Generic Algorithm Project Report

Jakub Suszwedyk

December 2022

Contents

1.Development of the Fitness Function

The fitness function tells us how close the individual is to reaching the target (it beeing HELLO WORLD) and presents that as a value in range of 0 to 1, one representing reaching a target and 0 meaning that none of the characters are in the right place, the space character in the middle is 0.2 of that rating, with every other character adding extra 0.08, that way the population is quickly composed of two words of the same length as the HELLO WORLD target.

2.Discussion and implementation

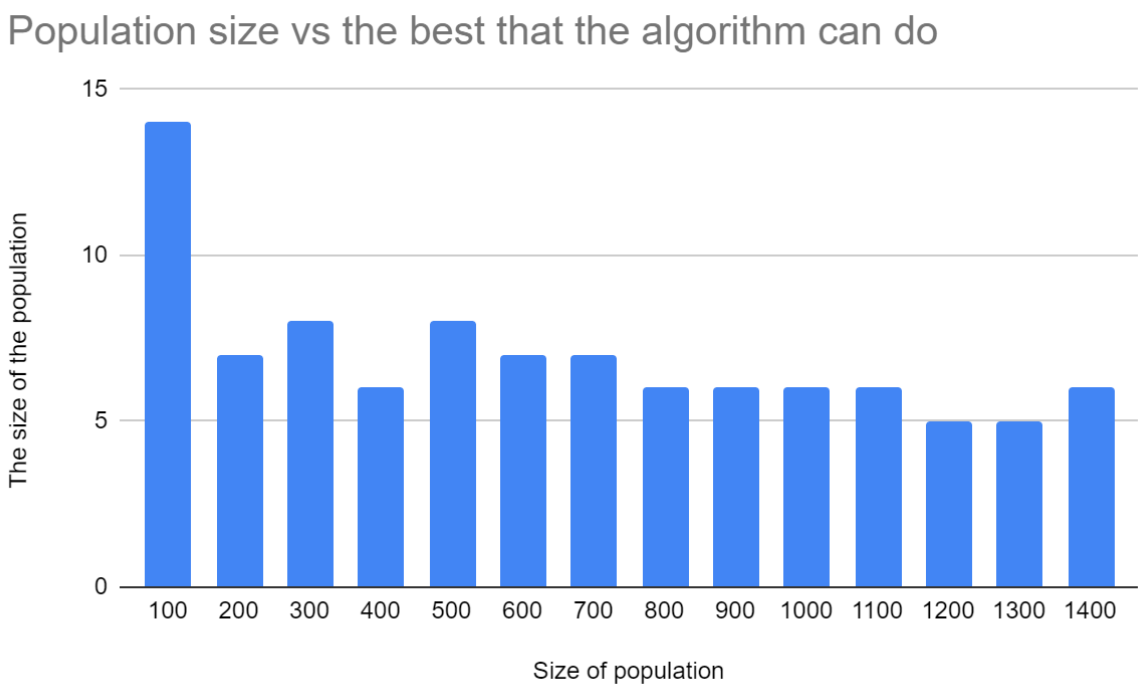
The GA is implemented with the elitist selection, meaning that only the best of each generation are able to evolve, by default it is 3 top individuals as the test showed that it is the best value for the population size of 100. The evolution is handled with a crossover method using the first 6 characters of the individual with the higher fitness value, then adding to the last 5 characters of the second individual. Next step in the evolution of every individual is mutation, here it is by default 2 random chromosomes being changed into a random uppercase letter or a space. In addition to that, we implemented a simple visual interface to allow the users to choose their own: mutation rate, population size and how many of the top individuals are allowed to evolve, because of that the user can carry out their own test on our GA.

3.Testing methodology

For all the tests, we constructed a different Testing class in our program, in there you can see all the test we carried out as well as all the data we got from them to create the graphs that you can see later in this report.
All tests were carried out multiple times with the GA running at least 100 times per test to get the result as close to the average as possible. The number presented in the graphs for the convergence of the algorithm are that for the best of the population to evolve into HELLO WORLD,.

