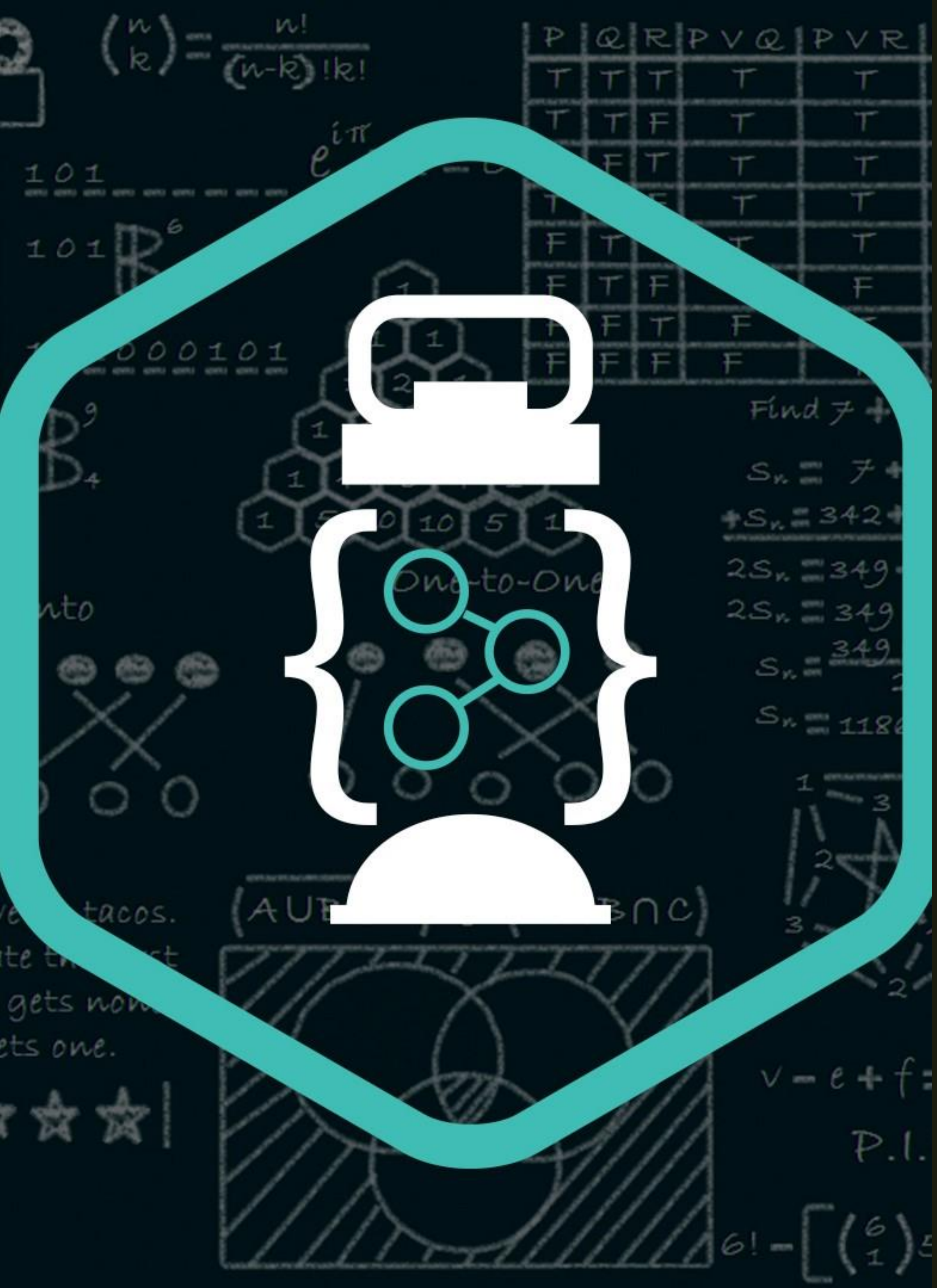


GENETIC ALGORITHM

A Project On Text Generation

By

Vaishali Tripathi
NUID: 001418444

Payal Zanwar
NUID: 001491724

Contents

Genetic Algorithms: What, Why & How

What.

- The term "Genetic Algorithm" refers to a specific algorithm implemented in a specific way to solve specific sorts of problems. [Daniel Shiffman, *Nature of Code – Chapter 9*]
- This algorithm is inspired by process of "natural selection" which basically belongs to the larger class of evolutionary algorithms (EA).
- These algorithms are used to **generate high quality solutions to optimize and search problems** by relying on bio-inspired operations such as "crossover", "mutation", "selection" etc.
- John Holland introduced **Genetic Algorithm** (**GA**) in 1960 based on the concept of **Darwin's theory of evolution**

> *"The fact that life evolved out of nearly nothing, some 10 billion years after the universe evolved out of literally nothing, is a fact so staggering that I would be mad to attempt words to do it justice."*
>
> — Richard Dawkins

Genetic Algorithms: What, Why & How

Why.

