Program Structures & Algorithms
INFO 6205 Fall 2019
Final Project - Game of Life

1. Team Member

Xiaoge Zhang / 001409334

Mingyu Liu / 001498402

Mibin Zhu / 001424937

2. Introduction

2.1 Game of Life

The Game of Life is actually a zero-player game, which includes a two-dimensional matrix, and each point in this matrix is an independent cell. Each cell interacts with its eight neighbors using the following four rules: firstly, any live cell with fewer than two live neighbors dies, as if caused by under-population; Secondly, any live cell with two or three live neighbors lives on to the next generation; Thirdly, any live cell with more than three live neighbors dies, as if by over-population; Finally, any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction. As the game progresses, the disordered cells will gradually evolve into various delicate and tangible structures, and these structures often have good symmetry, and each generation changes shape.

2.2 Genetic Algorithm

In computer science and operation research, the genetic algorithm is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolution algorithm. The genetic algorithm is commonly used to generate high-quality solutions to

optimization and search problems by relying on bio-inspired operators such as mutation, crossover, and selection. The goal of the project is to use a genetic algorithm to initialize the best random pattern for the game of life's cell structure.

2.3 Innovation & Problem realization

The genetic algorithm is very suitable for solving the problem with enormous solution space, so in other words, we can say the genetic algorithm can best optimize our Game of Life model. Professor had already given some models to test the Game of Life code, based on this, we used genetic algorithm to generate new random models to get the best solution with more generations of the cell.

3. Question Description

3.1 Overview

As the definition describes, the Game of Life code can show the changes from cell to cell. A group of simple 4 rules of action will produce many unexpected and complex behaviors. Some cell patterns can be extended indefinitely, moving from generation to generation in a certain direction; some cell patterns can appear alternately, forming a transition pattern without an end.

Based on these characteristics, we modified the Game of Life code given by the professor, to generate multiple generations of cell patterns that can change as infinitely as possible. The original cell model was not completely randomly generated, but the optimal solution that was continuously evolved through genetic algorithms.

3.2 Concepts in Problem Implementation

Individuals : The production of new individuals requires the reproduction of two other individuals, which are the father and mother. Each can inherit genes from parents, but the breeding algorithm determines the proportion of inheritance;

Genotype : Genotypes are determined at the beginning of individuals' birth. For the first generation, genotypes are randomly generated, and for the offspring of the first generation, their genotypes are produced by parents;

Phenotype : The phenotype is the external appearance of trait determined by chromosomes, or external appearance of individuals based on genotype;

Evolution : The population gradually adapted to the living environment, and the quality was continuously improved. The evolution of organisms is in the form of populations;

Crossover : DNA was cut off at the same position on two chromosomes, and the two strings before and after were crossed and combined to form two new chromosomes;

Fitness : Fitness is used to measure the adaptation of a species to its living environment, which is calculated by the fitness algorithm.

4. Algorithm Design

4.1 Genetic Algorithm Implementation

In the class of genetic algorithm, we create it in three steps. The first step is to use the eval function to create a connection between the implementation of the genetic algorithm and

the given Game class, and then pass the contemporary constructed object by returning generation.

Since the output we got for the chromosome is of type int, we also needed to use the transfer function to convert it to a string type. According to the definition of the genetic algorithm, we firstly get its fitness value, because we need to evolve as many generations as possible, so the fitness value here is the biggest generation when running the game.

In the geneticAlgorithm function, we initialized some chromosomes by using the Engine object, for each two points in these chromosomes correspond to a point coordinate, which is used to display the starting pattern in the Game of Life. For the length of chromosomes, we set them between the length of 8 and the length of 20. We also set the population size as 500, mutator probability as 0.3, also the crossover probability as 0.3. In this process, we got the best result in each generation and used the getGenotype function to save that into the result. In order to display the current generation in the main function every time to modify and optimize the code, we set the generations variable to store the current state for future UI use.

