გენეტიკური ალგორითმის შექმნა დამწყებთათვის

შესავალი

ძებნის რთული ამოცანების ამოხსნისათვის გენეეტიკური ალგორითმი არის საუცხოო საშუალება. ისინი ხშირად გამოიყენებიან ისეთ დარგებში როგორიცაა ტექნიკა, რომ შევქმნათ დაუჯერებლად მაღალი ხარისხის პროდუქტები. მაგალითად, მათ აქვთ შესაძლებლობა მოძებნონ მასალებისა და დიზაინების დიდი რაოდენობა განსხვავებულ კომბინაციებს შორის, რომ იპოვნონ უკეთესი კომბინაცია, რომელიც შედეგად გვაძლევს მაგარ, მსუბუქ და ყოველმხრივ უკეთეს პროდუქტს. ისინი შეიძლება აგრეთვე იყვნენ გამოყენებულნი კომპიუტერული ალგორითმების შესაქმნელად, განრიგის ამოცანებში და სხვა ოპტიმიზაციის საკითხებში. გენეტიკური ალგორითმები არიან დაფუძნებულნი ევოლუციურ პროცესზე ბუნებრივი შერჩევის გზით, რასაც ვხვდებით ბუნებაზე დაკვირვების დროს. ისინი არსებითად იმეორებენ ხერხს, რომელსაც ცხოვრება იყენებს რეალური სამყაროს პრობლემების გადაწყვეტის საპოვნელად. მიუხედავად იმისა, რომ გენეტიკური ალგორითმები შეიძლება გამოყენებულნი იყვნენ დაუჯერებლად რთული პროლემების ამოსახსნელად, მოულოდნელად ისინი არიან ძალზე მარტივნი გამოსაყენებლად და გასაგებად.

როგორ მუშაობენ ისინი

როგორც უკვე ვიცით, ისინი დაფუძნებული არიან ბუნებრივი შერჩევის პროცესზე. ეს ნიშნავს იმას, რომ ისინი იღებენ ბუნებრივი შერჩევის ფუნდამენტალურ თვისებებს და იყენებენ მათ ნებისმიერი პრობლემის გადასაწყვეტად, რომელსაც ისინი ცდილობენ, რომ ამოხსნან .

ძირითადი პროცესი გენეტიკური ალგორითმებისთვის არის:

1. ინიციალიზაცია - საწყისი პოპულაციის(დასახლების) შექმნა. ეს პოპულაცია ჩვეულებრივ გენერირებულია შემთხვევითად და შეიძლება იყოს ნებისმიერი ზომის, ცოტაოდენი რაოდენობა აღებული ათასებიდან.
2. შეფასება - შემდეგ პოპულაციის ყოველი წევრისათვის ვაფასებთ და ვითვლით მის ვარგისიანობას. ვარგისიანობის მნიშვნელობა გამოითვლება, თუ რამდენად კარგია ეს წევრი, ჩვენთვის სასურველი მოთხოვნებისათვის. ეს მოთხოვნები უნდა იყვნენ მარტივნი, უფრო სწრაფი ალგორითმები არიან

უკეთესნი, ან უფრო კომპლექსურად, მაგარი მასალები არიან უკეთესნი, მაგრამ ისინი არ უნდა იყვნენ ძალზე მძიმე.