- Because they are simple to program!
- Proven to find the global optimum if given enough time.
- Can be applied to diverse problem domains, etc.
- **Can be used when we know what's a good solution but we can't figure out the road to that solution.** It provides a good way to search and traverse the space of possible solutions in a smart way.
- Along with easy implementation, it is easy to trace the performance.

About this project – Problem Statement

Genetic Algorithms are one of the best ways to go when we know a good solution but we can't figure out the road to that solution.

Let's consider the string "***Program Structures and Algorithms is the best class!!***". This string is 54 characters long. If a system starts guessing this string character by character, the probability that the system guesses all the 54 characters right will be

(1/27) multiplied by itself 54 times!!! I.e., $(1/27)^{54}$

which equals a 1 in 50,857,702,033,867,822,607,895,549,241,096,482,953,017,615,834,735, 226,163,958,950 chance of getting it right!

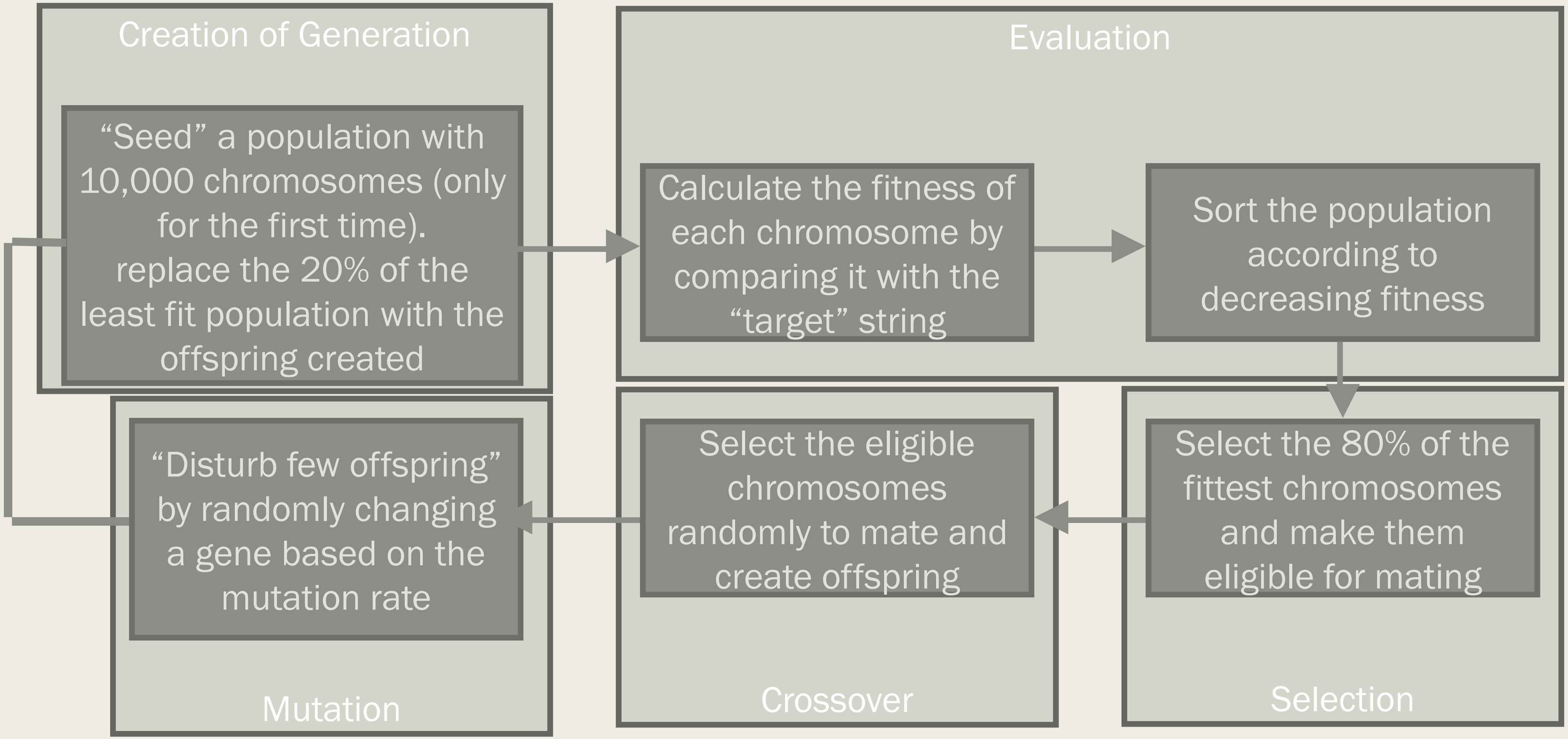
Now, the chances are, if this system has capability of generating one million phrase per second, for the system to have a 99% probability to get the string right, it will have to work for 9,719,096,182,010,563,073,125,591,133,903,305,625,605,017 years.

However, using Genetic Algorithm to do this would hardly take few seconds! And that's exactly what we are here to demonstrate!

