Genetic Algorithm Report

Alvaro Menendez i6289702, Eden Rochman i6293015

November 2022

Contents

1 Introduction

In this assignment we created a genetic algorithm that simulates a simplified version of the infinite monkey theorem. It simulates the process of evolution in a much shorter time. Starting from a random set of strings our goal is to evolve them to some desired sentence or word, in this case “Hello World”.

We implemented all of our code in Java, by making use of our knowledge in object oriented programming, populations and genetics.

The goal of this assignment is to have a better understanding of how natural selection works throughout programming.

2 Definitions

1. Individual: an entity containing a fitness value and a chromosome.
2. Fitness value: a number that represents how good an individual is. The bigger, the better.
3. Chromosome: a list of letters with the same length as the desired string.
4. Population: an array of individuals.
5. Parents: an array of the n best individuals of a population. An individual is better than another one if and only if it has a higher fitness value.(This is called elitist selection)

6. Generation: a population based on the individuals of the previous population. The first generation will always be randomly generated
7. To mutate: to randomly change a letter of an individual's chromosome
8. Mutation rate: the probability that an individual is mutated

3 Methods

In this section we will describe the most important methods implemented in our genetic algorithm code. Recall that in this section, we will only give an overview of the methods. See Appendix for the full implementation.

3.1 Prewritten methods

We received two prewritten methods. The first one of them creates a random population of a certain number of individuals. The second one sorts the population based on their fitness value, from high to low.

3.2 Our methods

3.2.1 Fitness

The first method we implemented is the fitness method. This method is needed to give each individual a score based on how similar an Individual is to the desired one. More specifically, our method gives an individual a point for every letter shared with the desired one at the correct place. See 1 for a visual representation of the method.

Figure 1: representation of fitness value of a given individual

3.2.2 Merge

This method receives two Individuals as the input. The method combines both Individuals chromosome randomly to produce a new Individual. see 2 for a visual representation of the method.

Figure 2: merge method representation

3.2.3 Crossover

This method creates a new generation by merging random Individuals from the parents of that generation.

3.2.4 Mutation

The mutation method makes sure that in each generation there is still a small element of randomness, simulating how evolution works. The mutation method randomly changes the chromosome of an Individual based on the mutation rate, the higher the mutation rate is, the more likely it is that an individual's chromosome is random and did not come from his parents.

4 Implementation

The implementation of the genetic algorithm works as follows. The program starts by creating a random population and assigning every Individual its determined fitness. After this, it assigns the parents for the next generations by elitist selection. This means that the parents will be the individuals with the highest fitness value. Then it creates a new generation by applying the crossover and the mutation methods to the parents of the generation. It will repeat this process until it has found the desired string or it has produced a maximum number of generations, meaning that the algorithm won't find a solution. This only happens in extreme cases.

5 Experiments

To test our algorithm, we conducted some experiments to answer the questions proposed in the report. If not stated otherwise, the setup for our algorithm is of population size = 100, mutation rate = 0.15 and parents size = 40 (as we are using elitist selection)

1. **What is the influence of a larger/smaller population? Do you see a difference in number of generations needed to find a solution?**
   In general, a larger population size can lead to a faster convergence to the optimal solution in a genetic algorithm, because it allows for a greater diversity of solutions to be explored and evaluated. We tested our algorithm with populations of 50, 100 and 200 individuals, and plotted the resulting histogram of the number of generations needed to get to the desired string. As you can see, as the population size gets bigger, the number of generations needed decreases. Here are the results:

