Chem 274B Final Project
14 December 2023

Team 1 Final Project Assessment:
Francine Bianca Oca, Kassady Marasigan, Korede Ogundele

Our implemented Cellular Automata (CA) application serves as a powerful tool for simulating the changes in allele frequency within a population. This application that was developed using the general-purpose CA library offers users the ability to specify the starting frequency of a recessive allele, initiating a simulation that changes over multiple generations. The CA model, configured in a 2D grid with Von Neumann neighborhood and no boundaries, uses a conditional transition rule based on the genetic makeup of neighboring cells.

This simulation is significantly relevant in the scientific realm of genetics and evolutionary biology. By allowing users to manipulate the initial conditions and observe the genetic evolution, the application provides insights into the factors influencing allele spread, genetic equilibrium, and population stability. The CA model captures the complex interplay of genetic traits, offering a visual representation of how different genetic configurations impact the overall population dynamics.

In the context of genetic research and education, this application becomes a valuable asset. Researchers and students can utilize the CA simulation to investigate the sensitivity of the model to various genetic interaction rules, exploring the spatial and temporal patterns of allele frequency changes. The results contribute to a deeper understanding of evolutionary processes, genetic diversity, and the role of initial conditions in shaping population genetics. The application's output enables users to connect it with real-world genetic scenarios, providing a bridge between theoretical genetic models and empirical observations. Overall, this CA modeling application can be an informative and illustrative tool for exploring the complexities of genetic systems and their evolutionary implications.

For our genetic model as mentioned in the beginning, a function for our general purpose library was created to set-up the model in terms of neighborhood utilization (set_neighborhood()) which adjusted neighbor settings depending on choice of neighborhood (Von Neumann vs. Moore). Additional functions in our general purpose library were created to set up the dimensions of our grid, choose our boundary type (Periodic/Fixed/No Boundaries), and even set the rules appropriate for our genetic simulation. For the purposes of our model to correctly demonstrate genetic evolution, we setup our application as follows (and this can be seen in our test_genotype.cpp test file under the Tests directory):

- set_dimensions(TWO_DIMENSIONAL)
- set_neighborhood(VON_NEUMANN)
- set_rule(CONDITIONAL_TRANSITION)
- set_boundaries(NO_BOUNDARIES)
- set_states(3)

As you can see, we set up a 2 dimensional grid of 10 rows and 10 columns (10x10) to simplify the output and make it readable for the user. This also allows users to view patterns easier in comparison with working with a large-scale grid. The Von Neumann neighborhood was best suited for our model to base a current cell's genotype off of the average of the neighbor's genotype and this can be seen in our update() function, which updates the grid by generation when applying the determine_genotype() function (will be talked about next). For the simulation to occur after initialization of the model, the update() and determine_genotype() functions are called to apply the specific genetic rules we set in place. Please note that these functions specifically are not a part of our general purpose library, and were created specifically for the simulation of our model. The determine_genotype() function, determines the cell's genotype by the following rules:

- If both parent cells are Homozygous Dominant -> Return Homozygous Dominant
- If both parent cells are Recessive -> Return Recessive
- If one parent cell is Homozygous Dominant and the other parent cell is Recessive -> Return Heterozygous
- If both cells are Heterozygous -> Random Decision for offspring based on probability:
  - 50% Probability Return Heterozygous
  - 25% Probability Return Homozygous Dominant
  - 25% Probability Return Recessive

The update() utilizes those rules (by calling determine_genotype() within it's function) and will apply it to the grid per generation.

When running our genetic application, the test code was programmed to allow users to define a starting recessive allele frequency (from 0 to 1). The grid is then filled with genotypes based on the starting recessive frequency and the genetic simulation updates the grid. The grid is filled by using a random number generator that is seeded to fill the rest of the grid with either heterozygous or homozygous dominant based on the recessive allele frequency chosen.

