PHYS 349, Winter 2022 - Final Report

Olivia Masella, Carlos Noble Curveira, Andrew Simmons

Our code can also be found in our GitHub repository: `https://github.com/cnoblec/automata`

1 Statement of contributions

Each person wrote their own class to handle cellular automata, with different implementations for different problems. Olivia worked on 1D and 2D traffic simulations, Carlos worked on the Game of Life, and Andrew worked on forest-fire simulations. For the interesting cases, Andrew focused on interesting cases for forest-fire simulations while Olivia and Carlos implemented and studied slime mould simulations.

2 Introduction

In this project, we are exploring cellular automata and its applications. In its simplest form, a cellular automaton features a discrete lattice with a finite number of cell representations. By imposing a set of rules on the lattice, we can simulate its evolution over time. These rules typically involve some sort of mathematical and/or logical relation between a single cell and its neighbours; where the neighbours are defined by the developer of the cellular automaton (i.e. adjacent cells with or without diagonals, etc.). Additionally, one of the advantageous features of cellular automata is their ability to simulate both determinisitc and probabilistic behaviour by simply changing the way in which the algorithm scans through the lattice and applies the rule(s).

3 Description (and comparisons) of algorithm(s)

3.1 Olivia's implementation

A CellularAutomata class was defined to take in the initial conditions and a function that defines the "rules". The rules function takes in: a tuple (x,y) defining coordinates, lattice dimensions, the current lattice, temporary lattice in the Automata's current update and outputs a Boolean determining whether the cell at (x,y) should move and a tuple (`next_pos`) that indicates where the cell should move to. Within the CellularAutomata class, the run method will apply the behaviour dictated by the rules function to every cell in the lattice. It's worth noting that this is not random, it filters through the lattice from the top-left to the bottom-right. Additionally, the class features a display method that displays the lattice at a given point in time (for 1D and 2D) or throughout time for 1D lattices (where the y-axis is time).

3.2 Carlos's implementation

Two classes are used to implement a generic N-D Cellular Automaton with the users choice of grid size and rules. The first class is a "Cell" object that defines the values that each cell can be, the data type that should be stored in the lattice, the possible values for initialization, and the frequencies of the cells to initialize from. Most importantly it is passed a function that returns the next value of a cell given a location in the lattice and a copy of the lattice.

There is also a "CA" class which represents the cellular automata and takes the number of dimensions, grid size (of a square grid), an instance of the "Cell" object, whether or not to use periodic boundaries, and potentially an initial lattice state. This class also creates a lattice where the cell data is stored. Instances of this class have a "generate" method which updates the "CA" lattice n times with the update function, and returns an iterable with the lattice at each generation. This method calls the cells update rule function on each coordinate in the grid to determine its value in the next generation. As the method is running a temporary lattice is created that stores the states of the next generation and then overrides the lattice with the values in the temporary one. The "CA" class instance also has an "animate" method that, given the number of generations n, a filename, and a list of colours corresponding to values in the lattice, saves an animated GIF of n generations via matplotlib's Animation packages.

The workflow of this implementation consists of creating a "Cell" object that contains the update rules etc. and pass that into the creation of a "CA" object from which you can call the "generate" or "animate" methods to interface the cellular automaton.

3.3 Andrew's implementation

An "Automata" class was defined that takes a neighbour order and either
(1) a pre-initialized lattice with at least one dimension
(2) a shape and a cell type (a type or a numpy.dtype) from which an empty lattice is created

Class "Automata" defined an evolve method which applies an update method (which should be defined by each subclass of "Automata" since, of course, updating one cell is different for each problem, but generally evolving the entire lattice is the same) to each cell. The update method should accept an array (the lattice to be updated) and an index (the index of the cell to be updated, and possibly the neighbouring cells depending on the problem).

