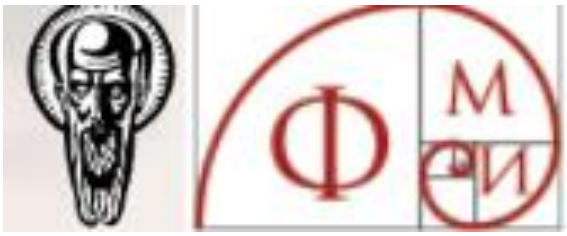

**СОФИЙСКИ УНИВЕРСИТЕТ "СВ. КЛИМЕНТ ОХРИДСКИ"**
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА

# Домашна работа 1

по Системи, основани на знания

НА ТЕМА: Knapsack problem

Изготвил:
Кенан Юсеин
фак. № 71947
спец. Информационни системи , 3 курс

София,
Ноември 2021

# Contents

Описание ..... 3

Описание чрез псевдокод ..... 4

Клас Cargo ..... 4

Клас Genome ..... 4

- Fitness – оценка ..... 5
- Кръстосване в една точка (Single point crossover) алгоритъм ..... 5
- Мутация – mutate() ..... 6

Клас Ship ..... 6

- getCurrentBest() – намира най-добрият genom от текушите комбинации ( най-добрата комбинация от товари ) ..... 6
- Селекция – selection(cullingLimit: Int) ..... 6
- nextGeneration(cullingLimit: Int) ..... 7
- Stop(bestFitness: Int) ..... 7

Main.kt ..... 7

- readDataInputFromTxtFile(filename: String) ..... 8
- getBestOptimalAnswer(cargoList: ArrayList<Cargo>, weightLimit: Int) ..... 8

Инструкции за компилация ..... 9

Примерни резултати ..... 9

# Описание

Задачата е пример за удовлетворяване на ограничения (Constraint Satisfaction Problem, CSP), който се решава чрез прилагане на генетичен алгоритъм.

Имаме кораб, който има максимална тежест на товара, който може да поеме и товари, които претендират да бъдат качени на борда – p1, p2, …, pn. За решение имаме масив от 1 или 0, който индикира кой от товарите ще бъде натоварен. Пример: при p1, p2, p3 товари, [1,1,1] означава, че всички товари ще бъдат натоварени.

Всеки от товарите p1 … pn има съответна тежест (weight) t1, t2, … tn и съответна приоритетна стойност s1, s2, … , sn (value), който индикира колко важен е самият товар p.

Задачата е да се напише генетичен алгоритъм, който дава резултат – комбинации от товари, които да бъдат натоварени така че корабът да е максимално пълен (максимална тежест без да се надвишава капацитета на кораба) и също така да се подберат и да се вземат тези товари, които са с най-висока приоритетна стойност.

За целта е написана програма, която реализира този алгоритъм. Първо се създават различни комбинации от масиви с 0/1 индикатори (индикиращи дали съответния товар ще бъде взет), които са родители (parent).

**( Повече за самите алгоритми – мутация, селекция, кръстосване в 1 точка – по-долу в „Описание чрез псевдокод“ ) :**

След това се прилага мутация на всичките комбинации, който в случая е с 10% шанс да бъде изпълнен за всеки от комбинациите. След това се прави селекция, който филтрира половината от тези комбинации, които са с по-висока фитнес оценка (фитнес оценката е сумата от приоритетните стойности на товарите, които в комбинацията са индикирани с 1; друго условие е тежестта на тези товари да не надвишава максималната допустима на кораба и ако я надвишава фитнес оценката е 0). Така се приближаваме повече към правилния отговор. Празните места на съответните комбинации се запълват чрез кръстосване в 1 точка като

родителите са именно тези с по-висока фитнес оценка, което означава, че и децата получени ще бъдат с по-висока оценка от минималната фитнес оценка от тези на родителите.