3. შერჩევა - ჩვენ გვჭირდება მუდმივად გავაუმჯობესოთ ჩვენი პოპულაციების მთლიანად ვარგისიანობა. შერჩევა გვეხმარება ჩვენ ცუდი დიზაინების მოცილებაში და პოპულაციის მხოლოდ უკეთესი ელემენტების დატოვებაში. არსებობენ მხოლოდ ცოტაოდენი მეთოდებისა, მაგრამ ძირითადი იდეაა, ვარგისი ელემენტები იყვნენ შერჩეულნი ჩვენი შემდეგი გენერაციისათვის.
4. შეჯვარება - შეჯვარების დროს ჩვენ ვქმნით ახალ ელემენტებს შერჩეული ელემენტების სახეების შეერთებით. ჩვენ შეგვიძლია მივბაძოთ იმაში, თუ როგორ მოშაობს სექსი ბუნებაში. ჩვენ ვიმედოვნებთ, რომ ერთი ან მეტი ელემენტების ზოგიერთი ნიშნების კომბინირებით, მივიღებთ მემკვიდრეობით მათი მშობლების უკეთეს ნიშნებს.
5. მუტაცია - ჩვენ გვჭირდება დავამატოთ ცოტაოდენი შემთხვევითობა ჩვენს პოპულაციურ გენეტიკაში. სხვა სიტყვებით, რომ ვთქვათ, ამოხსნების ყოველი კომბინაცია, რომელიც ჩვენ შეგვიძლია შევქმნათ, სასურველია იყოს ჩვენ საწყის პოპულაციაში. ტიპიურად მუტაცია მუშაობს ძალზე მცირე ცვლილებების გაკეთებით შემთხვევითად ელემენტების გენებში.
6. და გავიმეოროთ - ახლა ჩვენ გვაქვს შემდეგი გენერაცია, რომლისგანაც შეგვიძლია დავიწყოთ ახალი ნაბიჯი, სანამ არ მივაღწევთ დამთავრების პირობას.

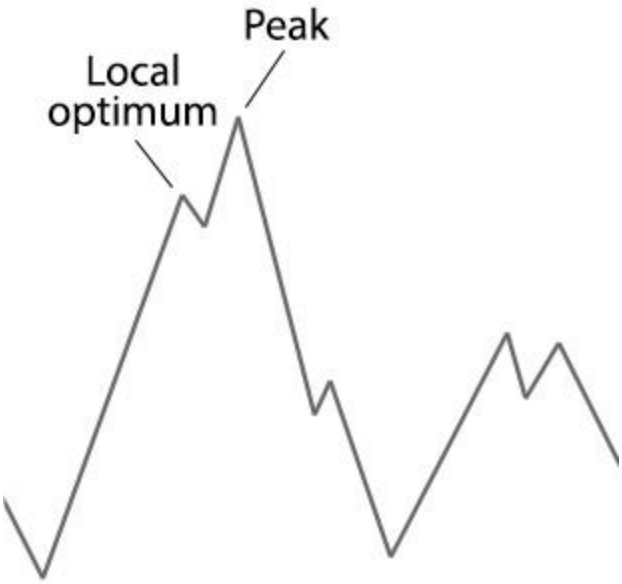
დამთავრება

+არსებობს რამოდენიმე მიზეზი, რომ დავამთავროთ ჩვენი გენეტიკური ალგორითმი. ყველაზე სასურველი მიზეზია ის, რომ ჩვენმა ალგორითმმა იპოვა ამონახსნი, რომელიც საკმარისად კარგია და აკმაყოფილებს წინასწარ განსაზღვრულ მინიმალურ კრიტერიუმს. სხვა მიზეზები შეიძლება იყოს შეზღუდვები, როგორიცაა დრო და ფული.

შეზღუდვები

წარმოიდგინეთ, რომ აგიხვიეს თვალები, დაგსვეს მთის ძირში და გითხრეს, რომ უნდა იპოვო ამ მთის უმაღლესი წვერო თქვენი საკუთარი ხერხით. თქვენ გაქვთ მხოლოდ ერთი არჩევანი, იაროთ მაღლა, სანამ ეს შესაძლებელია და როცა აღარ შეიძლება მეტად მაღლა სვლა, იგულისხმოთ, რომ მიაღწიეთ მთის მწვერვალს. მაგრამ თქვენ ვერ განსაზღვრავთ, ეს არის მთის ნამდვილი უმაღლესი წერტილი, თუ უბრალოდ ამაღლებული ადგილი(რასაც, ჩვენ ვუწოდებთ ლოკალურ ოპტიმუმს) და არსებობს სხვა უფრო მაღალი წვერო, რადგან თქვენ ხართ

