CHEM274B Final Project Write-Up

Group Members: Brandon Robello, Radhika Sahai, Curtis Wu

Sections

Introduction

A general purpose library is a collection of code that is already written that can be used to implement more specific functions. For this project, a general purpose library for a cellular automata was developed by us, which is a grid-like structure which each cell being a cellular automaton, each in one of a finite number of states, and evolving through discrete time steps based on a set of predefined rules.

For example, the ***Majority Rule*** aims to determine the prevailing state within a cells neighborhood. It examines the frequency of each state among neighboring cells and selects the state with the highest occurrence. This rule is particularly useful for scenarios where a cell tends to adopt the most common state in its surroundings, reflecting a consensus or majority influence. The ***Straight Conditional Rule*** facilitates state transitions based on a predefined order of states. A cell, represented by the center of its neighborhood, undergoes a transition to the next state in a specified sequence. This rule is applicable in situations where state changes follow a clear and predetermined pattern, providing a straightforward mechanism for sequential transitions. The Neighbor Conditional Rule introduces a conditional state transition mechanism based on the state of neighboring cells. It involves specifying a trigger state for the center cell and a target state for its neighbors. If the center cell is in the trigger state and encounters a neighbor in the specified target state, a transition to a new state occurs. This rule models scenarios where a cells state change depends on the presence of specific neighboring states. More specific details about the rules will be discussed in the General Purpose Cellular Automata.

A specific application was also developed in this project which was built on top of this general software library that models the spread of COVID-19. Which will use other application-specific rules, such as the `InfectionRule` which determines the likely of a person getting infected according the number of adjacent infected person, or `recoveryRule` which models the recovery rate of an infected individual (cell). More details will be discussed in theCOVID Modeling section.

General Purpose Cellular Automata

The general software library involves the representation of a grid using a Cell class, where each cell encapsulates its state and position information. The purpose is to simulate a Cellular Automaton, and the evolution of this automaton is governed by a set of rules. The description outlines the process of updating the grid states in a sequential manner.

```
class Cell {
    private:
        int x; // The x position on the grid
        int y; // The x position on the grid
        int state_t0; // Intitial state of the cell
        int state_tx = 0; // Updated state of the cell
    public:
        Cell();
        ~Cell();
        // x and y positions setters and getters
        void set_x(int x);
        void set_y(int y);
        int get_x() const;
        int get_y() const;

        int getState_t0() const; // Method to get current state
        int setState_t0(int state); // Methof to set current state
        int getState_tx() const; // Method to get new state
        int setState_tx(int state); // Methof to set new state

        vector<int> getPosition() const; // Method to get x and y positions

        void cell_update(); // Method to swap state_tx and state_to,
                               to make the new value (tx) the current value (t0)

        bool operator<(const Cell& other) const; // Operator for convenient map storing
};
```

The general purpose cellular automata the framework is defined by the CellularAutomata class, with partial header file code shown below. This class enables the simulation of dynamic systems through a grid of cells, each with defined states, features, and interactions. The primary functionalities of the code include the setup of CA dimensions, specification of neighborhood and boundary types, initialization of cell states at time t_0, and the implementation of a rule-based update mechanism.

The generation of neighborhoods is handled by the get_neighborhood function, which retrieves the neighborhood of a specific cell based on the defined neighborhood type and boundary conditions. This function is crucial for the application of rules and the evolution of the automaton over time.

Recording and output functionalities are implemented through the record_CAframe function, which records the current frame of the Cellular Automaton to a file, including dimensions and cell states. The print function outputs the current state of the Cellular Automaton to the console. The swapState function is responsible for swapping states from state_tx to state_t0 after recording for each new update, ensuring the integrity of the simulation.

```
class CellularAutomata
{
    private:

        vector<vector<Cell>> grid;    // 3D grid structure that stores cell data type
        std::map<string, int> states; // Map of the possible state values
```

```
        int neighbor_type;      // 1 -> Von Neumann neighborhood; 2 -> Moore neighborhood
        int bound_type;         // 3 -> Static; 4 -> Fixed; 5 -> Periodic
        int dim1;               // Number of rows for CA
        int dim2;               // Number of columns for CA
        int message_radius;     // Message passing radius
        Cell boundary_cell;     // Place holder cell that act as the boundary for static boundary type

    public:
        CellularAutomata();
        ~CellularAutomata();

        int setup_dimension(int ndims, int dim1, int dim2);
        int setup_neighborhood(int neigh_type);
        int set_boundtype(int bound_type, int radius);

        int set_states(std::map<string, int> states);
        std::map<string, int> list_states() const;

        int init_CA_state(int stat_t0);
        int init_CA_stateWprob(int stat_t0, double probability);

        int init_Cell_state(int stat_t0, vector<vector<int> > coords);
        int init_Cell_stateWprob(int stat_t0, double probability,  vector<vector<int> > coords);

        Neighborhood get_neighborhood(vector<int> coord);

        // Query the current cell states of the CA
        std::map<int, int> query_cellState();
        int record_cellState(string filepath);

        int record_CAframe(string filepath) const;
        //int record_CellState(string filepath) const;

        // Method to swap staes from tx to after recording for each new update
        int swapState();
        // Function that drive the timestep updates using a vector of rules
        int update(vector<Rule*>& rules);
};

#define VONNEUMANN  1
#define MOORE       2
#define STATIC      3
#define FIXED       4
#define PERIODIC    5
#define MAJORITY    6
#define PARITY      7
```

