Final Project Report

INFO6205 Algorithm

Group: 206

Team members:

Hongji Luo, UID: 001306756

Haonan Quan, UID:001305566

Date: Sunday, December 8, 2019

Table of Contents

[Introduction 2](#_Toc26662443)

[The Game of Life 2](#_Toc26662444)

[Genetic Algorithm 2](#_Toc26662445)

[Project Goal 2](#_Toc26662446)

[Basic concept 3](#_Toc26662447)

[Gene and Chromosome 3](#_Toc26662448)

[Genotype 3](#_Toc26662449)

[Phenotype 3](#_Toc26662450)

[fitness function 3](#_Toc26662451)

[Method and JA Settings 5](#_Toc26662452)

[Randomness/Repeatable result from JA 7](#_Toc26662453)

[Mutation 8](#_Toc26662454)

[unit tests 10](#_Toc26662455)

Introduction

The Game of Life

The Game of Life, also known simply as Life.

The game is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

• Is a cellular automation which imposes fixed rules on the cells.

• The game is “played” on an infinite two-dimensional discrete grid.

• Successive generations are dependent only on the previous generation: there is no input of any kind once the process starts.

• Because of the two-dimensional nature of the game, each cell has exactly eight neighbors.

## Genetic Algorithm

Genetic Algorithm refers to a algorithm inspired by the process of natural selection. It is used for generating high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection. There are a lot of Genetic algorithms library provided by some companies.

We chose the provided Jenetics Algorithm as the GA implementation in this project.

Project Goal

Our project goal is generating a starting pattern for the Game of Life by using GA. This pattern will survive as long as possible through Game of Life, with non-repeated generations, and a growth rate larger than 0.

# Basic concept

There actually three parts of this project – Game of Life, Genetic Algorithm and use genetic algorithm to generate the starting pattern for the Game of Life. Game of Life and Genetic Algorithm is provided directly. So we could make use of those two part to find the desired result.

Even though we do not need to actual implement the genetic algorithm, we still need to define some basic concepts of this project and decide to use which classes to achieve the final goal.

Gene and Chromosome

The Gene that we used for this project, is bits. Thus, the Chromosome would be the collection of the gene, which is bytes.

Genotype

Similar with the relationship between Chromosome and Gene, Genotype are the collection of the Chromosome. We use 64 bits to present an 8x8 grids, which can have at most 64 live cells. We use 8 Chromosome and each Chromosome contains 8 bits. So, whenever we generate a pattern from Genotype Factory, the corresponding pattern should be within the 8x8 grid.

A close up of a logo

Description automatically generated

Phenotype

Genotype is the 64 bits binary code, so the Phenotype is the corresponding pattern with those code. In this project, those patterns are represented as a String contain the coordinates of the live cells. By definition of the Phenotype, it should also contain a comparable fitness value from the fitness function. In this case, the fitness value is an integer representing the max generation that the genotype could reach within the game of the generation.

Also, the Phenotype also represent the Individual.



fitness function

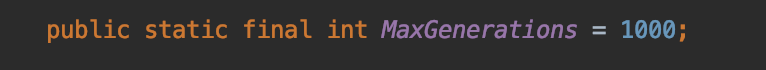
The fitness function would return a comparable value for Selector based on the input genotype. In this project, fitness function will return the max generation that the input genotype would survive through the game of life. The code of is provided directly from the professor and included the two most important parameters when pattern running through the processes – growth rate and max generation.

**Growth rate**: the attribute that fractions of the live cells between generations. A pattern would not be considered good if its growth rate smaller or equal than 0, which means the pattern is repeating itself in a short time period or it will die eventually. The game of life process will end once the growth rate <= 0, and return the current max generation.

A screenshot of a cell phone

Description automatically generated

**Max generation**: the upper bound of the this process. Once the pattern survived through the max generation, in this case, 1000, the process would end and return the max generation – 1000. The max generation, the upper bound number, of this project is 1000, so we could use Integer instead of Long to hold the value. However, if very large number are considered as the max generation, Long should be used as the return value.



So every genotype passing to fitness function, will be valued a score with its max generation running through the game of life. The max generation means unrepeated, with growth rate larger than 0, generations.

# Method and JA Settings

Before we actual setting up the evaluation, we need to set up a random seed, which will explain in randomness section.

