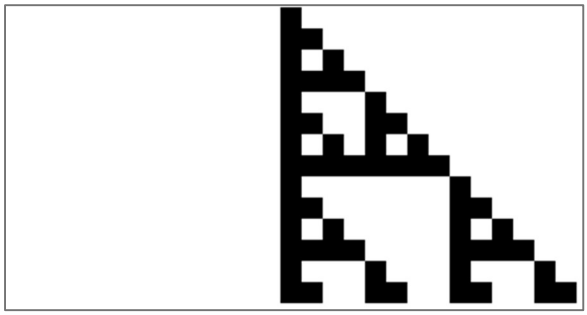

Figure 12. Rule 60 Generation

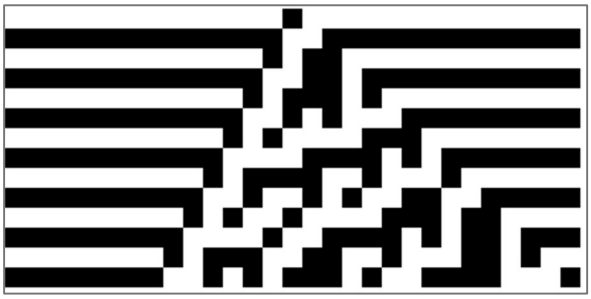

Figure 13. Rule 75 Generation

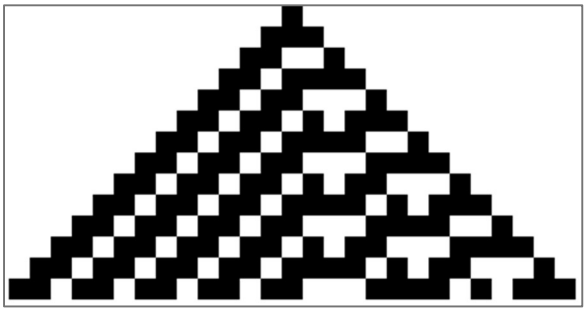

Figure 14. Rule 62 Generation

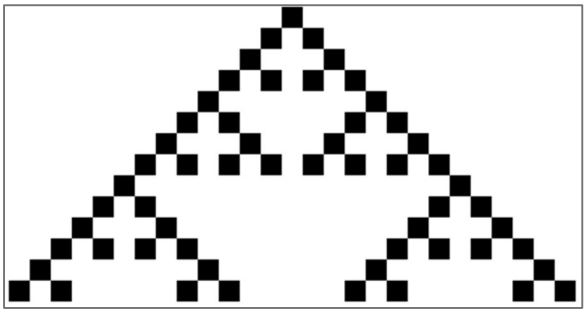

Figure 15. Rule 90 Generation

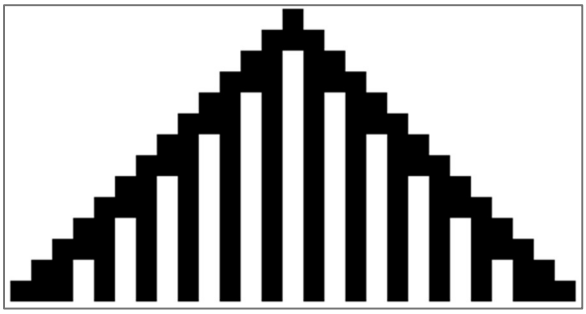

Figure 16. Rule 94 Generation

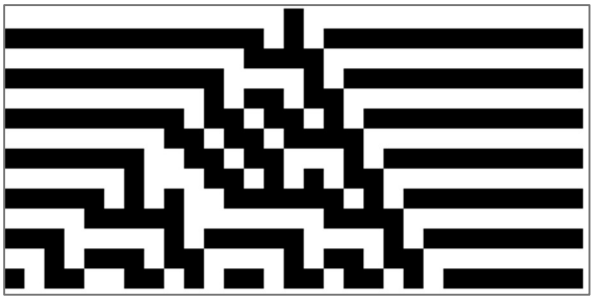

Figure 17. Rule 101 Generation

