Master of Molecular Science and Software Engineering

Chem 274 B - Software Engineering Fundamentals for Molecular Sciences

Final Project - Design and Implementation of A General Purpose Library for Basic Cellular Automata Modeling.

DUE Date: Friday, December 15, 2023 by 11:30 PM U.S. Pacific Standard Time

This a Software Development Team Assignment with individual student software engineering reflections and repository submission.

Contributors: Austin Glover, Destinee Maldonado, Sam Wollenburg

Introduction to a Cellular Automata:

Cellular Automata (CA) represent a discrete model of computation, consisting of a regular grid of cells, each embodying an agent that can exist in one of several states. These cells interact with their neighboring cells, forming the basis for the study of nonlinear dynamics in various fields like physics, theoretical biology, engineering, and computational sciences. In these systems, migration dynamics occur when a cell moves around the computational grid, while influence dynamics happen when a cell remains static but changes its state. The behavior of a cellular automaton is captured in the sequence of transformation states of its cells. This is governed by a function that links each cell's current state to its neighbors and the time step, expressed as:

$$x_t = F(x_{t-1}, t).$$

Here, x_t denotes a variable with a finite number of states housed in a particle cell at time t. This relationship allows for the generation of a time series which captures the evolution of variables within the model.

$$\{x_0, x_1, x_2, \ldots, x_t\},$$

Cellular automata undergo state changes based on inputs and previous states, noted as $S_t(x)$, which are finite and discrete. The system updates synchronously, with each update leading to a new configuration, or $S_{t+1}(x)$, representing a specific arrangement of states within a neighborhood.

$$S_{t+1}(x) = F[\overbrace{S_t(x+x_0), S_t(x+x_1), \ldots, S_t(x+x_{n-1})}^{\textit{neighborhood}}],$$

The dynamics of cellular automata are significantly influenced by the definition of neighborhoods, which are created by various boundary conditions such as Moore or Von Neumann neighborhoods. The CA can operate under different types of boundaries, including no boundaries (infinite space), periodic, fixed or walled, and cut-off boundaries where there are no neighbors at the ends.

To model a Cellular Automaton, the process starts with setting up an $n\ x\ m$ grid, with constraints on the grid size ($1 \leq\ n,\ m \leq\ MAX\_SIZE$). CA can function in either sequential or parallel modes. Essential steps in functionality include specifying boundary conditions, states for the variables, initial configurations, and rules. The initial configuration involves setting probabilities for a cell to enter a specific state from an empty state, with some states being established at initiation and others emerging through the application of CA rules. These rules are formulated through transition state functions, like the Majority rule, where a cell's state is influenced by its neighboring cells.

Another key functionality of a CA model includes time steps and updates. Compute functions update the CA based on predefined rules or user-provided transition functions. After every time step, users receive an updated CA configuration, which allows them to analyze the behavior of the system.. Ultimately, the model enables the extraction and plotting of analytical information, showing the current configuration and how its evolved over time with every nth timestep.

In a collaborative environment, building a cellular automaton involves distinct roles: the first user develops and tests the CA software library, the second writes a model using the CA framework, and the third runs the model crafted by the second user. This collaborative approach facilitates a comprehensive exploration and understanding of complex dynamical systems through the versatile and powerful framework of Cellular Automata.

We have organized the software development in repositories that follow the general structure illustrated in Figure 1.

