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 ”

Виконав:
Студент групи ІП-83
Карпюк І.В.
№ ЗК: ІП-8311

Перевірив:
викладач Регіда
П.Г.

Київ 2021

Мета роботи - ознайомлення з принципами реалізації генетичного алгоритму,
вивчення та дослідження особливостей даного алгоритму з використанням засобів
моделювання і сучасних програмних оболонок.

3.1. 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 вибирати 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+2·5=12 Δ=3 132724 = 0,7
2+3+2·1=7 Δ=8 182724 = 0,11
3+4+2·1=9 Δ=6 162724 = 0,15
3+6+2·4=17 Δ=2 122724 = 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:
(3 | 6,4) (3,1,5)
→
(1 | 1,5) (1,6,4)
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 Δ=0, тобто
його значення будуть розв’язками рівняння.

Лістинг програми із заданими умовами завдання

```
package ua.kpi.comsys.factorio

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import androidx.fragment.app.Fragment
import kotlin.math.absoluteValue
```

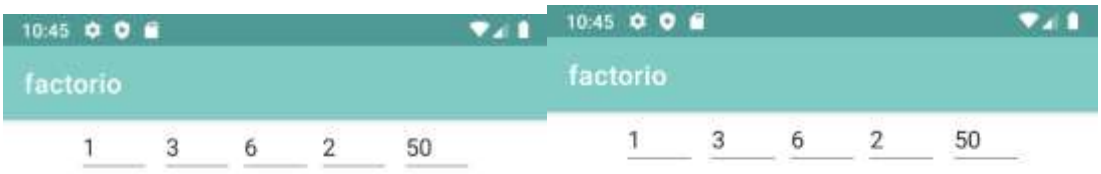
```
class Genetic : Fragment () {
    override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.genetic_layout, container, false)
    }
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        view.let { super.onViewCreated(it, savedInstanceState) }

        view.findViewById<Button>(R.id.calc)?.setOnClickListener { _ ->
            val x1 = view.findViewById<EditText>(R.id.x1)?.text.toString().toDouble()
            val x2 = view.findViewById<EditText>(R.id.x2)?.text.toString().toDouble()
            val x3 = view.findViewById<EditText>(R.id.x3)?.text.toString().toDouble()
            val x4 = view.findViewById<EditText>(R.id.x4)?.text.toString().toDouble()
            val y = view.findViewById<EditText>(R.id.y)?.text.toString().toDouble()
            val message = view.findViewById<TextView>(R.id.message)

            val generation = geneMachine(x1, x2, x3, x4, y).result()

            if (generation == null) {
                message?.text = "No result"
            } else {
                message?.text = generation.toString()
            }
        }

    }
}

class geneMachine
constructor(in_x1: Double, in_x2: Double, in_x3: Double, in_x4: Double, in_y: Double)
{
    val x1 = in_x1
    val x2 = in_x2
    val x3 = in_x3
    val x4 = in_x4
    val y = in_y
    val zero_population: MutableList<MutableList<Double>> = mutableListOf()
    var population: MutableList<MutableList<Double>> = mutableListOf()
    val fitness_list: MutableList<Double> = mutableListOf()
    val child_popul: MutableList<MutableList<Double>> = mutableListOf()
    var best_popul: MutableList<Double> = mutableListOf()

    fun searchFitness(popul_row: MutableList<Double>): Double {
        val fitness: Double = y -
                popul_row[0] * x1 -
                popul_row[1] * x2 -
                popul_row[2] * x3 -
                popul_row[3] * x4
        return fitness.absoluteValue
    }
```

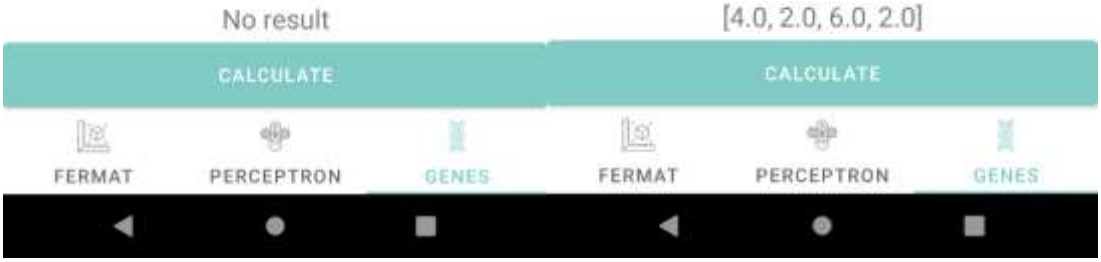
```
    fun zeroPopulationInit() {
        for (i in 0..3) {
            zero_population.add(mutableListOf())
            for (j in 0..3) {
                zero_population[i].add((1..8).random().toDouble())
            }
        }
    }

    fun calculateFitnesOfPopulation() {
        fitness_list.clear()
        if (population.isEmpty()) {
            zero_population.mapTo(population) { it }
        }
        for (i in 0..3) {
            fitness_list.add(searchFitness(this.population[i]))
        }
    }

    fun playRulet() {
        child_popul.clear()
        var rulet = 0.00
        val procent_rulet: MutableList<Double> = mutableListOf()
        val rulet_circle: MutableList<Double> = mutableListOf()
        this.fitness_list.forEach { rulet += 1 / it }
        for (i in 0..3) {
            procent_rulet.add(1 / fitness_list[i] / rulet)
        }

        for (i in 0..3) {
            if (i == 0) {
                rulet_circle.add(procent_rulet[i])
            } else {
                rulet_circle.add(rulet_circle[i - 1] + procent_rulet[i])
            }
        }
        var i = 0
        child_popul.clear()
        while (i < 4) {
            val piu: Double = (1..100).random().toDouble() / 100
            var k = 0
            var this_child = 0
            for (k in 0..3) {
                if (piu >= rulet_circle[k]) {
                    this_child = k
                }
            }
            child_popul.add(population[this_child])
            i++
        }
    }

    fun crossOver() {
        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(child_popul[p][j])
                    } else c.add(child_popul[p + 1][j])
                } else
                    if (j < 2) {
                        c.add(child_popul[p][j])
                    } else c.add(child_popul[p - 1][j])
            }
            population.add(c)
        }
    }

    fun calculateBestFitness(): Boolean {
        this.calculateFitnesOfPopulation()
        fitness_list.forEach { if (it == 0.0) return true }
        return false
    }

    fun lifeCycle(): Boolean {
        var q = 0
        while (!calculateBestFitness()) {
            // the limit of cycles
            if (q < 100) {
                this.calculateFitnesOfPopulation()
                this.playRulet()
                this.crossOver()
                q++
            } else {
                return false
            }
        }
        return true
    }

    fun result(): MutableList<Double>? {
        this.zeroPopulationInit()
        if (this.lifeCycle()) {
            while ((!this.fitness_list.contains(0.0)) &&
                    population[0] == child_popul[0] &&
                    population[1] == child_popul[1] &&
                    population[2] == child_popul[2] &&
                    population[3] == child_popul[3]
            ) {
                zero_population.clear()
                population.clear()
                fitness_list.clear()
                child_popul.clear()
                this.zeroPopulationInit()
                this.lifeCycle()
            }
            for (i in 0..3) {
```

```
                if (fitness_list[i] == 0.0) {
                    best_popul = population[i]
                }
            }
            return best_popul
        }
        return null
    }
}
```

Результати виконання програми





Висновки

Вивчив роботу генетичних алгоритмів.