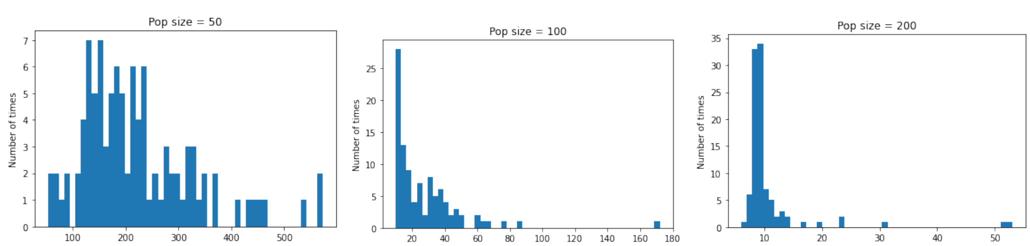

Figure 3: Different population sizes and generations needed

2. **What is the influence of a larger/smaller mutationrate? Do you see a difference in number of generations needed to find a solution?**
   A higher mutation rate can lead to a faster convergence to the optimal solution, because it allows for a greater exploration of the solution space and can help to avoid getting stuck in a local optimal solution. On the other hand, a lower mutation rate can lead to a slower convergence to the optimal solution, because it limits the

exploration of the solution space and may result in the algorithm getting stuck in a local optimal solution. Again, we tested our algorithm with mutation rates of 0.05, 0.1 and 0.2, and plotted the resulting histograms. Here are the results

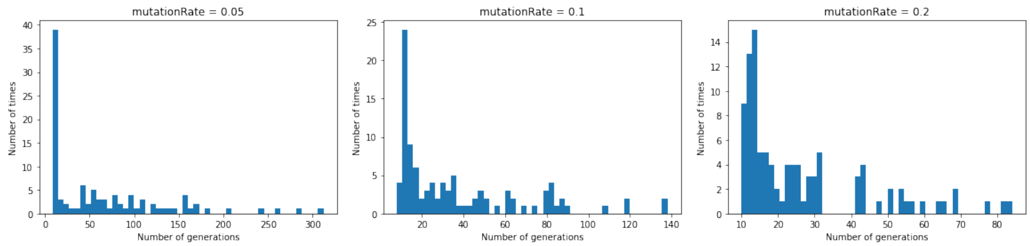

Figure 4: Different mutation rates and generations needed

3. **Does your GA still work without mutation? Why or why not?**
   Our GA rarely works without mutation. This is because we have to be lucky enough to get all the letters in the desired string in the first generation, otherwise, the algorithm would run forever, not converging to the solution. To test it, we ran our algorithm with no mutation rate and plotted the resulting histogram of generations needed. Note that, we capped the maximum number of generations to 600, so if a solution reaches that number, it means that it has not found a solution and it is stuck in an infinite loop. Here are the results:

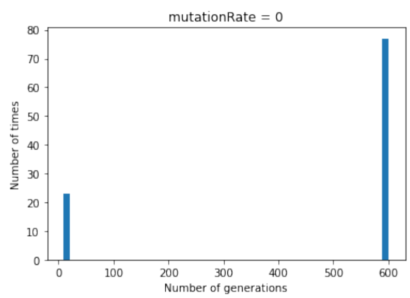

Figure 5: No mutation rate

4. **Now leave out the crossover operator. Do you see a difference in number of generations needed to find a solution?**
   Leaving out the crossover operator in a genetic algorithm can significantly impact the algorithm's ability to find the optimal solution, and may result in a slower convergence to the optimal solution. Without crossover, the algorithm will only be able to explore solutions that are present in the current population, which may limit its ability to explore a diverse set of solutions and find the optimal solution. This can result in a slower convergence to the optimal solution, or may prevent the algorithm from finding the optimal solution altogether.