A close up of a logo

Description automatically generated

We have to build a Genotype Factory which will generate a genotype when needed. And we use fitness function to evaluate those genotypes and give them a comparable value. Based on those value, selector will select from the highest valued individuals as survivors and offspring. Some of those offspring will mutate so we need to configure the mutator with the mutation rate / possibility. For each generation, we will have 100 individuals. A close up of a sign

Description automatically generated

**A screenshot of a cell phone

Description automatically generated**

Once we set up engine, we are able do the actual evolution. The evolution will stop when it reach 10th generation, and output the best individual / starting pattern for the game of life. At the same time, we create and refresh the statistic recorder for reference.

A screenshot of a cell phone

Description automatically generated

Some of the setting are not configured but needed for this evolution. The JA has the default setting for those configurations, and of course we will overwrite any overlapped setting.

**A screenshot of a cell phone

Description automatically generated**

As you may see, it is calling the executor. Which means this algorithm is running in multiple threads. This might cause some problem with randomness.

Randomness/Repeatable result from JA

Genetic algorithms heavily depend on pseudo random number generators (PRNG) for creating new individuals and for the selection- and mutation-algorithms. Jenetics is a multi-thread algorithm. So the normal Random would not have good performance. In other words, even though we configured the seeded random configuration within RandomRegistry, we are mostly like not to get the same result. So, instead of configure the seed for single thread, we have to configured globally. By doing researches, I found I actually could do that by using LCG64ShiftRandom. But we have to import the dependency before we can actually use it.

A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generated

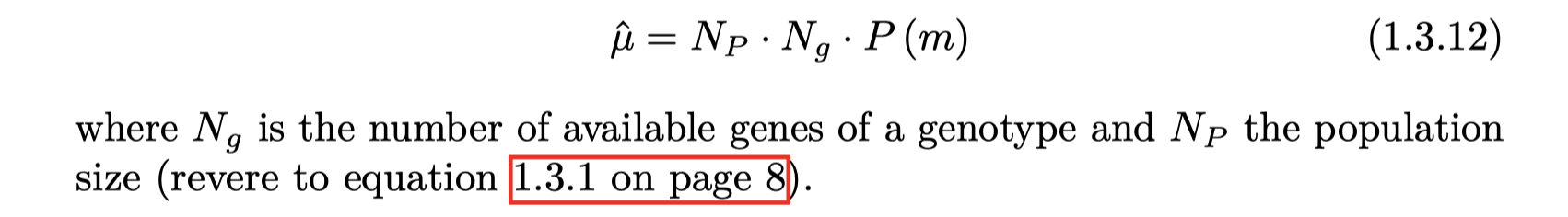
Even though we run the JA with the same random seed, we might still not get the exactly the same result. After several experiments we found we could found one of the four possible answers. This is demonstrated by the Junit test randomnessTest. We think the reason why we got four possible answer instead of exactly one, is that we had four best genotypes by using this random seed running through the same engine setting. All four have the highest fitness value 1000, and we only output one from the evolution result.

We could increase the upper bound of the max generation to significant large – eg. 10000, to demonstrate this idea. Since we increase the upper bound significantly, there might be only one phenotype with highest fitness value, instead of four. However, 1000 generation already took too long when we are running the algorithm. We might run out of time to demonstrate this idea.

Mutation

We are using the Jenetics Algorithm, which is already implemented full functionally, so we are using the Mutator from the library directly instead of building it by our own. Mutation and Crossover are provided from Jenetics library. We use mutation only for altering in this project. The mutation probability, P (m), is the parameter that must be optimized. The optimal value of the mutation rate depends on the role mutation plays. If mutation is the only source of exploration (if there is no crossover), the mutation rate should be set to a value that ensures that a reasonable neighborhood of solutions is explored.

The mutation probability, P (m), is deﬁned as the probability that a speciﬁc gene, over the whole population, is mutated. That means, the (average) number of genes mutated by a mutator is

****

By collecting the test result and analysis, we could actually demonstrate the mutation probability. But due to the limited documentation, experiments and understanding of genetics algorithm, I am not certain about my demonstration.

According to the 6 experiments data, it shows the average of the altering population for each generation is around 1950. And the default offspring fraction is 0.6. We have 100 Invidiuals for each generation and 64 genes for each Individual. Here is the demonstration.

Those data could be found in Data.docx.

**A screenshot of a cell phone

Description automatically generated**

unit tests

We wrote three Junit tests. fitnessTest will test the correctness of our fitness. genToPattern would transfer the binary code within genotype into pattern. And the randomnessTest could demonstrate the JA result repeatable by using the same random seed number.

Here is the test result. The test class could be found under base folder from test directory.

A screenshot of a cell phone

Description automatically generated