4.2 Game of Life Implementation

As mentioned before, because most of the Game of Life code the professor had already given to us, we had to do more work on linking them together. In the Point class, it compared each Point's location and returned the vector from this Point to other. In the Group class, it appended the specified point to the list of points and normalized the Group, there also had an overlap function to determine whether this Group overlaps with group and used merge function to combine them all.

In the Matrix class, it represented the physical 2-dimensional layout of a Group with clear rows and columns around any live cells. In order to calculate how many live cells are there, it also used getCount function to return the count. On the other hand, the existence of the neighbor cell at each point and the number of neighbor cells will affect the survival of the current cell, it also used Neighbors class to get that. The Grid class represented an infinite grid on which the game of life can be played.

Finally, the Game class is the main class of the entire Game of Life project, it had multiple functions to help this program run completely. The generation function is highly linked to the GUI interface, which will be introduced later. The growthRate function is used to get a very crude measure of growth rate based on this Game and the very first Game in the series. The Behavior function run is used to run a Game given a monitor method for Grids.

4.3 User Interface Implementation

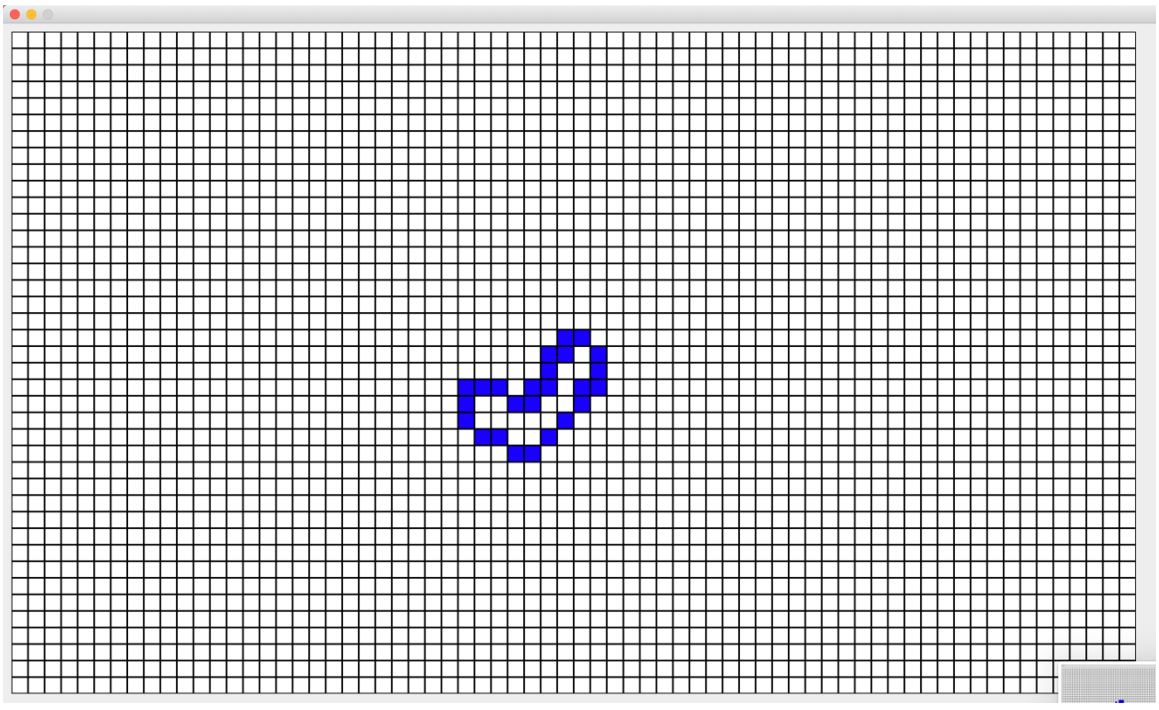
In the GUI class, we used java swing to make the UI, we set the map width as 68 and map height as 40. If cells multiplied to the border and need to continue to multiply, we can set map width and map height to higher value. Then we also need to create the JPanel object with panel size and set the layout to Grid layout.