6 Appendix

```java
    public static int popsize = 100;
    public static int parentsLength = 50;
    public static double mutationRate = 0.15;

    public static Individual[] parents = new Individual[parentsLength];
    static final String TARGET = "HELLO WORLD";
    static char[] alphabet = new char[27];


    public static void entrypoint() {
        Individual[] currentGeneration = randomGeneration(popsize);
        HeapSort sorter = new HeapSort();
        int generationCounter = 1;
        boolean solutionFound = false;

        int iter = 0;
        while (!solutionFound && iter < 600) {
            iter++;

            System.out.println();
            System.out.println("Generation number " + generationCounter);
            for (Individual i : currentGeneration) {
                i.setFitness(myFitness(i)); // sets each individual fitness of the current generation
                if(i.genoToPhenotype().equals(TARGET)){
                    solutionFound = true;
                }
            }
            sorter.sort(currentGeneration);
            printGeneration(currentGeneration);

            if(!solutionFound){
                //Creates the parents of the generation based on the elitist selection
                for (int i = 0; i < parents.length; i++) {
                    parents[i] = currentGeneration[i].clone();
                }
                currentGeneration = applyMutation(crossover(parents));
                generationCounter++;
            }

        }
        myWriter.println(generationCounter);

        System.out.println(currentGeneration[0].genoToPhenotype() + " found on generation "+ generationCounter);
    }

    public static int myFitness(Individual individual) {
        int fitness = 0;
        for (int i = 0; i < TARGET.length(); i++) {
            if (individual.getChromosome()[i] == TARGET.charAt(i)) {
                fitness += 1;
            }
        }
        return fitness;

    }
```

```java

    public static Individual[] randomGeneration(int popSize) {
        for (char c = 'A'; c <= 'Z'; c++) {
            alphabet[c - 'A'] = c;
        }
        alphabet[26] = ' ';
        Random generator = new Random(System.currentTimeMillis());
        Individual[] population = new Individual[popSize];
        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)];
            }
            population[i] = new Individual(tempChromosome);
        }
        return population;
    }

    /**
     *  Prints the most representative individuals of a generation
     * @param population
     */
    public static void printGeneration(Individual[] population)  {
        int individualnumber = 1;
        for (int i = 0; i < 20; i++) {
            System.out.println(population[i].genoToPhenotype() + "
                individual number " + individualnumber
                    + " fitness " + population[i].getFitness());
            individualnumber++;

        }
    }

    /**
     * Creates a new generation of individuals
     * @param parents Parents of the previous generation
     * @return
     */
    public static Individual[] crossover(Individual[] parents) {
        Individual[] newGeneration = new Individual[popsize];
        Random r = new Random();
        Individual parent1;
        Individual parent2;
        for (int i = 0; i < popsize; i++) {
            parent1 = parents[r.nextInt(parents.length)];
            parent2 = parents[r.nextInt(parents.length)];
            newGeneration[i] = merge(parent1, parent2);

        }
        return newGeneration;
    }

    public static Individual randomIndividual(){
        Random generator = new Random();
        char[] tempChromosome = new char[TARGET.length()];
        for (int j = 0; j < TARGET.length(); j++) {
            tempChromosome[j] = alphabet[generator.nextInt(alphabet.
                length)];
        }
        return new Individual(tempChromosome);
```

```java
    }

    /**
     * Combines the chromosome of two individuals to create a new
        Individual
     * @param one
     * @param two
     * @return
     */
    public static Individual merge(Individual one, Individual two){
        Random r = new Random();
        int maxCuts = 5;
        int minCuts = 1;
        int cuts = r.nextInt(maxCuts-minCuts)+ minCuts + 2;
        int[] cutsPos = new int[cuts];
        cutsPos[0] = 0;
        cutsPos[cutsPos.length-1] = one.getChromosome().length-1;

        char[] childChromosome = new char[one.getChromosome().length];
        for(int i = 1; i< cuts-1; i++){
            cutsPos[i] = r.nextInt(one.getChromosome().length - 1) + 1;
        }
        Arrays.sort(cutsPos);
        for(int i = 0; i < cutsPos.length-1; i++){
            for(int j = cutsPos[i]; j<= cutsPos[i+1]; j++){
                if(i % 2 == 0){
                    childChromosome[j] = one.getChromosome()[j];
                }else{
                    childChromosome[j] = two.getChromosome()[j];
                }
            }
        }
        return new Individual(childChromosome);
    }

    /**
     * Applies the mutation to a given generation
     * @param generation
     * @return
     */
    public static Individual[] applyMutation(Individual[] generation){
        Random r = new Random();
        double probability;
        int index;
        char toReplace;
        for(Individual i : generation){
            probability = r.nextDouble();
            if(probability < mutationRate){
                index = r.nextInt(generation[0].getChromosome().length)
                    ;
                toReplace = alphabet[r.nextInt(alphabet.length)];
                i.changeLetter(index, toReplace);
            }
        }
        return generation;
    }
```