

# Optimizing Solution of Conway's Game of Life Based on Genetic Algorithm

Kai Zhang, Bochao Wang

Northeastern University(NEU)

Email: [zhang.kai2@husky.neu.edu](mailto:zhang.kai2@husky.neu.edu), [wang.boch@husky.neu.edu](mailto:wang.boch@husky.neu.edu)

**Abstract** – To find a start **pattern** for **Conway's Game of Life**, which could have more **generations**, is always an interesting problem. While traditional solution is generate random pattern and test them one by one, it's not efficient and hard to find a good solution.

In this report, we propose an optimized solution to find a started pattern for Game of Life, which is based on **Genetic Algorithm**. Instead of generating random pattern to test, we generate different **genotypes** and **express** the genotypes to **phenotypes**. And by testing them in the same **environment**, we select those who get better **fitness**, make them **survive** and have **offspring**. Finally in the environment there would exist the best solution we want. Our evaluation result shows that with our proposal, it is much more easier to get a pattern with more generations.

## I. INTRODUCTION & BACKGROUND

In this section we briefly present different concepts and categories of state-of-the-art related to our proposed approach.

### A. Conway's Game of Life

The Game of Life is a cellular-automaton, zero player game, developed by John Conway in 1970. The game is played on an infinite grid of square cells, and its evolution is only determined by its initial state.

The rules of the game are simple, and describe the evolution of the grid:

▲ Birth: a cell that is dead at time  $t$  will be alive at time  $t + 1$  if exactly 3 of its eight neighbors were alive at time  $t$ .

▲ Death: a cell can die by:

▲ Overcrowding: if a cell is alive at time  $t + 1$  and 4 or more of its neighbors are also alive at time  $t$ , the cell will be dead at time  $t + 1$ .

▲ Exposure: If a live cell at time  $t$  has only 1 live neighbor or no live neighbors, it will be dead at time  $t + 1$ .

▲ Survival: a cell survives from time  $t$  to time  $t + 1$  if and only if 2 or 3 of its neighbors are alive at time  $t$ .

Starting from the initial configuration, these rules are applied, and the game board evolves, playing the game by itself! This might seem like a rather boring game at first, but there are many remarkable facts about this game. Today we will see the types of "life-forms" we can create with this game, whether we can tell if a game of Life will go on infinitely, and see how a game of Life can be used to solve any computational problem a computer can solve.

## *B. Genetic Algorithm*

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them.

Six phases are considered in a genetic algorithm.

### 1. Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.

An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome** (solution).

In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

### 2. Fitness Function

The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

### 3. Selection

The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.

Two pairs of individuals (**parents**) are selected based on their fitness scores. Individual with high fitness have more chance to be selected for reproduction.

### 4. Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes.

### 5. Mutation

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

Mutation occurs to maintain diversity within the population and prevent premature convergence.

### 6. Termination

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

### C. Jenetics

Jenetics is an advanced Genetic Algorithm, Evolutionary Algorithm and Genetic Programming library, respectively, written in modern day Java. It is designed with a clear separation of the several algorithm concepts, e. g. Gene, Chromosome, Genotype, Phenotype, population and fitness Function. Jenetics allows you to minimize or maximize a given fitness function without tweaking it. In contrast to other GA implementations, the library uses the concept of an evolution stream (EvolutionStream) for executing the evolution steps. Since the EvolutionStream implements the Java Stream interface, it works smoothly with the rest of the Java Stream API.

## II. SOLUTION

We write a java class called JeneticsAlgorithm to find the best pattern through the evolution.

For initial population, each **genotype** has 20 chromosome, and each **chromosome** has 8 bits. We use Jenetics to generate genotype. The genetic code generated by Jenetics is random. We can enter parameters of different rates when defining chromosomes, and our different rate is 50%.

```
Factory<Genotype<BitGene>> gtf = Genotype.of(BitChromosome.of(8,0.5), 20);
```

To express the genotype to **phenotype**, which just is to translate 8 bits to Point. We write an **Expression** function. As we mentioned before, our genetic code appears as an 8-bit chromosome. Factory contains multiple chromosomes. Each chromosome can represent the initial position of a cell, that is, each 8-bit represents the initial position of a cell.

We use the number of "0" in the first five digits of the eight-digit number to represent the position of the cell from the origin, and the last three digits of the eight-digit number to represent the direction of the cell from the origin. Here is a table of gene expression:

Last 3 digits	Direction	Example	Pattern(expression)
000	Up left	01100000	-3 3
001	Up	01100001	0 3
010	Up right	01100010	3 3
100	Right	01100100	3 0
011	Down right	01100011	3 -3
101	Down	01100101	0 -3
110	Down left	01100110	-3 -3
111	left	01100111	-3 0

Table 1 Table of Gene Expression

By expressing all the Chromosome in the Factory, we can get a random initial pattern.