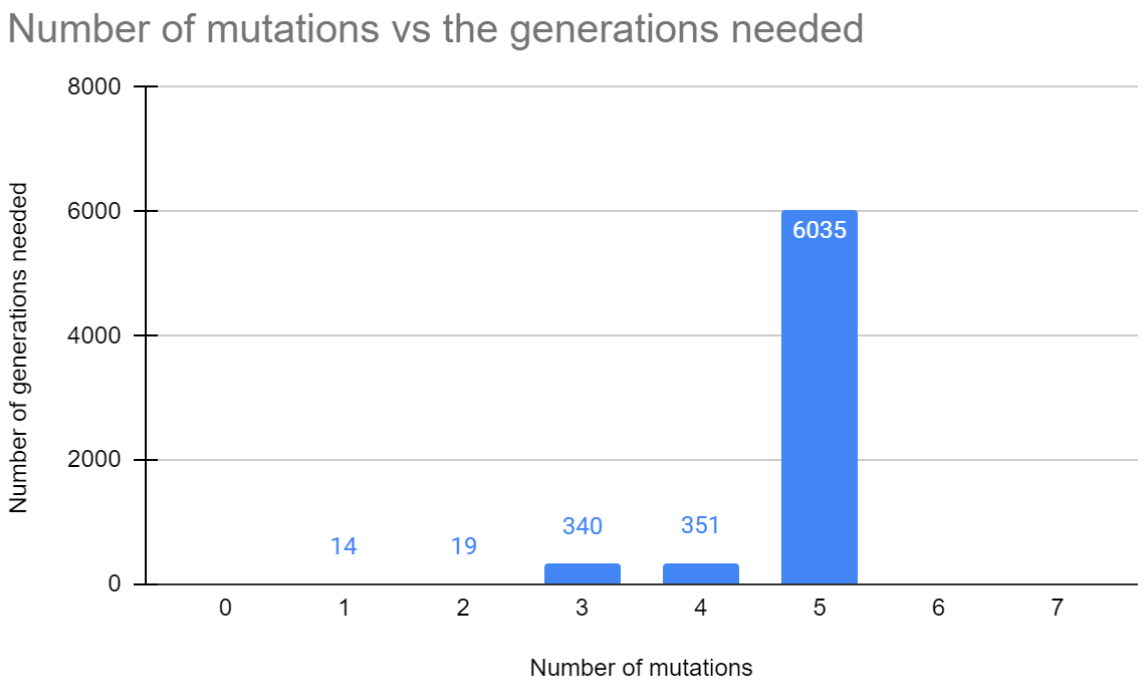
4.The influence of a larger/smaller population

Contrary to what we expected before conducting the experiments, the bigger population size can in fact help the algorithm to converge faster. The graph below, constructed from our test, shows that, every time the GA was run for 10 times for every number of individuals allowed to evolve (from 1 to 25 for every population size), giving us 250 times for every population size, the graph presents the best that we got for every population size. As you can see there, the bigger population helps the algorithm, but why is that? What we think is causing that is the fact the high mutation rate of 2 for every try allows the top individuals to have more tries at randomly hitting the target with more population size.



5.The influence of a larger/smaller mutation rate

For the experiments regarding the influence of the mutation rate on the number of generations needed, we tested the mutation from 0 to 7 chromosomes per individual. The mutation is always sure to occur, and it involves all individuals apart from the top 2 of each generation which are the base of evolution. The population size for our test was always 100 individuals. As shown on the graph, the lowest mutation rate of 0 did not allow the algorithm to converge, then, the only evolution was the crossover from the 2 of the best individuals of the population which caused the whole population to look the same (the first 6 letter of the best individual and last 5 letter of the second-to-best individual). The mutation rate of 1 to 2 is the best spot for the algorithm to converge as fast as possible, higher mutation rate of 3 or 4 still works but is significantly slower and the highest working rate of 5 chromosomes per individual is very slow. Higher mutation rate like 6 or 7 do not allow the algorithm to converge in 30,000 generations, that is because the individuals are pretty much random at this point, they would eventually hit the target but there is a very low chance of that and it never happened in 30,00 generations in our testing.