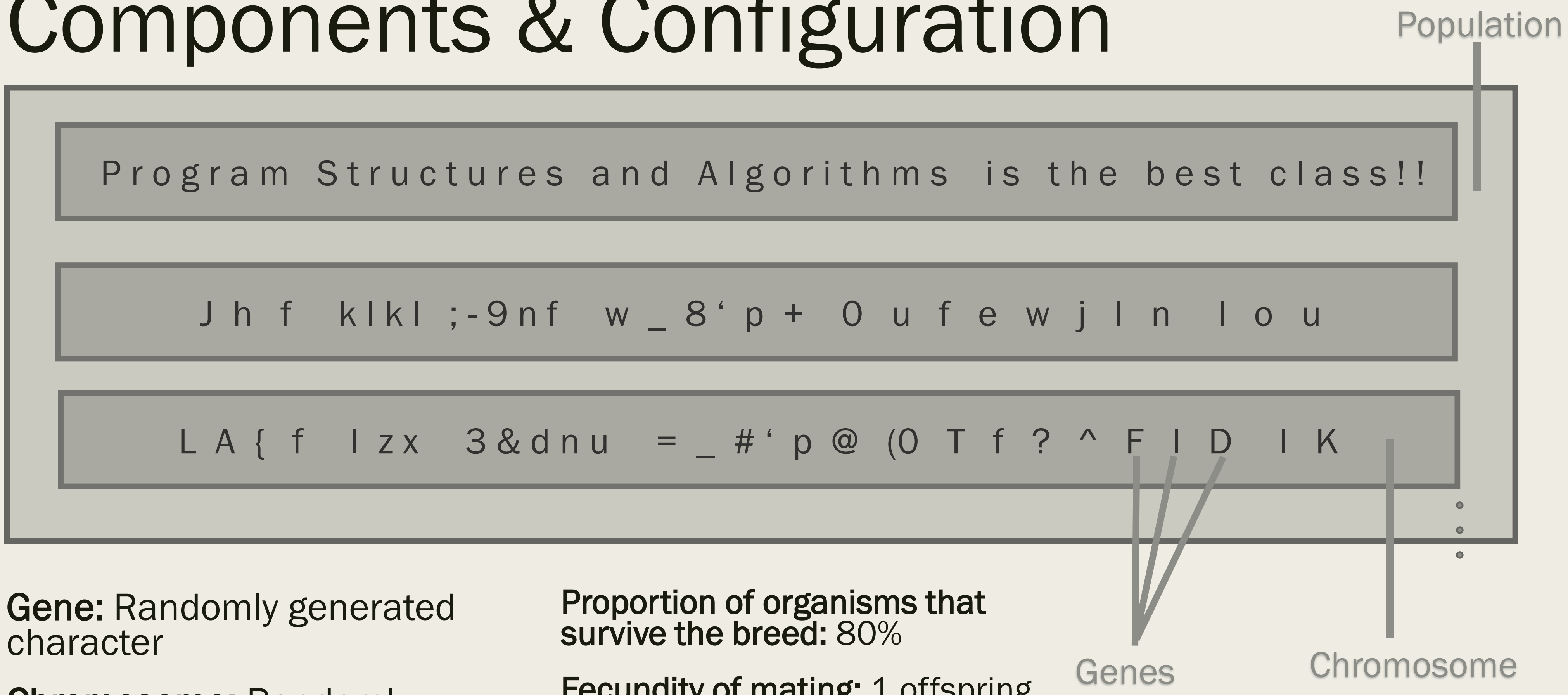
```
Progr m Siru.tures anF Aleorithms i9  he best cla;s!!
Progr m Siru.tures anF Aleorithms i9  he best cla;s!!
Progr m Siru.tures anF Aleorithms i9  he best cla;s!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
Prygram Struc9u es and Algo ithms i; 8he  est claNs!!
P7ogram Struc9u eJ and Algo ithms  s tne best clask!!
P7ogram Struc9u eJ and Algo ithms  s tne best clask!!
P7ogram Struc9u eJ and Algo ithms  s tne best clask!!
P7ogram Struc9u eJ and Algo ithms  s tne best clask!!
P7ogram Struc9u eJ and Algo ithms  s tne best clask!!
Progra- Structures Vnd Algo6ithms 4s thZ best claas!!
Progra- Structures Vnd Algo6ithms 4s thZ best claas!!
Progra- Structures Vnd Algo6ithms 4s thZ best claas!!
Progra- Structures Vnd Algo6ithms 4s thZ best claas!!
Progra- Structures Vnd Algo6ithms 4s thZ best claas!!
Progra- Structures Vnd Algo6ithms 4s thZ best claas!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Program Structures a d Adgo6ithms is the best cl2ss!!
Progra- Structures and Algorithms io t e best class!!
Progra- Structures and Algorithms io t e best class!!
Progra- Structures and Algorithms io t e best class!!
Program Siruc6ures{and Algorithms is the best class!!
Program Siruc6ures{and Algorithms is the best class!!
Program Siruc6ures{and Algorithms is the best class!!
Program Siruc6ures{and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Pr gram Structures and Algorithms is the best class!!
Program Structures and Algorithms is the best class!!
BUILD SUCCESSFUL (total time: 5 seconds)
```

