Project Report

INFO 6205 Team 207

Yezhi Miao 001401943 Ran Zhou 001448551

1. Background Introduction

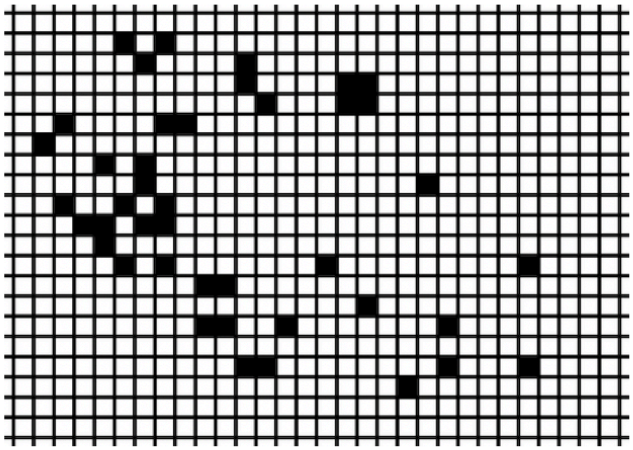
1.1 Game of Life

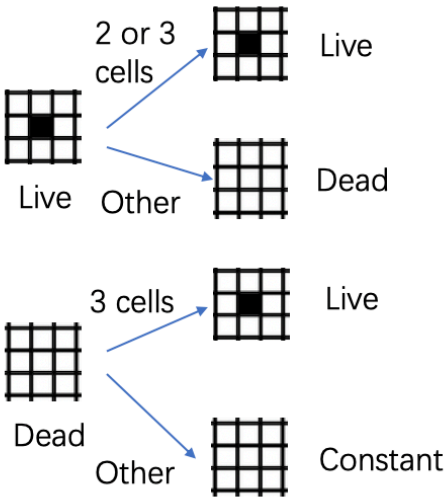
Following the guideline of project, we are required to implement Game of Life which is a cells' surviving model. Given an unlimited grid with cells, each cell has an initial state live or dead and they breed in next generation.

There are two rules for cells to survive:

- Each "live" cell will die if it has fewer than two or more than three neighbors
- Each "dead" cell will come alive if it has exactly three neighbors.

Clearly declare in following graph:





We have a starting pattern for the game which declares live cells and their position in the grid, there two conditions when the game running for a period of time: all cells dead and some cells lasting for unlimited generations.

1.2 Genetic Algorithm

Genetic Algorithm is a popular method to generate high-quality solutions to optimization and search better solutions by using some concepts of evolution in biology such as mutation, crossover and selection.

2. Problem Statement

In this project, we used Genetic Algorithm to find a best starting pattern which can make Game of Life last longer. The starting pattern is a string containing some coordinates.

3. Solutions

3.1 Basic Concepts of Genetic Algorithm

- **Gene**
  Gene is a basic element. In our project, we define a class named Gene, every instance of gene can have a value of Boolean[3 OR 4].

```
public class Gene {
    public static final int GENE_LENGTH = 4;
    private boolean[] gene;
    private double mutationRate = 0.01;



    //constructor for initializing the boolean[], totally random 0 OR 1
    public Gene(){
        gene = new boolean[GENE_LENGTH];
        init();
    }

    public void init(){
        for (int i = 0; i < GENE_LENGTH; i++) {
            gene[i] = Math.random() >= 0.5;
        }
    }
```

  In constructor, we use random figure to generate boolean[]. So for generation 0, all gene are totally randomly. The probabilities of generating 0 OR 1 are the same. The reason that we use boolean[] is that we all know that 0 represents false and 1 represents true.

  Each gene is half of coordinates(x coordinates OR y coordinates).

- **Genotype**
  Actually, gene is elements of genotype. So, genotype is all the binary number of an individual, consisting of many Gene instances.

- **Phenotype**
  In this case, phenotype is the starting pattern, which is translated from binary number to String.
  Each Gene(half coordinate) can be turned from a binary number to a int value, as a result, an individual's genotype can be converted into a int value. We use these int values to form a String according to the required format in Class Library.

