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

з дисципліни
«Інтелектуальні вбудовані системи»

на тему
«РЕАЛІЗАЦІЯ ЗАДАЧІ РОЗКЛАДАННЯ ЧИСЛА НА ПРОСТІ МНОЖНИКИ (ФАКТОРИЗАЦІЯ ЧИСЛА)
ДОСЛІДЖЕННЯ НЕЙРОННИХ МЕРЕЖ. МОДЕЛЬ PERCEPTRON
ДОСЛІДЖЕННЯ ГЕНЕТИЧНОГО АЛГОРИТМУ»

Виконав:	Перевірив:
студент групи ІП-84	викладач
Голубов Іван	Регіда Павло Геннадійович
номер залікової книжки: 8404	

Київ 2021

Основні теоретичні відомості

Факторизації лежить в основі стійкості деяких криптоалгоритмів, еліптичних кривих, алгебраїчній теорії чисел та кванових обчислень, саме тому дана задача дуже гостро досліджується, й шукаються шляхи її оптимізації.

На вхід задачі подається число n Є ℕ, яке необхідно факторизувати. Перед виконанням алгоритму слід переконатись в тому, що число не просте. Далі алгоритм шукає перший простий дільник, після чого можна запустити алгоритм заново, для повторної факторизації.

В залежності від складності алгоритми факторизації можна розбити на дві групи:

- Експоненціальні алгоритми (складність залежить експоненційно від довжини вхідного параметру);
- Субекспоненціальні алгоритми.

Існування алгоритму з поліноміальною складністю – одна з найважливіших проблем в сучасній теорії чисел. Проте, факторизація з даною складністю можлива на квантовому комп’ютері за допомогою алгоритма Шора.