თვალახვეული. ქვემოთ, მოცემულია თუ როგორ გამოიყურება ლოკალური ოპტიმუმი:



მიუხედავად, წვეროსკენ თვალახვეული სვლისა, გენეტიკურ ალგორითმს ხშირად შუძლია აიცილოს ლოკალური ოპტიმუმი, თუ ის არაა ძალზე მაღალი. თუმცა, ჩვენ არა ვართ ყოველთვის გარანტირებული, რომ გენეტიკური ალგორითმი იპოვის ჩვენი პრობლემის გლობალურ ოპტიმალურ ამოხსნას. უფრო რთული პრობლემებისათვის არსებობს გადაუწყვეტადი გამონაკლისები, რომ ვიპოვოთ გლობალური ოპტიმუმი. ყველაზე უკეთესი, რაც ჩვენ შეგვიძლია გავაკეთოთ არის ის, რომ ჩვენი ამონახსნი ახლოს იყოს ოპტიმალურ ამონახსნთან.

ძირითადი ბინარული გენეტიკური ალგორითმის რეალიზაცია Java-ზე

თუ თქვენ არა გაქვთ Java კომპილატორი. შეგიძლიათ იგი გადმოიწეროთ შემდეგი ვებ გვერდიდან:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

მოდით თვალი გადავავლოთ კლასებს, რომლებითაც ჩვენ ვაპირებთ გენეტიკური ალგორითმის პროგრამის შექმნას:

- Population - მართავს პოპულაციის ყველა წევრს
- Individual - მართავს პოპულაციის კონკრეტულ წევრს

- Algorithm - მართავს ჩვენს ევოლუციურ ალგორითმებს, როგორიცაა შეჯვარება და მუტაცია
- FitnessCalc - საშუალებას იძლევა განვსაზღვროთ ამონახსნი კანდიდატი და გამოვთვალოთ კონკრეტული წევრის ვარგისიანობა

Population.java

```
package simpleGa;

public class Population {

    Individual[] individuals;

    /*
     * Constructors
     */
    // Create a population
    public Population(int populationSize, boolean initialise) {
        individuals = new Individual[populationSize];
        // Initialise population
        if (initialise) {
            // Loop and create individuals
            for (int i = 0; i < size(); i++) {
                Individual newIndividual = new Individual();
                newIndividual.generateIndividual();
                saveIndividual(i, newIndividual);
            }
        }
    }

    /* Getters */
    public Individual getIndividual(int index) {
        return individuals[index];
    }

    public Individual getFittest() {
        Individual fittest = individuals[0];
        // Loop through individuals to find fittest
        for (int i = 0; i < size(); i++) {
            if (fittest.getFitness() <= getIndividual(i).getFitness()) {
                fittest = getIndividual(i);
            }
        }
        return fittest;
    }

    /* Public methods */
    // Get population size
    public int size() {
        return individuals.length;
```

```
    }

    // Save individual
    public void saveIndividual(int index, Individual indiv) {
        individuals[index] = indiv;
    }
}
```

Individual.java

```
package simpleGa;

public class Individual {

    static int defaultGeneLength = 64;
    private byte[] genes = new byte[defaultGeneLength];
    // Cache
    private int fitness = 0;

    // Create a random individual
    public void generateIndividual() {
        for (int i = 0; i < size(); i++) {
            byte gene = (byte) Math.round(Math.random());
            genes[i] = gene;
        }
    }

    /* Getters and setters */
    // Use this if you want to create individuals with different gene lengths
    public static void setDefaultGeneLength(int length) {
        defaultGeneLength = length;
    }

    public byte getGene(int index) {
        return genes[index];
    }

    public void setGene(int index, byte value) {
        genes[index] = value;
        fitness = 0;
    }

    /* Public methods */
    public int size() {
        return genes.length;
    }

    public int getFitness() {
        if (fitness == 0) {
```

```
            fitness = FitnessCalc.getFitness(this);
        }
        return fitness;
    }

    @Override
    public String toString() {
        String geneString = "";
        for (int i = 0; i < size(); i++) {
            geneString += getGene(i);
        }
        return geneString;
    }
}
```