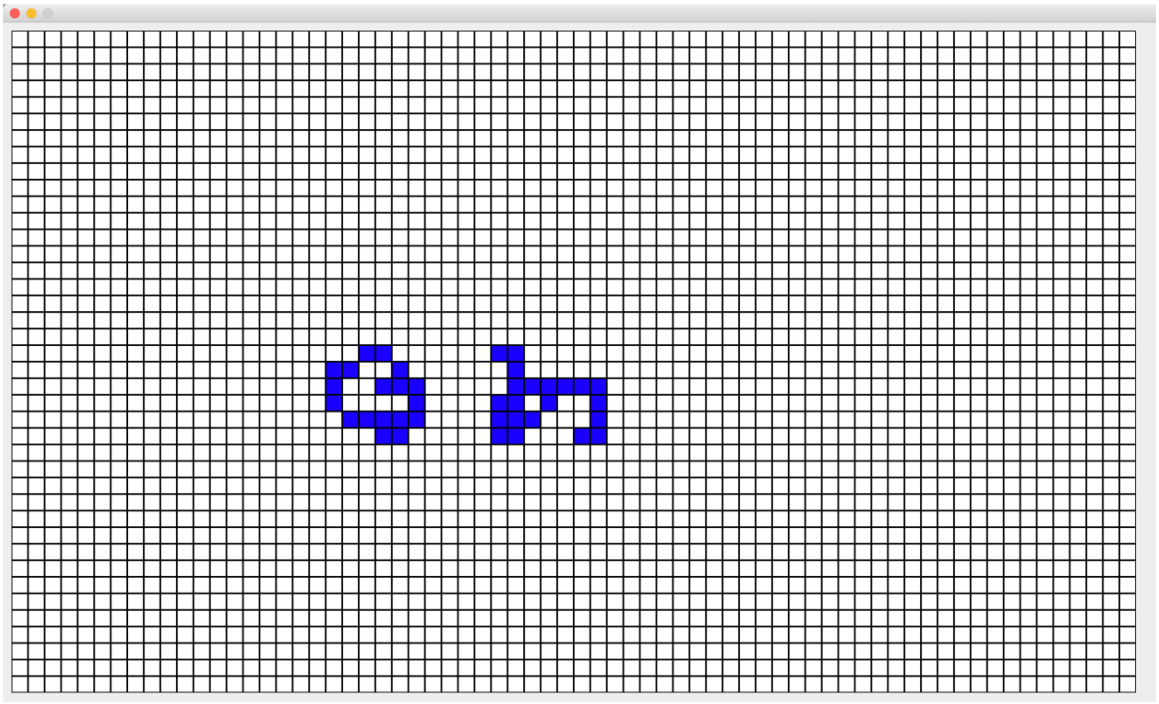
As we all know, the generation of cells in each generation is based on the origin point, so we need to set the origin point first, and then display the cell shape of each generation on the panel based on the origin point. For example, the current abscissa curX of each point is equal to the abscissa of the origin point getX plus the node's original abscissa startX.