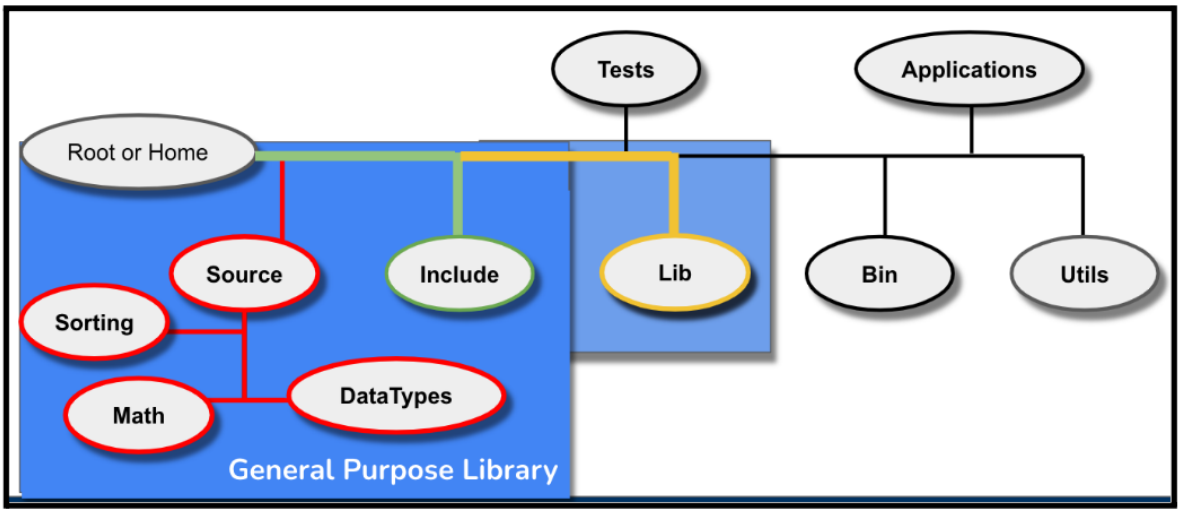


Figure 1. Core software development space. General purpose functionality is built in this area of the software repository. The core development team designs and implements the core functionality, this team has access to all source files and provides Application Programming Interfaces (APIs) for third party users to use their software. They also provide means for installing the core software functionality in the third party users' computational platforms (e.g. makefiles)

Part I. Problem 1.

Design and implement a C++ General Purpose library to support Cellular Automata modeling:

This project is focused on developing a General Purpose library for a Cellular Automata (CA), followed by the creation of a COVID-19 simulation model utilizing this CA framework. The core of the CA general purpose library interface is defined in the Cellular_Automata.h file in the include directory, and the functionality is defined in the Cellular_Automata.tpp file, located within the Source directory of this repository. This class is designed with templating to offer flexibility in both the data type, the dimensions of the cellular automata grid, and the update rule, and the boundary condition rule. Thereby accommodating the various user requirements in the given guidelines. The application of this library in a real world scenario is demonstrated through the COVID-19 simulation model, the code for which can be found in the COVID-19_Simulation.cpp file within the Applications directory. This structure ensures a clear separation between the general-purpose CA library and its specific application in modeling COVID-19 dynamics. A more in depth analysis of each component related to the project guidelines is discussed below.

1. **Setup and Initialization:**

1.1. A Cellular Automata has a computational grid of dimensions: The Cellular_Automata.h code for the General Purpose library uses template parameters '$size\_t \;\; n\_rows$', and '$size\_t \;\; n\_cols$' to and the size of

"intial state" to define the dimensions of tha CA grid, this templated approach allows flexibility in the dimensions setup.

1.2. Your CA library is designed and implemented without explicit parallelization:

The implementation of the CA General Purpose library does not include explicit parallelization constructs and instead relies on C++ features and data structures such as $std::vector$ and $std:function$.

1.3. The CA library needs to manage the following Boundary Conditions and types of neighborhoods:

A requirement of the cellular automata (CA) library is its capability to manage cells located at the grid's edges, where neighboring cells might fall outside the predefined space. To address this, our code incorporates template parameters and specialized functions designed to handle various boundary conditions, including periodic, fixed (or walled), and No/Cut off boundary scenarios. These functions are integrated into the CA constructor, allowing them to be specified for each instance of an automaton. This ensures that edge cells are appropriately updated according to the selected boundary condition. This design not only facilitates easy customization of boundary behaviors but also simplifies the extension of the behaviors in the library. This behavior is accomplished under-the-hood by created a larger hidden state, with boundary cells, and separately calculating the state for these cells after the user-facing cells are processed.

1.3.1. Required neighborhoods: Moore and Von Neuman neighborhoods:

The code is designed to support both Moore and Von Neumann neighborhoods which is achieved through the std::functions which transform a 3x3 grid into a std::vector of neighbors, so that code can be written to act on a std::vector and be agnostic to how that vector is created. This is demonstrated in the "conditional transition rule" implementation.

1.3.2. Required Boundary types:

1.3.2.1. Periodic Boundaries:

'$PeriodicBoundary()$' wraps the space around each spatial axis, connecting border cells with cells at the opposite end. This is accomplished by updating the boundary space in the hidden layer with the cells that "would be adjacent" if the space was truly periodic, that way when the update rule is applied, each cell is provided with an accurate neighborhood.

