Character Generation Using Genetic Algorithm

By

Dhairyasheel Surve

Chetan Shirsath

Sarthak Hoshangabade

Genetic Algorithm

- It is a method to solve constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The **genetic algorithm** repeatedly modifies a population of individual solutions
- Mutation and Crossover are two important features to generate healthiest population

Problem Statement

- Consider a sentence where we want to predict or regenerate exact same characters, a normal system can be used for small lengths but if the length goes on increasing then time required increases exponentially
- For eg : The string “Say hello to my little friend!!” contains 30 characters and the probability to determine them is (1/27)*30 and millions of years time will taken to determine the exact string
- Genetic Algorithm can be applied where the solution is already known and we want to figure out the steps or way to the solution

Stepwise Flow

- Generation Creation

20% of least fit population is replaced by chromosomes

- Evaluation

Fitness is calculated depending on the input characters and population is sorted depending on decreasing fitness

- Selection

80% of the fittest population is selected

- Crossover

Chromosomes are selected randomly for mating and offspring is created

- Mutation

Characters are changed randomly based on mutation rate and offsprings are disturbed

Code snippets

1) Crossover() : Accepts two chromosomes and mates them to get a offspring
2) Evaluate () : returns the chromosome with highest fitness
3) getAverageFitness() : Returns the average fitness of the generation

```
 * @return average fitness level
 */
public double getAverageFitness() {
    double total = 0;
    for (int i = 0; i < generation.size(); i++) {
        total += generation.get(i).getChromosomeModel().getFitness();
    }
    return total / (generation.size());
}
}
```

```
 */
public double evaluate(ArrayList<Chromosome> generation) {
    double worldrecord = 0.0;
    int index = 0;
    for (int i = 0; i < generation.size(); i++) {
        if (generation.get(i).getChromosomeModel().getFitness() > worldrecord) {
            index = i;
            worldrecord = generation.get(i).getChromosomeModel().getFitness();
        }
    }
    if (worldrecord == perfectScore) {
        finished = true;
    }
    best = new Chromosome(inputString);
    best.getChromosomeModel().setCandidateString(generation.get(index).getChromosomeModel().getCandidateString());
return worldrecord;
}
```

```
 */
public Chromosome crossover(Chromosome partner1, Chromosome partner2, String target) {
    Chromosome cObj = new Chromosome(target);
    int midpoint = (random.nextInt(target.length()));
    char[] chr = new char[target.length()];
    for (int i = 0; i < target.length(); i++) {
        if (i > midpoint) {
            chr[i] = partner1.getChromosomeModel().getCandidateString().charAt(i);
        } else {
            chr[i] = partner2.getChromosomeModel().getCandidateString().charAt(i);
        }
    }
    String string = String.valueOf(chr);
    cObj.getChromosomeModel().setCandidateString(string);
    cObj.getChromosomeModel().setFitness(cObj.getFitness(string));
    return cObj;
}
```

4) getPopulation() : Returns a sorted arraylist of the fit population
5) Mutate() : Accepts a mutation rate value and disturbs the generated offspring to get better results

```
public void getPopulation(ArrayList<Chromosome> generation) {
    while (!(generation.size() == populationSize)) {
        generation.add(new Chromosome(inputString));
    }
    Collections.sort(generation);
}
```