For our CA, there are several key functionalities that collectively define the structure, initialization, evolution, and output of the automaton. The grid setup and initialization are encapsulated in the setup_dimension function, which configures the grid based on specified dimensions. Additionally, the setup_neighborhood and set_boundtype functions allow for the definition of neighborhood types (Von Neumann or Moore) and boundary conditions (Static or Periodic).

Cell states and their initialization are addressed by functions like set_states, facilitating the definition of possible discrete states for grid cells. The init_CA_state function initializes all cells to a specified state, while init_CA_stateWprob allows for initialization with a certain probability. init_Cell_state and init_Cell_stateWprob enable the initialization of specific cells, either deterministically or probabilistically. The rules are inherited from a virtual parent class `Rule`, which allows child functions to define specific behaviours and attributes belonging to that defined class.

```
class Rule {
public:
    virtual void apply(Neighborhood &neighborhood);
    virtual ~Rule() {}
};
```

The ***Majority Rule*** and ***Straight Conditional Rule*** has been described in the introduction section. The ***Activation-Inhibition Rule*** implements a more complex mechanism where a cells state is influenced by both activation and inhibition factors. The rule considers the states of neighboring cells and their distances to calculate weighted sums for activation and inhibition. The cell transitions to an active state if activation surpasses a threshold and inhibition remains below another threshold. This rule is suitable for systems where cells respond to both positive and negative influences from their neighbors. The Parity Rule focuses on state transitions determined by the parity (even or odd) of the count of neighboring cells in a specific target state. The center cell transitions to one state if the count is even and another state if the count is odd. This rule is effective in scenarios where the collective evenness or oddness of neighboring states plays a role in determining the state of the center cell, adding a layer of complexity to the cellular automaton dynamics.

The generation of the `neighborhood` class is handled by the get_neighborhood function, which retrieves the neighborhood of a specific cell based on the defined neighborhood type and boundary conditions. This function is crucial for the application of rules and the evolution of the automaton over time.

```
struct Neighborhood
{
    public:
    map<Cell, int> subgrid; // Store neighboring cells and their distances
    int dim;               // Number of cells in the subgrid
    Cell center_cell;       // Copy of the center cell of the neighborhood
};
```

The core evolution of the Cellular Automaton is orchestrated by the update function, which drives timestep updates using a vector of rules. Each rule is applied to every cell in the grid, leading to the progression of the automatons state.

The state of each cell is determined by the states of its neighboring cells, typically within a specified neighborhood using the `update` function. The evolution of the entire automaton is characterized by the repeated application of these rules to update the states of all cells simultaneously. Cellular automata exhibit dynamic and complex behavior emerging from simple local interactions, making them valuable tools for studying self-organization, pattern formation, and computational processes. Popularized by mathematician John Conways Game of Life, cellular automata find applications in various fields, including computer science, physics, biology, and artificial life research.

```
int CellularAutomata::update(vector<Rule*>& rules) {
   for (Rule* rule : rules) {
       for (int i = 0; i < dim1; i++) {
```

```
            for (int j = 0; j < dim2; j++) {

            Neighborhood neighborhood = get_neighborhood({i, j});
            int newState = rule->apply(neighborhood);
            grid[i][j].setState_tx(newState);
            }
        }
    }
    swapState();
    return 0;
}
```

