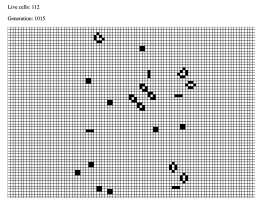
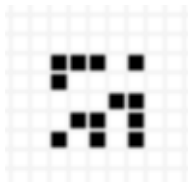


Conway's Game of Life (GA)

Part 1. Conclusion and Proof of Random Generation



We found a pattern that could expand to 112 nodes from 13 nodes, after 1000 generation of Conway's game of life. We thought this is a great pattern that worth to take a test like others. And we put it in a more larger canvas. We found out that after around 800 generations, it turns to follow a specific growth rule like the third picture. Based on our investigation, we thought we could think this model as the model which could infinite growth.

Although it seems like we have gave too much constrains to the scale of the pattern. But actually, we tried from 100*100 to 5*5 for a long range of times. We have run several times and took a long time to get this conclusion and tring to deducted the scale of the original canvas is the method we use to make the things easier. Before we tried the 5*5, we get some patterns more bigger and get some good values, but they didn't show a sign of infinite. So, we try to make this more likely happened. So, we investigate some papers and found that the 5*5 also worked, that why we get this small scale of the pattern.

Part 2. Proof of Using of Generic Algorithm

We created our own genetic algorithm, in which we implemented the fitness function. And also, we did the mutation and the crossover functions based on the fitness evaluation of each elements in the population.

Our main argument shows below:

* Number of population: 10

To get a better performance, we set the population as 10.

```
GameOfLife life = new GameOfLife();  
life.geneticAlgorithm(desiredPop: 10);
```

* The max generation of a single game : 500-1000

During the experiments, we found that when the max generation set to 500 (or smaller that 500), the output are too optimistic, when we enlarge the generation to 1000, the output data will be

not that good but more trusted. But in order to keep the efficiency, we choose the 1000 which is in a more stable level for the max generation.

```
private void setUp(int [][] array)
{
    // fill the current board with 0's
    fillBoard(array);
    box = new int [30][30];

    randomStart(box);

    // used in putting the 30*30 middle box
    // into the 1000*1000
    int spot = (array.length / 2) - 15;

    for (int i = spot; i < spot + 30; i++)
    {
        for (int j = spot; j < spot + 30; j++)
        {
            array[i][j] = box[i-spot][j-spot];
        }
    }
}
```

* The initial pattern scale: from 100*100 to 5*5

During our experiments and based on the investigation, we try to control the scale of the pattern's canvas, in which way we could get the more confirmed output as soon as possible. Cuz if we just focus on a 100*100 scale, there is less chance to find a worked pattern way more smaller, which will waste bunch of times.

* The Monitoring range (main canvas) : set dead board to 100*100 to 1000*1000

During the experiments we found that the infinite growth pattern usually not stay on where it started, so that we need to monitor a larger scale. Also, in order to make sure the high efficiency, we choose to control the canvas within 100*100 and within 1000*1000 to monitor.

Part 3. Fitness Function

In our Fitness Function, we calculated the growth rate (float num) of the endLiveCell (which is the number of cells in latest pattern when it's died) and the startLiveCells (which is the number of cells in the first pattern when it born). As we mentioned above, when one pattern approached the max generation, it would stop. So the max end live cells number is the max number when it died naturally or the max number when it grow enough generation for an infinite pattern.

```
float num = endLiveCells / ((float)2.0 * startLiveCells);
fitnessVals[j] = num;
```

Part 4. Genotype and Phenotype

(A) Genotype:

We built a 2D “**matrix**” (int[][]) to represent the genotype in the GA algorithm.

(B) Phenotype:

For the phenotype, it's the expression of the matrix, which is a “**pattern**”. In a patter, we express the matrix of genotype on a canvas and add transforming function on it.

And also, phenotype has **the method of get Fitness score** of a specific genotype.