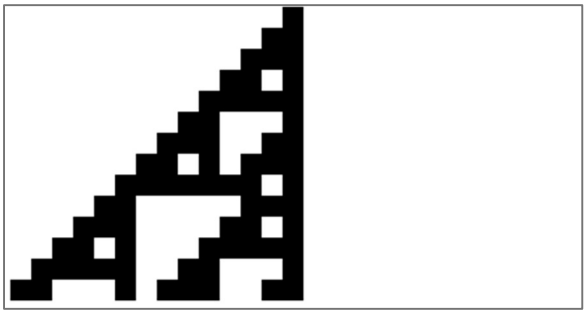

Figure 18. Rule 110 Generation

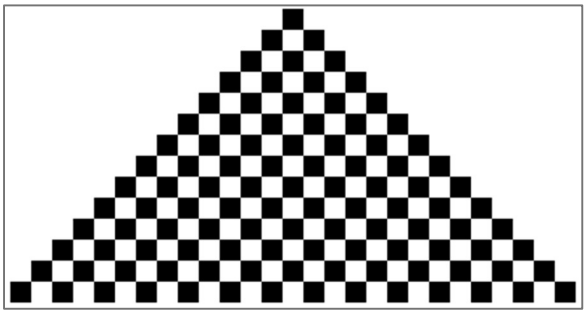

Figure 19. Rule 122 Generation

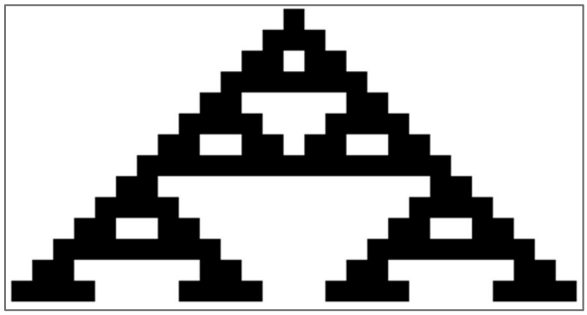

Figure 20. Rule 126 Generation

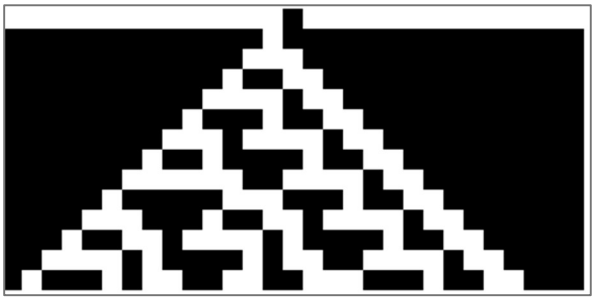

Figure 21. Rule 149 Generation

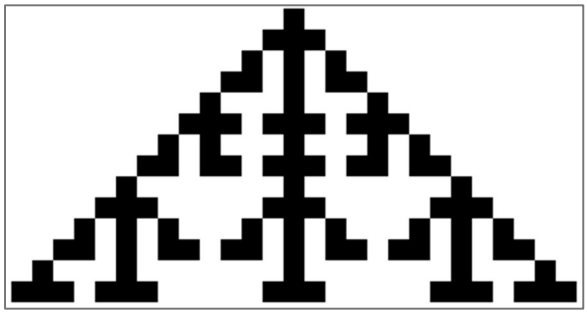

Figure 22. Rule 150 Generation

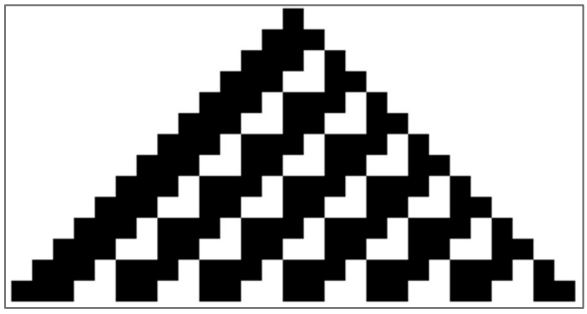

Figure 23. Rule 158 Generation