1.3.2.2. Fixed or Walled boundary conditions:

'$FixedBoundary()$' initializes boundary cells with a state that remains unchanged during iteration, practically this is achieved by always copying the specified state to the boundary condition before the next update is calculated.

1.3.2.3. No Boundaries:

There are two practical ways to create a "No Boundary Condition." The simplest option is possible to use an update rule which only considers its own state, this effectively eliminates the requirement for the hidden state to be any larger than the user state, and truly a "no boundary condition" is achieved. The second way is necessary

if you want to consider a neighborhood in the update rule and also want "no boundary later." Essentially you must create a 'sentinel value boundary' or '$FixedBoundary()$' . By utilizing this concept, the user can effectively establish a boundary that behaves as though the cells at the edges are bordering an empty space.

1.3.2.4. Custom Boundaries:

The design of the code is structured to support the implementation of customizable boundary conditions since the neighborhood calculation and boundary handling are modular and customizable. The user could define time dependent or gradient boundaries or boundaries with specific conditions.

1.3.3. Variable Radius of a Neighborhood:

The '$get\_neighborhood\_around$' method is specifically designed to extract neighborhoods based on a specified radius. The handling of a neighborhood with a radius of 1 is facilitated through the use of templated parameters, n_rows and n_cols, which in turn define the $\_rpad$ and $\_cpad$ variables. These variables are instrumental in defining the size of the neighborhood surrounding each cell. In the context of a radius of 1, the method considers the immediate neighbors in all directions—up, down, left, right, and diagonally. This configuration is essential for cellular automata models, ensuring that the immediate vicinity of each cell is taken into account.

1.4. Your general purpose library must handle different discrete states for the cells in the CA:

The library supports different discrete state cells in the CA. Each state can be represented by a unique int ID as the class template 'T' can be an integer type. This enables users to specify how many different states each cell in the model can be.

1.5. Your general purpose library must enable users to set up an initial configuration for the CA model:

The constructor of the '$Cellular\_Automata$' class allows users to set up an initial configuration for the CA model. The initial state of the CA is passed as a parameter to the constructor, allow for the specification of the CA's initial configuration at time $t = 0$.

1.6. Your general purpose library must enable users to specify which rules are applied in the CA transformations from one configuration (at time t) to another (at time t+1):

1.6.1. Straight conditional transition rule:

'$StraightConditionalRule$' is implemented to allow a cell's state to change based on its current state. It takes a std::map as input and transitions the cell state from key -> value, if a key can be found, otherwise no behavior takes place.

1.6.2. Conditional transition rule on a neighbor:

'$NeighborConditionalRule$' is implemented to allow a cell's state to change based on the states of its neighbors. The user specifies a specific neighborhood they want to use, "moore" or "von neumann" as well as two std::maps, one that maps internal transition, moving a cell from key->value, and if no internal state transition is available, then it looks at the external state transitions and sets the cell equal to the value of one of

the key states that a neighboring cell possesses. This functionality is sufficient to define the "fire spreading" cellular automata example we studied in class. More specific update rules would have to be defined by the user as their application requires.

1.6.3. Majority rule:

'$MajorityRule$' is implemented to change a cell's state based on the dominant state of the majority of its neighbors, as defined in class.

1.6.4. The Parity rule and The Activation-Inhibition rules presented in Week 13 and Week 15 are optional: Custom Rules and Additional Attributes for Part II:

Some custom rules, functions and their uses for Part II of the simulation portion of this project include the following:

- $Person\ Struct$: representing the basic unit of the simulation, contains several attributes and various thresholds for state transitions.
- $checkSick3$: determines if any neighbor in a 3 x 3 grid is sick with variations based on the input parameters
- $superSpread3$ and $superSpread3Wrapped$: Models the spread of the sickness. A central person's state changes based on the states of their neighbors.
- $specifySteps3$ and $specifySteps3Wrapped$: Governs the progression or regression of a person's state based on the neighborhood's condition and individual thresholds.

2. Compute services: Time Stepping and Update:

The '$Cellular\_Automata$' class manages the simulation of CA using a 2D vector, '$\_hidden\_state$', which holds the current state of the system and the boundary zones. During the update phase, the $update()$ method creates another 2D vector, '$next\_hidden\_state$', to store the state for the next time step. This approach ensures that the grid for the current configuration (representing the $CA\ at\ time\ t$) is kept separate from the grid containing the updates ($CA\ at\ time\ t\ +\ 1$).

