Infinite Monkey Theorem

Genetic Algorithms

Noah Scharrenberg (i6254952)[1] and Lorenzo Pompigna (i6233748)[2]

[1] Cognitive and Computational Neuroscience
Department of Data Science & AI at Maastricht University
n.scharrenberg@student.maastrichtuniversity.nl
[2] Cognitive and Computational Neuroscience
Department of Data Science & AI at Maastricht University
l.pompigna@student.maastrichtuniversity.nl

1. Background

1.1. Infinite Monkey Theorem

The infinite monkey theorem states that a bunch of "writing" monkeys tapping keys at random, will almost surely type any given text when given an infinite amount of time. However, the probability of this happening is so tiny that it may take a very long time to achieve this, but the chances of this event from occurring is not zero. [2]

Suppose we provide each monkey with a keyboard that has 27 keys (a to z and spaces), and we let them type the word "hello world". Monkey's know how to type, but they can't read or understand what they are typing. So each key is pressed randomly and independent from the other, meaning each key has an equal change of being pressed. Then, the chance of the first letter typed being "h" is $1/27$ and the chance of the second letter being "e"is also $1/27$, and so on. So the chances of the first eleven letters to be "hello world" is:

$$\begin{aligned}(1/27) \times (1/27) \times (1/27) \times (1/27) \times (1/27) \\ \times (1/27) \times (1/27) \times (1/27) \times (1/27) \times (1/27) \\ \times (1/27) = 1.7988650924514296 \times 10^{-16}\end{aligned} \tag{1}$$

As you can see it's close to zero, but never zero.

So a monkey will type "hello world" every 1 out of 17.988.650.924.514.296 times. If we assume that the monkey hits one key per second, the amount of time it could take for the monkey to type "hello world" would be multiple human lifetimes.

As you may imagine, when using a brute force search for "hello world" the amount of computation and time will be a huge. But if we type it ourselves, it wouldn't take more then 5 seconds to do, for the reason that we know how to read and write, and we know how to spell "hello world".

We can improve the brute force search through the use of a genetic algorithm.

1.2. Genetic Algorithm

Genetic Algorithms are inspired by how living beings (animals, bacteria, ...) evolve over generations, mainly "survival of the fittest" also known as selection. They simulate evolution, where they start with an initial population and generate successive "generations" from the population. [4]

The genetic algorithm uses chromosome (a list of array, string, integer, ...) to represent the genetic code and a fitness function of an individual. The population represent a species, and the species that is most fit will represent the best solution that has been found so found.

In order to go to the next generation, the population needs to reproduce. As per "survival of the fittest" we assume the fittest individual will be "selected" to potentially produce offspring with another individual from the "mating pool". So the "mating pool" has all the fittest individuals, and they'll participate in a so called "tournament" where they may or may not be chosen to actually reproduce an offspring.

When two individuals are chosen to mate with each other, a certain part of the parents chromosomes are inherited by the offspring. This is called "crossover", a point is decided where the chromosome is split up and the left half is inherited from the mother and the right side is inherited from the father. [1]

During the crossover it may be possible for a mutation to occur, thus changing a value in the chromosome (as also happens in evolution). This mutation is implemented to ensure diversity within the population and to prevent premature convergence. [3]

With natural evolution it's survival being the target, genetic algorithms usually have a different target they're trying to achieve. As per the "infinite monkey theorem" we aim to use a genetic algorithm to get "hello world".

In order to answer some questions about genetic algorithms, a genetic algorithm implementation has been developed to help answer these questions. The code can be found in Appendix A.4.

2. Population

The genetic algorithm starts with an initial population, which consists of individuals, which we'll call "monkeys". Each monkey has a set characters, which characterize the genes. These genes together form a chromosome (the word in the form of a string).

The initial population size that is chosen, may impact the amount of generations is takes for the word "hello world" to be typed.

As seen in table 1, the size of the population influences the amount of generations or the amount of time it takes to reach "hello world". It'll take too long and too

Table 1: Genetic Algorithm results for "hello world" with a mutation rate of 0.001.

Population	Run 1	Run 2	Run 3	Run 4	Run 5
10	7471061*	17975061*	9481351*	17995163*	21685512*
100	1484 in 282ms	799 in 167ms	3906 in 500ms	1189 in 242ms	3452 in 503ms
1000	14 in 42ms	73 in 114ms	17 in 42ms	39 in 72ms	15 in 41ms
10000	10 in 125ms	13 in 150ms	12 in 146ms	13 in 162ms	13 in 160ms
100000	10 in 1103ms	9 in 1000ms	12 in 1360ms	9 in 1036ms	10 in 1161ms
1000000	9 in 9643ms	8 in 8182ms	8 in 8054ms	8 in 9594	8 in 8158

* Stopped after 5 minutes of running

many generations when the initial population is low. Having to stop testing on a population of ten monkeys after 15 minutes, and a population of a hundred monkeys varies to much in the amount of generations and time it take to reach the goal. However, an initial population that is too high may take less generation but takes significantly more time, the generations will also stagnate and therefore be very similar to each other.

