

**Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки**

Лабораторна робота №3.3

з дисципліни

«Інтелектуальні вбудовані системи»

на тему

«Дослідження генетичного алгоритму»

Виконала:

студентка групи ІІІ-84

Шахова Поліна Миколаївна

номер залікової книжки: 8424

Перевірив:

ас. Регіда П. Г.

Київ 2020

Основні теоретичні відомості:

Генетичні алгоритми служать, головним чином, для пошуку рішень в багатовимірних просторах пошуку.

Можна виділити наступні етапи генетичного алгоритму:

- (Початок циклу)
- Розмноження (схрещування)
- Мутація
- Обчислити значення цільової функції для всіх особин
- Формування нового покоління (селекція)
- Якщо виконуються умови зупинки, то (кінець циклу), інакше (початок циклу).

Розглянемо приклад реалізації алгоритму для знаходження цілих коренів діофантового рівняння $a+b+2c=15$.

Згенеруємо початкову популяцію випадковим чином, але з дотриманням умови – усі згенеровані значення знаходяться у проміжку від одиниці до $y/2$, тобто на відрізку $[1;8]$ (узагалі, границі випадкового генерування можна вибрати на свій розсуд):

(1,1,5); (2,3,1); (3,4,1); (3,6,4)

Отриманий генотип оцінюється за допомогою функції пристосованості (fitness function). Згенеровані значення підставляються у рівняння, після чого обраховується різниця отриманої правої частини з початковим y . Після цього рахується ймовірність вибору генотипу для ставання батьком – зворотня дельта ділиться на сумму сумарних дельт усіх генотипів.

$1+1+2\cdot5=12$	$\Delta=3$	$\frac{\frac{1}{3}}{\frac{27}{24}} = 0,7$
$2+3+2\cdot1=7$	$\Delta=8$	$\frac{\frac{1}{8}}{\frac{27}{24}} = 0,11$
$3+4+2\cdot1=9$	$\Delta=6$	$\frac{\frac{1}{6}}{\frac{27}{24}} = 0,15$
$3+6+2\cdot4=17$	$\Delta=2$	$\frac{\frac{1}{2}}{\frac{27}{24}} = 0,44$

Наступний етап включає в себе схрещування генотипів по методу кросоверу – у якості дітей виступають генотипи, отримані змішуванням коренів – частина йде від одного з батьків, частина від іншого, наприклад:

$$\left. \begin{array}{l} (3 \mid 6,4) \\ (1 \mid 1,5) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} (3,1,5) \\ (1,6,4) \end{array} \right.$$

Лінія кросоверу може бути поставлена в будь-якому місці, кількість потомків також може вибиратися. Після отримання нових генотипів вони перевіряються функцією пристосованості та створюють власних потомків, тобто виконуються дії, описані вище.

Ітерації алгоритму відбуваються, поки один з генотипів не отримає $\Delta=0$, тобто його значення будуть розв'язками рівняння.

Завдання за варіантом:

Варіант 24

Лістинг програми:

```
package com.example.lab_33

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import java.lang.Math.abs
import java.util.Random
import kotlin.math.absoluteValue

class MainActivity : AppCompatActivity() {

    private fun gen_alg(x1_base: Double,
                       x2_base: Double,
                       x3_base: Double,
                       x4_base: Double,
                       y_base: Double) : MutableList<Double>{

        var counter = 0
        val populationZero: MutableList<MutableList<Double>> =
mutableListOf()
        val population: MutableList<MutableList<Double>> = mutableListOf()
        val listOfFitnesses: MutableList<Double> = mutableListOf()
        val populationOfChild: MutableList<MutableList<Double>> =
mutableListOf()
        var bestPopulation: MutableList<Double> = mutableListOf()

        fun fitness(population: MutableList<Double>): Double {
            val fitness: Double = y_base -
                population[0] * x1_base -
                population[1] * x2_base -
                population[2] * x3_base -
                population[3] * x4_base
            return fitness.absoluteValue
        }

        fun populationZeroFind() {
            for (i in 0..3) {
                populationZero.add(mutableListOf())
                for (j in 0..3) {
                    populationZero[i].add((1..8).random().toDouble())
                }
            }
        }

        fun findFitnessOfPopulation() {
            listOfFitnesses.clear()
            if (population.isEmpty()) {
```

```

        populationZero.mapTo(population) { it }
    }
    for (i in 0..3) {
        listOfFitnesses.add(fitness(population[i]))
    }
}

fun findRoulette() {
    populationOfChild.clear()
    var roulette = 0.00
    val roulettePercent: MutableList<Double> = mutableListOf()
    val circleRoulette: MutableList<Double> = mutableListOf()
    listOfFitnesses.forEach { roulette += 1 / it }
    for (i in 0..3) {
        roulettePercent.add(1 / listOfFitnesses[i] / roulette)
    }

    for (i in 0..3) {
        if (i == 0) {
            circleRoulette.add(roulettePercent[i])
        } else {
            circleRoulette.add(circleRoulette[i - 1] +
roulettePercent[i])
        }
    }

    var i = 0
    populationOfChild.clear()
    while (i < 4) {
        val piu: Double = (1..100).random().toDouble() / 100
        var thisChild = 0
        for (k in 0..3) {
            if (piu >= circleRoulette[k]) {
                thisChild = k
            }
        }
        populationOfChild.add(population[thisChild])
        i++
    }
}

fun crossingOver() {
    counter++
    population.clear()
    for (p in 0..3) {
        val c: MutableList<Double> = mutableListOf()
        c.clear()
        for (j in 0..3) {
            if (p % 2 == 0) {
                if (j < 2) {
                    c.add(populationOfChild[p][j])
                } else c.add(populationOfChild[p + 1][j])
            } else
                if (j < 2) {
                    c.add(populationOfChild[p][j])
                } else c.add(populationOfChild[p - 1][j])
        }
        population.add(c)
    }
}

fun bestFitnessFind(): Boolean {
    findFitnessOfPopulation()
    listOfFitnesses.forEach { if (it == 0.0) return true }
    return false
}

fun life() {