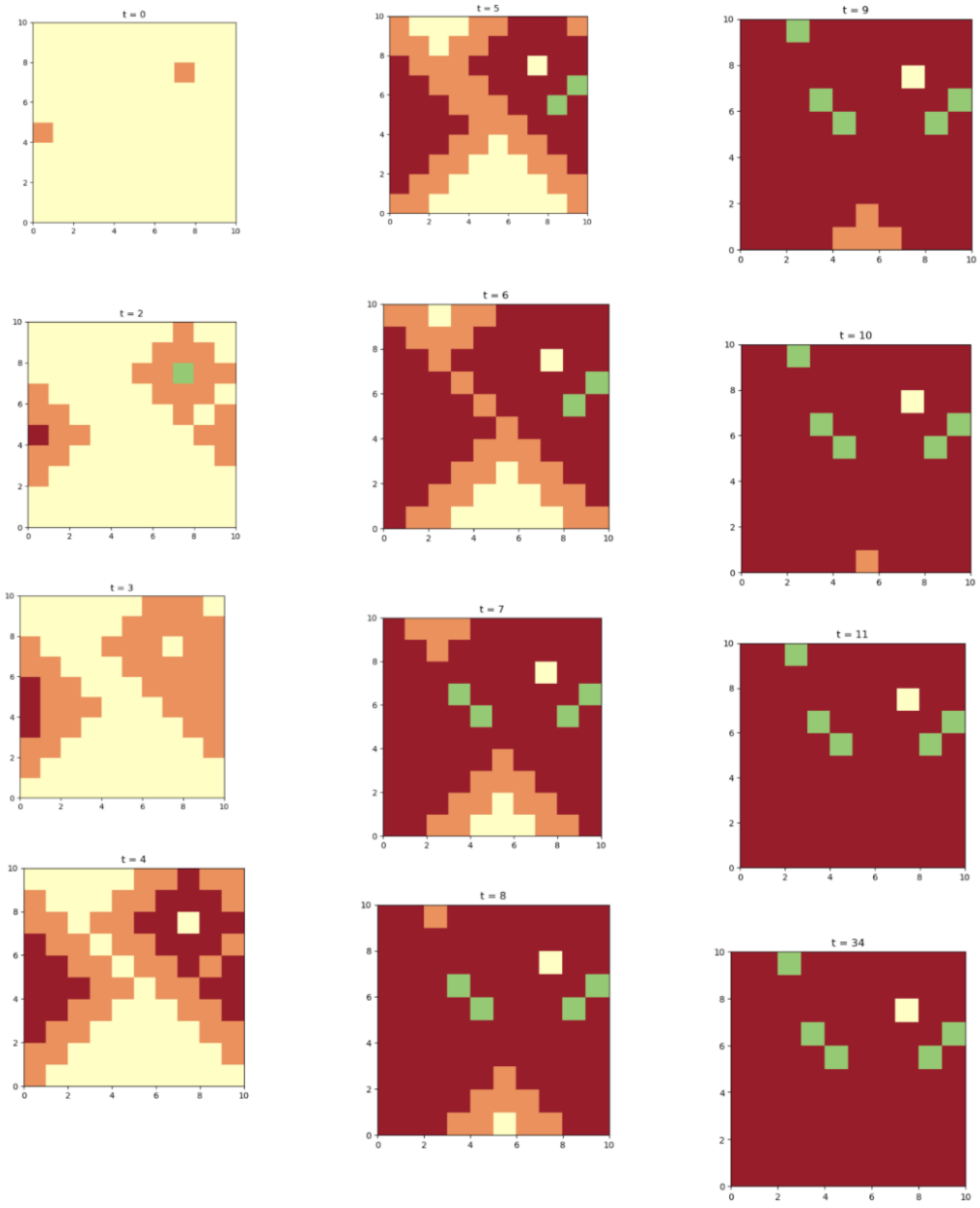
To apply the update rules to each cell, the '$get\_neighborhood\_around$' function retrieves the neighborhood of cells around a given cell. The rules are then applied through the '$\_updateRule$' function, based on the current state of each cell and its neighbors. This guarantees that the update rules reference the current state of the cells, avoiding any influence from states that have already been updated in the current cycle.