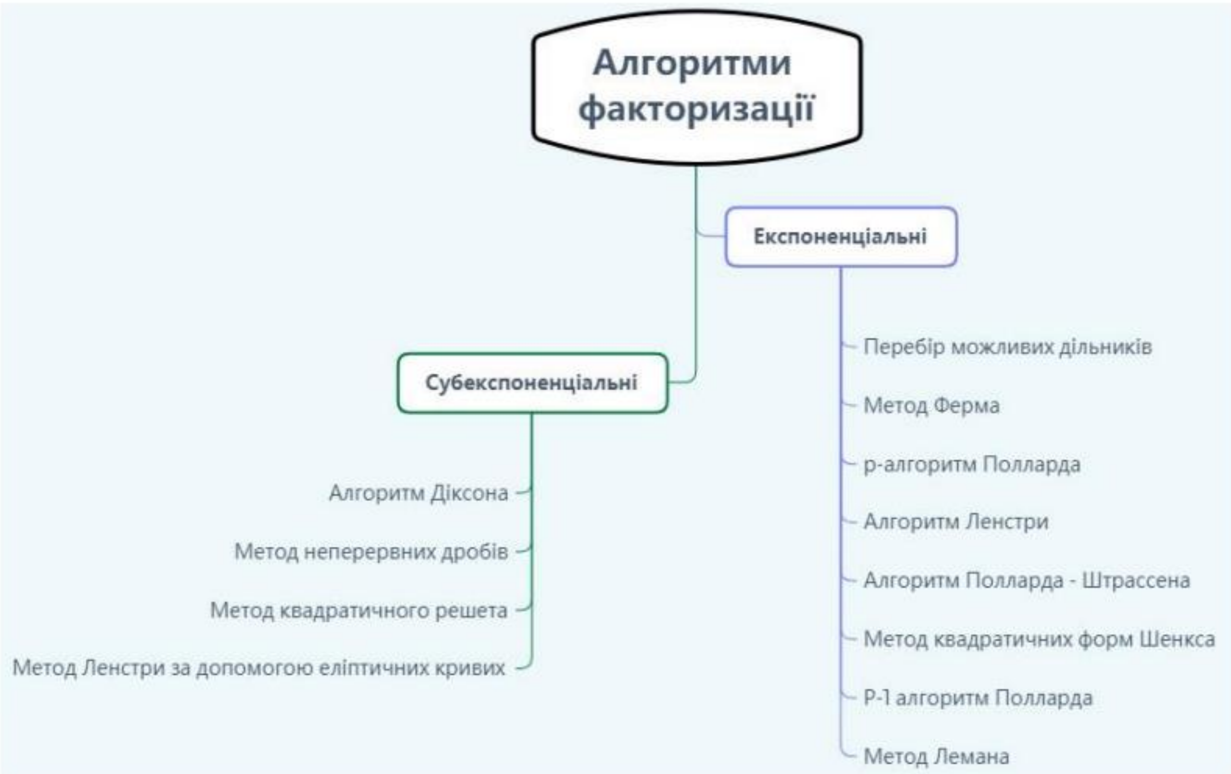

Рис1. Алгоритми факторизації

Розглянемо принципи роботи найпростіших алгоритмів факторизації.

Метод перебору можливих дільників.

Один з найпростіших і найочевидніших алгоритмів заключається в тому, щоб послідовно ділити задане число n на натуральні числа від 1 до $\lfloor\sqrt{n}\rfloor$. Формально, достатньо ділити лише на прості числа в цьому інтервалі, але для цього необхідно знати їх множину. На практиці складається таблиця простих чисел і на вхід подаються невеликі числа (до 2^{16}), оскільки даний алгоритм має низьку швидкість роботи.

Приклад алгоритму:

1. Початкова установка: t = 0, k = 0, n = N (t,k,n такі, що $n = N / p_1 \dots p_n$ і n не мають простих множників, менших за d_k).
2. Якщо n = 1, закінчуємо алгоритм.
3. Присвоюємо $q = [n / d_k]$, $r = n \bmod d_k$.
4. Якщо $r \neq 0$, переходимо на крок 6.
5. Присвоюємо t++, $p_t = d_k$, n = q і повертаємось на крок 2.
6. Якщо $q > d_k \rightarrow$ k++ і повертаємось на крок 3.
7. Присвоїти t++, p_t = n і закінчити виконання алгоритму.

Модофікований метод факторизації Ферма.

Ідея алгоритму заключається в пошуку таких чисел A і B, щоб факторизоване число n мало вигляд: $n = A^2 – B^2$. Даний метод гарний тим, що реалізується без використання операцій ділення, а лише з операціями додавання й віднімання.

Приклад алгоритму:

1. Початкова установка: $x = 2[\sqrt{n}] + 1$, y = 1, $r = [\sqrt{n}]^2 – n$.
2. Якщо r = 0, то алгоритм закінчено: $n = \frac{x-y}{2} * \frac{x+y-2}{2}$
3. Присвоюємо r = r + x, x = x + 2.
4. Присвоюємо r = r - y, y = y + 2.
5. Якщо r > 0, повертаємось до кроку 4, інакше повертаємось до кроку 2.

Метод факторизації Ферма.

Ідея алгоритму заключається в пошуку таких чисел A і B, щоб факторизоване число n мало вигляд: $n = A^2 – B^2$. Даний метод гарний тим, що реалізується без використання операцій ділення, а лише з операціями додавання й віднімання.

Приклад алгоритму:

Початкова установка: $x = [\sqrt{n}]$ – найменше число, при якому різниця x^2-n невід’ємна. Для кожного значення k Є ℕ, починаючи з k = 1, обчислюємо $(\lceil\sqrt{n}\rceil + k)^2 - n$ і перевіряємо чи не є це число точним квадратом.

- Якщо не є, то k++ і переходимо на наступну ітерацію.
- Якщо є точним квадратом, тобто $x^2 - n = \left(\lceil\sqrt{n}\rceil + k\right)^2 - n = y^2$, то ми отримуємо розкладання: $n = x^2 - y^2 = (x+y)(x-y) = A * B$, в яких
$$x = (\lceil\sqrt{n}\rceil + k)$$

Якщо воно є тривіальним і єдиним, то n - просте

Важливою задачею, яку система реального часу має вирішувати є отримання необхідних для обчислень параметрів, її обробка та виведення результату у встановлений дедлайн. З цього постає проблема отримання водночас точних та швидких результатів. Модель Перцпептрон дозволяє покроково наближати початкові значення.