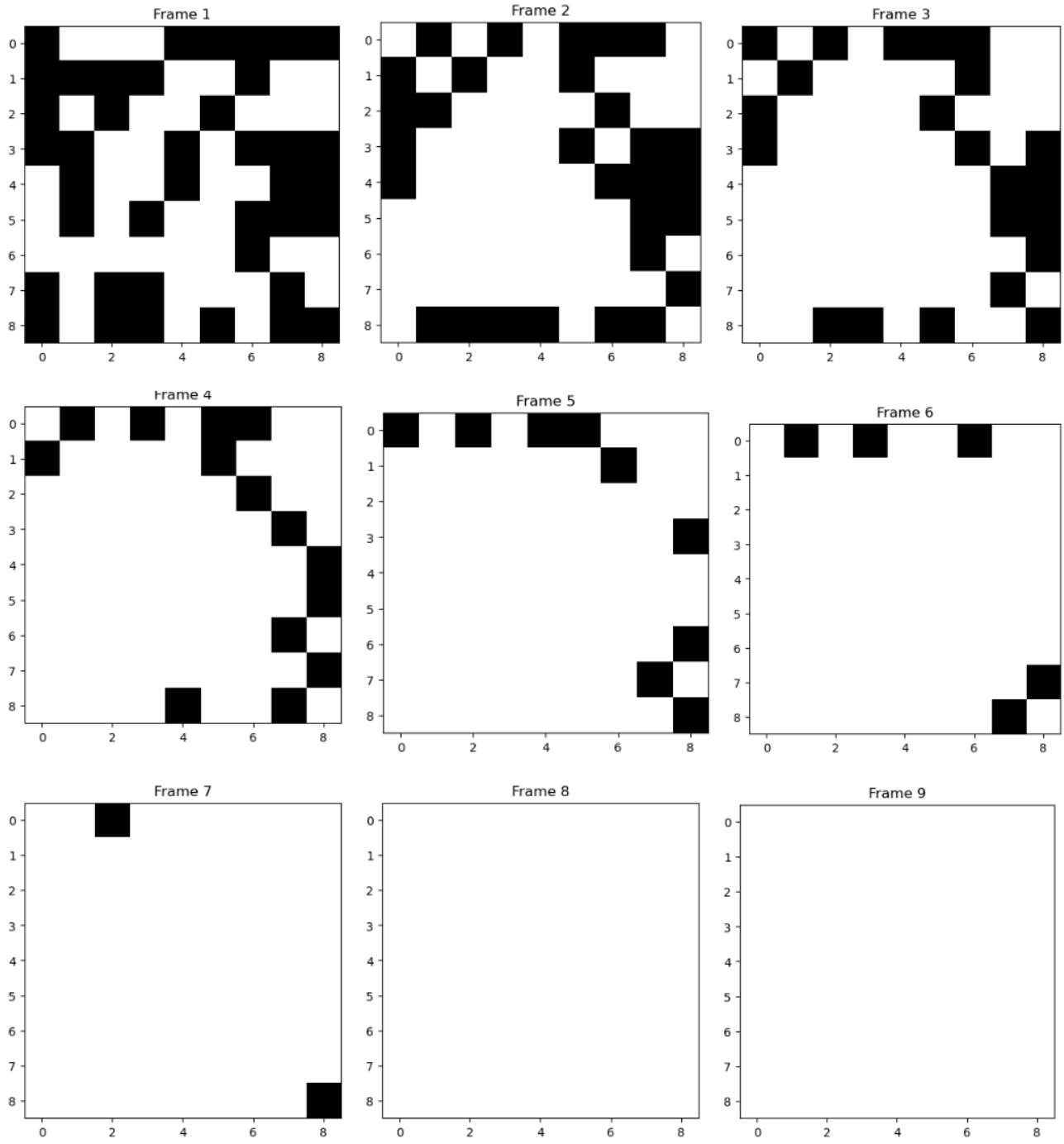
Finally, querying and recording cell states are facilitated by the query_cellState function, which retrieves the frequency of each state in the current cell configuration. The record_cellState function complements this by recording the frequency of each state in a CSV file, capturing the dynamic evolution of the Cellular Automaton over time. Together, these functionalities provide a comprehensive framework for simulating, analyzing, and visualizing complex cellular automaton dynamics.

The overall algorithmic complexity of these methods is influenced by the size of the grid and the number of iterations performed. The update method, in particular, may have a complexity of O(dim1 * dim2 * num_rules) based on the nested loops iterating over the grid and rule set. The complexity of other methods generally depends on the specific operations they perform, such as initializing states or recording frames.

In summary, the process involves a systematic iteration through the grid, applying specific rules to update the states of individual cells, and then swapping the current and updated states to progress the simulation to the next step. This approach reflects the typical dynamics of a Cellular Automaton, where the state of each cell is influenced by its neighbors and evolves over discrete time steps.

The visualization of the general CA shows each of the frames (there is also a gif in the directory Utils/Plots to show the change over time)

From the visualization of the general purpose CA it is seen that the on state is black and the off state and each grid represents one frame of the CA. From this visualization we can see that over time all the states eventually become 0.



