Факультет математики и информатики

Департамент Информатики

**Лабораторная работа**

**по дисциплине "Автоматизация проектирования программных продуктов"**

Cтудент

группы MIA2302 :

Ulmanu Cristian

Проверил : Căpățînă Gheorghe

Кишинев 2023

Оглавление

[1. Описание алгоритма 3](#_Toc1326122405)

[2. Описание кода алгоритма 4](#_Toc1553972269)

[3. Тестирование алгоритма 8](#_Toc2063293916)

[Выводы 9](#_Toc214914456)

[Код алгоритма 10](#_Toc2147251385)

# Описание алгоритма

1. Инициализация популяции:

Начнем с создания начальной популяции, представленной набором индивидов (геномов). Значения геномов выбираются случайным образом в пределах заданных границ (в данном случае, от 0 до 64).

2. Оценка приспособленности:

Каждый индивид в популяции оценивается по приспособленности, которая определяется целевой функцией. В данном случае используется функция -1 \* (x - 14) \* (x - 21) \* (x - 30) \* (x - 50).

3. Рулеточное колесо (Roulette Wheel) - выбор особей:

Рассчитываем общую приспособленность популяции. Для каждого индивида рассчитываем вероятность выбора на основе его приспособленности.

Выбираем индивидов с использованием рулеточного колеса, где вероятность выбора пропорциональна приспособленности.

4. Скрещивание (Crossover):

Выбранные индивиды подвергаются операции скрещивания с определенной вероятностью. В данной реализации используется одноточечное кроссоверное скрещивание.

5. Мутация:

Индивиды подвергаются мутации с определенной вероятностью. В данной реализации мутация заключается в изменении случайного гена у индивида.

6. Повторение:

Шаги 3-5 повторяются в течение нескольких поколений (итераций). Каждое поколение представляет собой эволюцию популяции.

7. Оценка лучшего индивида:

После завершения всех поколений выбирается лучший индивид на основе его приспособленности.

8. Вывод результатов:

Выводится информация о лучшем индивиде и его приспособленности.

Генетический алгоритм создает новые поколения путем комбинирования лучших индивидов с использованием операций скрещивания и мутации. Это позволяет эффективно искать оптимальные значения в пространстве поиска для решения задач оптимизации.

# Описание кода алгоритма

1. Инициализация популяции:

На начальном этапе создается популяция из индивидов (особей), которые представляют собой значения в пределах от 0 до 64. Эти значения выбираются случайным образом.

private static double[] initializePopulation() {

double[] population = new double[POPULATION\_SIZE];

Random rand = new Random();

for (int i = 0; i < POPULATION\_SIZE; i++) {

population[i] = RANGE\_MIN + (RANGE\_MAX - RANGE\_MIN) \* rand.nextDouble();

}

return population;

}

2. Определение целевой функции:

В данном случае целевая функция представляет собой -1 \* (x - 14) \* (x - 21) \* (x - 30) \* (x - 50), где x - значение индивида.

private static double targetFunction(double x) {

return -1 \* (x - 14) \* (x - 21) \* (x - 30) \* (x - 50);

}

3. Оценка приспособленности:

Приспособленность каждого индивида определяется значением целевой функции. Чем выше значение функции, тем лучше приспособлен индивид.

private static double fitness(double x) {

return targetFunction(x);

}

4. Рулеточное колесо (Roulette Wheel) - выбор особей:

* Рассчитывается общая приспособленность популяции.
* Для каждого индивида рассчитывается вероятность выбора на основе его приспособленности.
* Выбираются индивиды, пропорционально их вероятностям.

private static double[] rouletteWheelSelection(double[] population) {

double[] selectedIndividuals = new double[POPULATION\_SIZE];

// Рассчитываем общую приспособленность

double totalFitness = Arrays.stream(population).map(FitnessBasedGeneticAlgorithm::fitness).sum();

// Рассчитываем вероятность выбора для каждого индивида на основе приспособленности

double[] selectionProbabilities = new double[POPULATION\_SIZE];

for (int i = 0; i < POPULATION\_SIZE; i++) {

selectionProbabilities[i] = fitness(population[i]) / totalFitness;

}

// Выполняем выбор особей с использованием рулеточного колеса

for (int i = 0; i < POPULATION\_SIZE; i++) {

double rouletteValue = Math.random();

double cumulativeProbability = 0;

for (int j = 0; j < POPULATION\_SIZE; j++) {

cumulativeProbability += selectionProbabilities[j];

if (rouletteValue <= cumulativeProbability) {

selectedIndividuals[i] = population[j];

break;

}

}

}

return selectedIndividuals;

}

5. Скрещивание (Crossover) и мутация (Mutation):