Class "LandUse", subclass of class "Automata", accepts either a pre-initialized lattice with cell type "LandUseCell" (a class with a land use and an age attribute) or a shape (which will fill a lattice with random land use types and ages). Also defines an "update" method which will turn an earth cell into a tree cell if near a water or tree cell, will turn a tree cell into a fire cell after the tree grows for a certain amount of time or if near a fire cell, and will turn a fire cell into an earth cell after it spreads for a certain amount of time.

3.4 Major strengths and weaknesses of all three

From Olivia's version, the major strength that will be applied to the final implementation is the use of a temporary lattice to store information during a step that may affect further movements within said step. Additionally, a major limitation is the use of the "rules" function. This function has to have very specific inputs and outputs that limit the versatility of the class.

From Carlos' version, one major strength to be brought to the final implementation is the use of a temporary lattice similar to Olivia's version. Another strength is the flexibility to create any given cellular automata with ease. A weakness of this version is the update function because it can only change the value the coordinates passed in. This adds unnecessary complexity when creating an update function for more complex cellular automata. A takeaway from this is to allow the user to update neighbouring cell's in the lattice as well as have access to the temporary array to see what other updates have already been determined (e.g. cars moving in traffic at intersections.)

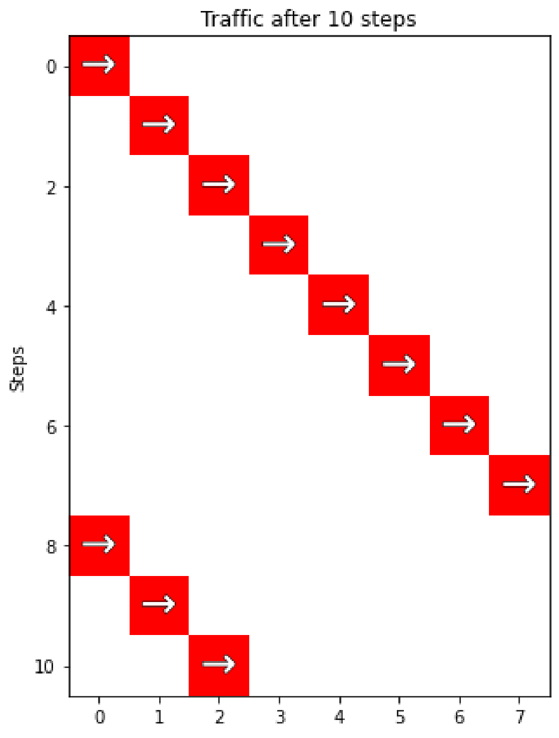
From Andrew's version, the major strength is that most implementation details are within class "Automata", meaning that the subclasses need minimal code in order to work. Additionally, an updating cell can also update its neighbours if need be. A weakness is that no temporary lattice is used; cells which move may be updated multiple times within one lattice update. This is not an issue for stationary automata, like "LandUse", but it severely limits its versatility for other types of automata.

4 Validation: Application to Test cases

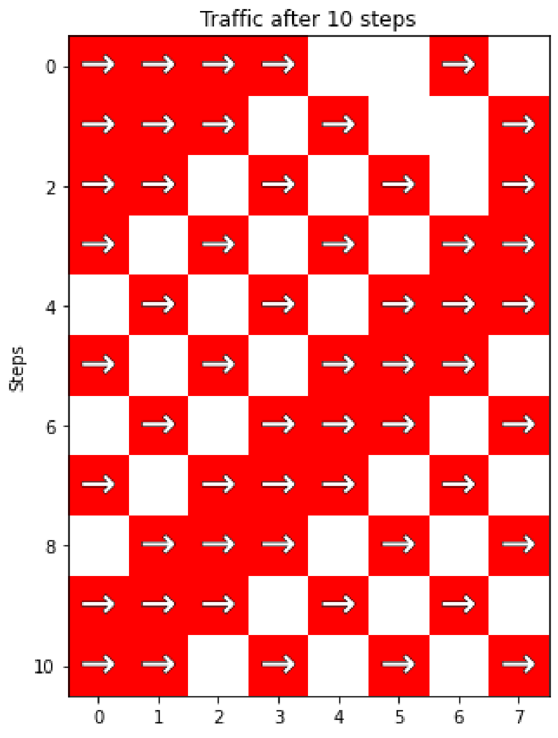
4.1 Traffic Simulations [1]