- **Chromosome**
  We defined that an individual has a chromosome, each chromosome has 10 Gene[]. The number of Gene[] can be changed. Chromosome includes genotype and phenotype.

```java
public class Chromosome {
    public static final int numOfGene = 8;
    private Gene[] genotype;
    private String phenotype;
    private double score;

    public Chromosome(){

    }

    //constructor for initializing Gene[]
    public Chromosome(int size){
        initGeneNum(size);
        for(int i = 0; i < size; i++){
            Gene gene = new Gene();
            genotype[i] = gene;
        }

    }
```

- **Fitness**
  In our project, fitness is the how long the candidate can survive. In addition, we can also use the rate of growth to calculate the fitness:

```java
@Override
public double calculateFitness(String phenotype) {
    // TODO Auto-generated method stub
    Game game = new Game();
    Game.Behavior g = game.start(phenotype);
    double score = g.generation * g.growth;
    return score;
}
```

3.2 Genetic Algorithm Implement

3.2.1 Step1

First, randomly initialize a population of 100, 0 or 1. Each individual in the population has a chromosome with 10 gene fragments. We use a boolean [] to represent the gene fragment. Each gene segment represents an x / y coordinate. So 10 gene fragments represent 5 initial coordinates.

3.2.2 Step2

Each individual in the population calls Game to calculate fitness. We use the product of surviving generations and growth rates as an indicator of fitness. The first population then evolved for the first time. Each individual has a 0.01 chance of a genetic mutation at each locus. We use Math.random to generate a random number. When less than 0.01, the gene mutates. The principle of mutation is that 0 becomes 1, and 1 becomes 0.

3.2.3 Step3

In addition to mutation, we also use crossover. The principle of crossover is to use roulette to select two chromosomal gene fragments in the population whose fitness is higher than the average. The swapped fragments are also generated from random numbers. It should be noted that in our actual operation, a string with duplicate coordinates may be generated. We should try to avoid selecting individuals with this phenotype. In the process of generating the next generation, individuals of the current population may be combined with individuals of populations elsewhere.

3.2.4 Step4

The population evolved until the 500 generations stopped. Then the program ends and the console outputs the starting pattern of the maximum fitness achieved in 500 iterations. The population evolved until the 500 generations stopped. Then the program ends, the console outputs the starting pattern and fitness of the maximum fitness achieved in 500 iterations, and the population algebra of the current starting pattern.

3.3 Best Result

We find our best solution, a string with 5 coordinates is : “2 1, 0 2, 1 0, 3 3, 3 1”
Here is the result:

```
count=116
Group generation: 1997
generation 1998; grid=Grid{generation=1998, groups=[generation 1998, origin = {-21, -15}, extents = [{-444, -484}, {505, 489}]
    [{0, 0}, {1, 0}, {1, -1}, {0, -1}, {-19, 5}, {-17, 5}, {-18, 4}, {-19, 4}, {-18, 6}, {-17, 6}, {-7, -4}, {-8, -4}, {-8, -5}, {-7, -5}, {13, 2}, {15, 3}, {13, 3}, {
generation 1998;
count=116
Group generation: 1998
generation 1999; grid=Grid{generation=1999, groups=[generation 1999, origin = {-21, -15}, extents = [{-444, -484}, {506, 490}]
    [{0, 0}, {1, 0}, {1, -1}, {0, -1}, {-19, 5}, {-17, 5}, {-18, 4}, {-19, 4}, {-18, 6}, {-17, 6}, {-7, -4}, {-8, -4}, {-8, -5}, {-7, -5}, {13, 2}, {15, 3}, {13, 3}, {
generation 1999;
count=116
Group generation: 1999
Terminating due to: having exceeded 2000 generations
Ending Game of Life after Behavior{generation=2000, growth=0.0555, reason=2} generations
```

```
the worst fitness is:-5.0
the average fitness is:-0.26
the total fitness is:-26.0
geneI:266   Starting pattern:2 1, 0 2, 1 0, 3 3, 3 1    y:159.0
```

In this result, the worst fitness is -5.0, average fitness calculated as -0.26, and total fitness is -26.0. The fitness is the less the better, we run the code for many times and try to find the relatively low value.

This starting pattern runs beyond 2000 generations if we set the max generation as 2000.