Using an initial population of around a thousand monkeys seems to be the most optimal choice, given that it takes the least amount of time to reach "hello world", even though it is not as consistent, in the amount of generations, as bigger populations have.

3. Mutation

When questioning the influence of population sizes on the amount of generations and time it takes to reach "hello world", a mutation rate of 0.01 was used. However, what impact would a bigger or smaller mutation rate have on the populations? and what would happen if we have no mutations?

The mutation rate gives a certain probability for the gene of an offspring to be mutated, which means that a gene could be flipped to another value. This mutation happens to maintain diversity within the population and to prevent premature convergence. [3]

As seen in table 2, the mutation rate influences the amount of generations depending on the population. A couple of things were noted during the tests.

To many mutations occurred the mutation rate to high, decreasing the changes of reaching "hello world", as it's more-or-less randomized. Thus each generation having a different "fittest" monkey.

When the mutation rate is low and the population size is small, there were a lot of generations that had the same "fittest" monkey, resulting in a high generation count with the same chromosomes. This Happened less often in bigger populations, as the chance of having at least one monkey to be mutated is bigger when having

Table 2: Genetic Algorithm results for "hello world" with different mutation rates and population sizes

	1×10^1		1×10^2		1×10^3		1×10^4		1×10^5		1×10^6	
	Gen	Time	Gen	Time	Gen	Time	Gen	Time	Gen	Time	Gen	Time
1×10^{-1}	+2mil	300s[1*]	2555	334ms	65	94ms	25	294ms	15	1.5s	11	11s
1×10^{-2}	+1mil	300s[1*]	501	108ms	30	61ms	15	166ms	10	1s	9	9s
1×10^{-3}	+500k	300s[1*]	4937	627ms	72	110ms	17	151ms	134	2s	8	8s
1×10^{-4}	+2mil	300s[1*]	27437	2794ms	365	1415ms	13	156ms	10	1s	9	9s
1×10^{-5}	+2.5mil	300s[1*]	330820	31013ms	670	598ms	16	219ms	10	1s	8	8s
0	+4mil	300s[1*]	+3mil	300s[1*]	+200k	300s[1*]	12	141ms	10	1s	9	9s

* Left vertical column indicates the population size.

* Top horizontal row indicates the mutation rate.

* Average over 10 runs

* Time indicated by: "ms" for milliseconds or "s" for seconds.

1* Stopped after 5 minutes

a 10k population than having a population of 10, even though the mutation rate is the same. However, it heavily depended on the initial populations chromosomes, as repetitive "fittest" monkey did also occur in bigger populations, even though not as often.

When a mutation rate is zero, it purely depends on the initial population in order to reach "hello world". Populations of around a 1000 monkeys and lower didn't reach the target in time for any performed test, while populations of 10k or higher managed to reach the target most of the time, even though still having the possibility of reaching premature convergence.

Having a mutation rate of approximately $1x10^{-2}$ (0.01) seems to be the most optimal choice in terms of generations and time, where enough mutations happens to get to "hello world" relatively fast, but where not to many mutations happen for it to randomize to much.

4. Crossover

Last we question ourselves, what happens if there was no crossover? During the crossover the two parents that "mated", would produce an offspring. The so called "crossover point" is a random point that determines at what point to take the genes of the other parent. The offspring would than have a certain amount of genes from the mother and the remainder genes from the father. [3]

So when no crossover happens we can see it as either one of two things:

- No offspring are created
- An offspring is created with only the mothers genes.

The first one is relatively easy to get results from, as it'll always go extinct during the first generation except when a monkey with the exact "hello world" chromosome is present in the population.

When still creating offspring but removing crossover, we notice that it'll be entirely up to the mutations occurring to reach "hello world". This however, also requires the selection mechanism to be changed, since it could potentially happen to not have any "fittest" monkey remaining, which would result in an empty "mating pool", thus the monkey population going extinct, which shouldn't happen in this case. This can be solved by having the whole population of the previous generation to be in the "mating pool", regardless if they got better or worse, which as previously mentioned brings is all up to the mutation, thus being completely random.

Shortly said, when no off springs are created as there is no crossover, then the population will most likely go extinct within the first generation. When there is no crossover, but they can still produce offspring, it brings it all up to mutations becoming exactly in the correct order. Obviously. Neither of these would produce a useful genetic algorithm.

