

**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-84
Гудь В.В.
№ залікової книжки: ІП-8405

Перевірив:
викладач
Регіда П.Г.

Київ 2021

Теоретичні відомості

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.

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

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:

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

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 на лабораторну роботу

Налаштувати генетичний алгоритм для знаходження цілих коренів діофантового рівняння $ax_1+bx_2+cx_3+dx_4=y$. Розробити відповідний мобільний додаток і вивести отримані значення. Провести аналіз витрат часу на розрахунки.

Вихідний код

```
private fun calculateDiophantine(): String {
    var population = generateSemiRandomPopulation()
    var fitness = calculateFitness(population)

    repeat(1000) {
        val index = fitness.indexOfFirst {it == 0}

        if (index != -1) {
            val solution = population[index]
            val deviation = (y - solution[0] * a - solution[1] * b - solution[2] * c - solution[3] * d)
            return "x1 = ${solution[0]}\nx2 = ${solution[1]}\nx3 = ${solution[2]}\nx4 = ${solution[3]}\nDeviation = $deviation"
        }

        population = cross(population, fitness)
        mutate(population)
        fitness = calculateFitness(population)
    }

    val bestLoser = findBestFitness(fitness)
    val closestChromosome = population[bestLoser]
    val deviation = (y - closestChromosome[0] * a - closestChromosome[1] * b - closestChromosome[2] * c - closestChromosome[3] * d)

    return "x1 = ${closestChromosome[0]}\nx2 = ${closestChromosome[1]}\nx3 = ${closestChromosome[2]}\nx4 = ${closestChromosome[3]}\nDeviation = $deviation\n"
```

```

}

private fun generateSemiRandomPopulation(): Array<IntArray> {
    val maxGene = y / 2
    val pool = 1..maxGene
    val population = Array(generations) { IntArray(4) }
    for (chromosome in 0 until generations) {
        population[chromosome] = intArrayOf(pool.random(), pool.random(), pool.random(),
pool.random())

    }
    return population
}

private fun calculateFitness(population: Array<IntArray>) : IntArray {
    val res = IntArray(population.size)
    for (i in population.indices) {
        res[i] = fitness(population[i])
    }
    return res
}

private fun fitness(solution: IntArray) : Int {
    var res = y
    val coefficient = intArrayOf(a, b ,c ,d)
    for (i in coefficient.indices) {
        res -= coefficient[i] * solution[i]
    }
    return abs(res)
}

private fun cross(population: Array<IntArray>, fitness: IntArray) : Array<IntArray>{
    val probabilities = calculateProbabilities(fitness)
    val parentsPool = decideParents(population, probabilities)
    val couples = createCouples(parentsPool)
    return nextGen(couples)
}

private fun calculateProbabilities(fitness: IntArray): FloatArray {
    var sum = 0f
    val probabilities = FloatArray(fitness.size)
    for (i in fitness.indices) {
        val probability = 1f / fitness[i]
        sum += probability
        probabilities[i] = probability
    }
    val res = FloatArray(fitness.size)

```

```

    for (i in fitness.indices){
        res[i] = probabilities[i] / sum
    }
    return res
}

```

```

private fun createCouples(parentsPool: Array<IntArray>): Array<Pair<IntArray, IntArray>>
{
    val res = Array(parentsPool.size / 2) { Pair(IntArray(4), IntArray(4)) }
    for (i in res.indices) {
        val random1 = (parentsPool.indices).random()
        val random2 = (parentsPool.indices).random()
        val couple = Pair(parentsPool[random1], parentsPool[random2])
        res[i] = couple
    }
    return res
}

```

```

private fun decideParents(population: Array<IntArray>, probabilities: FloatArray) :
Array<IntArray> {
    val parents = Array(population.size) { IntArray(4) }
    val values = calculateValues(probabilities)
    for (i in population.indices) {
        val factor = Math.random()
        val index = calculateIndex(values, factor)
        val parent = population[index]
        parents[i] = parent
    }
    return parents
}

```

```

private fun calculateValues(probabilities: FloatArray) : FloatArray {
    var sum = 0f
    val res = FloatArray(probabilities.size)
    for (i in probabilities.indices) {
        res[i] = sum
        sum += probabilities[i]
    }

    return res
}

```

```

private fun calculateIndex(values : FloatArray, factor : Double) : Int {
    var index = values.size / 2

    var valuesStart = 0
    var valuesEnd = values.size - 1
    var start = values[index]

```

```

var end = values[index + 1]
while(true) {
    if (factor > end) {
        valuesStart = index + 1
    } else if (factor < start) {
        valuesEnd = index - 1
    } else {
        break
    }
    index = (valuesStart + (valuesEnd - valuesStart) / 2)
    start = values[index]
    end = if (index != values.lastIndex) {
        values[index + 1]
    } else {
        1f
    }
}
return index
}

private fun nextGen(couples: Array<Pair<IntArray, IntArray>>) : Array<IntArray>{
    val nextGen = Array(couples.size * 2) { IntArray(4) }

    for (i in couples.indices) {
        val couple = couples[i]
        val indices = (0..3).toList().toIntArray()
        val randomChange = (1..3).random()
        val randomizedGens = IntArray(randomChange)
        repeat(randomChange) {
            val index = (0 until (indices.size - it)).random()
            randomizedGens[it] = indices[index]
            indices[index] = indices[indices.lastIndex - it]
        }
        val first = couple.first.clone()
        val second = couple.second.clone()
        for (g in randomizedGens) {
            first[g] = couple.second[g]
            second[g] = couple.first[g]
        }

        nextGen[i * 2] = first
        nextGen[i * 2 + 1] = second
    }
    return nextGen
}

private fun mutate(population: Array<IntArray>) {
    for (i in population.indices) {
        if(Math.random() < 0.01) {

```

```
        val chromosome = population[i]
        chromosome[chromosome.indices.random()] += intArrayOf(-2, -1, 1, 2).random()
    }
}
}
private fun findBestFitness(populationFitness: IntArray): Int {
    var index = 0
    var value = populationFitness[0]
    for (i in 1 until populationFitness.size) {
        if (populationFitness[i] < value) {
            index = i
            value = populationFitness[i]
        }
    }
    return index
}
```

Результати роботи програми



10 23 5 7 456
x1 = 10
x2 = 13
x3 = 10
x4 = 1
Deviation = 0

SOLVE

51 74 32 15 452
x1 = -3
x2 = -3
x3 = 22
x4 = 8
Deviation = 3

SOLVE



Висновки

Під час даної лабораторної роботи ми ознайомились з принципами реалізації генетичного алгоритму, вивчили як за його допомогою можна знаходити наближені розв'язки діофантових рівнянь.