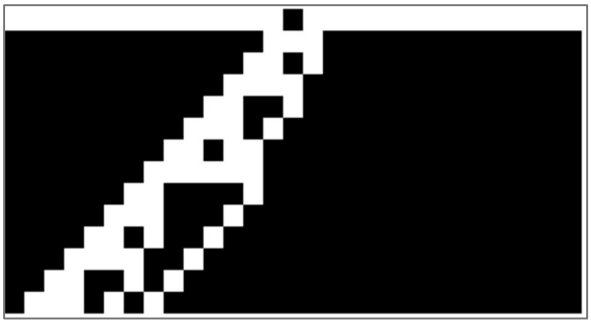

Figure 24. Rule 169 Generation

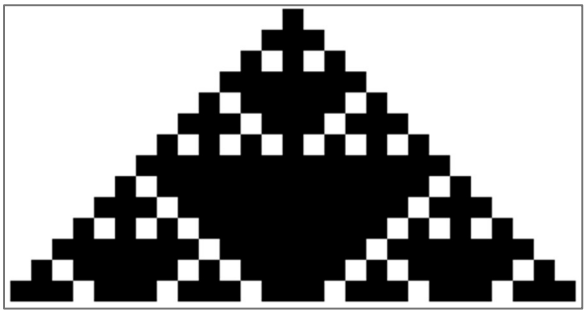

Figure 25. Rule 182 Generation

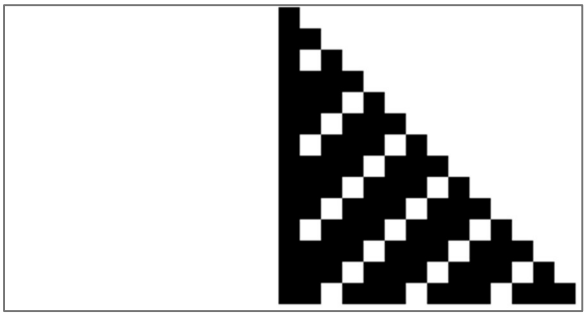

Figure 26. Rule 188 Generation

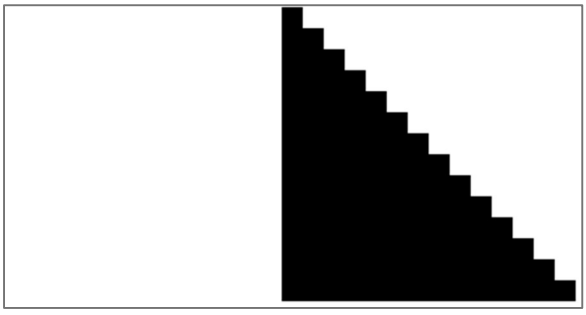

Figure 27. Rule 220 Generation

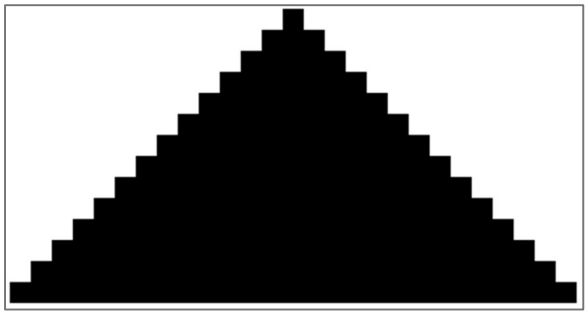

Figure 28. Rule 222 Generation

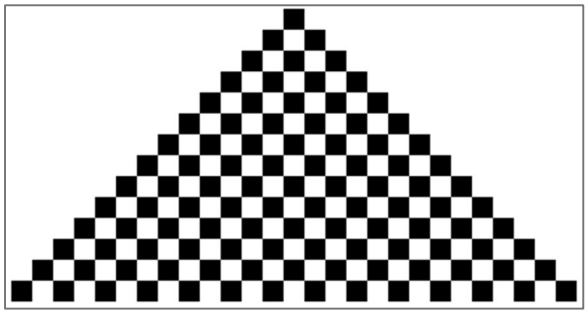

Figure 29. Rule 250 Generation