Traffic simulations were used to test the algorithm. The main mechanism behind these traffic simulations featured a lattice of integers where $0 =$ empty, $1 =$ car going right, $2 =$ car going down, $3 =$ car going left, $4 =$ car going up. At every step, a car can only move forward one slot and will only do so if and only if another car is not already occupying that slot or if another car will not occupy that slot first. Something worth noting is priority. The way in which priority is handled is by giving priority to the top-left most car.

Starting with the one-dimensional traffic simulations, the algorithm was tested with the simplest problem: a single car moving to the right in an otherwise empty lattice. Figure 1a below shows the results over time where the y-axis is the step number and the x-axis is the lattice. This case proved to be an excellent test since the behaviour of a single car moving forward is trivial; the car has no choice but to always move forward one space to the right every step.



(a) Basic case

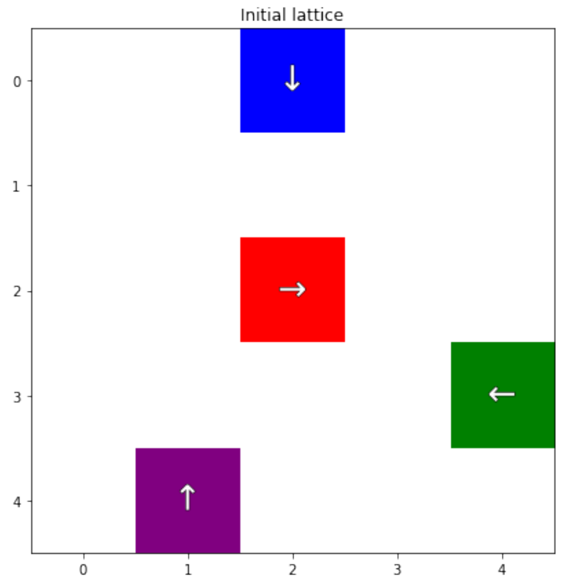


(b) Additional case

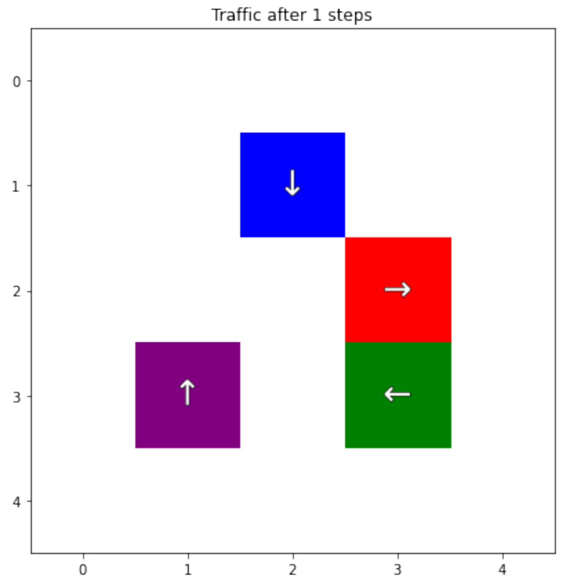
Figure 1: 1D Traffic Simulations

Additionally, 1D cases were also tested with less trivial examples such as random configurations of cars; the results of such are given in figure 1b. We can still work out the logic and verify that figure 1b yields the correct results based on how the cars should logically move forward in time.