In the '$COVID\ -\ 19\_Simulation.cpp$' main function, the simulation progresses through a loop that iterates for a specified number of steps ($numSteps$). At each iteration, the $update()$ method is called to advance the simulation by one time step, reflecting the changes according to the defined rules.

3. Output, Display or Draw:

3.1. Requirement to display an image of the current CA configuration.

The CellularAutomatPractice.ipynb notebook, located in the Utils/MonteCarlo directory, is set up to demonstrate the outcomes detailed below. This code effectively models and visualizes a Cellular Automaton (CA) simulation, depicting the progression of COVID-19 over time. The visualization is based on a sequence of CSV files generated by the COVID-19_Simulation. A reference to the looped visualization from our jupyter notebook can be seen below:

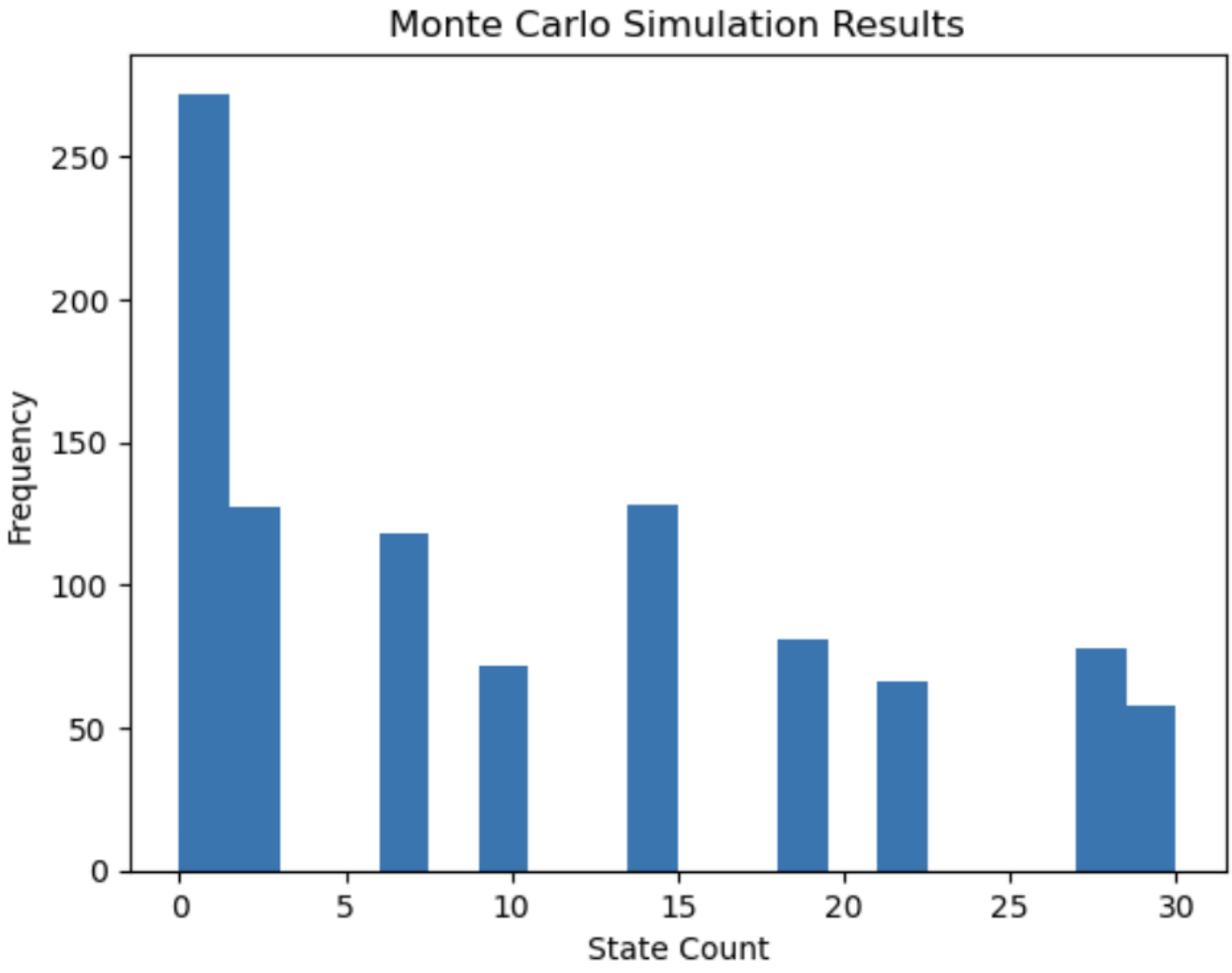


3.2. Requirement to count the number of cells in the CA that are in a particular state.

The 'count' function is a templated method within the '*Cellular_Automata*' class. This function serves as an important tool that is designed to count the number of cells in the CA grid that conform to a user defined predicate condition. It enables users to easily and efficiently count cells that meet a specific criteria, enhancing the analytical capabilities of the CA model.

3.3. Other analytical information relevant to the CA simulation and analysis of application (Problem 2 in Part II).

A Monte Carlo simulation integrated into the simulation of the spread of COVID-19, can offer significant value in assessing the variability and likelihood of different outcomes under uncertain conditions. By randomly sampling from different states of the CA simulation, an MC simulation can provide a probabilistic understanding of disease spread dynamics. In epidemiological modeling, this can be utilized in quantifying risks, understanding the possible range of infection rates, and preparing for various scenarios.



The code in Utils/Visualizations/Data was used to statistically analyze the variability of COVID-19 spread in a simulated environment using our CA model. This simulation provides insights into the probable distributions of infection counts at various stages of spread. From the histogram we can deduce the most common outcome is a lower count of infected cells, as indicated by the tallest bar on the left. There is a wide range of possible outcomes which reflects the stochasticity of the disease spread in the CA model. The variability in the histogram bars suggests that the spread of infection is unpredictable and subject to fluctuations over time.

Part II. CA Modeling and Simulation Summary:

Problem 2. <u>Design and implement a Cellular Automata Model that Uses Your General Purpose Library (from Part I) Showcase the application of the CA library:</u>