Розглянемо приклад: дано дві точки A(1,5), B(2,4), поріг спрацювання P = 4, швидкість навчання δ = 0.1. Початкові значення ваги візьмемо нульовими W1 = 0, W2 = 0. Розрахунок вихідного сигналу у виконується за наступною формулою:

$$x1 * W1 + x2 * W2 = y$$

Для кожного кроку потрібно застосувати дельта-правило, формула для розрахунку похибки:

$$\Delta = P - y$$

де y – значення на виході.

Для розрахунку ваги, використовується наступна формули:

$$W1(i+1) = W1(i) + W2 * x11$$
$$W2(i+1) = W1(i) + W2 * x12$$

де i – крок, або ітерація алгоритму.

Розпочнемо обробку:

1 ітерація:

Використовуємо формулу обрахунку вихідного сигналу:

0 = 0 * 1 + 0 * 5 значення не підходить, оскільки воно менше зазначеного порогу. Вихідний сигнал повинен бути строго більша за поріг.

Далі, рахуємо Δ:

Δ = 4 – 0 = 4

За допомогою швидкості навчання δ та минулих значень ваги, розрахуємо нові значення ваги:

W1 = 0 + 4 * 1 * 0,1 = 0,4

W2 = 0 + 4 * 5 * 0.1 = 2

Таким чином ми отримали нові значення ваги. Можна побачити, що результат змінюється при зміні порогу.

2 ітерація:

Виконуємо ті самі операції, але з новими значеннями ваги та для іншої точки.

8,8 = 0,4 * 2 + 2 * 4 , не підходить, значення повинно бути менше порогу.

Δ = -5 , спрощуємо результат для прикладу.

W1 = 0,4 + 5 * 2 * 0,1 = -0,6

W2 = 2 - 5 * 4 * 0.1 = 0

3 ітерація:

Дано тільки дві точки, тому повертаємось до першої точки та нові значення ваги розраховуємо для неї.

-0,6 = -0,6 * 1 + 0 * 5 , не підходить, значення повинно бути більше порогу.

Δ = 5 , спрощуємо результат для прикладу.

W1 = -0,6 + 5 * 1 * 0,1 = -0,1

W2 = 0 + 5 * 5 * 0.1 = 2,5

По такому самому принципу рахуємо значення ваги для наступних ітерацій, поки не отримаємо значення, які задовольняють вхідним даним.

На восьмій ітерації отримуємо значення ваги W1 = -1,8 та W2 = 1,5.

5,7 = -1,8 * 1 + 1,5 * 5, більше за поріг, задовольняє

2,4 = -1,8 * 2 + 1,5 * 4, менше за поріг, задовольняє

Отже, бачимо, що для заданого прикладу, отримано значення ваги за 8 ітерацій. При розрахунку значень, потрібно враховувати дедлайн. Дедлайн може бути в вигляді максимальної кількості ітерацій або часовий.

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.

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

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:

$$\left.\begin{matrix}(3\mid 6{,}4)\\ \\(1\mid 1{,}5)\end{matrix}\right]\rightarrow\left[\begin{matrix}(3{,}1{,}5)\\ \\(1{,}6{,}4)\end{matrix}\right.$$

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 Δ=0, тобто його значення будуть розв'язками рівняння.

Завдання на лабораторну роботу

Розробити програма для факторизації заданого числа методом Ферма. Реалізувати користувацький інтерфейс з можливістю вводу даних.

Поріг спрацювання: P = 4

Дано точки: A(0,6), B(1,5), C(3,3), D(2,4).

Швидкості навчання: δ = {0,001; 0,01; 0,05; 0.1; 0.2; 0,3}

Дедлайн: часовий = {0.5с; 1с; 2с; 5с}, кількість ітерацій = {100;200;500;1000}

Обрати швидкість навчання та дедлайн. Налаштувати Перцептрон для даних

точок. Розробити відповідний мобільний додаток і вивести отримані значення.

Провести аналіз витрати часу та точності результату за різних параметрах навчання.

Налаштувати генетичний алгоритм для знаходження цілих коренів діофантового рівняння ax1+bx2+cx3+dx4=y. Розробити відповідний мобільний додаток вивести отримані значення. Провести аналіз витрат часу на розрахунки.