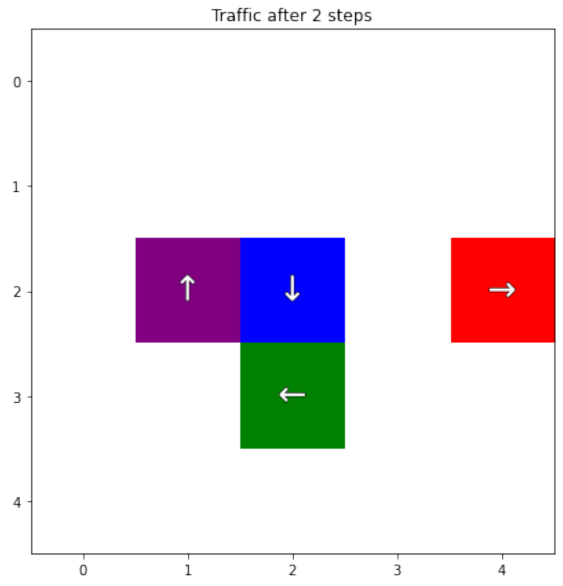
Two-dimensional simulations were also ran to further complicate and test the algorithm. Since most 2D cases are fairly complicated, a simple case was tested for the sake of validation, as shown in figure 2. If you follow the cars and logically map out their movements along with the correct order of priority, the simulated movements are correct.



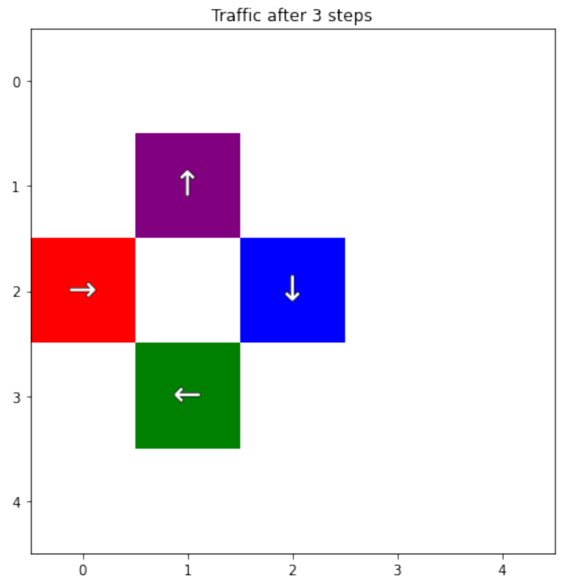
(a) Initial lattice



(b) After 1 step



(c) After 2 steps

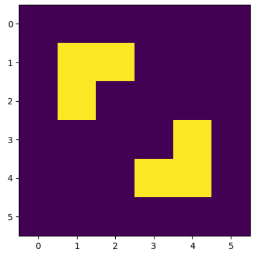


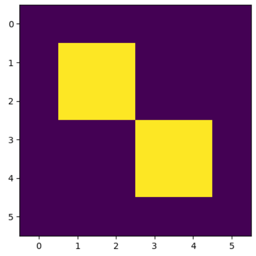
(d) After 3 step

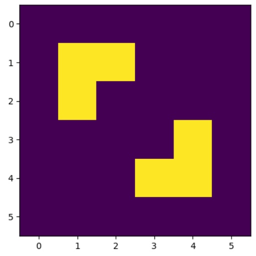
Figure 2: 2D Traffic Simulations

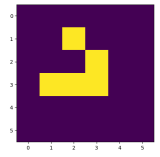
4.2 Game of Life

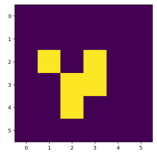
Conway's Game of Life is a very well known and studied cellular automaton which has a 2D lattice of binary variables, 0 (dead) and 1 (alive), representing some arbitrary cells. The game works by, each generation, going through each grid point and counting how many of the possible 8 neighbours to that grid point are 1 (alive). The rules are built to roughly mimic a population of living beings or cells and are as follows. If the value of the grid is 1 (alive) it stays alive if it has two or three neighbours, and dies with fewer or more. If the value of the grid is 0 (dead) it come to life if it has exactly 3 neighbours, and stays dead with any other number. This is a deterministic system with very results and can be verified by some well known structures that appear. Below are two examples, one of a glider which is a pattern that moves along the diagonal, and a blinker which cycles the same pattern over and over unless it is disturbed. Yellow is 1 (alive) and purple is 0 (dead).