COVID-19 Simulation Model

Cellular Automata (CA) can simulate the spread of infectious diseases like COVID-19 by representing individuals or groups in a grid, with each cell signifying a person and states indicating their health status. To set up the model, individuals are initialized with random states, including some initially infected individuals. The grids neighborhood and boundary types are defined to represent the range of interactions between individuals and the conditions of the population (closed or open). Rules governing the dynamics of the epidemic are established. These rules include the spread of infection from infected to susceptible individuals, recovery and immunity mechanisms, and the simulation of preventive measures like vaccination. The simulation evolves over time through iterative updates to the CA grid, representing the progression of the disease. The states of the grid are recorded at each time step, allowing for the analysis of the temporal evolution of the epidemic.

```
#define EMPTY           0
#define HEALTHY         1
#define HEALTHY_VACCED  2
#define INFECTED        3
#define RECOVERED       4
```

CellularAutomata Class defines a cellular automaton with methods for setting up dimensions, neighborhoods, boundary types, states, and initializing cell states. It includes methods for updating the CA based on a set of rules, querying and recording cell states, and printing the current state of the CA. The program defines several rules that govern the behavior of cells in the CA.

`VaccinationRule`: Represents the rule for vaccinating healthy cells based on a specified probability.

`InfectionRule`: Models the infection process, considering the infection rate and immunity multiplier for vaccinated cells.

`ReinfectionRule`: Deals with the possibility of reinfection for recovered cells, considering the reinfection rate and recovery immunity multiplier.

`RecoveryRule`: Handles the recovery process based on a recovery rate.

The implementation of `InfectionRule` has been shown below, which could utilize custom `infection_rate_` and `immunity_multiplier_` values to achieve generalizability and flexibility of the simulation.

```
class InfectionRule: public Rule {
    private:
        double infecion_rate_;
        double immunity_multiplier_;
    public:
        InfectionRule(double infecion_rate, double immunity_multiplier):infecion_rate_(infecion_rate), immunity_multiplier_(immunity_multiplier) {}
        int apply(Neighborhood &neighborhood) override{

            int current_state = neighborhood.center_cell.getState_t0();
            int current_state_tx = neighborhood.center_cell.getState_tx();
            // Skip for empty, infected, and recovered cells
            if (current_state == EMPTY || current_state == INFECTED || current_state == RECOVERED) {
                return current_state_tx;
            }
            int infected_count = 0;
            for (const auto &cellEntry : neighborhood.subgrid) {

                const Cell &cell = cellEntry.first; // Gets the cell

                int cell_state = cell.getState_t0();// Gets the state of the cell

                if (cell_state == INFECTED) {
                    infected_count++;
                }
            }
            // Probability
            //srand(static_cast<unsigned int>(time(0)));
            double randomValue = static_cast<double>(rand()) / RAND_MAX;

            if (current_state == HEALTHY){
                if (randomValue <= infected_count*infecion_rate_) {
                    return INFECTED;
                }
                else {
                    return current_state_tx;
                }
            }
            // Additional immunity multiplier for vaccinated cells
            if (current_state == HEALTHY_VACCED){
                if (randomValue <= infected_count*infecion_rate_*immunity_multiplier_) {
                    return INFECTED;
                }
                else {
                    return current_state_tx;
                }
            }
            return current_state_tx;
        }
};
```

The main function initializes the simulation, sets up the CA, defines rules, and runs the simulation for a specified number of steps. It records the state of the CA at each step, prints the current state, queries cell states, and records statistics into output files (`Test_COVID_CAframe_output.txt` and `Test_COVID_CAcellState_output.txt`).

The CA is initialized with a 9x9 grid, Moore neighborhood, and static boundaries. The initial states include empty space, healthy individuals, and a small percentage of infected individuals. Rules are applied in a specific order during each simulation step.

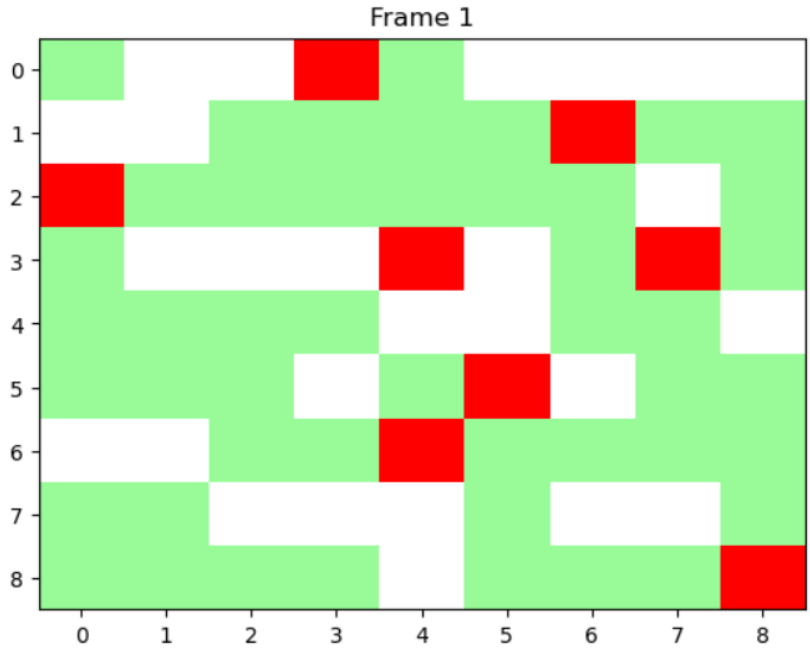
These rules collectively define the dynamics of the CA, simulating aspects such as infection, recovery, vaccination, and reinfection.Choices made in each rule, such as the rates and probabilities, affect the overall behavior of the CA. For example, a higher recovery rate or vaccination probability would lead to faster recovery or higher vaccination coverage, respectively.The interaction between rules influences the emergence and spread of infections, the effectiveness of vaccination campaigns, and the potential for reinfection in the population.