Part 5. Crossover and Mutation

(A) Crossover:

The crossover process includes two parts, select 2 parents for each child in the population and crossover two parents generate the

* Choose the parents

We have 10 population in each generation. When the crossover function is called, we get the fitness ranking of each game in the parent population. And we built a selection mechanism in which we pick up the better parents of the current population to “Crossover” and get the children to replace the bad one.

```
ArrayList<int []> populationWithWeight = getWeightedPopulation(fitnessVals);

for(int i=0; i< population.size(); i++){
    Random r = new Random();
    int pick = r.nextInt( bound: 100);
    int pick2 = r.nextInt( bound: 100);
    population.remove(i);
    population.add(crossOver(populationWithWeight.get(pick),populationWithWeight.get(pick2)));
}
```

We choose the parent games by the the `getWeightedPopulation()` function, in which we generate a new `ArrayList` to save those parents by different weight (1st one get 25%, 2nd one get 20, 3rd one get 15%, 4th one get 10%, 5th one get 8, 6th one get 7, and so on). In this way, when we pick up the parents game, it will follow the rule we want that the better one get the more chance to be selected.

```
459
460
461 /**
462  * According the fitness array, return a weighted population
463  * |
464  * @param fitnessVals
465  * @return
466  */
467 public ArrayList<int []> getWeightedPopulation(float[] fitnessVals) {
468     ArrayList<int []> weithedP = new ArrayList<>();
469
470     float totalFitness = 0;
471
472     int[] possibilites = new int [fitnessVals.length];
473
474     for (float fitness: fitnessVals) {
475         totalFitness += fitness;
476     }
477
478     for (int i = 0; i < fitnessVals.length; i++) {
479         possibilites[i] = (int) (fitnessVals[i] / totalFitness * 100);
480     }
481
482     for (int i = 0; i < possibilites.length; i++) {
483         for (int j = 0; j < possibilites[i]; j++) {
484             weithedP.add(population.get(findDesiredFit(fitnessVals, i + 1)));
485         }
486     }
487
488     return weithedP;
489 }
490
491
492
493 }
```

* Crosseover Function:

In the crossover function, we crossover the two parents (original pattern) based on the mechanism that if the two nodes in one specific position in two parents pattern are the same, the node value in this position will set to 0; else if they are not same, we change the value to 1.

```
/**
 * perform a crossover over with the two parents
 * and then return their child
 *
 * @param parent1
 * @param parent2
 * @return
 */
private int [][] crossover(int [][] parent1, int [][] parent2)
{
    int [][] child = new int [1000][1000];

    int spot = (parent1.length / 2) - 2;
    for (int row = 0; row < parent1.length; row ++)
    {
        for (int column = 0; column < parent2.length; column ++)
        {
            // use XOR
            if (parent1[row][column] == parent2[row][column])
            {
                child[row][column] = 0;
            }
            else if (parent1[row][column] != parent2[row][column])
            {
                child[row][column] = 1;
            }
        }
    }
}
```

(B) Mutation:

As shown in the former getWeightedPopulation() function, we called the mutate() function every time we got the child node to guarantee the efficiency of the GA algorithm.

In the mutate() function, we set 1% of the chance to mutate the pattern (child just generated). If this function be triggered, the every node in the pattern will set from 1 to 0 and from 0 to 1.

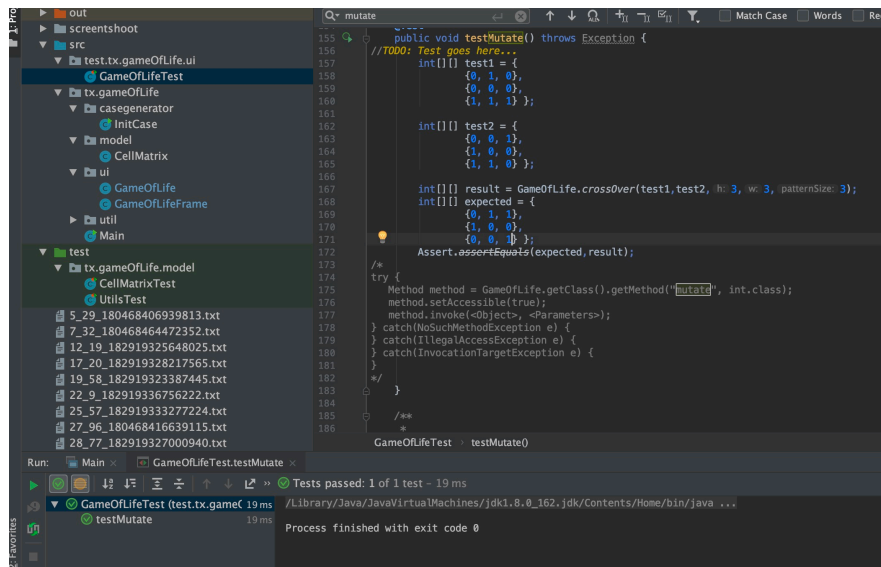
```
{ boolean val = rand.nextInt( bound 101) < 1;
  if (val && !atBorder(original, row, column))
  {
      if (original[row][column] == 0)
      {
          mutatedArray[row][column] = 1;
      }
      else if (original[row][column] == 1)
      {
          mutatedArray[row][column] = 0;
      }
  }
  else
  {
      mutatedArray[row][column] = original[row][column];
  }
}
```

Part 6. Unit Test

(A) 1.Expression function (assuming that genotype != phenotype)

Here we tested the function that the phenotype (the change of the genotype(matrix)) during the Conway's game of life runtime.