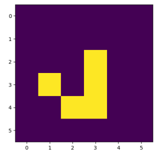

(a) Initial Beacon

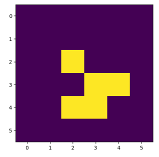

(b) After 1 step


(c) After 2 steps


(d) Initial Glider


(e) After 1 step


(f) After 2 steps


(g) After 3 steps

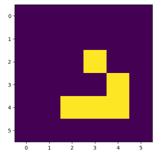

(h) After 4 steps

Figure 3: Conway's Game of Life Validation

4.3 Forest Fire Simulations [1]

The forest-fire problem does not have a standard set of deterministic rules for behaviour, so there are no exact results to compare directly against. However, the general idea is that fires will spread radially, trees will grow faster than they are being burned, and the forest will never burn completely down. You can view a deterministic example of the forest-fire simulation at `https://github.com/cnoblec/automata/blob/main/output/land_use_tests/fire_spread_3.gif`.

5 Description of interesting cases, and results of applications

5.1 Slime Mould

Slime mould is a field of cellular automata that models the behavior Physarum polycephalum, a slime mould that is used to model transport networks. This was implemented as a lattice of cells that store two layers of information, one is the individual cells and their directions, the other is a layer of pheromones that are used to signal the direction that nearby cells should move. The algorithm outlined by Jones is followed in our implementation [2]. Slime cells are initialized on a grid and secrete a pheromone as they move that diffuses outward and signals orientation of nearby slime cells.

The slime mould has three main parameters that control the behaviour which are the coverage, sensor offset, and decay constant. The coverage represents the fraction of the initialized grid to be covered with randomly directed slime mould cells. The sensor offset is the radial distance in pixels that the slime cells sample when moving. The decay is the rate at which the pheromone field multiplicatively decays during the diffusion process.

A parametric study was performed to analyse the slime mould's behaviour. The coverage, sensor offset, and decay parameters were specifically chosen for this study. By first looking at the sensor offset versus the coverage, we obtained the results in figure 4. For the sensor offset, we get a clear pattern: an increased offset increases the size of the holes of low pheromone concentrations. It gives the impression that we're simply zooming into a section of the lattice when we increase the offset. While, for the coverage, we get more noise with a higher coverage percentage that ultimately leads to little to no movement, hence the lower amount of pheromones present in the 90% column. When we combine variations of both of the parameters, we see how they depend on each other.