3.4 Seed of This Result

The seed of this result(the generation0 of population) is:

3 3, 0 0, 1 0, 1 3, 3 2
0 -4, 3 3, -3 -4, -4 3, 2 0
-4 -4, 3 -3, 3 -4, -3 0, 1 3
-4 3, -3 -3, -4 3, 1 2, -4 1
2 3, -4 -4, 3 -4, 1 2, -4 3
3 -4, 0 -4, 2 -3, 0 3, 3 1
3 -4, 3 -4, 1 1, 1 -3, -4 3

-2 1, 1 1, 3 -4, 2 -2, 2 3
3 -2, -2 0, 3 -2, -4 1, 0 -3
3 -4, 3 0, 3 -4, 3 1, -3 3
2 3, -4 -4, 1 -3, 3 0, 1 2
-2 -3, 1 3, 2 3, 3 1, 2 1
3 -4, 0 -4, 1 0, -3 -4, 3 -3
0 3, 0 1, 3 0, 0 2, -2 0
3 1, -4 2, -4 3, 3 3, 2 1
-4 2, 3 3, 3 0, 3 3, 1 0
3 1, 0 -4, 1 2, 1 -2, 1 -2
-4 3, 3 3, 3 0, 1 -4, -3 -3
-4 3, 3 3, 2 -3, -4 3, 2 2
2 0, 0 0, 2 3, 3 -3, 0 2
2 3, 3 3, -4 3, -4 3, 3 3
-4 2, 3 1, -2 3, 3 2, 1 -3
0 3, -2 0, -3 3, 0 1, 3 -3
1 -4, -3 3, 1 2, -3 3, -3 3
3 3, 0 3, 1 -4, -2 0, 0 -3
-3 0, 3 -3, 1 -3, 1 0, 0 3
0 -4, 3 2, -4 3, 0 -3, 2 2
2 3, 2 3, 1 1, 3 0, 0 1
1 -3, -4 -4, -4 -3, -4 3, 3 -4
-3 1, 3 -2, 2 0, -4 -3, -4 3
3 0, 3 -4, -4 -2, -4 -4, 3 1
3 -3, 2 3, -3 -3, 3 1, 3 3
1 -2, -3 -3, 3 -2, -3 -3, 1 -3
-2 -4, 3 0, -2 -3, 1 3, 1 -4
-4 0, 1 3, 1 3, -4 -3, -2 2
3 -4, 2 -4, 3 1, -4 3, 2 -3
-2 1, 0 1, -3 3, 1 3, -2 3
-2 3, 2 -2, 3 -3, 0 3, 2 -2
3 2, 3 -4, -2 2, 1 0, 2 1
-4 1, -3 3, 3 -4, -4 1, -2 -3
-2 3, 2 1, 3 2, -4 2, 3 3
2 0, -3 0, -4 3, -3 -3, -4 -3
-4 -4, 2 -3, 3 2, 3 -3, 3 3
0 3, 3 1, 1 1, 3 3, 3 -2
3 3, 3 3, -3 -3, -2 -4, 0 0
0 -3, -4 -3, -4 1, -2 3, 2 -3
3 3, 3 -3, 1 3, 3 3, 3 2
-2 0, 2 -4, -4 3, 0 3, 3 -2
-2 -2, 3 3, -2 -2, 1 0, 2 -3
1 3, -3 -4, 3 0, -2 3, -3 -2
-4 0, -2 3, 0 3, -3 -3, -4 2
3 3, -3 3, 3 1, 3 3, -3 -3
-2 0, 2 0, 0 0, 3 -4, 1 -4