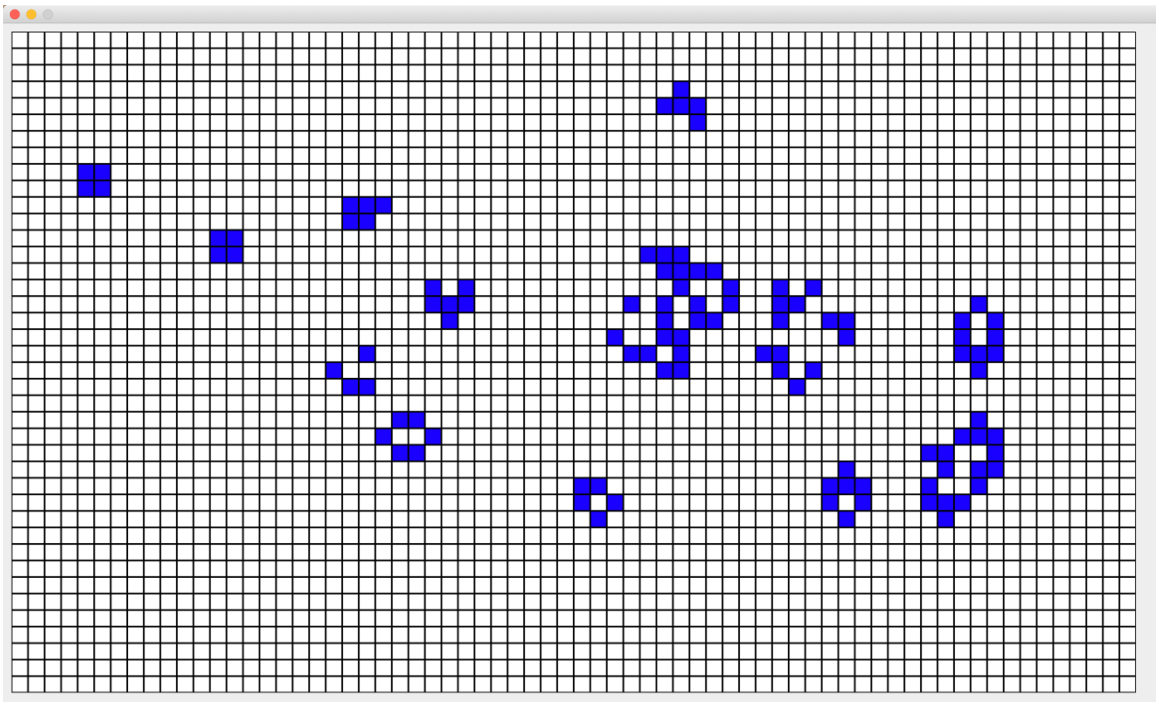
When generating each generation of cell nodes, there was a small trick that is the origin point of the first generation cell style has nothing to do with the location of other points, so when the generation is equal to 0, we directly print the cell nodes generated by the algorithm. When the generation is greater than 0, we must firstly calculate the coordinates of the origin point based on the generation distance of the origin point, and then print the coordinates of other cell nodes based on the origin point coordinates. In addition, if the origin point of the new generation is equal to the origin point of the previous generation, we do not need to move the origin point, and just calculate the node coordinates.

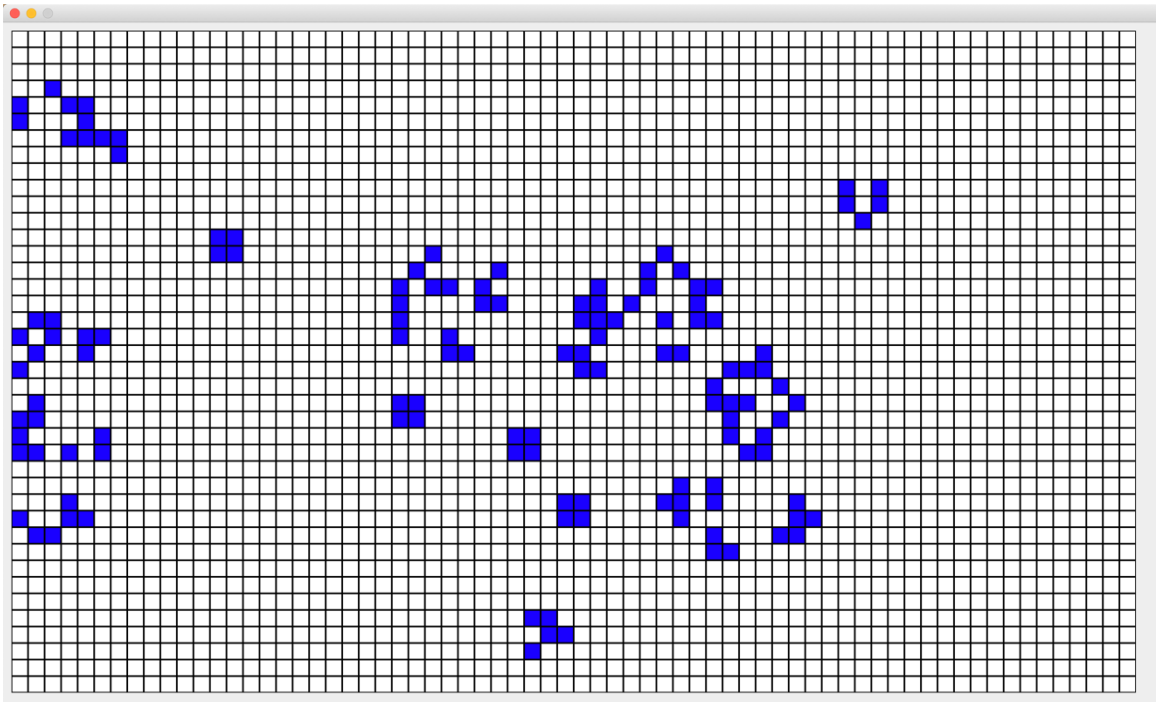
Finally, when we got the node coordinates for each target cell, we need to get that point by calculating gui.curY * gui.MAP_WIDTH + gui.curX, which is the index value in the labels object. In order to create a delayed display of each generation, we added the Thread.sleep function for interval display, the amount of time depended on how fast we want to display.