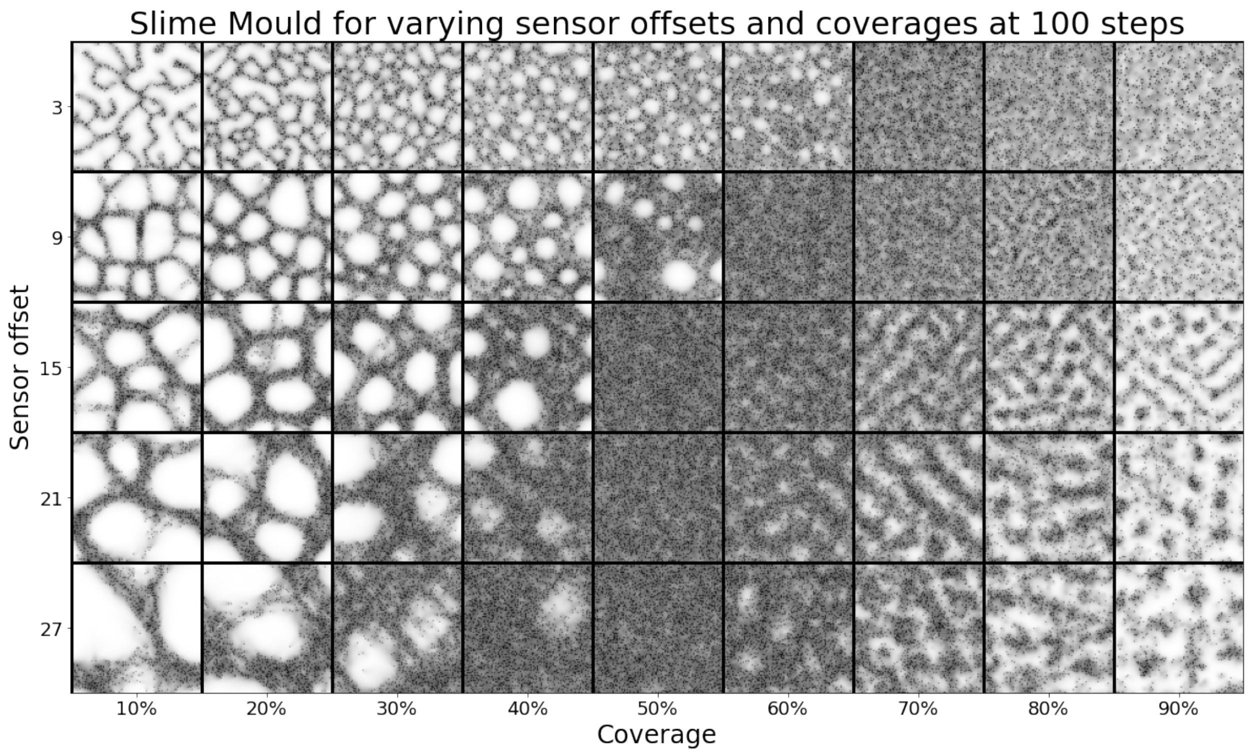


Figure 4: Slime mould results after 100 steps for varied sensor offset and coverage. Each image shows a 200 $\times$ 200 pixel lattice for the pheromone field depicting the trail left by the slime mould cells, where a darker shade of grey indicated a higher concentration of the pheromone with a scale from 0 to 1.

Next, we varied the decay constant along with the coverage, yielding the results in figure 5.

The coverage behaved as we expected it to from the previous figure where a larger coverage leads to more holes then eventually leading to static noise. The static noise is a result of slime cells not being able to move since the lattice is too overpopulated with slime cells. Meanwhile, increasing the decay constant increases the contrast of the image. This occurs because pheromones decay very quickly with a larger decay constant resulting in only newly deposited trails being displayed. The relationship between the decay constant and the coverage is very subtle. With a small enough decay constant, we can see some holes forming at a high coverage percentage compared to the larger decay constants as shown in the last column of figure 5.