5. Conclusion

All in all, we noticed that the initial population and mutation rate influence the amount of generations and time it takes to reach the target. Having a too small of a population would result in many generations before reaching the goal, while bigger populations may take fewer generation, they take longer to get to a new generation.

Having a mutation rate that is too high would result in to many mutations from happening, while a mutation rate that is too low may result in not enough mutations from happening, and therefore getting many generations with the same individual being the fittest.

A good balance between the initial population and mutation rate needs to be found. During the experiments we noticed a population of a thousand individuals and a mutation rate of 0.01 to be the most optimal, as seen in table 2.

Last but not least, when no off springs are created as there is no crossover, then the population will most likely go extinct within the first generation. When there is no crossover, but they can still produce offspring, it brings it all up to mutations becoming exactly in the correct order. Obviously. Neither of these would produce a useful genetic algorithm.

References

[1] D. H. Ballard. “An introduction to natural computation”. In: MIT Press, 1997. Chapter 12.

[2] *Infinite monkey theorem*. URL: `https://en.wikipedia.org/wiki/Infinite_monkey_theorem`. (accessed: 2020/11/24).

[3] V. Mallawaarachchi. *Introduction to Genetic Algorithms — Including Example Code*. URL: `https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3`. (accessed: 2020/11/24).

[4] T. V. Mathew. "Genetic algorithm". In: *Report submitted at IIT Bombay* (2012).

A. Code Implementation

A.1. Population.java

```java
package nl.scharrenberg.imt.domain;

import nl.scharrenberg.imt.exceptions.NoPopulationException;
import nl.scharrenberg.imt.utils.HeapSort;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Population {
    public static char[] VOCABULARY = new char[56];
    public static String TARGET_PHRASE = "Hello world";
    public static int POPULATION_SIZE = 100000;
    public static double MUTATION_RATE = 0.01;
    public static double REQUIRED_SCORE = 1;
    public static double CONVERGENCE_RATE = 1;
    public static boolean ENABLE_CONVERGENCE = false;

    private boolean isDone = false;
    private int generation = 1;
    private ArrayList<Individual> population = new ArrayList<>();
    private ArrayList<Individual> matingPool = new ArrayList<>();

    public Population() {
        getVocabulary();

        for (int i = 0; i < POPULATION_SIZE; i++) {
            this.population.add(new Individual());
        }

        this.calculateFitness();
    }

    public void calculateFitness() {
        for (Individual individual : this.population) {
            individual.calculateFitness(TARGET_PHRASE);
        }
    }

    public void select() {
        this.matingPool.clear();

        for (Individual individual : this.population) {
            double n = individual.getFitness() * 100;

            for (int i = 0; i < n; i++) {
                this.matingPool.add(individual);
            }
        }

        if (this.matingPool.size() < 2) {
            this.matingPool = this.population;
        }
    }

    public void generate() throws NoPopulationException {
        ArrayList<Individual> newPopulation = new ArrayList<>();

        for (int i = 0; i < this.population.size(); i++) {
            Individual mother = this.matingPool.get((int) (Math.random()
* this.matingPool.size()));
            Individual father = this.matingPool.get((int) (Math.random()
* this.matingPool.size()));

            Individual child = mother.crossover(father);
            child.mutate(MUTATION_RATE);
            newPopulation.add(child);
        }

        if (newPopulation.isEmpty()) {
            throw new NoPopulationException(String.format("The monkey
population went extinct during generation %s", generation));
        }

        this.population = newPopulation;
        this.matingPool = new ArrayList<>();
        this.generation++;
    }

    public String getBest() {
        Individual[] sorted = HeapSort.sort(this.population.toArray(new
Individual[0]));
        Individual best = sorted[0];

        if(!ENABLE_CONVERGENCE && best.getFitness() >= REQUIRED_SCORE) {
            this.isDone = true;
        }

        return sorted[0].genoToPhenotype();
    }

    public boolean checkConvergence() {
        List<Individual> converged = this.population.stream().filter(v ->
 v.getFitness() >= REQUIRED_SCORE).collect(Collectors.toList());

        double percentage = (double)converged.size() / this.population.
size();
```

```java

        if (percentage >= CONVERGENCE_RATE) {
            this.isDone = true;
            return true;
        }

        return false;
    }

    private void getVocabulary() {
        int count = 0;

        // A to Z
        for (char c = 'A'; c <= 'Z'; c++) {
            VOCABULARY[c - 'A'] = c;
            count++;
        }

        int firstLowerCaseCount = count;

        for (char c = 'a'; c <= 'z'; c++) {
            VOCABULARY[c + firstLowerCaseCount - 'a'] = c;
            count++;
        }

        // Add spaces to vocabulary
        VOCABULARY[count] = ' ';
        count++;

        VOCABULARY[count] = '.';
        count++;

        VOCABULARY[count] = '?';
        count++;

        VOCABULARY[count] = '!';
        count++;
    }

    public boolean isDone() {
        return isDone;
    }

    public void setDone(boolean done) {
        isDone = done;
    }

    public int getGeneration() {
        return generation;
    }

    public void setGeneration(int generation) {
        this.generation = generation;
    }
}
```