2 -3, 2 1, -2 0, 0 0, 1 1
0 1, 2 0, 0 -4, 2 -2, 1 3
3 -4, 0 -2, 2 3, 1 -3, 0 2
-4 2, -3 0, 3 1, 1 -4, 3 2
3 3, -3 3, -4 -4, 3 -3, 3 3
-2 1, 3 3, 0 0, 1 2, -2 -3
1 3, -3 -2, -3 3, -3 0, -2 2
3 -3, 3 0, -4 3, -3 2, 0 2
-2 -2, -2 2, 1 3, 3 2, 3 2
1 3, 0 1, 0 1, 3 0, -2 3
-3 2, -2 0, 3 -4, -4 -4, -4 3
0 -2, 1 -3, -3 2, 3 3, 2 2
0 2, 1 -4, -2 1, 1 2, 2 -3
3 -3, 1 -4, 0 -3, -3 0, 3 2
0 3, 0 3, -4 3, 2 2, -4 2
3 0, -4 3, 3 3, 3 -4, 1 1
1 2, -4 -3, -3 -4, -4 -4, -3 -3
1 1, 1 3, 3 3, 3 3, -4 3
2 -4, -4 0, -4 -3, 2 3, -4 3
-4 2, 1 1, 3 -2, 0 3, 1 2
0 -4, 3 -4, -4 2, 3 -3, 1 3
-4 2, -2 -3, 3 1, -3 2, 3 -3
2 2, -2 3, 1 -2, 2 2, 0 -2
3 -3, -3 -3, 1 2, -3 -4, 1 -4
0 -4, -2 0, 2 -2, 0 -4, -3 -3
1 1, 3 3, 0 2, -2 -4, 0 0
-3 3, 1 1, 3 -2, 2 0, -2 0
-4 1, 3 3, 1 3, 3 -3, -3 3
2 -2, 3 3, -4 3, 2 3, 3 2
1 3, 3 -2, 3 3, -2 2, -3 1
2 -3, 2 -4, 1 -2, -2 3, 1 3
-3 3, 0 -4, -3 -3, 1 3, -4 -2
-3 0, 3 -3, -4 3, 3 1, -3 1
-2 3, 3 2, 0 1, 2 2, 1 -4
2 -3, 0 3, 3 0, 2 3, 3 3
1 2, -3 -4, -3 3, 3 -3, 1 3
-4 2, 3 -4, 3 3, -3 3, 1 3
3 -4, 2 2, 3 3, -2 1, 3 -3
3 3, -3 2, 0 0, 2 -4, -2 0
0 2, -4 -4, 3 -4, 2 2, 1 3
1 3, -4 -3, 2 1, 0 -3, 0 0
3 2, -3 0, -4 3, 3 -4, 2 -4
-3 2, -4 1, 0 2, -3 2, 2 1
-3 -2, -3 1, 2 -2, 0 -2, -2 0
2 3, -4 -2, -2 -3, 2 3, 0 -4
-4 3, 1 0, 1 3, 1 -3, 0 3

1 3, -4 -3, 1 2, 3 0, -2 3

3.5 Framework of Detailed Implementation

Class Gene:

```
public class Gene {
    public static final int GENE_LENGTH = 3;
    private boolean[] gene;
    private double mutationRate = 0.01;

    //constructor for initializing the boolean[], totally random 0 OR 1
    public Gene(){
        gene = new boolean[GENE_LENGTH];
        init();
    }

    public void init(){
        for (int i = 0; i < GENE_LENGTH; i++) {
            gene[i] = Math.random() >= 0.5;
        }
    }

    //get int value of boolean[] gene,convert binary to int
    public int getNum() {}

    //if random figure < mutation Rate, 0 to 1 OR 1 to 0$
    public void mutation() {}

    public boolean[] getGene(){}

    public void setGene(boolean[] b){}
}
```

Class Chromosme:

```
public class Chromosome {
    public static final int numOfGene = 8;
    private Gene[] genotype;
    private String phenotype;
    private double score;

    public Chromosome(){}

    //constructor for initializing Gene[]
    public Chromosome(int size){}

    public void initGeneNum(int size){}
```

```
    //clone gene for generate next generation
    public static Chromosome clone(final Chromosome c) {}

    //let two Chromosome exchange part of gene
    public static List<Chromosome> genetic(Chromosome p1, Chromosome p2) {}


    //every gene bit has probability to mutate
    public void mutation(){}


    //get phenotype
    public String getPhenotype(){}


    //score is the fitness of candidate
    public double getScore() {}

    public void setScore(double score) {}


    //check it phenotype has duplicates, duplicates can cause some error in Class Game, some
duplicates can last forever
    public boolean ifDuplicate(){}

    public Gene[] getGenotype() {}
}
```