6. Evolution without the crossover

For the evolution without the crossover the algorithm menaged to converse every time out of 100 we run it, the average generation needed was 63. The evolution there was composed of choosing the best individuals and then filling the next generation with the copies of it with slight mutations (2 random chromosomes with random values). The experiments were carried out on the population size of 100, the generations needed to converge were in range from 21 to 160. As you can see the lack of crossover can hinder the speed at which the algorithm converges but it never stopped it from working.

7.Code as an appendix

The code below is only the part responsible for running the algorithm in its simple version, for the testing as well a a graphical implementation please check the files attached

```
import javax.sound.midi.Soundbank;
import java.util.Formattable;
import java.util.Random;
import java.util.Scanner;

public class Practical2 {

   static final String TARGET = "HELLO WORLD";
   static final char[] splitTarget = {'H', 'E', 'L', 'L', 'O', ' ', 'W',
'O', 'R', 'L', 'D'};
   static char[] alphabet = new char[27];
   private static final int popSize = 100;

   public static void main(String[] args) {
       //creating a population
       for (char c = 'A'; c <= 'Z'; c++) {
           alphabet[c - 'A'] = c;
       }
       alphabet[26] = ' ';
       Random generator = new Random(System.currentTimeMillis());
       Individual[] population = new Individual[popSize];
```

```
        // we initialize the population with random characters
        for (int i = 0; i < popSize; i++) {
            char[] tempChromosome = new char[TARGET.length()];
            for (int j = 0; j < TARGET.length(); j++) {
                tempChromosome[j] =
alphabet[generator.nextInt(alphabet.length)]; //choose a random letter in
the alphabet
            }
            population[i] = new Individual(tempChromosome);
        }
        HeapSort.sort(population);
        //asking the user for their choice
        int howManyToEvolve = 2;
        elitist(population, howManyToEvolve);
    }
    public static void elitist(Individual[] population, int howManyToEvolve) {
        int popSize = population.length;
        for (int safety = 0; safety < 30000; safety++) {
            if (population[0].getFitness() == 1) {
                System.out.println("this is the: " + safety + " iteration, the
best is our target: ");
                System.out.println(population[0].genoToPhenotype());
                System.out.println("the rest of the population with their
fitnes values:");
                boolean first = true;
                System.out.println("done");
                break;
            }
            Individual[] nextIteration = new Individual[popSize];
            for (int index = 0; index < popSize; index++) {
                nextIteration[index] = Evolve(population[index %
howManyToEvolve], population[(index % howManyToEvolve) + 1]);
            }
            population = nextIteration.clone();
            HeapSort.sort(population);
        }
    }
    public static Individual Evolve(Individual one, Individual two, int
mutationRate) {
        char[] chromosome = new char[TARGET.length()];
        for (int i = 0; i < 6; i++) {
            chromosome[i] = one.getChromosome()[i];
        }

        for (int i = 6; i < TARGET.length(); i++) {
            chromosome[i] = two.getChromosome()[i];
        }
        Random r = new Random();
        //random
        for (int a = 0; a < mutationRate; a++) {
            char random_char = alphabet[r.nextInt(alphabet.length)];
            int random_int = r.nextInt(TARGET.length());
            chromosome[random_int] = random_char;
```

```
        }

        Individual re = new Individual(chromosome);
        return re;
    }
    public static Individual Evolve(Individual one, Individual two) {
        return Evolve(one, two, 2);
    }

}
```