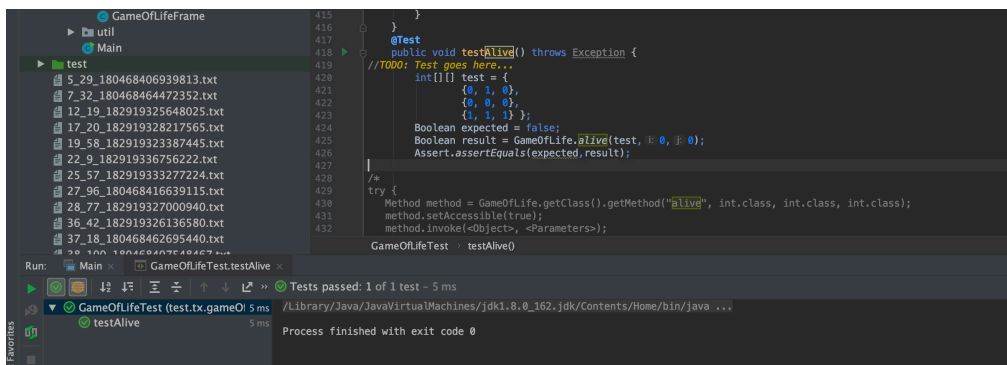
(C) Genetic Algorithm - Mutation function (and/or crossover if used)



```
155 public void testMutate() throws Exception {
156     //TODO: Test goes here...
157     int[][] test1 = {
158         {0, 1, 0},
159         {0, 0, 0},
160         {1, 1, 1} };
161
162     int[][] test2 = {
163         {0, 0, 1},
164         {1, 0, 0},
165         {1, 1, 0} };
166
167     int[][] result = GameOfLife.crossover(test1, test2, 3, 3, patternSize: 3);
168     int[][] expected = {
169         {0, 1, 1},
170         {1, 0, 0},
171         {0, 0, 0} };
172     Assert.assertEquals(expected, result);
173 }
174
175 /*
176 try {
177     Method method = GameOfLife.getClass().getMethod("mutate", int.class);
178     method.setAccessible(true);
179     method.invoke(<Object>, <Parameters>);
180 } catch (NoSuchMethodException e) {
181 } catch (IllegalAccessException e) {
182 } catch (InvocationTargetException e) {
183 }
184 */
185 }
186
187 GameOfLifeTest - testMutate()
```

Run: Main x GameOfLifeTest.testMutate x
Tests passed: 1 of 1 test - 19 ms
GameOfLifeTest (test.tx.gameOfLife) 19 ms
testMutate 19 ms
Process finished with exit code 0

(D) Candidate selection function



```
415 }
416
417 @Test
418 public void testAlive() throws Exception {
419     //TODO: Test goes here...
420     int[][] test = {
421         {0, 1, 0},
422         {0, 0, 0},
423         {1, 1, 1} };
424     Boolean expected = false;
425     Boolean result = GameOfLife.alive(test, 0, 0, 0);
426     Assert.assertEquals(expected, result);
427 }
428
429 /*
430 try {
431     Method method = GameOfLife.getClass().getMethod("alive", int.class, int.class, int.class);
432     method.setAccessible(true);
433     method.invoke(<Object>, <Parameters>);
434 }
435 */
436 }
437
438 GameOfLifeTest - testAlive()
```

Run: Main x GameOfLifeTest.testAlive x
Tests passed: 1 of 1 test - 5 ms
GameOfLifeTest (test.tx.gameOfLife) 5 ms
testAlive 5 ms
Process finished with exit code 0

Part 7. UI

Since when we do the experiment, we make the canvas to 1000*1000, so that the java swing could not show the best output since the refresh requirement is too much. So that we decided to output the matrix into a txt file and write a js program to run it in the web browser, in which way we get a better performance. Our output is shown below:

Live cells: 78

Generation: 167