IMPLEMENTATION

The Flow


Creation of Generation
"Seed" a population with 10,000 chromosomes (only for the first time).
replace the 20% of the least fit population with the offspring created
Evaluation
Calculate the fitness of each chromosome by comparing it with the "target" string
Sort the population according to decreasing fitness
Selection
Select the 80% of the fittest chromosomes and make them eligible for mating
Crossover
Select the eligible chromosomes randomly to mate and create offspring
Mutation
"Disturb few offspring" by randomly changing a gene based on the mutation rate

Components & Configuration



Gene: Randomly generated character

Chromosome: Randomly generated string of genes

Population: Array List of chromosomes

Population Size: 10,000

Proportion of organisms that survive the breed: 80%

Fecundity of mating: 1 offspring per pair

Generations to reproductive maturity: 2^{nd} generation

Maximum number of generations: 10,000

Phenotype: String made of characters

Genotype: The type of character – i.e., punctuation, number, lowercase letter, uppercase letter

Classes and Methods

- **Gene**

generateGene() – whenever called, generates and returns uppercase letter, lowercase letter, punctuation, space or a number.

- **Chromosome**

generateChromosome() – generates a string type chromosome using genes.

calculateFitness() – calculates fitness of each chromosome by comparing their genes to the target string and assigns fitness value to each chromosome in the generation.

```java
public class Gene {

    Random r = new Random();

    public char generateGene(){
        char[] punctuations = new char []{',', '.','!','-','{','}','&','*',';'};
        char[] numbers = new char[]{'1','2','3','4','5','6','7','8','9','0'};
        char[] space = new char[]{' '};

        if(r.nextBoolean())
            if(r.nextBoolean())
              return (char)(r.nextInt(26) + 'a');
            else
                if (r.nextBoolean())
                return punctuations[r.nextInt(punctuations.length)];
                else
                return (char) (r.nextInt(26) + 'A');
        else
            if(r.nextBoolean())
                return numbers[r.nextInt(numbers.length)];
            else
                return space[r.nextInt(space.length)];
    }

}
```

```java
public void generateChromosome(int chromozomeLength) {
    int minRange = 0;
    int maxRange = chromozomeLength;
    String str = generateRandomWord(r.nextInt(maxRange - minRange));
    while (!(str.length() == maxRange)) {
        str = str + " " + generateRandomWord(r.nextInt(maxRange - str.length()));
    }
    candidateString = str;
    setCandidateString(candidateString);
    setFitness(calculateFitness(candidateString));
}
```

```java
public double calculateFitness(String candidateString) {
    double score = 0;
    for (int i = 0; i < candidateString.length(); i++) {
        if (candidateString.charAt(i) == target.charAt(i)) {
            score++;
        }
    }
    fitness = (score / target.length()) + 0.01;
    return fitness;
}
```

Classes and Methods

- **Population**

createPopulation() – creates Array List of chromosomes of the given size and sorts the generation (or population) obtained on the basis of their fitness.

NaturalSelection() – creates a **mating pool** of type array list and adds the fittest 80% population to it. Then it chooses 2 partners (chromosomes) at random, performs **crossover**, **mutates** the offspring and replaces the least fit 20% of the generation with the children.

crossover() – accepts two chromosomes, randomly selects a mating point and merges the two chromosomes to create a child.

```
public void createPopulation() {
    while (!(generation.size() == populationSize)) {
        generation.add(new Chromosome(target));
    }
    Collections.sort(generation);
}
```

```
public void NaturalSelection() {
    matingPool.clear();
    for (int i = 0; i < generation.size() * 0.8; i++) {
        matingPool.add(generation.get(i));
    }
    Collections.sort(generation);
    for (int j = 0; j < generation.size() * 0.2; j++) {
        int a = (r.nextInt(matingPool.size()));
        int b = (r.nextInt(matingPool.size()));
        Chromosome partnerA = matingPool.get(a);
        //System.out.println("Partner A: "+partnerA.getCandidateString());
        Chromosome partnerB = matingPool.get(b);
        //System.out.println("Partner B: "+partnerB.getCandidateString());

        Chromosome child = crossover(partnerA, partnerB, target);
        mutate(child, mutationRate);
        generation.set(generation.size() - (j + 1), child);

    }
}
```

```
public Chromosome crossover(Chromosome partnerA, Chromosome partnerB, String target) {
    Chromosome child = new Chromosome(target);
    int midpoint = (r.nextInt(target.length()));
    char[] str1 = new char[target.length()];
    // Half from one, half from the other
    for (int i = 0; i < target.length(); i++) {
        if (i > midpoint) {
            str1[i] = partnerA.getCandidateString().charAt(i);
        } else {
            str1[i] = partnerB.getCandidateString().charAt(i);
        }
    }
    String str = String.valueOf(str1);
    child.setCandidateString(str);
    child.setFitness(child.calculateFitness(str));
    return child;
}
```