Class GeneticAlgorithm:

```
public abstract class GeneticAlgorithm {

    private List<Chromosome> population = new ArrayList<>();//种群
    private List<Chromosome> farPopulation = new ArrayList<>();
    private int popSize = 100;//size of pupulation
    private int geneSize;//max length of gene
    private int maxIterNum = 500;//max number of evolution
    private double mutationRate = 0.01;//probability of mutation
    private int maxMutationNum = 3;//max length of mutation
    private int generation = 0;//current generation of population

    private double bestScore;//best fitness
    private double worstScore;//worst fitness
    private double totalScore;//total fitness
    private double averageScore;//average fitness
```

```
private String x; //best  solution(phenotype) in all population
private double y; //best fitness
private int geneI;//the generation of population which contain x & y
private List<Chromosome> generation0 = new ArrayList<>();//save seed


public GeneticAlgorithm(int geneSize) {}

public void init() {}

   //calculate each candidate's fitness and get best,worst average score
private void calculateScore() {}

private void setChromosomeScore(Chromosome c) {}

//get phenotype of a chromosome
   public abstract String changeX(Chromosome c);

   //using abstract method to calculate candidate
public abstract double calculateFitness(String phenotype);

//select Chromosome with hight fithness
   public Chromosome selectChromosome(){}

//generate a population with the same size
   private void generateagain(){}

//the generation evolve by selecting and mutation
   private void evolve() {}

private void mutation() {}

public void caculte() {}

//print information of best worst ave score and best solution
   private void print() {}

public void setPopulation(List<Chromosome> population) {}

public void setPopSize(int popSize) {}

public void setGeneSize(int geneSize) {}

public void setMaxIterNum(int maxIterNum) {}

public void setMutationRate(double mutationRate) {}
```

```
    public void setMaxMutationNum(int maxMutationNum) {}

    public double getBestScore() {}

    public double getWorstScore() {}

    public double getTotalScore() {}

    public double getAverageScore() {}

    public String getX() {}

    public double getY() {}

    public void setBestScore(double bestScore) {}

    public void setWorstScore(double worstScore) {}

    public void setTotalScore(double totalScore) {}

    public void setAverageScore(double averageScore) {}

    public List<Chromosome> getPopulation() {}
}
```

Class Driver:

```
public class Driver extends GeneticAlgorithm {

    public static final int NUM = 1 << 10;

    public Driver() {
        super(10);
    }

    @Override
    public String changeX(Chromosome chromosome){}

    @Override
    public double calculateFitness(String phenotype) {
        // TODO Auto-generated method stub
        Game game = new Game();
        Game.Behavior g = game.start(phenotype);
        double score = g.generation * g.growth;
        return score;
    }
```

```
    public static void main(String[] args) {
        Driver test = new Driver();
        test.caculte();



    }
}
```

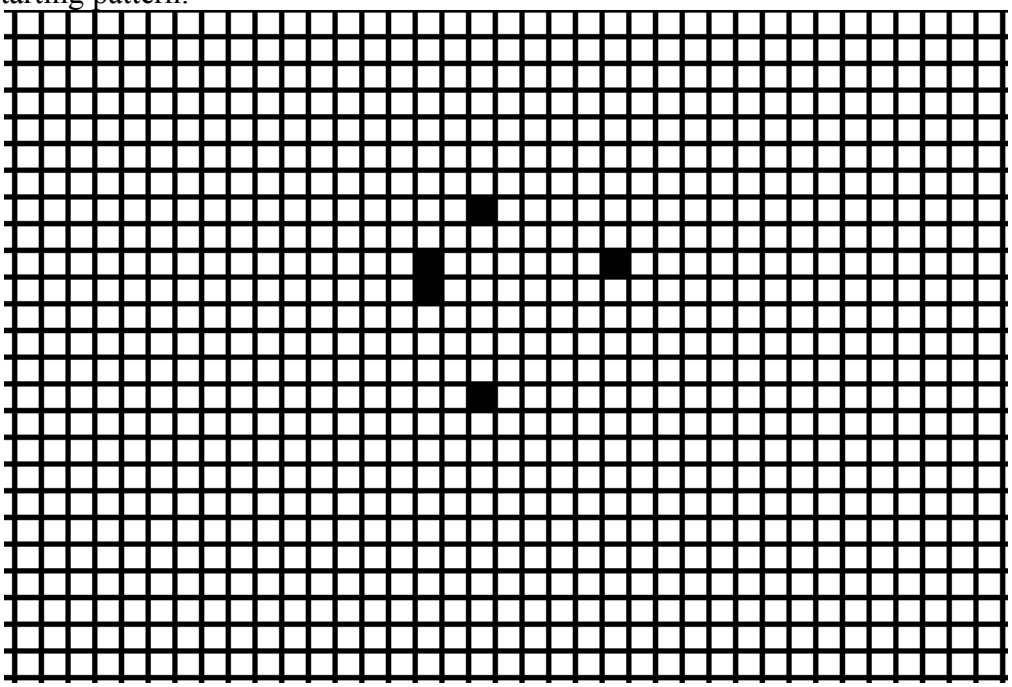
This class has main method and we should run this class.