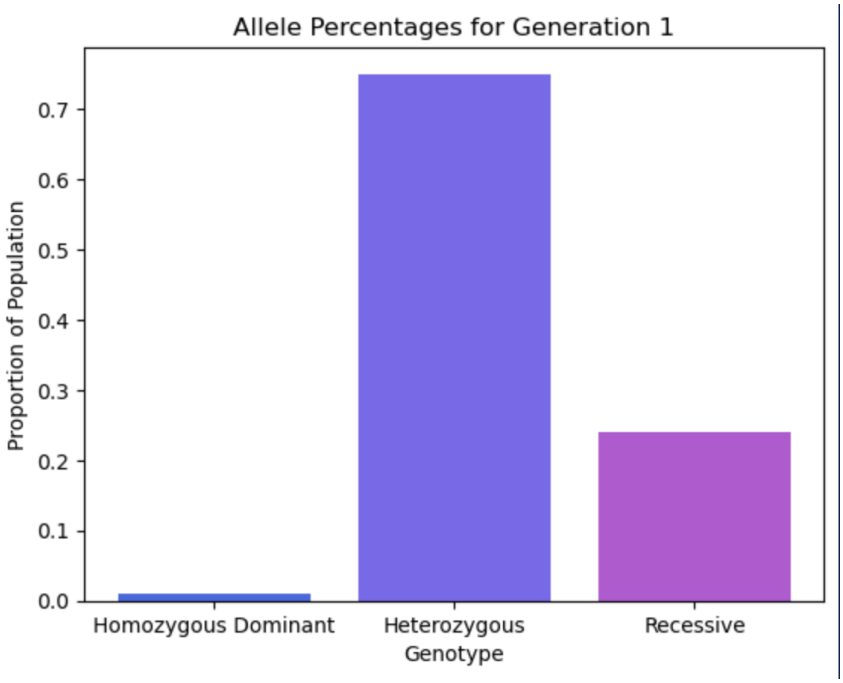
The output of the grid is saved to a file called 'simulation_output.txt'. These outputs are then imported by a Jupyter notebook (found in the Plots/ directory) and relevant visualizations are created. The first function plots the current state of the cellular automaton using the arguments generation number and map color scheme (an optional parameter). Given a generation number, it plots an automata with the

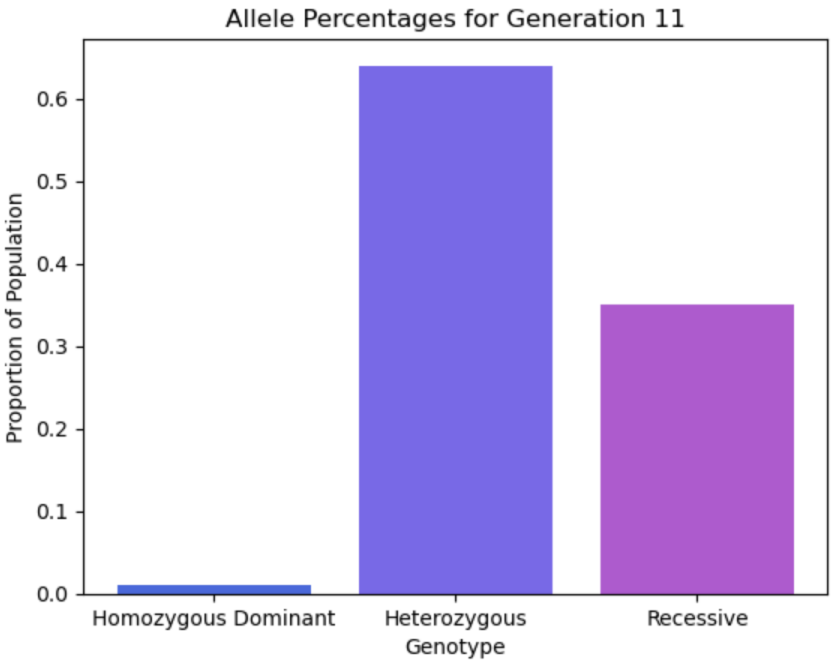
The second function, display_genotype_count, takes the genotype and generation number and produces a simple output statement telling the user how many individuals of a specific genotype are in a chosen generation.

```
display_genotype_count(genotype = "recessive", generation_number= 5)
```
✓ 0.0s Python

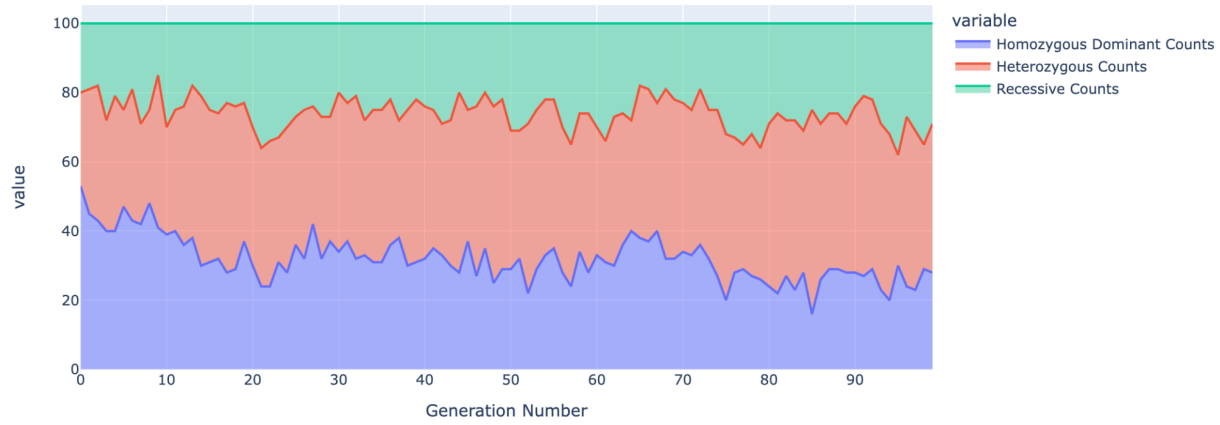
```
There are 37 individuals with the recessive genotype in Generation 5.
```

The third function, plot_generation_counts, takes in the generation number and total population to produce a bar graph that displays the proportion of each genotype in a certain generation. This is a simple and useful visualization to compare allele frequencies from generation to generation. Below are two examples of this graph in use.