Classes and Methods

- **Population**

mutate() – this function accepts the child and a mutation rate value and modifies (disturbs) random genes of the child chromosome according the mutation rate

```java
public void mutate(Chromosome child, double mutationRate) {
    double rangeMin = 0.0f;
    double rangeMax = 1.0f;
    double createdRanNum = 0;

    char[] childChars = child.getCandidateString().toCharArray();
    for (int i = 0; i < childChars.length; i++) {
        createdRanNum = rangeMin + (rangeMax - rangeMin) * r.nextDouble();
        if (createdRanNum < mutationRate) {
            gene = new Gene();
            childChars[i] = (char) gene.generateGene();
        }
    }
    child.setCandidateString(String.valueOf(childChars));
}
```

evaluate() – evaluates the complete generation by comparing the fitness of each chromosome and returns the chromosome with highest fitness

```java
public void evaluate() {
    double worldrecord = 0.0;
    int index = 0;
    for (int i = 0; i < generation.size(); i++) {
          System.out.println("Generation fitness of index ["+i+"]:["+genera
        if (generation.get(i).getFitness() > worldrecord) {
            index = i;
            worldrecord = generation.get(i).getFitness();
        }
    }
    if (worldrecord == perfectScore) {
        finished = true;
    }
    best = new Chromosome(target);
    best.setCandidateString(generation.get(index).getCandidateString());
}
```