Algorithm.java

```
package simpleGa;

public class Algorithm {

    /* GA parameters */
    private static final double uniformRate = 0.5;
    private static final double mutationRate = 0.015;
    private static final int tournamentSize = 5;
    private static final boolean elitism = true;

    /* Public methods */

    // Evolve a population
    public static Population evolvePopulation(Population pop) {
        Population newPopulation = new Population(pop.size(), false);

        // Keep our best individual
        if (elitism) {
            newPopulation.saveIndividual(0, pop.getFittest());
        }

        // Crossover population
        int elitismOffset;
        if (elitism) {
            elitismOffset = 1;
        } else {
            elitismOffset = 0;
        }
        // Loop over the population size and create new individuals with
        // crossover
        for (int i = elitismOffset; i < pop.size(); i++) {
            Individual indiv1 = tournamentSelection(pop);
```

```
            Individual indiv2 = tournamentSelection(pop);
            Individual newIndiv = crossover(indiv1, indiv2);
            newPopulation.saveIndividual(i, newIndiv);
        }

        // Mutate population
        for (int i = elitismOffset; i < newPopulation.size(); i++) {
            mutate(newPopulation.getIndividual(i));
        }

        return newPopulation;
    }

    // Crossover individuals
    private static Individual crossover(Individual indiv1, Individual indiv2) {
        Individual newSol = new Individual();
        // Loop through genes
        for (int i = 0; i < indiv1.size(); i++) {
            // Crossover
            if (Math.random() <= uniformRate) {
                newSol.setGene(i, indiv1.getGene(i));
            } else {
                newSol.setGene(i, indiv2.getGene(i));
            }
        }
        return newSol;
    }

    // Mutate an individual
    private static void mutate(Individual indiv) {
        // Loop through genes
        for (int i = 0; i < indiv.size(); i++) {
            if (Math.random() <= mutationRate) {
                // Create random gene
                byte gene = (byte) Math.round(Math.random());
                indiv.setGene(i, gene);
            }
        }
    }

    // Select individuals for crossover
    private static Individual tournamentSelection(Population pop) {
        // Create a tournament population
        Population tournament = new Population(tournamentSize, false);
        // For each place in the tournament get a random individual
        for (int i = 0; i < tournamentSize; i++) {
            int randomId = (int) (Math.random() * pop.size());
            tournament.saveIndividual(i, pop.getIndividual(randomId));
        }
        // Get the fittest
        Individual fittest = tournament.getFittest();
```

```
        return fittest;
    }
}
```

FitnessCalc.java

```
package simpleGa;

public class FitnessCalc {

    static byte[] solution = new byte[64];

    /* Public methods */
    // Set a candidate solution as a byte array
    public static void setSolution(byte[] newSolution) {
        solution = newSolution;
    }

    // To make it easier we can use this method to set our candidate solution
    // with string of 0s and 1s
    static void setSolution(String newSolution) {
        solution = new byte[newSolution.length()];
        // Loop through each character of our string and save it in our byte
        // array
        for (int i = 0; i < newSolution.length(); i++) {
            String character = newSolution.substring(i, i + 1);
            if (character.contains("0") || character.contains("1")) {
                solution[i] = Byte.parseByte(character);
            } else {
                solution[i] = 0;
            }
        }
    }

    // Calculate inidividuals fittness by comparing it to our candidate solution
    static int getFitness(Individual individual) {
        int fitness = 0;
        // Loop through our individuals genes and compare them to our cadidates
        for (int i = 0; i < individual.size() && i < solution.length; i++) {
            if (individual.getGene(i) == solution[i]) {
                fitness++;
            }
        }
        return fitness;
    }

    // Get optimum fitness
    static int getMaxFitness() {
        int maxFitness = solution.length;
```

```
        return maxFitness;
    }
}
```

Now let's create our main class.