To compare different genotypes, we have to choose a parameter to be compared. Since we would like to choose those genotypes with more generations to survive and breed, we just have to compare their generations. But the generation may be very large and exceed the maxGeneration. So we choose to use growthRate to represent fitness.

```
GrowthRate = (game.getCount-firstgame.getCount)/game.generations;
```

So we write a **fitness** function which is called **eval()**. This function gets a genotype as input, then starts its game of life and returns its growthRate() as output.

```

public static long eval(Genotype<BitGene> gt) {
    String gtS = gt.toString();
    Pattern p = Pattern.compile("[^0-9]");
    Matcher m = p.matcher(gtS);
    gtS = (m.replaceAll("").trim()).toString();
    String gtStr = GenoType.getPetternStr(gtS);
    Game game=new Game();
    game.myRunWithoutPrint(gtStr);
    int endlength = game.counted;
    double growthrate = 0;
    if(generations>0){
        growthrate = (endlength-20)*1.000/generations;
    }
    else{
        growthrate = 0;
    }
    run++;
    return growthrate;
}

```

The evolution driver in Jenetics, which is also known as the environment, is defined as **Engine**. It controls how the evolution steps are executed. Once the Engine is created, via a Builder class, it can't be changed. It doesn't contain any mutable global state and can therefore safely used/called from different threads. This allows to create more than one EvolutionStreams from the Engine and execute them in parallel.

To control the evolution, there are several Engine parameters can be configured. In our solution, we use like

offspringFraction() to define the fraction of offspring (and survivors) for evaluating the next generation;

populationSize() to defines the number of individuals of a **population**;

survivorsSelector() & offspringSelector to defines the **Selector** used for selecting the survivors and offspring population;

alterers(new Mutator<>(), new SinglePointCrossover()) to implement the mutation and crossover of offspring;

executor() to implement the parallelization and control the number of execution threads.

Here is our code:

```

Engine<BitGene, Long> engine
    = Engine.builder(JeneticsAlgorithm::eval, gtf)
        .offspringFraction(0.5)
        .populationSize(10)
        .survivorsSelector(new TournamentSelector<>(5))
        .offspringSelector(new RouletteWheelSelector<>())
        .alterers(new Mutator<>(0.1), new SinglePointCrossover(0.1))
        .executor(executor)
        .build();

```

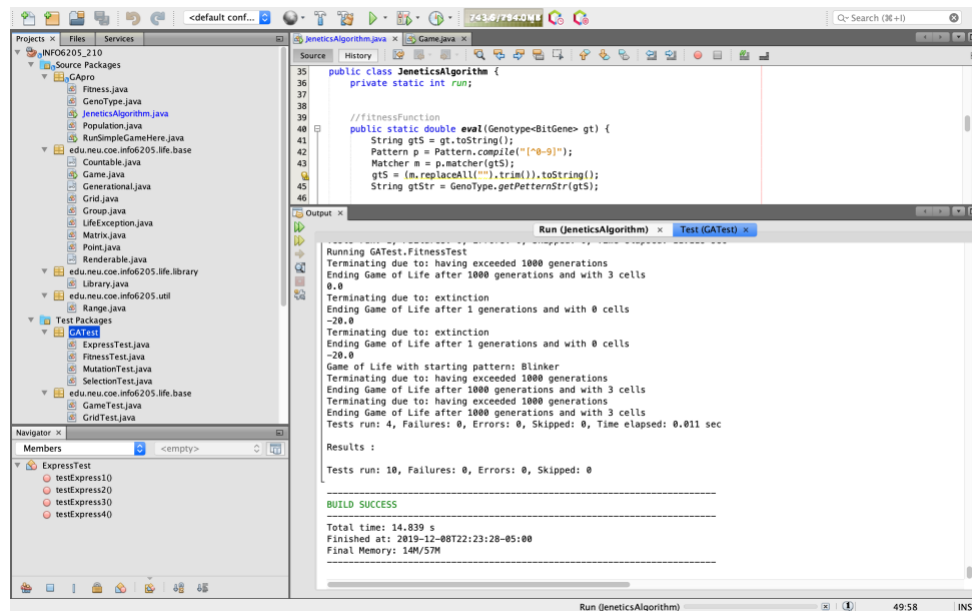
After termination, we use **engine.stream()** to choose the best pattern.

```
Genotype<BitGene> result  
    = engine.stream()  
        .limit(Limits.bySteadyFitness(999))  
        .collect(EvolutionResult.toBestGenotype());
```

### III. EVALUATION

#### A. Unit Test:

To evaluate our solution, first we run the Unit test package. GATest contains ExpressTest, FitnessTest, MutationTest and SelectionTest, and all tests are successful.