getAverageFitness() – returns the average fitness of the generation

DEMO

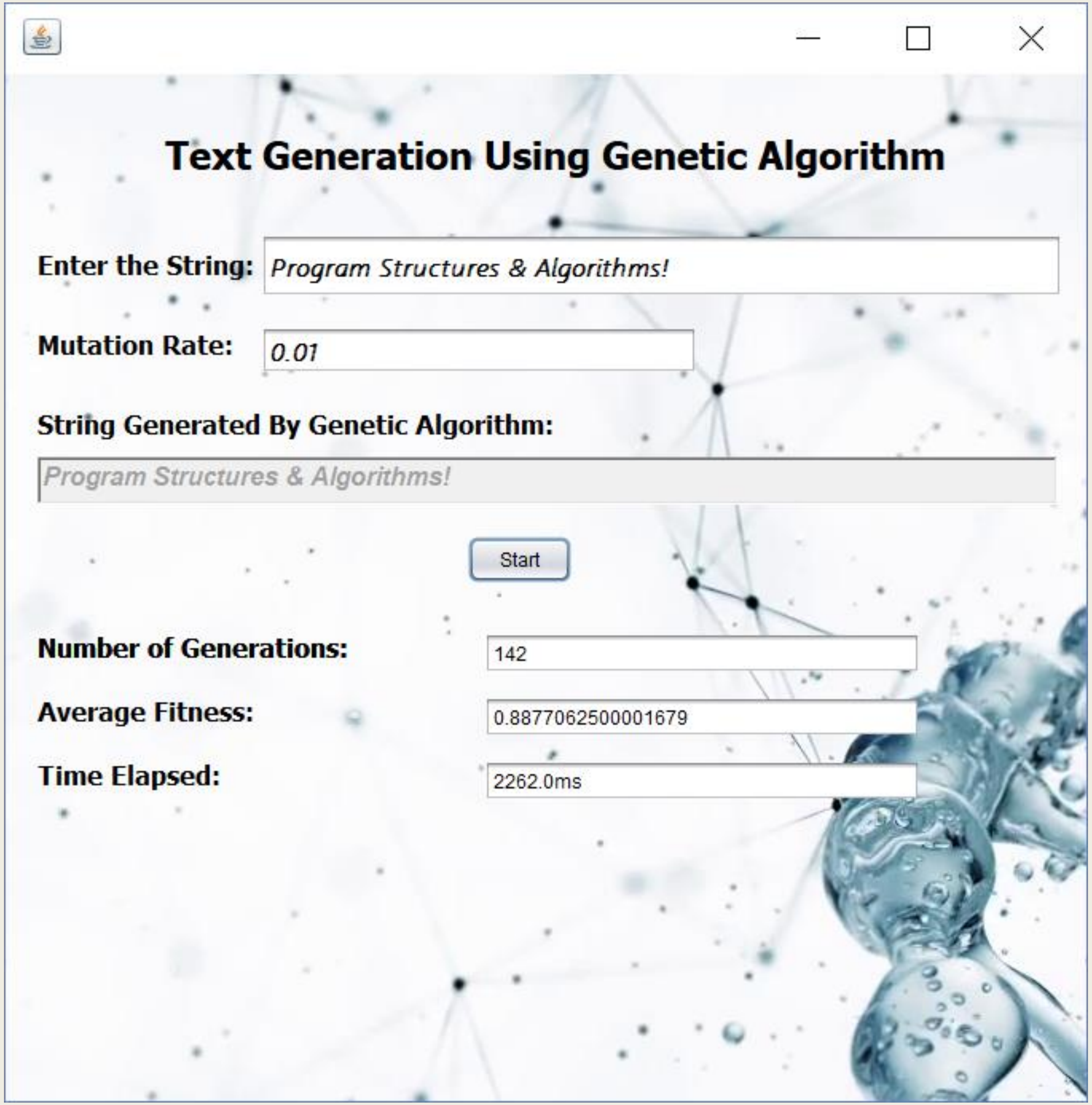
CLICK HERE FOR THE DEMO

GIT

LINK TO GITHUB REPOSITORY

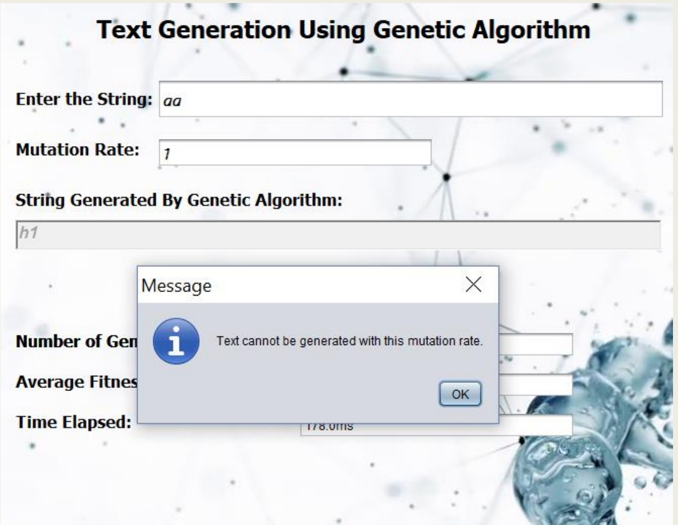
Output Details

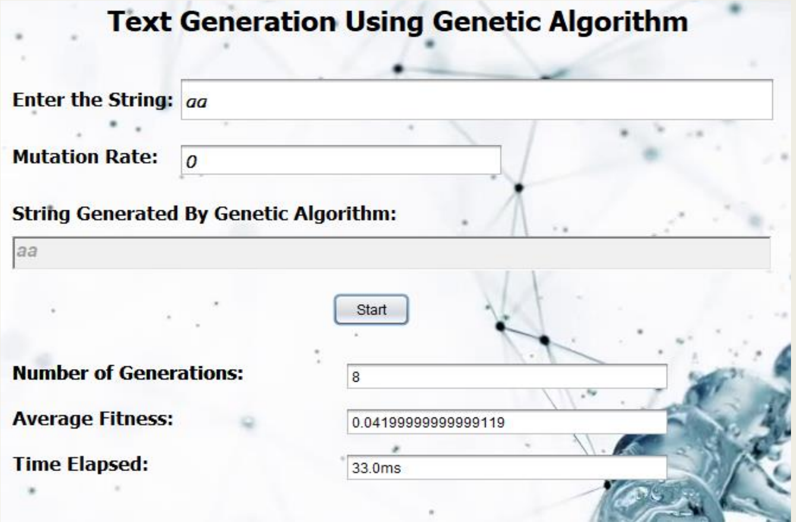
- The application accepts any **string of user's choice** and applies Genetic Algorithm to generate the same string as an output based on the **mutation rate entered by the user.**
- The total **number of generations** involved along with the **average fitness** of the generation and total **time taken by the algorithm (benchmarking)** to generate the output.