Алгоритъмът продължава до намиране на комбинация, на която фитнес оценката е равна на максималната приоритетна стойност ( това се намира като се сумират всички приоритетни стойности на всички от p1,…,pn). Ако не се достигне до такъв край, алгоритъмът прави n! / 2 (факториела на бройката на товарите p1…pn  / 2) и след приключването на циклите се връща тази комбинация, която има най-висока фитнес оценка.

Програмата чете списъка от товари чрез текстов файл ( повече в „Инструкции за компилиране“) и връща най-добрата комбинация от товари.

# Описание чрез псевдокод

Проектът е разработен на Kotlin и Gradle. Използван е ООП подход за реализирането на задачата. Програмата чете лист от товарите от текстов файл, прочита максимален капацитет на кораба в килограми и реализира генетичен алгоритъм за намиране на най-оптималната комбинация от товари, които да бъдат натоварени в кораба като идеята е той да се напълни максимално много и да се вземат най-високо приоритетни товари.

Под пакета Model можем да намерим  програмните представители на реални обекти, необходими за осъществяване на алгоритъма.

## Клас Cargo

Класът Cargo представлява товар, който може да бъде качен на кораба. Всеки товар има тежест (weight) и приоритетна стойност (value).

```kotlin
package model

data class Cargo(
    val value: Int,
    val weight: Int
)
```

( Data класовете в котлин ни предоставят автоматично гетъри, сетъри, конструктори и други методи, които се използват имплицитно. )

## Клас Genome

Класът Genome е обект, който индикира кои от товарите ще бъдат натоварени в кораба. За целта имаме масив от цели числа, които са 0 / 1. За даден масив от товари(cargo[]), един геном индикира за всеки един от товарите с 0 – ако съответния товар няма да бъда натоварен и с 1 – ако ще бъде натоварен.

Всеки Genom първоначално се конструира чрез случайно създаден такъв масив от 0 / 1.

```kotlin
class Genome {
    private val random = Random()

    private var dna: ArrayList<Int> = ArrayList()

    constructor(length: Int) {
        (1..length).forEach { _ ->
            dna.add(if (random.nextBoolean()) 1 else 0)
        }
    }

    constructor(dnaP1: List<Int>, dnaP2: List<Int>) {
        dna.addAll(dnaP1)
        dna.addAll(dnaP2)
    }
```

- Fitness – оценка.

Всеки Genom има оценка (Fitness), който се получава спрямо масив от товари и максималната допустима тежест на кораба.

```
fun getFitness(cargos: ArrayList<Cargo>, weightLimit: Int): Int {
    require( value: cargos.size == dna.size) { " ERROR: Cargo size and dna size must have equal size" }

    var value = 0
    var weight = 0
    cargos.forEachIndexed { index, cargo →
        if (dna[index] == 1) {
            weight += cargo.weight
            value += cargo.value
        }
        if (weight > weightLimit) return 0;
    }
    return value
}
```

Фунцкията за намиране на fitness-а на един геном преминава през него и за товарите, които ще бъдат натоварени ( с индикатор 1 ) им взима сумата от приоритетните стойности (value).

В същото време сумира и тежестите на всеки един от товарите и ако общата им тежест надвишава лимита на кораба то fitness-а (genom оценката) е 0, т.е. комбинацията не е валидна

- Кръстосване в една точка (Single point crossover) алгоритъм.

Функцията singlePointCrossover(other: Genome) взима друг геном и прилага алгоритъма за кръстосване в една точка върху двата генома ( двата масива от 0 / 1 ).

При кръстосването двама родители винаги връщат 2 деца. По случаен принцип се избира точка (index) на който ще се cross-нат двата масива – тоест до кой индекс ще се взимат стойности (0 / 1) от първия масив и след този индекс се взимат стойности от 2-рия масив. За второто дете се прави обратното – до този индекс взимаме първо от 2рия родител и след индекса от 1-вия родител.