The '$COVID - 19\_Simulation.cpp$' file demonstrates the application of the '$Cellular\_Automata$' (CA) General Purpose library in the modeling of the spread of COVID-19. It defines a '$Person$' struct to represent individuals with different states that indicate stages of infection related to age, step count and thresholds for state transitions like toRecover, toHealthy, toSick, toDead. The program reads simulation parameters and initial states from an input file. The parameters include the number of steps for simulation, output frequency, dimensions of the simulation grid and initial state and attributes of each individual in the population. This information is used to initialize a cellular automata board with these specified settings. Key to the simulation are the custom update rules: '$superSpread3$'and '$specifySteps3$'. The former models the contagion process, altering the state of the central individual based on the infection status of their neighbors. The latter further manages the progression or regression of an individual's state, taking into account their current state, states of their neighbors and individual specific thresholds.

As the simulation progresses, it iterates over a specified number of steps. At each step the board's state is updated according to the defined rules. The frequency of output generation, as specified in the input file, dictates when the current state of the simulation is saved to CSV files. These files contain a dataframe that shows the simulation at different time steps, allowing for a detailed analysis of the pattern of the spread of infection and the implementation of the rules over time. This application showcases how the CA library is utilized for a complex simulation like disease spreading, emphasizing state management, neighborhood effects, boundary conditions, rule application and time- stepping mechanisms, offering a comprehensive tool in understanding and analyzing the spread of COVID-19 within a simulated population.

The application of the CA library in modeling the spread of COVID-19 is highly relevant to the fields of science and engineering, showcasing the potency of computational modeling. Computational modeling is

crucial in understanding complex biological phenomena such as disease transmission. Simulating various scenarios facilitated by the CA can enable policy makers to visualize how disease is spread, thereby providing valuable insights. These insights are instrumental in devising effective intervention strategies that can significantly impact public health decisions. Furthermore, this approach is pivotal in epidemiological studies. It aids in predicting the behavior of diseases, guiding response strategies and shaping public health policies. The application of the CA library in modeling COVID-19 not only contributes to epidemiological research but also exemplifies the intersection of computational biology, showing how a concept like cellular automata can be applied to model and analyze real world scenarios, in this context of pandemic response and management.