Для создания новых индивидов производится скрещивание двух родителей с определенной вероятностью. Происходит также мутация индивидов с некоторой вероятностью.

private static double crossover(double parent1, double parent2) {

Random rand = new Random();

if (rand.nextDouble() < CROSSOVER\_RATE) {

return (parent1 + parent2) / 2.0;

} else {

return rand.nextDouble() < 0.5 ? parent1 : parent2;

}

}

private static double mutation(double individual) {

Random rand = new Random();

if (rand.nextDouble() < MUTATION\_RATE) {

return RANGE\_MIN + (RANGE\_MAX - RANGE\_MIN) \* rand.nextDouble();

} else {

return individual;

}

}

6. Запуск генетического алгоритма:

* В основном цикле происходит несколько поколений генетического алгоритма.
* Выполняется рулеточное колесо для выбора особей.
* Производится скрещивание и мутация.
* Повторяется до завершения всех поколений.

public static void main(String[] args) {

double[] population = initializePopulation();

for (int generation = 0; generation < MAX\_GENERATIONS; generation++) {

// Рулеточное колесо для выбора особей

double[] selectedIndividuals = rouletteWheelSelection(population);

// Скрещивание и мутация

for (int i = 0; i < POPULATION\_SIZE; i++) {

double parent1 = selectedIndividuals[(int) (Math.random() \* POPULATION\_SIZE)];

double parent2 = selectedIndividuals[(int) (Math.random() \* POPULATION\_SIZE)];

double child = crossover(parent1, parent2);

population[i] = mutation(child);

}

}

// Находим лучшего индивида в конечной популяции

double bestIndividual = population[0];

double bestFitness = fitness(population[0]);

for (int i = 1; i < POPULATION\_SIZE; i++) {

double currentFitness = fitness(population[i]);

if (currentFitness > bestFitness) {

bestFitness = currentFitness;

bestIndividual = population[i];

}

}

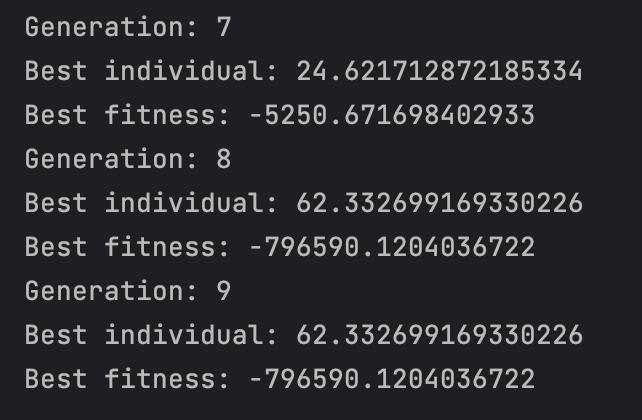
System.out.println("Лучший индивид: " + bestIndividual);

System.out.println("Лучшая приспособленность: " + bestFitness);

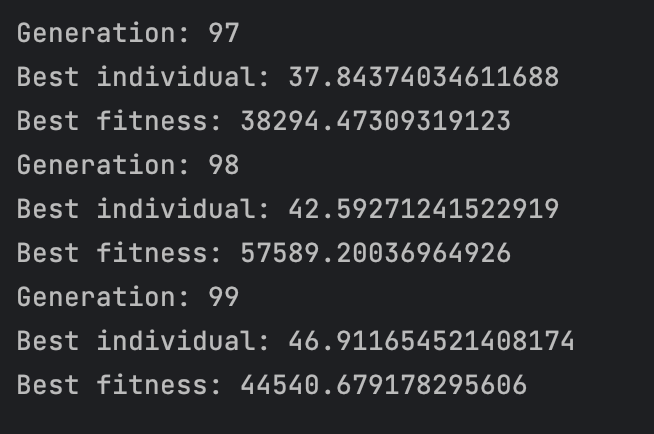
}

# Тестирование алгоритма

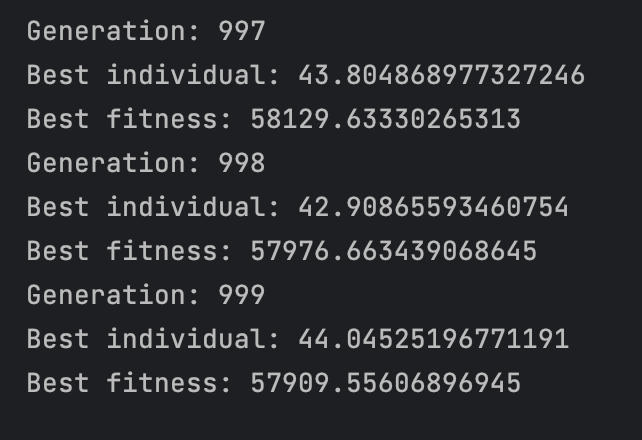
* После 10 поколений и 10 особей



* После 100 поколений и 100 особей



* После 1000 поколений и 1000 особей



# Выводы

Генетический алгоритм — это эвристический оптимизационный метод, вдохновленный процессами естественного отбора и генетики. Он используется для решения задач оптимизации, где требуется найти наилучшее решение в большом пространстве поиска.