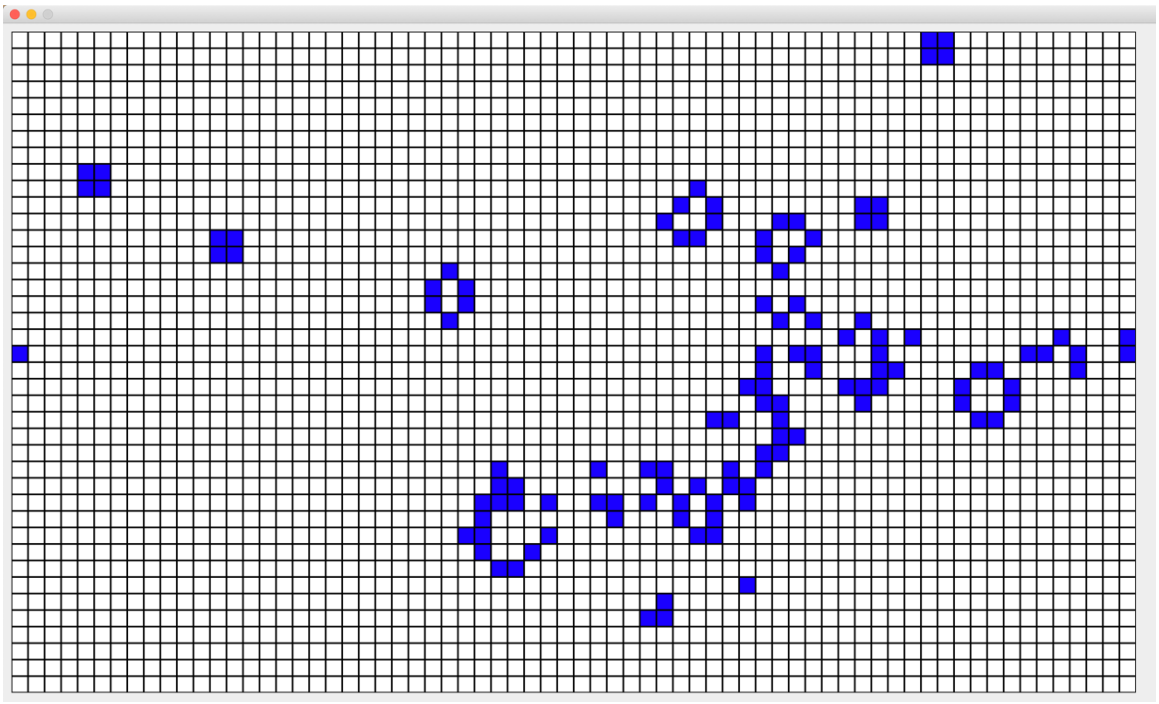
5. Result

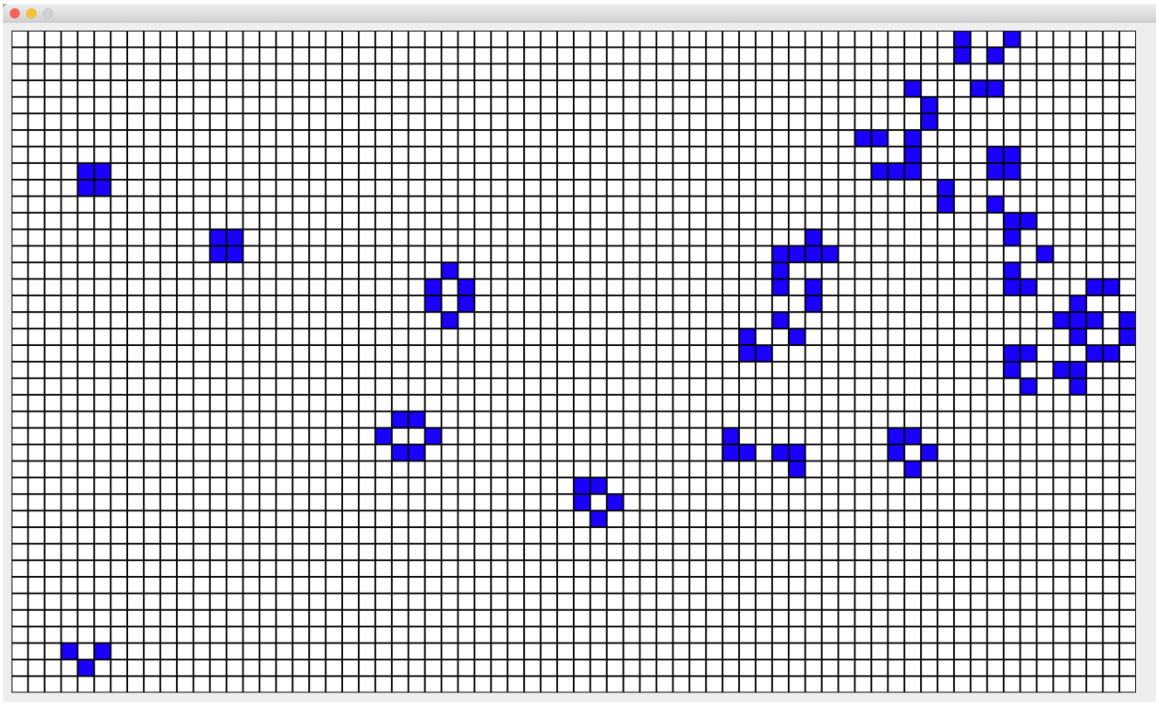


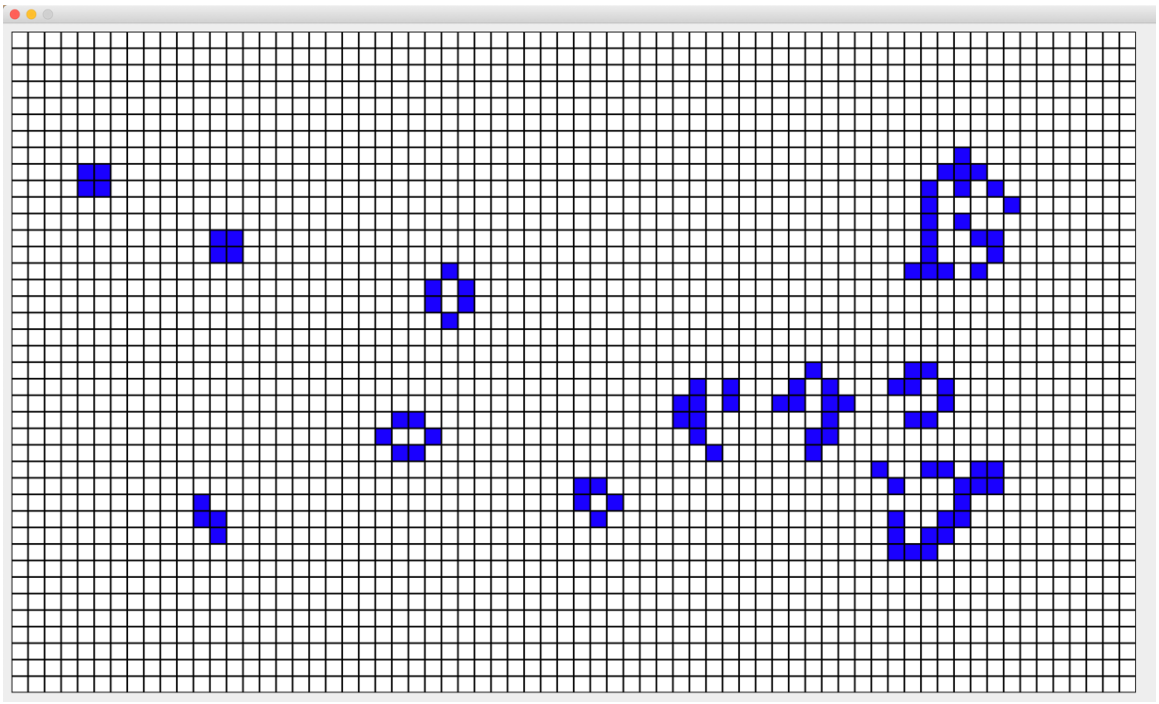