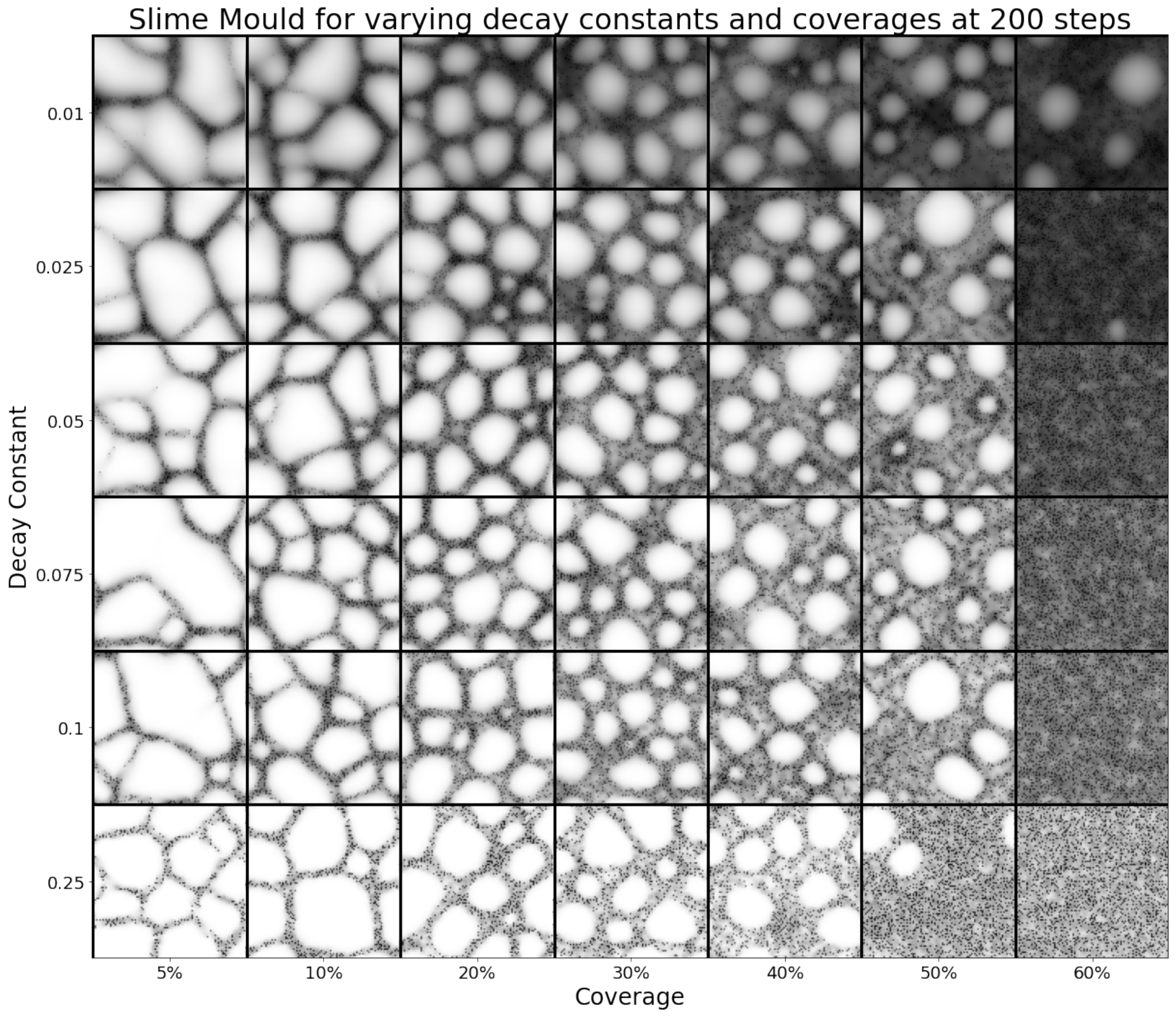


Figure 5: Slime mould results after 200 steps for varied decay constant and coverage. Each image shows a 200 × 200 pixel lattice for the pheromone field depicting the trail left by the slime mould cells, where a darker shade of grey indicated a higher concentration of the pheromone with a scale from 0 to 1.

5.2 Modifications to Forest Fire

The forest-fire simulations are composed of a small lattice of four land types: water, earth, fire, and tree. By default, lattices will not initially have any fire; fires are started by random lightning events. There are four other random events: a tree grows near a lake, a tree grows near another tree, rain extinguishes a fire, and fire spreads to a nearby tree.

Since trees can grow near a lake, a forest can always continue growing, even if it has been entirely burned down. Similarly, since lightning exists, a forest can always continue burning, even if all fire has been extinguished.

If these options are turned off, there are two states of equilibrium; the entire forest burns down, leaving a lattice of water and earth, or all fires are extinguished, leaving a lattice of water and trees. In these situations, dramatic differences are seen when varying the fire spread and tree spread. With a minor increase/decrease in fire/tree spread-ability, a simulation can go from trees being dominant to a forest burning down in seconds. Minor decreases in the occurrence of rain have a similar effect; fires will burning for longer, fires will spread more, and eventually cover the whole lattice.