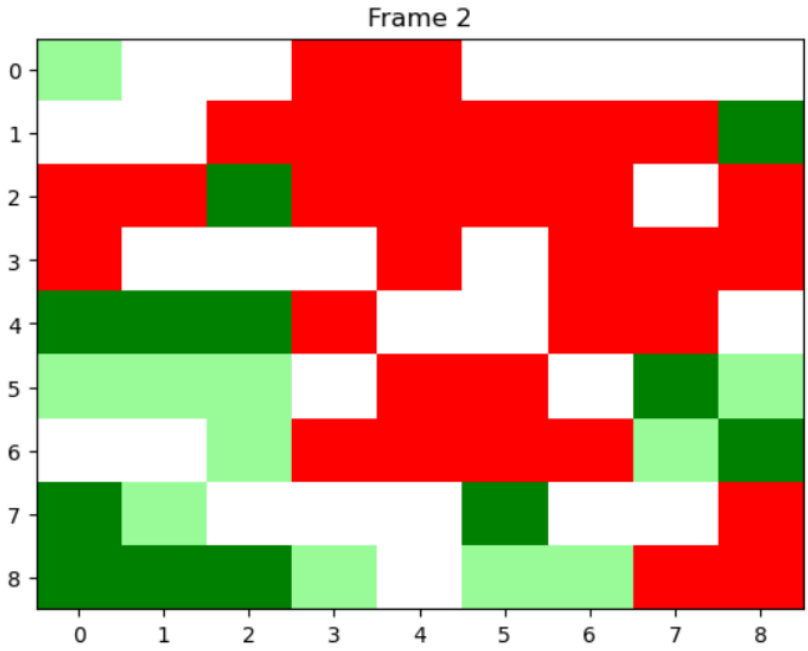
The General Purpose Cellular Automata allows the rules to be applied in different contexts and scenarios, providing flexibility for simulating various infectious

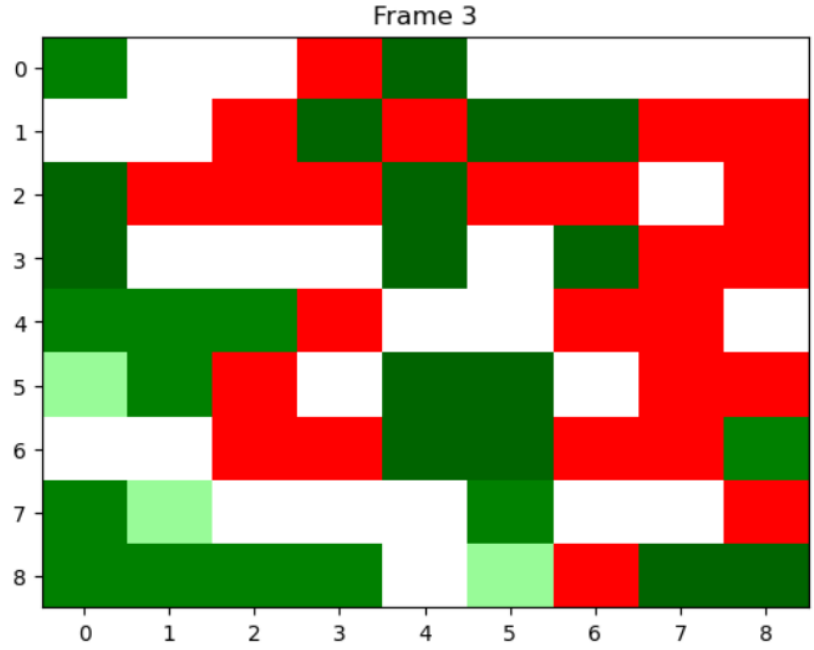
disease scenarios. The choices made in these rules may need to be adjusted based on the specific characteristics of the modeled system and the desired simulation outcomes. In summary, the choices in the general-purpose cellular automaton rules determine how infection, recovery, vaccination, and reinfection processes are modeled. These choices collectively shape the dynamics of the simulated population in response to infectious diseases.