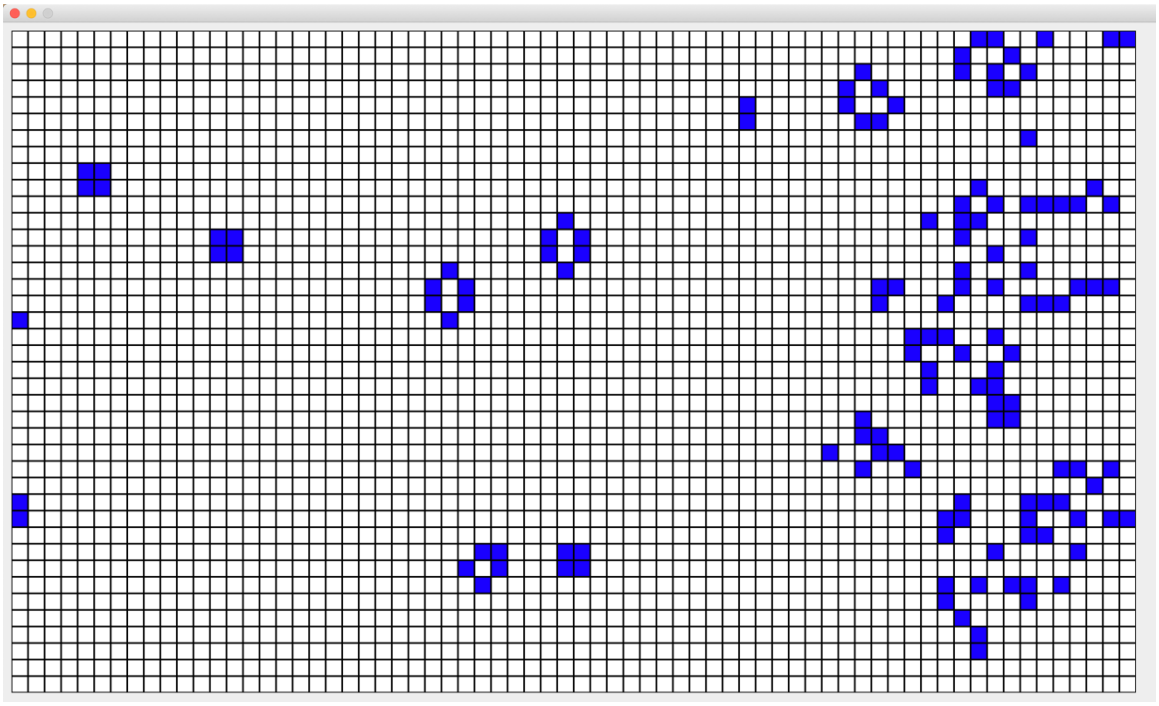












6. Conclusion

Throughout this project, we applied a genetic algorithm to obtain the optimized initial point and initial pattern for running the Game of Life. We can successfully find the most suitable pattern as our starting value to run the generation as long as possible.