Лістинг програми

```
package com.example.lab3

import kotlin.math.sqrt

class lab3_1 {
    public fun gen_alg(n: Int): MutableList<Int> {
        val x: Int = sqrt(n.toDouble()).toInt() + 1
        var k: Int = 0
        var i: Int = 0
        var y: Int = 0
        while (k == 0) {
            y = Math.pow(((x + i).toDouble()), 2.00).toInt() - n
            if ((sqrt(y.toDouble())) % 1.0 == 0.0) {
                k = i
                break
            } else i++
        }
        val a: Int = x + k
        val b: Int = sqrt(y.toDouble()).toInt()
        return mutableListOf(a, b)
```

```
    }
}

package com.example.lab3

class lab_3_2
constructor(in_speed: Double, in_time: Double, in_iter: Int) {
    var speed = in_speed
    var time = in_time
    var iter = in_iter
    private var W1 = 0.00
    private var W2 = 0.00
    private var P = 4.00
    var points = arrayListOf(Pair(0.00, 6.00), Pair(1.00, 5.00), Pair(3.00, 3.00), Pair(2.00, 4.00))
    private fun validate(): Boolean {
        val y1 = W1 * points[0].first + W2 * points[0].second
        val y2 = W1 * points[1].first + W2 * points[1].second
        val y3 = W1 * points[2].first + W2 * points[2].second
        val y4 = W1 * points[3].first + W2 * points[3].second
        if ((y1 > P) && (y2 > P) && (y3 < P) && (y4 < P)) {
            return true
        }
        return false
    }

    fun find_answer(): Pair<Double, Double> {
        val time_in = System.currentTimeMillis()
        for (i in 0..iter) {
            if ((System.currentTimeMillis() - time_in) <= time * 1000) {
                for (k in 0 until points.size) {
                    val y = W1 * points[k].first + W2 * points[k].second
                    val delta = P - y
                    W1 += delta * points[k].first * speed
                    W2 += delta * points[k].second * speed
                    if (validate()) {
                        return Pair(W1, W2)
                    }
                }
            }
        }
        return Pair(W1, W2)
    }
}
package com.example.lab3

import kotlin.math.absoluteValue

class lab3_3
constructor(in_x1: Double = 1.00, in_x2: Double = 1.00, in_x3: Double = 2.00, in_x4: Double = 0.00, in_y: Double
= 15.00) {
    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 find_fitness(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 create_zero_population() {
    for (i in 0..3) {
        zero_population.add(mutableListOf())
        for (j in 0..3) {
            zero_population[i].add((1..8).random().toDouble())
        }
    }
}

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

fun play_rulet() {
    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 crossing_over() {
    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 find_best_fitness(): Boolean {
    this.find_fitness_of_popul()
    fitness_list.forEach { if (it == 0.0) return true }
    return false
}

fun life() {
    var q = 0
    while (!find_best_fitness() && q < 10) {
        this.find_fitness_of_popul()
        this.play_rulet()
        this.crossing_over()
        q++
    }
}

fun find_answer(): MutableList<Double> {
    this.create_zero_population()
    this.life()
    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.create_zero_population()
            this.life()
        }
        for (i in 0..3) {
            if (fitness_list[i] == 0.0) {
                best_popul = population[i]
            }
        }
        return best_popul
    }
}
package com.example.lab3

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.EditText
import android.widget.Spinner
import android.widget.TextView
import kotlin.math.sqrt


class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
    fun lab1(view: View){
        val editText = findViewById<EditText>(R.id.EnterNLab31)
        val aResult = findViewById<TextView>(R.id.aResultLab31)
        val bResult = findViewById<TextView>(R.id.bResult)
        val number = editText.text.toString().toInt()
        val errorMessage = findViewById<TextView>(R.id.errorMessage)

        var err =""
        if (number % 2 == 0) {
            errorMessage.apply {