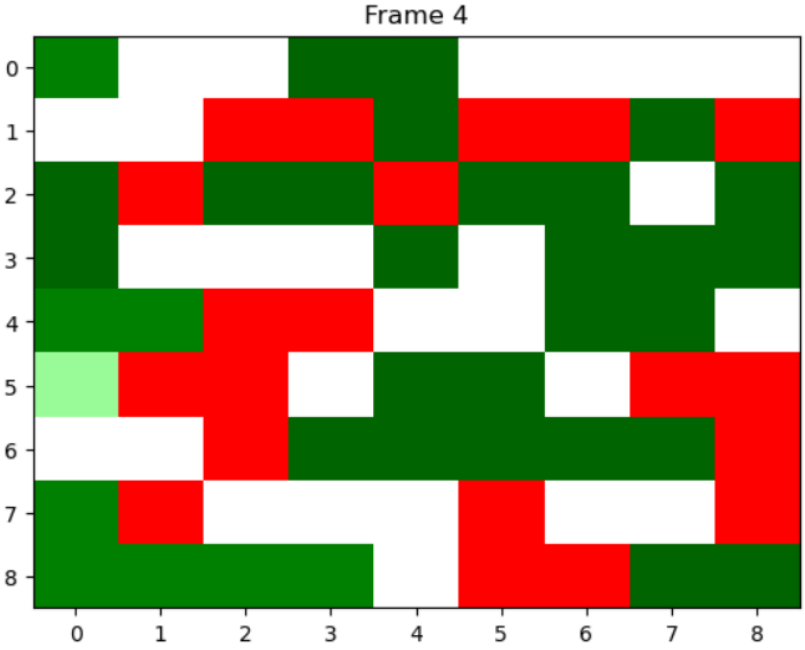
The visualization of the COVID simulation it shows each of the frames (there is also a gif in the directory Utils/Plots to show the change over time)

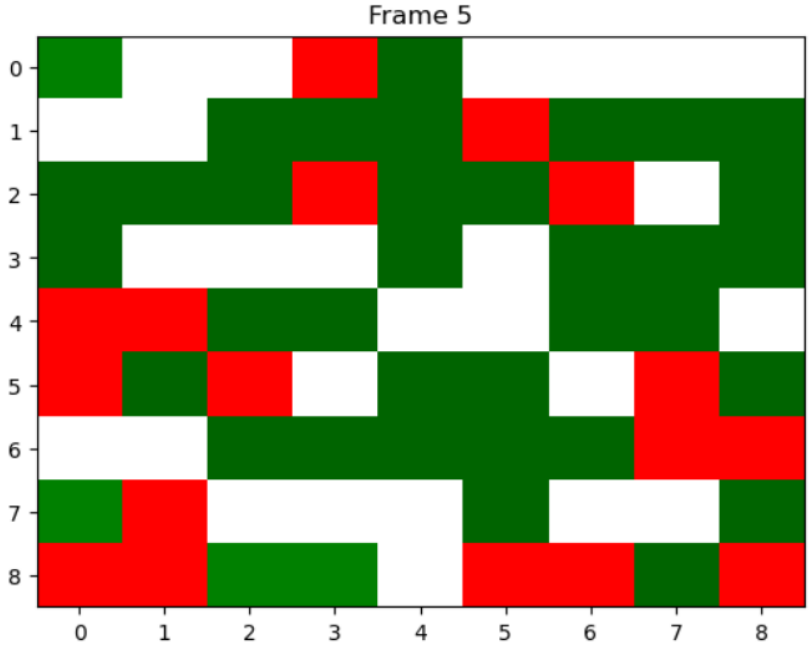
Input Number	State	Color
0	Empty	White
1	Healthy	Pale Green
2	Healthy + Vaccinated	Green
3	Infected	Red
4	Recovered	Dark Green