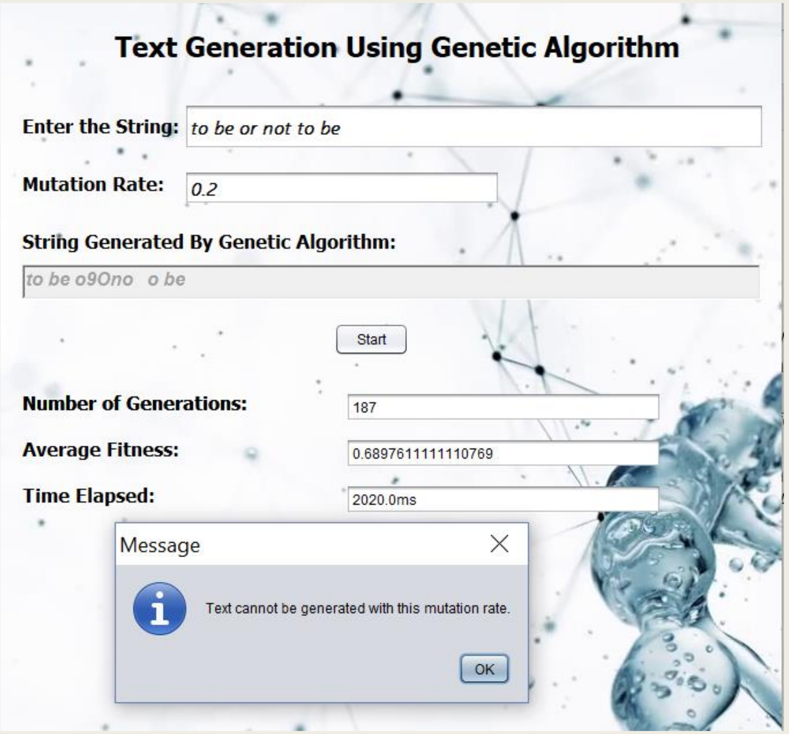


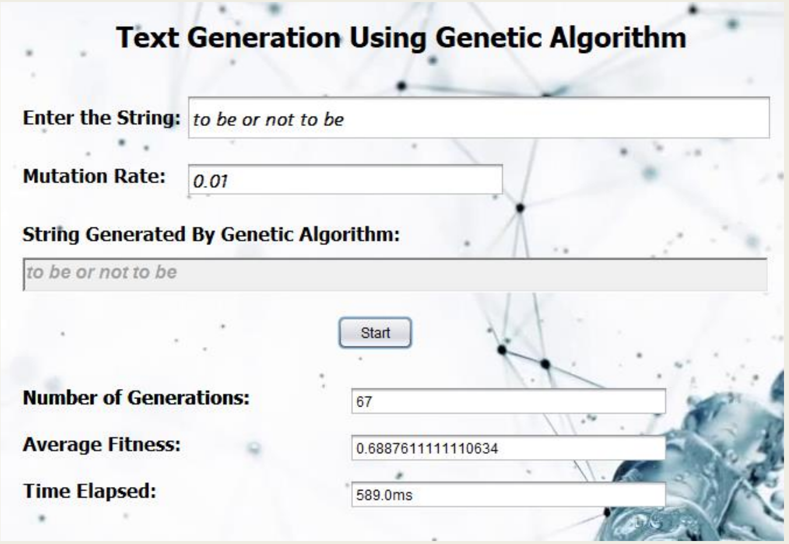
Observations

- The number of generations required to evolve a sentence vary with each execution.
- The number of generations and time required for evolution change drastically with change in the mutation rate
- For a given population size, the algorithm with get the results fast and with least number of generations only for a given range of mutation rate. As we go on considering lower or higher mutation rates, the number of generations to reach the target string may increase or the final string may not be correct at all
- If string size is too less, then is it normal to go in a infinity loop for a large mutation rate



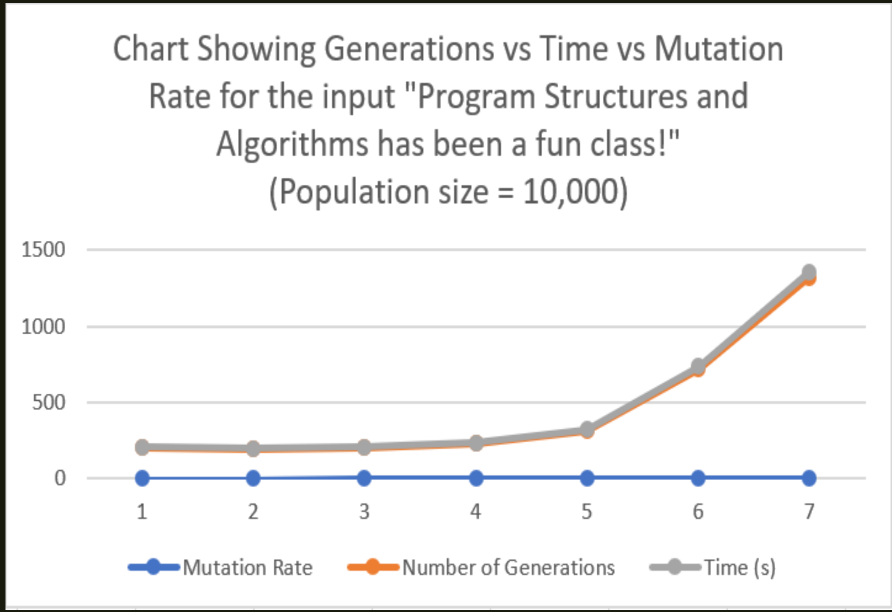






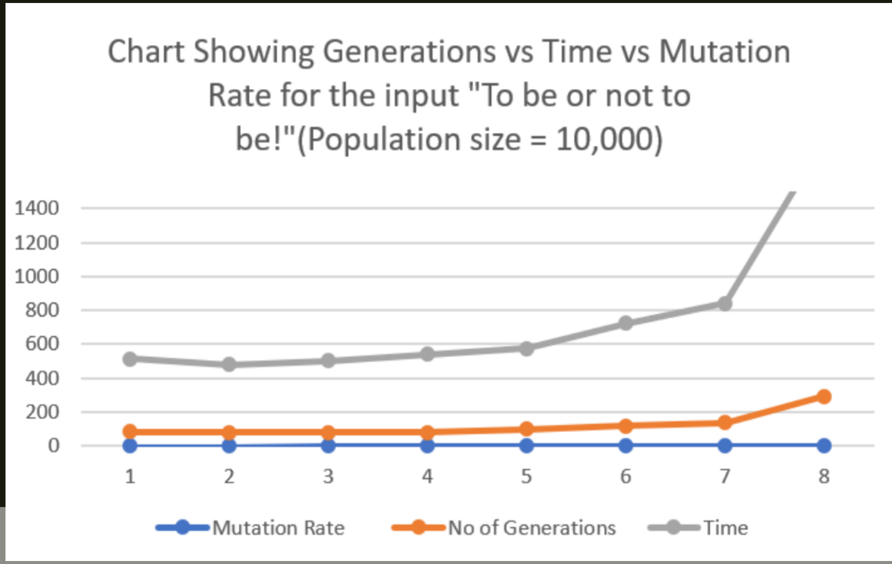
“Program Structures and Algorithms has been a fun class!” (String length: 55)