За да е по-генетичен алгоритъма, функцията добавя 10% вероятност в която просто се връщат двата родителя като деца, без да бъдат променени.

```
fun singlePointCrossover(other: Genome): Array<Genome> =
    when {
        random.nextInt( bound: 100) > 90 → arrayOf(this, other)
        else → {
            val crossOverIdx = random.nextInt(dna.size)
            arrayOf(
                Genome(dna.subList(0, crossOverIdx), other.dna.subList(crossOverIdx, other.dna.size)),
                Genome(other.dna.subList(0, crossOverIdx), dna.subList(crossOverIdx, dna.size))
            )
        }
    }
```

- Мутация – mutate()

Функцията реализира алгоритъма за мутация на Genom. Мутацията е необходима част от всеки един генетичен алгоритъм. Идеята му е да обиколи всички стойности на масива, определящ кои товари ще бъдат натоварени и на случаен принцип да смени техните стойности. Тоест при обхождане на масива на случаен принцип се избира точно кои от стойностите да бъдат сменени. Мутацията в моя проект има 10% шанс да се осъществи. В останалите 10 процента функцията връща Genom-а без да го мутира.

```
fun mutate() {
    if (random.nextInt( bound: 100) > 90) {
        dna = ArrayList(dna.map {  it: Int
            when {
                random.nextBoolean() → if (it == 0) 1 else 0  ^map
                else → it  ^map
            }
        })
    }
}
```

7

## Клас Ship

Класът Ship представлява един обект – кораб, който има максимална тежест, масив от товари, които претендират да бъдат натоварени и комбинации от genom-и, които представляват комбинации от товари, които да бъдат натоварени. Останалите функции реализират идеята да се намери най-добрата такава комбинация, според условията зададени горе, която да бъде натоварена в кораба.

```
class Ship(numberOfCargos: Int, cargos: ArrayList<Cargo>, wLimit: Int) {
    private var genomes: ArrayList<Genome> = ArrayList()
    private var cargos: ArrayList<Cargo> = ArrayList()
    private var weightLimit: Int = 0

    private val r = Random()

    init {
        this.weightLimit = wLimit
        this.cargos = cargos
        (1..numberOfCargos).forEach { _ → genomes.add(Genome(cargos.size)) }
    }
```

- getCurrentBest() – намира най-добрият genom от текушите комбинации ( най-добрата комбинация от товари )

```
fun getCurrentBest() = genomes.maxByOrNull { it.getFitness(this.cargos, weightLimit) }!!
```

Функцията обхожда всички genom-и, за всеки товар намира фитнеса и сумира фитнесите на тези товари, които ще бъдат натоварени според генома. Накрая връща генома който има най-висока фитнес оценка.

- Селекция – selection(cullingLimit: Int)

```
private fun selection(cullingLimit: Int) {
    require( value: cullingLimit > 0 && cullingLimit ⩽ genomes.size) { " Wrong culling limit value provided! " }
    this.genomes = ArrayList(this.genomes.sortedByDescending { it.getFitness(cargos, weightLimit) }
        .subList(0, cullingLimit))
}
```

Селекцията взима първите най-добри геноми. Алгоритъма сортира всички геноми по фитнес оценки и взима първите (половината) най-добри геноми (комбинации за товарене)

- nextGeneration(cullingLimit: Int)
- Намиране на следваща генерация (нова генерация на геноми – получени от чрез селекция, кръстосване в една точка, мутация )

```
fun nextGeneration(cullingLimit: Int) {
    require( value: cullingLimit ≤ genomes.size) { " ERROR: Culling limit must be smaller than to the number of individuals " }
    var cullingLimit1 = cullingLimit

    val nextGenomes = ArrayList<Genome>()
    selection(cullingLimit)
    genomes.forEach { it.mutate() }

    while (cullingLimit1 * 2 > 0) {
        var p1Idx = 0
        var p2Idx = 0
        while (p1Idx == p2Idx) {
            val numIndividuals = genomes.size
            p1Idx = r.nextInt(numIndividuals)
            p2Idx = r.nextInt(numIndividuals)
        }
        nextGenomes.addAll(genomes[p1Idx].singlePointCrossover(genomes[p2Idx]))
        --cullingLimit1
    }
    this.genomes = nextGenomes
}
```

Първо алгоритъмът прави мутация на всичко геноми (5% шанс да се осъществи, както е споменато по-горе).