Some examples of these can be found at:
`fire_spread_2.gif` and `fire_spread_3.gif`
`earth_to_tree_4.gif` and `earth_to_tree_6.gif`

6 Discussion

Another extension of slime mould is its use in mapping out points of interest like subway stations in cities [3] and dark matter between galaxies [4]. A simplified version of these cases is that "food" sources secrete a strong signal of the attractant pheromone and are placed at points of interest but do not move. These points serve as a guide for the slime mould to form a network between, connecting the points in the most efficient way possible [2]. Simple implementations of these were tested and are on the project github however further improvements could be made to include food sources in a more structure implementation. Additionally the slime mould cellular automata could be implemented to handle a 3D lattice to map out 3D transport networks.

A possible improvement would be to implement a sparse-lattice of sorts. For cellular automata which can have empty cells (cells which do nothing when updated), such as traffic simulations or the game of life, it might be better to store a list of cell values and locations. You would not need the space to store the empty cells, and it might improve speed and readability when updating the state of each cell. You would not have to write special cases for empty cells (since they would be skipped in looping), and you would only have to loop over a few non-empty cells instead of hundreds/thousands/millions of empty ones. We might have also tried a method for determining if a lattice can be better stored as sparse or full, keeping in mind that we store more information per cell with a sparse lattice (a value AND location), but might store less cells in total.

One of the bigger limitations with our cellular automata class implementation using subclasses is its use for specific cases. We encountered a bit of a run time issue with the slime mould simulations where the parametric studies took roughly 15-30 minutes to run. Luckily, 15-30 minutes is not entirely unreasonable for a simulation, however, upon further thought, we realized that a more tailored implementation could have decreased the run time. This cellular automata superclass is great for smaller examples like the ones we showcased in this project, however, it would not be ideal for an extensive study of one particular cellular automata example, especially a computationally extensive one like slime mould that requires a larger lattice.

7 Conclusions

Overall, we were able to code a generic cellular automata superclass to be used with specific examples as subclasses. We implemented traffic simulations (both 1D and 2D), game of life examples, forest fires, and finally, slime mould simulations. Simple simulations of traffic, game of life, and forest fires were used to ensure that the cellular automata class was behaving as expected. Then, more complicated cases were explored for forest fires and a newly implemented slime mould. For forest fires, fire spreading and tree spreading parameters were explored to yield drastic differences from slight modifications. For the slime mould simulations, the coverage, sensor offset, and decay constant parameters were explored. An increase in the coverage increased the number of holes in the trail network. Increasing the sensor offset leads to a smaller number of holes in the trail network, causing it to look as if it's zoomed in. Finally, increasing the decay constant leads to an increase in contrast, where a value of 0.1 provides an ideal balance of contrast to blur. To conclude, the model we developed is a useful tool for both simple automata cases and more complicated ones. Further improvements would include a sparse lattice to reduce redundant scans through the lattice and ultimately improve run time. Additionally, this cellular automata implementation can be used to test out more non-analytic cases such as ant colony simulations and biological cell modelling.

References

[1] B. Chopard, *Cellular Automata Modeling of Physical Systems.* New York, NY: Springer US, 2018, pp. 657–689. [Online]. Available: https://doi.org/10.1007/978-1-4939-8700-9_57

[2] J. Jones, "Characteristics of pattern formation and evolution in approximations of physarum transport networks," *Artificial Life*, vol. 16, pp. 127–153, 2010. [Online]. Available: https://uwe-repository.worktribe.com/output/980579

[3] A. Tero, S. Takagi, T. Saigusa, K. Ito, D. P. Bebber, M. D. Fricker, K. Yumiki, R. Kobayashi, and T. Nakagaki, "Rules for biologically inspired adaptive network design," *Science (American Association for the Advancement of Science)*, vol. 327, no. 5964, pp. 439–442, 2010. [Online]. Available: https://www.science.org/doi/10.1126/science.1177894

[4] J. N. Burchett, O. Elek, N. Tejos, J. X. Prochaska, T. M. Tripp, R. Bordoloi, and A. G. Forbes, "Revealing the dark threads of the cosmic web," *The Astrophysical Journal*, vol. 891, no. 2, p. L35, mar 2020. [Online]. Available: https://doi.org/10.3847/2041-8213/ab700c