                text = "Enter an odd number"
            }
        }
        else if  (number <= 0) {
            errorMessage.apply {
                text = "Enter a positive number"
            }
        }
        else {
            val result = lab3_1().gen_alg(number)
            aResult.apply { text= result[0].toString() }
            bResult.apply { text= result[1].toString()}
            errorMessage.apply {
```

```
            text = "no errors"
        }
    }
}
fun lab2(view: View){
    val spinnerSpeed = findViewById<Spinner>(R.id.speed)
    val spinnerTime = findViewById<Spinner>(R.id.time)
    val spinnerIter = findViewById<Spinner>(R.id.iter)
    val w1Result =  findViewById<TextView>(R.id.w1Result)
    val w2Result =  findViewById<TextView>(R.id.w2Result)
    val selectedSpeed = spinnerSpeed.selectedItem.toString().toDouble()
    val selectedTime = spinnerTime.selectedItem.toString().toDouble()
    val selectedIter = spinnerIter.selectedItem.toString().toInt()
    val result = lab_3_2(selectedSpeed,selectedTime,selectedIter).find_answer()
    val errorMessage = findViewById<TextView>(R.id.errorMessage)
    val illegalValues = listOf(
        Double.NaN,
        Double.NEGATIVE_INFINITY,
        Double.POSITIVE_INFINITY
    )
    if (result.first in illegalValues || result.second in illegalValues) {
        errorMessage.apply {
            text = "Solution couldn`t be found"
            }
    }else{
        w1Result.apply { text= result.first.toString() }
        w2Result.apply { text= result.second.toString() }
        errorMessage.apply {
            text = "no errors"
        }
    }

}
fun lab3(view: View){
    val editX1 = findViewById<EditText>(R.id.editx1).text.toString().toDouble()
    val editX2 = findViewById<EditText>(R.id.editx2).text.toString().toDouble()
    val editX3 = findViewById<EditText>(R.id.editx3).text.toString().toDouble()
    val editX4 = findViewById<EditText>(R.id.editx4).text.toString().toDouble()
    val editY = findViewById<EditText>(R.id.edity).text.toString().toDouble()
    val errorMessage = findViewById<TextView>(R.id.errorMessage)
    val resultA =  findViewById<TextView>(R.id.aResultLab33)
    val resultB =  findViewById<TextView>(R.id.bResultLab33)
    val resultC =  findViewById<TextView>(R.id.cResultLab33)
    val resultD =  findViewById<TextView>(R.id.dResultLab33)
    val result = lab3_3(editX1,editX2,editX3,editX4,editY).find_answer()
    if (result.isEmpty()) {
        errorMessage.apply {
            text = "No possible answer in range [1;y/2]"
        }
    }else{
        resultA.apply {text= result[0].toString()}
        resultB.apply {text= result[1].toString()}
        resultC.apply {text= result[2].toString()}
        resultD.apply {text= result[3].toString()}
        errorMessage.apply {
```

```
                text = "no errors"
            }
        }
    }
}
fun lab3_1(n : Int = 1) : MutableList<Int> {
    var n : Int =899
    if(n % 2 == 0) error("Введите нечётное число")
    if (n <= 0) error("Введите положительное число")
    var x : Int = sqrt(n.toDouble()).toInt()+1
    var k : Int = 0
    var i : Int =0
    var y : Int = 0
    while (k==0){
        y= Math.pow(((x + i).toDouble()), 2.00).toInt()-n
        if((sqrt(y.toDouble()))% 1.0 == 0.0){
            k=i
            break
        }else i++
    }
    var a : Int =x+k
    var b : Int = sqrt(y.toDouble()).toInt()
    println(a)
    println(b)
    println((a+b)*(a-b))
    return mutableListOf(a,b)
}
```

Результати роботи програми

https://github.com/vano7577/RTS/blob/main/lab3/lab3.gif

Висновки

Під час даної лабораторної роботи я ознайомився з принципами розкладання числа на прості множники з використанням різних алгоритмів факторизації, з принципами машинного навчання за допомогою математичної моделі сприйняття інформації Перцептрон(Perceptron), з принципами реалізації генетичного алгоритму. Змоделювати роботу нейронної мережі та дослідити вплив параметрів на час виконання та точність результату, вивчив та дослідив особливості даних алгоритмів з використанням засобів моделювання і сучасних програмних оболонок.