Mutation Rate	Number of Generations	Time (s)	Comments
0	203	5.147	
0.0001	195	4.837	Least number of generations Fastest text generation
0.001	203	5.051	
0.01	229	6.132	
0.04	315	7.882	
0.08	719	18.714	
0.1	1322	35.72	
0.2	Above 10,000	NA	Too much mutation to maintain the stability
0.5	Above 10,000	NA	Too much mutation to maintain the stability



“To be or not to be!” (String length: 19)

Mutation Rate	Number of Generations	Time (ms)	Comments
0	83	514	
0.0001	80	479	Fastest text generation
0.001	79	504	Lowest no. of generations
0.01	82	540	
0.04	101	576	
0.08	117	723	
0.1	139	841	
0.2	294	1810	
0.5	Above 10,000	NA	Too much mutation to maintain the stability



The Test Cases and Test Results

testGenerateChromosome() – generates a chromosome and tests against the passed chromosome size

testCalculateFitness() – calculates fitness of a chromosome and tests against the target string

testGenerateGene() – tests the generated gene based of the gene size (1 character in this case) and sample space

testCreatePopulation() – creates a population based on the population size passed

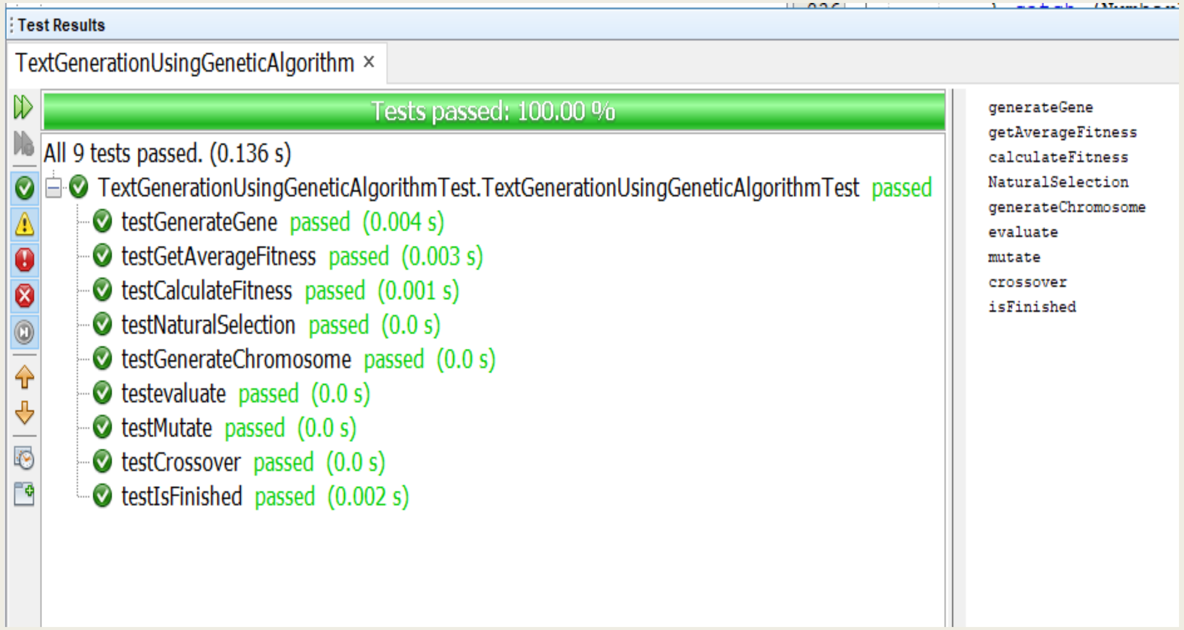
testNaturalSelection() – checks if the method NaturalSelection() creates a mating pool of 80% of the most fit generation

testCrossover() – tests if two chromosomes are merged to generate a new offspring successfully

testMutate() – tests if the method mutates a chromosome genes based on passed mutation rate

testIsFinished() – passes the target fitness value and checks if the isFinished() method is returning true

testGetAverageFitness() – tests the output of average fitness of a generation



Exclusive Features & Applications

Features.

- **Logger:** Used Apache *log4j* for logging the program
- **User Interface:** User interface for accepting user defined string and mutation rate dynamically
- The program terminates if the number of generations exceed 10,000

Applications.

- This implementation can be useful for Genetic Algorithm Search for predictive patterns in Multidimensional time series
- Useful for efficient dictionary search
- Mutation testing applications
- Rare event analysis
- Selection of optimal mathematical model to describe biological systems
- Timetabling problems, such as designing a non-conflicting class timetable for a large university

Conclusion

From this project, we understood that

- Genetic Algorithm can offer efficient way of searching and text generation if the target is known.
- It is not necessary to get the right result every time, unless the combinations of mutation rate and population size selected are in right range
- For population size of 10,000, the mutation rate that suited almost all sizes of string is 0.01.

References

- Daniel Shiffman, “Nature of Code”, Chapter 09
- Wikipedia
- Vijini Mallawaarachchi’s article on “Introduction To Genetic Algorithms” featured at www.towardsdatascience.com
- Paper on “A NEW APPROACH FOR DATA ENCRYPTION USING GENETIC ALGORITHMS” featured at www.researchgate.net

Thank You ☺