След това се прави селекция, която отрязва половината геноми, които са с по-ниски фитнес оценки. Така се елиминират комбинации, които не са кандидати за отговор.

След това 2-рата  половина, която е била отрязана чрез селекцията, се запълва чрез наследници на получените от селекция геноми – тоест деца на родителите, които са с най-висок фитнес. Така отрязваме слабите геноми и комбинациите от геноми, които имаме след операцията са с по-висока оценка. За получаване на тези деца се прави кръстосване в една точка.

- Stop(bestFitness: Int)
  **Индикира дали да се спре алгоритъма – намерен ли е най-добрият резултат?**

```
fun stop(bestFitness: Int): Boolean {
    if (genomes.find { it.getFitness(cargos, weightLimit) == bestFitness } ≠ null) return true
    genomes.forEach {  it: Genome
        if (it.getFitness(cargos, weightLimit) ≠ genomes.first().getFitness(cargos, weightLimit)) return false
    }
    return true
}
```

Ако в списъка от геноми има такъв, която фитнес оценка съвпада с максималната фитнес оценка ( която се намира чрез сумиране на всички приоритетни стойности на всички товари) означава, че сме намери най-добрият резултат, което индикира край на алгоритъма.

## Main.kt

Main функцията, където се реализира логиката на програмата.

- Програмата изисква от потребителя да въведе weightLimit – капацитет на кораба по тежест.

```
fun main(args: Array<String>) {
    print("Please input ship's weight capacity (weight limit) : ")
    val weightLimit = readLine()?.trim()?.toIntOrNull()
    require( value: weightLimit ≠ null) { "Wrong input" }
```

- readDataInputFromTxtFile(filename: String)

```
val cargoList = readDataInputFromTxtFile( filename: "test_data_weights_values.txt")
```

Чете от текстовия файл – „test_data_weights_values.txt“ 2 масива, отговарящи на тежести и приоритетни стойности на товарите (cargo) .Правят се всички необходими валидации за правилния формат на файла и се създава масив от товари, който се връща на main функцията.

```
private fun readDataInputFromTxtFile(filename: String): List<Cargo> {
    val dataFile = File(filename)
    val lines = dataFile.readLines()
    require( value: lines.isNotEmpty() && lines.size ≥ 2)
    { "Wrong data input file format. First line - values, second line - weights." }

    val values = lines[0].split( ...delimiters ",").map { it.trim().toIntOrNull() }
    val weights = lines[1].split( ...delimiters ",").map { it.trim().toIntOrNull() }

    val totalCargo = values.size

    require( value: values.size == weights.size) { " The number of values and weights should be the same! " }
    return (0 until totalCargo).map { it: Int
        require( value: values[it] ≠ null && weights[it] ≠ null) { "Not all weights entered are integers! Fix the data.txt file." }
        Cargo(values[it]!!, weights[it]!!)  ^map
    }
}
```

- getBestOptimalAnswer(cargoList: ArrayList<Cargo>, weightLimit: Int)

```
println("Calculating . . .")
val result = getBestOptimalAnswer(ArrayList(cargoList), weightLimit)
println("Answer: $result")
```

Main фунцкията подава листа от товари и капацитета на кораба и се прилага генетичния алгоритъм за намиране на решение.

BestValue / Best Fitness – най-добрата фитнес оценка се калкулира като се съберат всички value-та (приоритетни стойности) на всички товари от масива.