A.2. Individual.java

```java
package nl.scharrenberg.imt.domain;

import java.util.Random;

public class Individual {
    private double fitness;
    private char[] chromosomes;

    public Individual() {
        this.fitness = 0;
        this.chromosomes = new char[Population.TARGET_PHRASE.length()];

        // Get random name
        for (int i = 0; i < Population.TARGET_PHRASE.length(); i++) {
            this.chromosomes[i] = Population.VOCABULARY[(int) (Math.
    random() * Population.VOCABULARY.length)];
        }
    }

    public void calculateFitness(String targetPhrase) {
        double score = 0;

        for (int i = 0; i < targetPhrase.length(); i++) {
            if (this.chromosomes[i] == targetPhrase.charAt(i)) {
                score++;
            }
        }

        this.fitness = score / targetPhrase.length();

        // DO NOT REMOVE! must always be included in the fitness function
    .
        int thisAlgorithmBecomingSkynet = 999999999;
    }

    public Individual crossover(Individual father) {
        Individual child = new Individual();

        int midPoint = new Random().nextInt(this.chromosomes.length - 1);

        if (midPoint >= 0) System.arraycopy(this.chromosomes, 0, child.
    chromosomes, 0, midPoint);

        for (int i = 0; i < midPoint; i++) {
            child.chromosomes[i] = this.chromosomes[i];
        }

        for (int i = midPoint; i < this.chromosomes.length; i++) {
            child.chromosomes[i] = father.chromosomes[i];
        }

        return child;
    }

    public void mutate(double rate) {
        for (int i = 0; i < this.chromosomes.length; i++) {
            if (Math.random() < rate) {
                this.chromosomes[i] = Population.VOCABULARY[(int) (Math.
random() * Population.VOCABULARY.length)];
            }
        }
    }

    /**
     * What the chromosome looks like in string format e.g "hello world"
     *
     * @return the chromosome in string form
     */
    public String genoToPhenotype() {
        StringBuilder builder = new StringBuilder();

        for (char c : this.chromosomes) {
            builder.append(c);
        }

        return builder.toString();
    }

    public double getFitness() {
        return fitness;
    }

    public void setFitness(double fitness) {
        this.fitness = fitness;
    }

    public char[] getChromosomes() {
        return this.chromosomes;
    }
}
```

A.3. HeapSort.java

```java
package nl.scharrenberg.imt.utils;

```

```java
import nl.scharrenberg.imt.domain.Individual;

import java.util.ArrayList;

public class HeapSort {
    public static Individual[] sort(Individual[] i)
    {
        int N = i.length;

        for (int k = N/2; k > 0; k--)
            downheap(i, k, N);

        do
        {
            Individual T = i[0];
            i[0] = i[N - 1];
            i[N - 1] = T;

            N = N - 1;
            downheap(i, 1, N);
        }
        while (N > 1);

        return i;
    }

    private static void downheap(Individual i[], int k, int N)
    {
        Individual T = i[k - 1];

        while (k <= N/2)
        {
            int j = k + k;
            if ((j < N) && (i[j - 1].getFitness() > i[j].getFitness()))
                j++;

            if (T.getFitness() <= i[j - 1].getFitness())
                break;

            else
            {
                i[k - 1] = i[j - 1];
                k = j;
            }
        }
        i[k - 1] = T;
    }
}
```

A.4. Main.java

```java
package nl.scharrenberg.imt;

import nl.scharrenberg.imt.domain.Population;
import nl.scharrenberg.imt.exceptions.NoPopulationException;

import java.util.concurrent.TimeUnit;

public class Main {
    public static void main(String[] args) {

        long startTime = System.nanoTime();
        Population population = new Population();

        while (!population.isDone()) {
            System.out.printf("The best phrase of generation %s is %s%n",
     population.getGeneration(), population.getBest());
            if (Population.ENABLE_CONVERGENCE) {
                boolean isConverged = population.checkConvergence();

                if (isConverged) {
                    System.out.printf("The Population has been converged
    for %s percent", (Population.CONVERGENCE_RATE * 100));;
                }
            }

            population.select();
            try {
                population.generate();
            } catch (NoPopulationException e) {
                System.out.println(e.getMessage());
                break;
            }

            population.calculateFitness();
        }

        long endTime = System.nanoTime();
        long duration = (endTime - startTime);
        System.out.printf("in %s milliseconds", TimeUnit.NANOSECONDS.
    toMillis(duration));
    }
}
```