3.6 The Genetic Algorithm UI

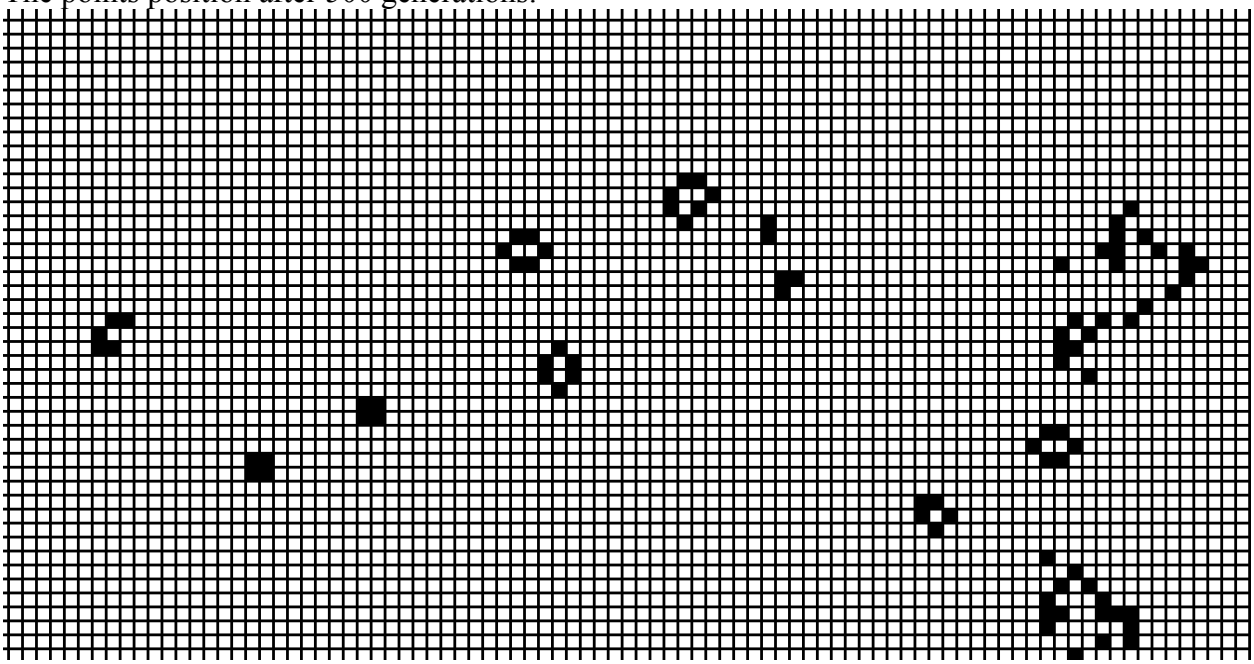
We use Java Swing package to implement UI part in our project. We show the all points in an limited grid, because our gird in this project is unlimited, but it's hard to show all coordination in the screen, so we ignore the coordination out of the bound and make sure the programming keeping running.

In our UI frame, we demonstrate all coordination showing positions in our 500 times Genetic Algorithm iteration process. Because Here is an example of the UI graph:

The starting pattern:



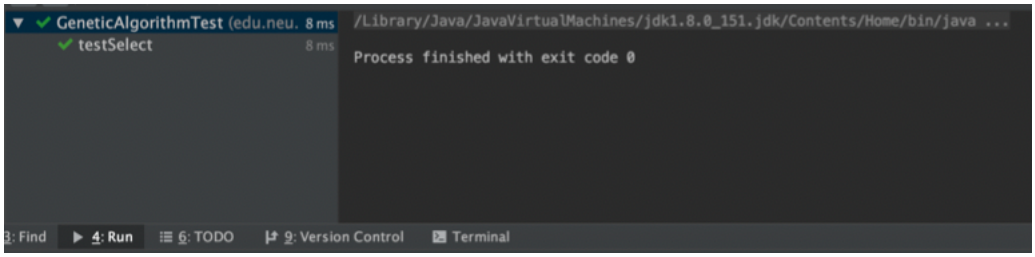
The points position after 500 generations:

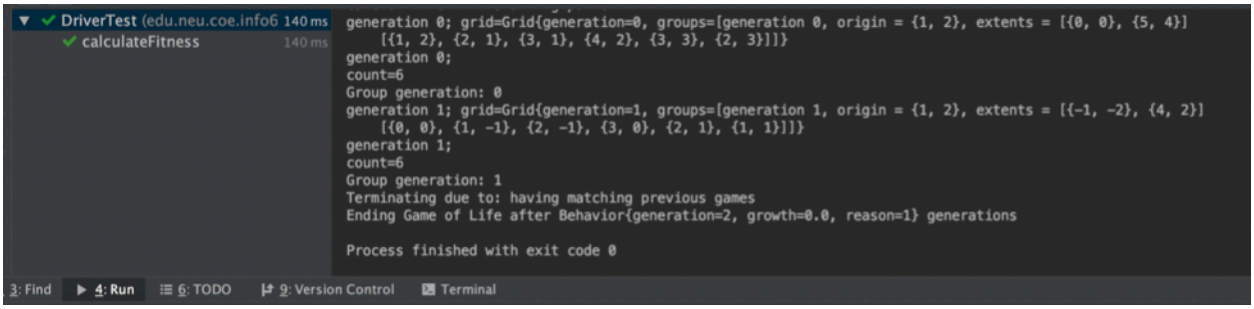


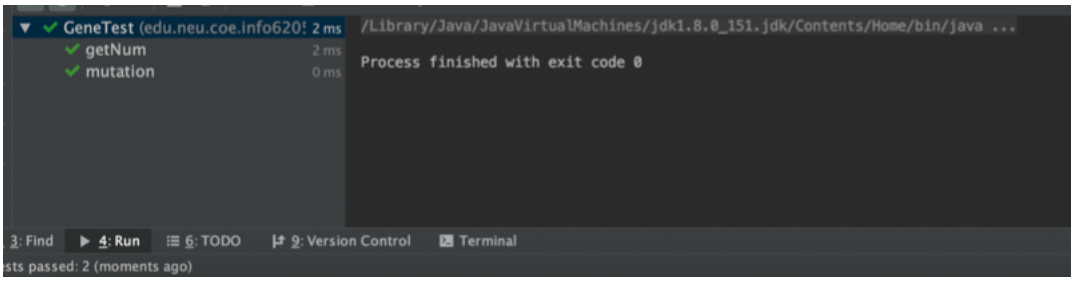
4. Conclusion

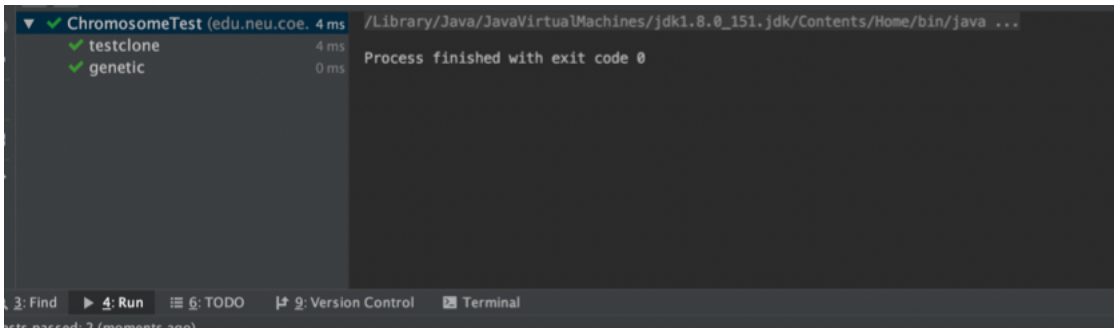
After running the genetic algorithm programming, we can find the local optimal solution as our best starting pattern to make the game of life last for many generations beyond 999 generations.

5. Unit Test






GeneTest (edu.neu.coe.info620! 2 ms
getNum 2 ms
mutation 0 ms
/Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/bin/java ...
Process finished with exit code 0
3: Find
4: Run
6: TODO
9: Version Control
Terminal
sts passed: 2 (moments ago)


ChromosomeTest (edu.neu.coe. 4 ms
testclone 4 ms
genetic 0 ms
/Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/bin/java ...
Process finished with exit code 0
3: Find
4: Run
6: TODO
9: Version Control
Terminal