```

```

        var q = 0
        while (!bestFitnessFind() && q < 10) {
            findFitnessOfPopulation()
            findRoulette()
            crossingOver()
            q++
        }
    }

fun result(): MutableList<Double> {
    populationZeroFind()
    life()
    while ((!listOfFitnesses.contains(0.0)) &&
        population[0] == populationOfChild[0] &&
        population[1] == populationOfChild[1] &&
        population[2] == populationOfChild[2] &&
        population[3] == populationOfChild[3]
    ) {
        populationZero.clear()
        population.clear()
        listOfFitnesses.clear()
        populationOfChild.clear()
        populationZeroFind()
        life()
    }
    for (i in 0..3) {
        if (listOfFitnesses[i] == 0.0) {
            bestPopulation = population[i]
        }
    }
    return bestPopulation
}

val answer : MutableList<Double> = result();

if (answer.isEmpty()) {
    return answer;
}

answer.add(counter.toDouble())
return answer
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val result: TextView = findViewById(R.id.show_result)
    val input_a: EditText = findViewById(R.id.input_data_a)
    val input_b: EditText = findViewById(R.id.input_data_b)
    val input_c: EditText = findViewById(R.id.input_data_c)
    val input_d: EditText = findViewById(R.id.input_data_d)
    val input_y: EditText = findViewById(R.id.input_data_y)

    val buttonCalculate: Button = findViewById(R.id.button_calculate)
    buttonCalculate.setOnClickListener {
        if (input_a.text.isEmpty()){
            result.text = "There was no number entered!"
        } else {
            val a = input_a.text.toString().toDouble()
            val b = input_b.text.toString().toDouble()
            val c = input_c.text.toString().toDouble()
            val d = input_d.text.toString().toDouble()
            val y = input_y.text.toString().toDouble()

            val ans = gen_alg(a, b, c, d, y)

            if (!ans.isEmpty()) {
                ans.removeAt(4)
            }
        }
    }
}

```

```
        result.text = "X1 = ${ans[0].toInt()}, X2 =  
${ans[1].toInt()}, X3 = ${ans[2].toInt()}, X4 = ${ans[3].toInt()}"  
    } else {  
        result.text = "Couldn't find in range y/2. Try again"  
    }  
}  
}  
}
```

Приклад роботи програми:


$$X_1 + X_2 + X_3 + X_4 =$$

CALCULATE

Here wil be a result

$$2x_1 + 2x_2 + 2x_3 + 2x_4 = 14$$

CALCULATE

$$X_1 = 1, X_2 = 1, X_3 = 3, X_4 = 2$$


Висновки:

Під час виконання лабораторної роботи я ознайомилась з принципами реалізації генетичного алгоритму, вивчення та дослідження особливостей даного алгоритму з використанням засобів моделювання і сучасних програмних оболонок. Я налаштувала генетичний алгоритм для знаходження цілих коренів діофантового рівняння $ax^1 + bx^2 + cx^3 + dx^4 = y$, розробила відповідну програму та реалізувала користувацький інтерфейс з можливістю вводу даних за допомогою Android.