Picture 3.1 Unit Test

#### B. Run a simple game:

In “RunSimpleGameHere” class, user the can define the size of population. By running this class, multiple genotypes are generated from the population class and inherited under natural conditions through their phenotypes. During this progress, fitnessScore is calculated.

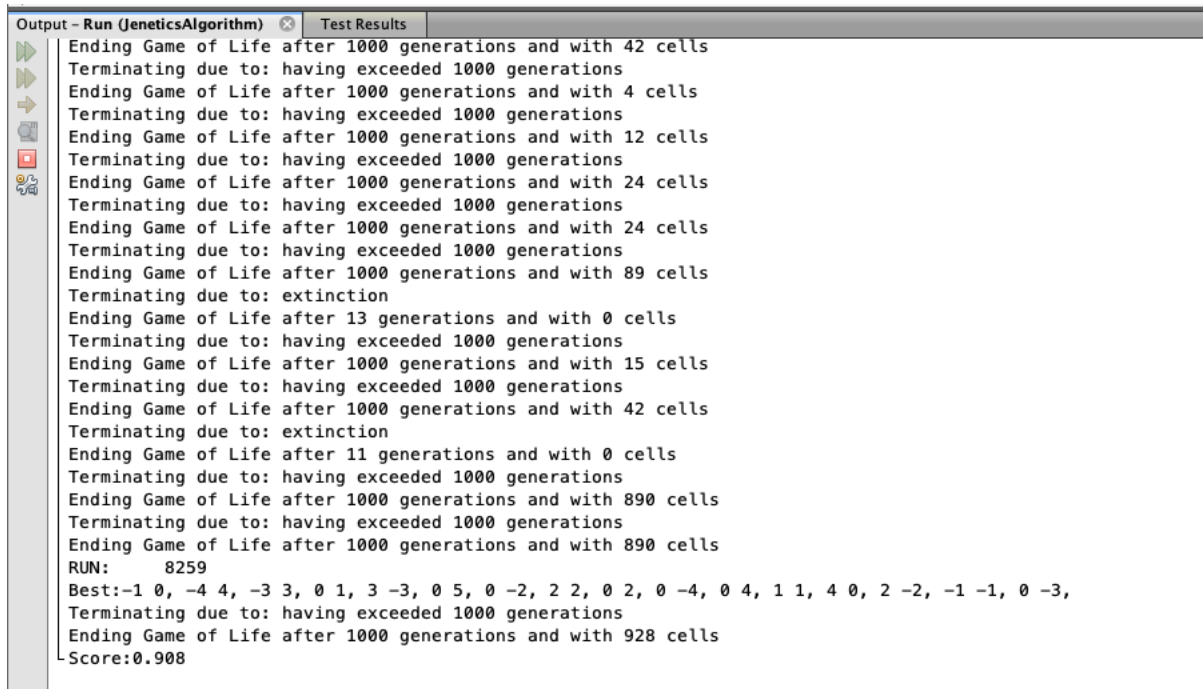
The output includes three parts:

- 1) Generate each genotype and its phenotype. We print patterns first, and then they will be saved in a HashMap.
- 2) Run each pattern in the game(without print of every generation) and then generate the rank of individual genotypes according to the FitnessScore.
- 3) the genetic process with the highest FitnessScore genotype.

In this class, we can screen the population with the most breeding generations under simple conditions (no mutation, natural selection, Crossover, etc.), and observe the breeding process.







```
Output - Run (GeneticsAlgorithm) Test Results
Ending Game of Life after 1000 generations and with 42 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 4 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 12 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 24 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 24 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 89 cells
Terminating due to: extinction
Ending Game of Life after 13 generations and with 0 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 15 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 42 cells
Terminating due to: extinction
Ending Game of Life after 11 generations and with 0 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 890 cells
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 890 cells
RUN: 8259
Best:-1 0, -4 4, -3 3, 0 1, 3 -3, 0 5, 0 -2, 2 2, 0 2, 0 -4, 0 4, 1 1, 4 0, 2 -2, -1 -1, 0 -3,
Terminating due to: having exceeded 1000 generations
Ending Game of Life after 1000 generations and with 928 cells
Score:0.908
```

Picture 3.6 JeneticsTest

#### IV. CONCLUSION

From the learning and implementing GA and Jenetics, we studied how to use Genetic Algorithm to solve problem that to find best solution. And the genetics do help to optimize the start pattern for Conway's Game of Life. However the results would be different every time.

#### REFERENCE

1. <http://web.mit.edu/sp.268/www/2010/lifeSlides.pdf>
2. <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>
3. <http://jenetics.io/manual/manual-5.1.0.pdf>