7. Unit Test

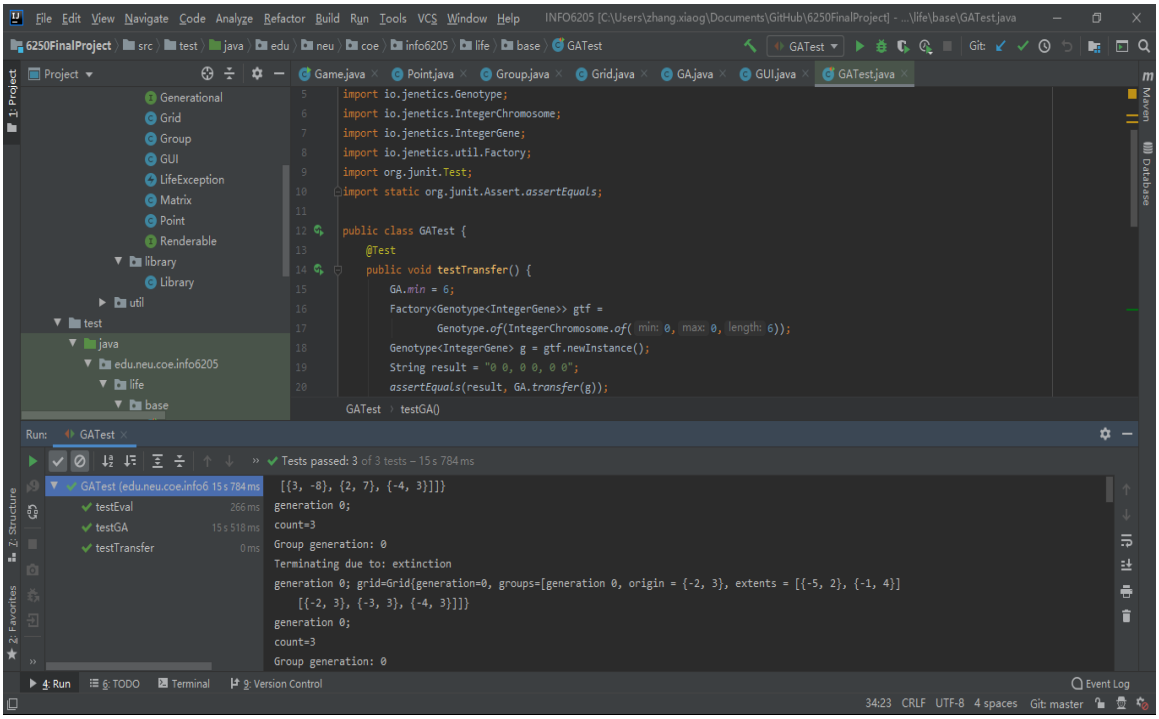
The Unit Test part is used to check whether the transfer function, eval function and ga functions (also including the mutation function) can be accurately executed. We gave them some initial value for running, the following is the result.

```java
@Test
public void testTransfer() {
    GA.min = 6;
    Factory<Genotype<IntegerGene>> gtf =
            Genotype.of(IntegerChromosome.of( min: 0, max: 0, length: 6));
    Genotype<IntegerGene> g = gtf.newInstance();
    String result = "0 0, 0 0, 0 0";
    assertEquals(result, GA.transfer(g));
}

@Test
public void testEval(){
    GA.min = 10;
    Factory<Genotype<IntegerGene>> gtf =
            Genotype.of(IntegerChromosome.of( min: -2, max: 2,  length: 10));
    Genotype<IntegerGene> g = gtf.newInstance();
    String test = GA.transfer(g);
    assertEquals(Game.run( generation: 0L, test).generation, GA.eval(g));
}

@Test
public void testGA(){
    GA.min = 6;
    long result = 4L;
    long test = GA.geneticAlgorithm( test: true);
    assertEquals(result, test);
}
```

7 - 1 Unit Test Code



7 - 2 Unit Test Result