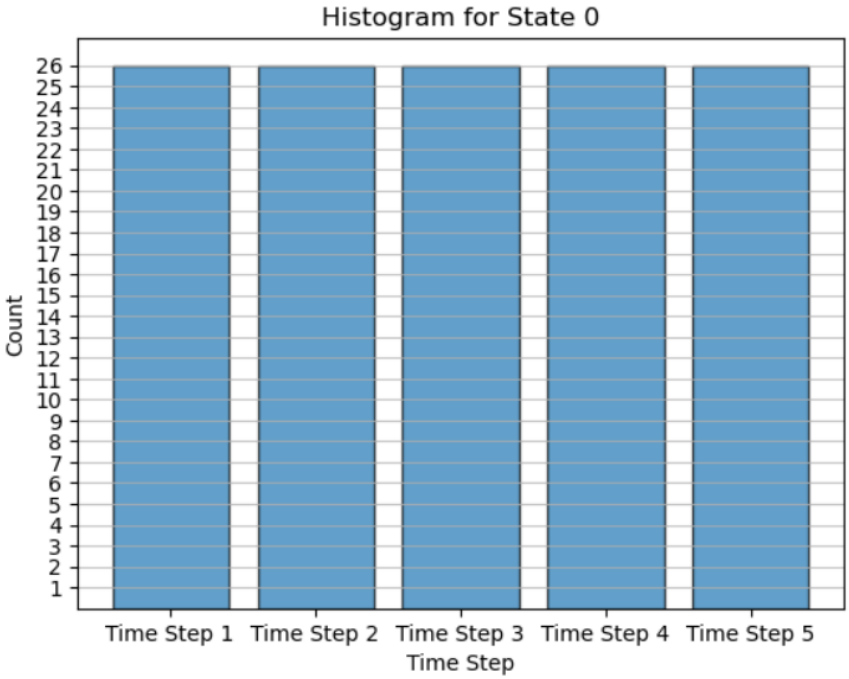


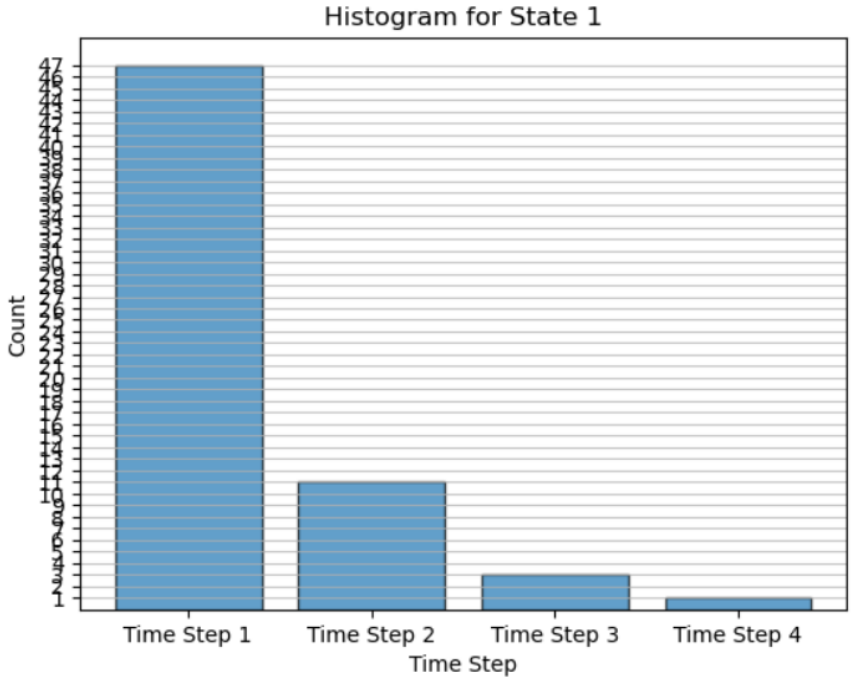


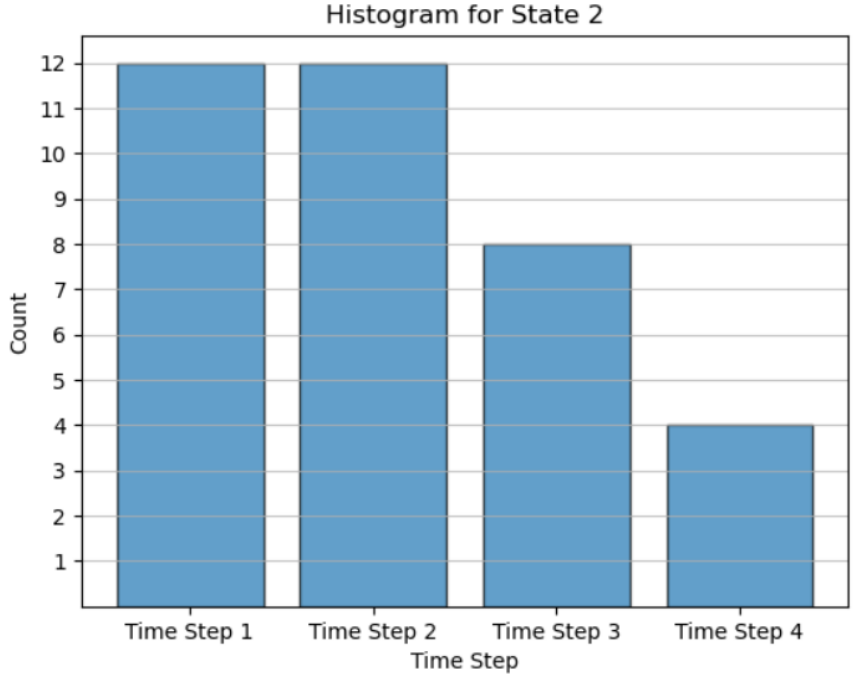

Frame 3
0
1
2
3
4
5
6
7
8
0 1 2 3 4 5 6 7 8

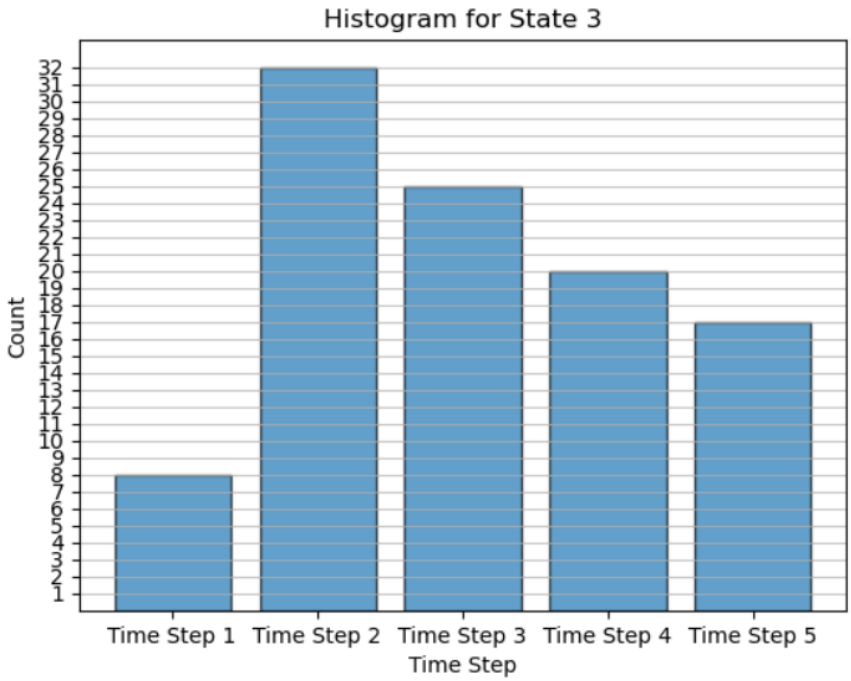

Frame 4
0
1
2
3
4
5
6
7
8
0 1 2 3 4 5 6 7 8

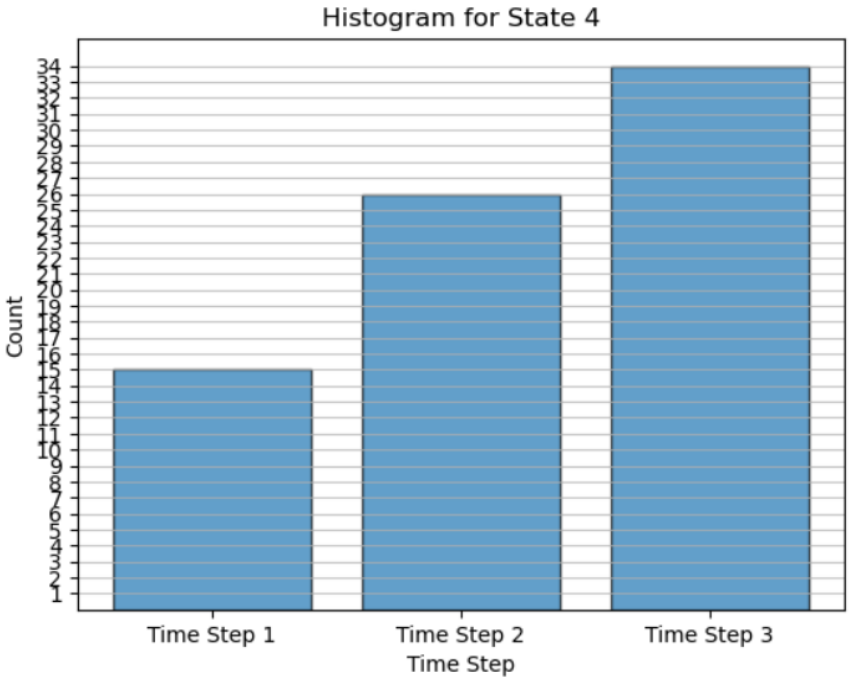

Frame 5
0
1
2
3
4
5
6
7
8
0 1 2 3 4 5 6 7 8


Histogram for State 0
Count
Time Step 1
Time Step 2
Time Step 3
Time Step 4
Time Step 5
Time Step


Histogram for State 1
Count
Time Step 1
Time Step 2
Time Step 3
Time Step 4
Time Step


Histogram for State 2
Count
Time Step 1
Time Step 2
Time Step 3
Time Step 4
Time Step





These images show the cumulative number of each state in the simulation over each time step. This is a function from the general CA applied to find each state of the cells

In summary, employing a Cellular Automata framework for COVID-19 modeling allows researchers to gain insights into the potential impact of interventions, understand the spatial and temporal dynamics of the epidemic, and simulate various scenarios to inform public health strategies.