Epochs – това е колко пъти ще се направи генетичния алгоритъм в опит да се намери най-оптималното решение. В случая съм избрат да е факториел на бройката на товарите / 2.

```
private fun getBestOptimalAnswer(cargoList: ArrayList<Cargo>, weightLimit: Int): Genome {
    val numberOfIndividuals = cargoList.size * 10
    val epochs = getFactorial(cargoList.size) / 2

    val bestValue = cargoList.sumOf { it.value }

    val p = Ship(numberOfIndividuals, cargoList, weightLimit)
    (0..epochs).forEach { _ →
        p.nextGeneration( cullingLimit numberOfIndividuals / 2)
        if (p.stop(bestValue)) return p.getCurrentBest()
    }
    return p.getCurrentBest()
}
```

Ако в рамките на epchos не бъде намерен най-добрия резултат, който е геном, който има фитнес оценка равен на bestFitness, програмата връща резултата, който до момента е с най-висока фитнес оценка.

# Инструкции за компилация

В папката „executable program for testing“ има папка lib, където се намира jar-файла, който е build-нат и готов за пускане чрез Gradle (той съдържа псевдо кода, необходим за пускане на програмата).

За да се пусне програмата е достатъчно да имате инсталирана Java.

Отваряте скрипта “runProgram.bat”, който пуска програмата и отваря конзола, в която въвеждате капацитет на кораб. След което програмата процедира както е описано по-горе в главта за Main.kt.

Текстовият файл “test_data_weights_values.txt” е файлът, от който програмата чете масив от товари. Първият ред на файла представлява масив от приоритетни стойности на товарите (value), а вторият ред съответно тежестите на всеки от товарите.

Примерно съдържание на данните от файла:

| | | | |
|---|---|---|---|
| приоритетни стойности (s) | 82 | 47 | … |
| тегла (t) | 13 | 31 | … |

# Примерни резултати

1. Вход:
   Максимална вместимост: 3235

| Приоритетни стойности (s) | 500, 150, 60, 40, 30 |
|---|---|
| Тегла (t) | 2200, 160, 350, 333, 192 |

В случая се получават товари както следва:

Cargo( value: 500, weight: 2200),
Cargo( value:150, weight: 160),
Cargo (value: 60, weight: 350),
Cargo( value: 40, weight: 333),
Cargo( value: 30, weight: 192)

Best Fitness = 780

Изход: [1,1,1,1,1],
защото в този пример капацитета на кораба позволява всички товари да бъдат натоварени.

2. Вход
   Максимална вместимост: 3648

| Приоритетни стойности (s) | 500, 150, 60 , 40, 30, 5, 10 , 15, 500, 100 |
|---|---|
| Тегла (t) | 2200, 160, 350, 333, 192, 25, 38, 80, 200, 70 |

В случая се получават товари както следва:

Cargo( value: 500, weight: 2200),
Cargo( value:150, weight: 160),
Cargo (value: 60, weight: 350),
Cargo( value: 40, weight: 333),
Cargo( value: 30, weight: 192)
Cargo( value: 5, weight: 25 ),
Cargo( value: 10, weight: 38 ),
Cargo( value 15, weight: 80),
Cargo( value: 500, weight: 200),
Cargo( value: 100, weight: 70),
Изход: [1,1,1,1,1,1,1,1,1].

3. Вход
   Максимална вместимост: 600

| Приоритетни стойности (s) | 200, 40, 3000, 125, 50, 300, 150, 35, 800 |
|---|---|
| Тегла (t) | 50, 150, 10, 25, 200, 60, 30, 50 , 35 |

Cargo( value: 200, weight: 50),

Cargo( value: 40, weight: 150),

Cargo( value: 3000, weight: 10 ),

Cargo( value: 125, weight: 25),

Cargo( value: 50, weight: 200),

Cargo( value: 300, weight: 60),

Cargo( value: 150, weight: 30 ),

Cargo( value: 35, weight: 50),

Cargo( value: 800, weight: 35),

Изход: [1,1,1,1,1,1,1,0,1]

Нямаме място за всички товари. В този случай не се взима товарът с най-нисък приоритет и тежест достатъчно голяма за да се вместят останалите товари в капацитета. В този случай това е Cargo( value: 35, weight: 50)