Part III. [Individual Work]

Problem 3. Individual student software engineering reflection on their final project:

3.1. A description of their role in the project.

In this project, my primary role was to contribute to the conceptual development of the Cellular Automata (CA) General Purpose Library, focusing on the discussion and formulation of its functionality. This included the creation of custom rules, attributes, and various thresholds essential for state transitions in the simulation. While the majority of the C++ code was developed by my colleagues, Austin and Sam, my responsibility extended to a thorough analysis of this code. This in-depth examination was crucial to fully grasp the capabilities and applications of the CA General Purpose Library, particularly in the context of the COVID-19 Simulation.

Leveraging this understanding and through continuous dialogue with my team, I took the lead in composing the majority of our project paper. At the same time, this period also involved supporting my teammates as they refined and debugged the code. I played a particularly vital role in the Python coding aspect, especially in visualizing the simulation modeling of the spread and the Monte Carlo simulation. This contribution proved to be instrumental, as my teammates were able to further modify my code to produce the insightful visualizations featured in question 3 - 3.1 and 3.3 of our paper.

The success of this project is a testament to the effectiveness of cross-functional teamwork. I deeply value the contributions of my colleagues and the shared knowledge that significantly enriched our collective experience throughout the project's progression. This collaborative effort not only culminated in the successful completion of the project but also fostered a dynamic exchange of ideas and skills that were integral to our achievement.

3.2. How much do they contribute to the successful project completion (e.g. lead work, help others, coordinated meetings, etc):

As members of the full-time MSSE program, our trio effectively synchronized our schedules, allowing us to collaborate on the project amidst other class assignments and final projects. We utilized Discord for coordination and meetings, where screen sharing and discussions of ideas played a key role in the successful culmination of our project.

3.3. Describe any challenges/problems you and/or your team dealt with and how you solved them:

Collaborating in cross-functional teams posed its challenges, particularly due to our remote working setup. The task of debugging issues was notably more complex without the possibility of in-person, direct interaction to pinpoint problems. However, we overcame these obstacles by effectively using GitHub. We leveraged its functionality to push and pull changes, ensuring smooth collaboration by tagging each other in our approaches.

3.4. Comment on the algorithmic and performance analysis of your library (you can base this on the data structures and functions you included). You do not need to provide a detail of every function that you implemented but rather provide a high level description of the design considerations that make you choose the data structures and algorithmic implementations that you used in this project. In particular, the parts of the project that you lead.

The algorithmic and performance aspects of running the CA library for the COVID-19 simulation are influenced by the strategic design choices in data structures and algorithmic implementations. The use of a 2D vector grid coupled with a templated program enabled flexibility and specificity, allowing for various simulation types while encapsulating relevant attributes efficiently. This choice results in O(n^2) spatial complexity, typical of grid based models, which supports dynamic resizing and easy access, crucial to CA simulations. Algorithmically the functions governing cell state transitions and neighborhood interactions adhered to the CA's principle of localized interactions. These functions, including boundary rules, ensure that the model accurately represents the spread of infection. Performance wise the most significant consideration is the update step, which has O(n^2) time complexity per iteration due to the necessary recalculating of each cell's state.

3.5. Comment on how effective your showcase CA application from Problem 2 worked (i.e., how realistic was the model, any suggestions on how to improve its accuracy, etc)

Showcasing our CA application from Problem 2 went extremely well. Utilizing the CA framework, we successfully modeled the spread of COVID-19 over time on a micro scale over time. By employing individual states for each cell (representing persons) it captures a range of scenarios like infection spread, recovery and mortality. For future research we could fine tune parameters like infection rates, recovery times and mortality

rates based on real world data, in order to enhance realism. Public health measures like masking, vaccination rates or population density could improve the model by integrating external variables.

3.6. What you would have done differently :

3.6.1. Library development:

To reflect on the development process, more comprehensive documentation and further testing could have been done. Due to the time constraints of the full time program we were unable to fully document within the code itself.

3.6.2. Software project management

Had we been given more time, we would have distributed responsibilities more evenly amongst each other. Initially, we underestimated the complexity of each project component, and as we delved deeper, this became evident. Despite these challenges, our collaboration was ultimately successful. Each team member developed a thorough understanding of the entire project, including the CA general library, its application in the COVID-19 simulation, the intricacies of creating visualizations, and the implementation of the Monte Carlo simulation

3.6.3. Product improvements

For future research, addressing performance challenges associated with large-scale simulations should be a key focus. This might include advanced code optimization strategies and the exploration of parallel computing techniques to enhance efficiency.