First we need to set a candidate solution (feel free to change this if you want to).

```
FitnessCalc.setSolution("1111000000000000000000000000000000000000000000000000000001
111");
```

Now we'll create our initial population, a population of 50 should be fine.

```
Population myPop = new Population(50,true);
```

Now we can evolve our population until we reach our optimum fitness

```
int generationCount = 0;
while(myPop.getFittest().getFitness() < FitnessCalc.getMaxFitness()){
  generationCount++;
  System.out.println("Generation: "+generationCount+" Fittest:
"+myPop.getFittest().getFitness());
  myPop = Algorithm.evolvePopulation(myPop);
}
System.out.println("Solution found!");
System.out.println("Generation: "+generationCount);
System.out.println("Genes:");
System.out.println(myPop.getFittest());
```

Here's the complete code for our main class:

GA.java

```
package simpleGa;

public class GA {

    public static void main(String[] args) {

        // Set a candidate solution
        FitnessCalc.setSolution("1111000000000000000000000000000000000000000000000000000000
00000001111");

        // Create an initial population
        Population myPop = new Population(50, true);

        // Evolve our population until we reach an optimum solution
        int generationCount = 0;
        while (myPop.getFittest().getFitness() < FitnessCalc.getMaxFitness()) {
```

```
            generationCount++;
            System.out.println("Generation: " + generationCount + " Fittest: " +
myPop.getFittest().getFitness());
            myPop = Algorithm.evolvePopulation(myPop);
        }
        System.out.println("Solution found!");
        System.out.println("Generation: " + generationCount);
        System.out.println("Genes:");
        System.out.println(myPop.getFittest());

    }
}
```

თუ ყველაფერი სწორადაა გაკეთებული, თქვენ უნდა მიიღოთ ქვემოთ მოცემულის მსგავსი შედეგი:

```
Generation: 1 Fittest: 40
Generation: 2 Fittest: 43
Generation: 3 Fittest: 50
Generation: 4 Fittest: 50
Generation: 5 Fittest: 52
Generation: 6 Fittest: 59
Generation: 7 Fittest: 59
Generation: 8 Fittest: 61
Generation: 9 Fittest: 61
Generation: 10 Fittest: 61
Generation: 11 Fittest: 63
Generation: 12 Fittest: 63
Generation: 13 Fittest: 63
Generation: 14 Fittest: 63
Generation: 15 Fittest: 63
Solution found!
Generation: 15
Genes:
1111000000000000000000000000000000000000000000000000000001111
```

გაითვალისწინეთ, რომ თქვენი ამონახსნი ზუსტად შეიძლება არ დაემთხვეს ზედა ამონახსნს, გენეტიკური ალგორითმის მემკვიდრეობითი მახასიათებლების გამო და თქვენ აქ გაქვთ ძირითადი ბინარული გენეტიკური ალგორითმი. გენეტიკური ალგორითმის დიდი უპირატესობა არის ის, რომ იგი არის მარტივი ნებისმიერი პრობლემის წარმოსადგენად, თუმცა იგი ყოველთვის შეიძლება არ იყოს საუკეთესო გზა მის გადასაწყვეტად.

თუ გსურთ გამოიყენოთ გენეტიკური ალგორითმი რეალური ძებნის ამოცანისათვის, გაეცანით შემდეგ სახელმძღვანელოს: [applying a genetic algorithm to the traveling salesman problem]

ამ საკითხთან დაკავშირებული სტატიები:

Our Artificial World!
Consciousness, coming to a machine near you.
Applying a genetic algorithm to the traveling salesman problem
Introduction to Artificial Neural Networks Part 2 - Learning
Solving the Traveling Salesman Problem Using Google Maps and Genetic Algorithms