Генетические алгоритмы обладают свойствами параллелизма и способностью исследовать большие пространства поиска, что делает их привлекательным инструментом в различных областях, где требуется оптимизация сложных задач.

Из тестов данного алгоритма мы видим что чем больше база для тестирования тем лучше становится результат потому что остаются только самые приспособленные особи. Данный алгоритм показывает как работает эволюция и как мы можем использовать такие же принципы для обработки больших массивов данных и выявлению самых нужных нам данных.

# Код алгоритма

import java.util.Arrays;  
import java.util.Random;  
  
public class RouletteWheelGeneticAlgorithm {  
  
 private static final int POPULATION\_SIZE = 1000;  
 private static final int MAX\_GENERATIONS = 1000;  
 private static final double MUTATION\_RATE = 0.1;  
 private static final double CROSSOVER\_RATE = 0.7;  
 private static final double RANGE\_MIN = 0;  
 private static final double RANGE\_MAX = 64;  
  
 private static double targetFunction(double x) {  
 return -1 \* (x - 14) \* (x - 21) \* (x - 30) \* (x - 50);  
 }  
  
 private static double[] initializePopulation() {  
 double[] population = new double[POPULATION\_SIZE];  
 Random rand = new Random();  
 for (int i = 0; i < POPULATION\_SIZE; i++) {  
 population[i] = RANGE\_MIN + (RANGE\_MAX - RANGE\_MIN) \* rand.nextDouble();  
 }  
 return population;  
 }  
  
 private static double fitness(double x) {  
 return targetFunction(x);  
 }  
  
 private static double[] rouletteWheelSelection(double[] population) {  
 double[] selectedIndividuals = new double[POPULATION\_SIZE];  
  
 double totalFitness = Arrays.stream(population).map(RouletteWheelGeneticAlgorithm::fitness).sum();  
  
 double[] selectionProbabilities = new double[POPULATION\_SIZE];  
 for (int i = 0; i < POPULATION\_SIZE; i++) {  
 selectionProbabilities[i] = fitness(population[i]) / totalFitness;  
 }  
  
 for (int i = 0; i < POPULATION\_SIZE; i++) {  
 double rouletteValue = Math.random();  
 double cumulativeProbability = 0;  
  
 for (int j = 0; j < POPULATION\_SIZE; j++) {  
 cumulativeProbability += selectionProbabilities[j];  
 if (rouletteValue <= cumulativeProbability) {  
 selectedIndividuals[i] = population[j];  
 break;  
 }  
 }  
 }  
  
 return selectedIndividuals;  
 }  
  
 private static double crossover(double parent1, double parent2) {  
 Random rand = new Random();  
 if (rand.nextDouble() < CROSSOVER\_RATE) {  
 return (parent1 + parent2) / 2.0;  
 } else {  
 return rand.nextDouble() < 0.5 ? parent1 : parent2;  
 }  
 }  
  
 private static double mutation(double individual) {  
 Random rand = new Random();  
 if (rand.nextDouble() < MUTATION\_RATE) {  
 return RANGE\_MIN + (RANGE\_MAX - RANGE\_MIN) \* rand.nextDouble();  
 } else {  
 return individual;  
 }  
 }  
  
 public static void main(String[] args) {  
 double[] population = initializePopulation();  
  
 for (int generation = 0; generation < MAX\_GENERATIONS; generation++) {  
 System.out.println("Generation: " + generation);  
 double[] selectedIndividuals = rouletteWheelSelection(population);  
  
 for (int i = 0; i < POPULATION\_SIZE; i++) {  
 double parent1 = selectedIndividuals[(int) (Math.random() \* POPULATION\_SIZE)];  
 double parent2 = selectedIndividuals[(int) (Math.random() \* POPULATION\_SIZE)];  
 double child = crossover(parent1, parent2);  
 population[i] = mutation(child);  
 }  
  
 double bestIndividual = population[0];  
 double bestFitness = fitness(population[0]);  
 for (int i = 1; i < POPULATION\_SIZE; i++) {  
 double currentFitness = fitness(population[i]);  
 if (currentFitness > bestFitness) {  
 bestFitness = currentFitness;  
 bestIndividual = population[i];  
 }  
 }  
 System.out.println("Best individual: " + bestIndividual);  
 System.out.println("Best fitness: " + bestFitness);  
 }  
  
  
 }  
}