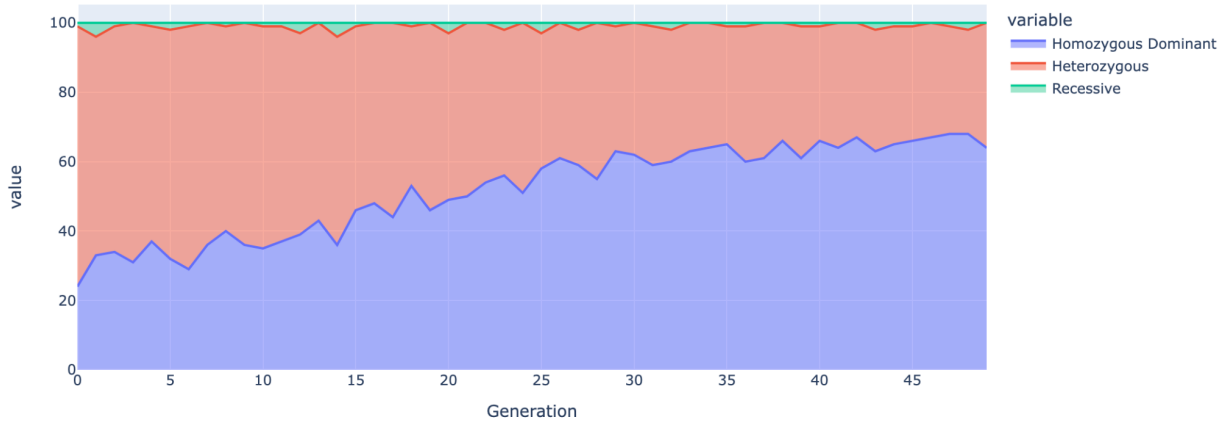


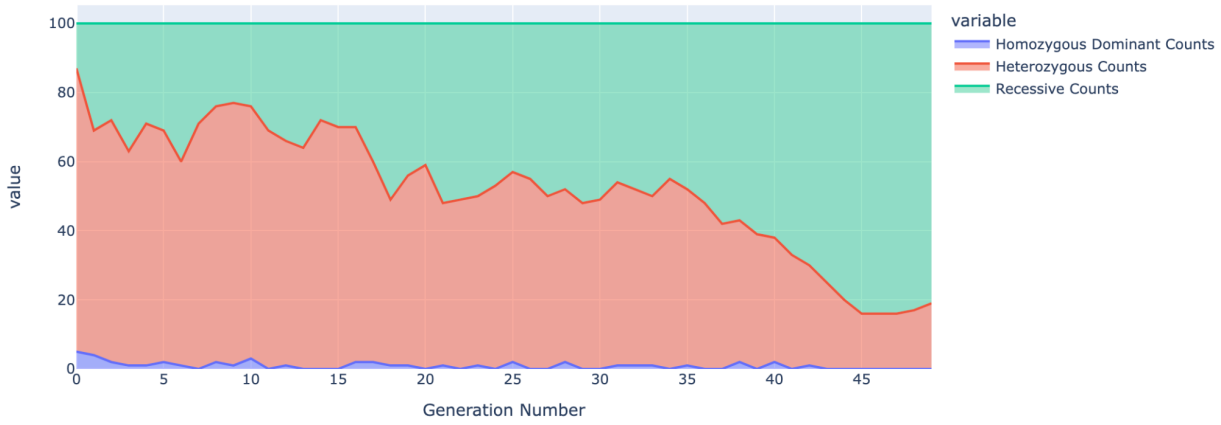


The fourth and final function produces an area plot that shows proportions of all allele frequencies over time. This area plot is similar to the bar graph produced by the third function, but displays genotype proportions through all generations at the same time. The area plot will vary vastly depending on the initial random state of the population (determined by recessive frequency entered by the user and other genotypes determined by the random seeding of the grid). Three example outputs are shown below.



In the example above, we started with a recessive frequency of 0.2. In the example below, we started with a recessive frequency of 0.01.





In the third example, we started with a recessive frequency of 0.2, and (due to the random filling of the matrix) a homozygous dominant frequency of 0.05. The area plot function illustrates how changes in allele frequency over generations (microevolution) is heavily dependent on the initial state of the population, which can be swayed by chance events such as a natural disaster (leading to a genetic bottleneck) or migration of individuals (leading to a new population whose allele frequency does not reflect that of the original population).