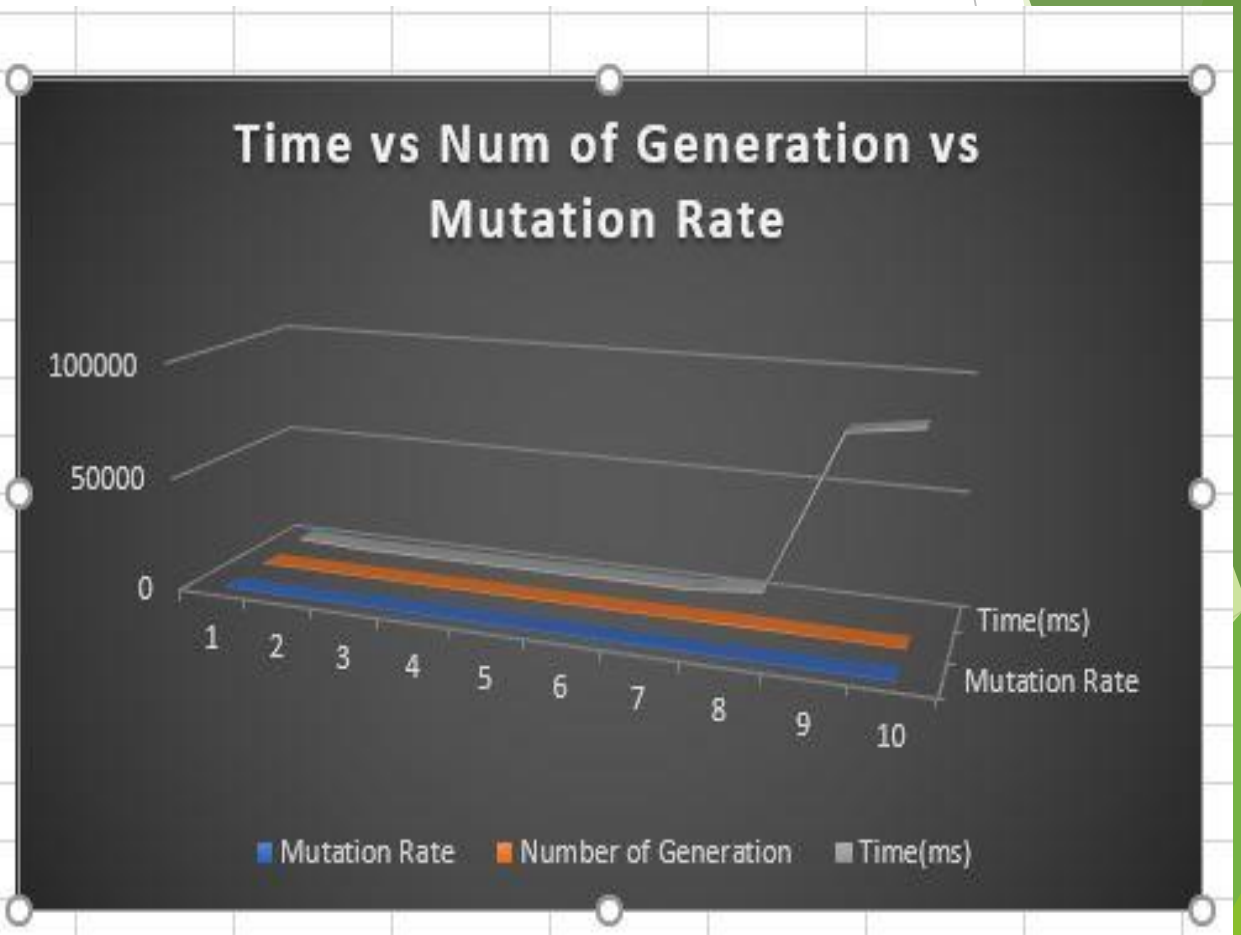
```
 */
public void mutate(Chromosome cObj, double mutationRate) {
    double rangeMin = 0.0f;
    double rangeMax = 1.0f;
    double randomNum = 0;

    char[] childChars = cObj.getChromosomeModel().getCandidateString().toCharArray();
    for (int i = 0; i < childChars.length; i++) {
        randomNum = rangeMin + (rangeMax - rangeMin) * random.nextDouble();
        if (randomNum < mutationRate) {
            gene = new Gene();
            childChars[i] = (char) gene.randomGene();
        }
    }
    cObj.getChromosomeModel().setCandidateString(String.valueOf(childChars));
}
/*
```

Observations

- Change in mutation rate is responsible for number of generation and time required
- If the Characters are too few it results in infinite loop if the mutation rate is high
- Number of generations required to evolve or predict given characters vary with each execution
- For the particular mutation rate characters are predicted is lesser generation and in less time
- If the mutation rate is increased uniformly then the number of generations may increase or decrease

Graphical Analysis

String :	*PSA final project*	
Mutation Rat	Number of Generatio	Time(ms)
0	80	1700
0.01	84	669
0.02	86	691
0.07	105	907
0.08	105	912
0.09	122	1098
0.1	119	1017
0.3	503	4042
0.5	greater than 10000	76381
0.7	greater than 10000	80806



Test cases

1) testNaturalSelection() : checks if 80% pool of the most fit generation is created
2) testGetAverageFitness() : tests the average fitness of the generation
3) testGenerateChromosome() : generates a chromosome and tests against the passed chromosome size
4) testMutate() : checks if the method mutates the gene as per the given mutation rate
5) testGetFitness() : calculates fitness and tests against target String
6) testCrossover() : tests if two chromosomes are mated successfully to generate a offspring