Софийски университет „Св. Климент Охридски“
Факултет по математика и информатика

ДОМАШНА РАБОТА №1
по
Системи основани на знания
зимен семестър 2021/2022

Knapsack problem(implemented in Ships)

Изготвил: Никола Петров Кирилов,
специалност ИС, ф.н 71986

Ноември 2021
София

1.Описание на използвания метод за решаване на задачата

Основната идея в задачата се базира на така наречената “**KnapSack problem**”. Задачата се базира на това да имплементираме решение на този проблем, но в случая с кораби. За целта имаме променливата **weightLimit**, която ни представлява максималното тегло, което може да се натовари на даден кораб. Съответно има товари(контейнери), които имат **value** и **weight** параметри. **Value** е приоритетната стойност, а **weight** е теглото на товара. С тези стойности е постигната целта да се вземат максимално ценни неща без да надвишава максималната допустима маса. Имаме и един списък(масив) от бинарните стойности 0 и 1**(индикатор)**, описващи дали даден товар ще бъде натоварен, като с 0 ще означава, че няма да бъде натоварен, а с 1 ще означава, че ще бъде натоварен.
В задачата използвам генетичен алгоритъм, който се състои от 3 алгоритъма: Първият алгоритъм е “**Single point crossover”** още известен като рекомбинация, базиран на селекцията, който по случайно избрана точка от ДНК на единия родител, като след тази точка двете ДНК на родителите се разменят. Следващият алгоритъм е “**Fitness”,** който пресмята колко близо е дадено решение до оптималното решение, като сумира стойностите и тежестите, като използва списъка(ДНК), и ако общата тежест на товарите които сме взели надвишава максималната допустима маса приоритета става 0 и нашата товарителница не може да бъде направена, за всяка Последният алгоритъм е “**Mutate”**, при който нашият списък бива мутиран с различни стойности на произволен принцип.

2. Описание на реализацията с псевдокод

За реализирането на решението аз съм използвал езикът Java. Имплементирал съм следните няколко основни класа:
Genome.java - този клас е основният, който държи нашият масив(индикатор) , който представям като **ArrayList**. Има конструктор с конкретна дължина, който инициализира списъка и му слага брой колкото конкретната дължина елементи посредством една променлива **Random**, която определя нашите “гени” на случаен принцип за максимална правдоподобност на задачата. В даденият клас се намират и основните методи, които ще представя с псевдокод:
(с “//” и син цвят по псевдокода ще бележа коментарите, с които обяснявам)
(ще използвам думите “родител” за нашите масиви, които репрезентират товаренето на кораби)

2.1 Single point crossover algorithm:

//Псевдо име на функцията
func singlePointCrossover(Genome other)

```
 {
//Правим валидация дали не е нулева стойност
   if (other == null) {
      error;
   }
//Създаваме си нов масив, в който да съхраняваме резултатните
   Genome[] resultCrossover = new Genome[] { this, other };

//Създаваме си една променлива от тип Integer, която ще ни генерира
//случаен индекс
   int crossOverIndex = RANDOM.nextInt(this.zeroOneList.size());

 //Правим Crossover на двамата "родители"
//В частта до индекса се  вземат сегашните данни и след индекса данните
//на втория "родител"
   resultCrossover[0] = new Genome(this.zeroOneList.subList(0,
crossOverIndex), other.zeroOneList.subList(crossOverIndex,
other.zeroOneList.size()));
   resultCrossover[1] = new Genome(other.zeroOneList.subList(0,
crossOverIndex), this.zeroOneList.subList(crossOverIndex,
this.zeroOneList.size()));


//Връщаме новите кръстосани в една точка масиви
   return resultCrossover;
}
```

2.2 Fitness algorithm:

```
//Псевдо име на функията
func haveFitness(containers, weightLimit, zeroOneList) {
//Правим проверка дали дължината на контейнерите е равна на дължината
//на индикатора, ако не са с еднаква дължина изкарваме грешка
   if (containers.size() != zeroOneList.size()) {
      error;
   }

   int fitness = 0;
   int currentWeight = 0;
//Правим един for цикъл до големината на нашите контейнери
   for (i i < containers_size; i++) {
```

```
//Правим проверка дали съответния контейнер трябва да бъде натоварен, ако е
//1 трябва да го сложим, да сметнем неговия товар и да пресмятаме колко е
//ценен и колко е неговото тегло
      if (zeroOneList.get(i) == 1) {
         weight += containers[i].getWeight();
         value += containers[i]getValue();
//Отново правим проверка дали нашето текущо тегло не е превишило
//максималното допустимо, в случая, когато е превишено ние не го
//добавяме, защото не отговаря на нашето условие
         if (weight > weightLimit) {
            return 0;
         }
      }
   }

//Тук връщаме фитнеса
   return fitness;
}
```

2.3 Mutation algorithm:

```
//Псевдо име на функцията Mutate
func mutate() {
//Създаваме обект от клас Random, който генерира //произволни стойности
      Random r;

//Ръчно слагаме процентна възможност за алгоритъма с //цел леко
//ограничаване на задачата, защото //компилаторите имат непостоянни
//произволни стойности
      if(r > 91) {
         return;
      }
//For цикъл, с който на произволен принцип правим мутацията на нашият
//масив
      for (int i = 0; i < zeroOneList.size(); i++) {
         if(r.boolean) {
            indicator[i] = indicator[i] == 0 ? 1 : 0;
         }
      }
   }
```

Population.java - Метод **„nextGeneration**“ е метод, с който създавам нови генерации, чрез използването на методите „**singlePointCrossove**r“ и **„mutate**“ на **Genom** класа, който описах по-горе.

Container.java - този клас ми представлява въпросните предмети, които трябва да бъдат натоварени на кораба. В него имам една променлива от тип String, която ми репрезентира името на артикула, и две променливи от тип Integer, които са съответно стойността и теглото на предметите. Имам тук е и една от функциите, които съм имплементирал спрямо условието, а именно данните да се четат от файл. Функцията ще опиша с псевдокод:

```
readFile(name_of_the_file or filePath, containers){
//

      Scanner scanner = null;
//Правим непосредствена проверка дали можем да отворим файла
      try
      {
         scanner = new Scanner(name_of_the_file);
      } catch (FileNotFoundException e)
      {
        error;
      }

//От файла четем ред по ред, като разделяме файла със специален символ
//“#”, който отделя елементите във файла.
 while(scanner.hasNextLine())
      {
         String[] stringBuffer = scanner.nextLine().split("#");
         String nameBuffer = stringBuffer[0];
         int valueBuffer = Integer.parseInt(stringBuffer[1]);
         int weightBuffer = Integer.parseInt(stringBuffer[2]);

//Създаваме нов контейнер с артикула, който добавяме от файла
         Container newContainer = new Container(nameBuffer, valueBuffer,
weightBuffer);
//Добавяме в масива от контейнери въпросният предмет
         containers.add(newContainer);
      }
//Връщаме масивът от контейнери
```

```
        return containers;
}
```

Main.java - в него се съдържа основната логика, показана чрез тестове на методите, които решават нашата задача.

3. Инструкции за компилиране на програмата

За да се компилира кодът е нужно да потребителя да има инсталирана Java virtual machine, или по-висока. Въпросната програма може да бъде стартирана от команден прозорец:

1.Влезте в терминала през src папката на проекта
2. Въвеждате: javac Main.java Genome.java Population.java Container.java
3. .cd ../../out/production/Homeword_KBS_1/shipContainer
4. java Main.class

Също така кодът може да се компилира на всяко IDE, което поддържа Java(Например IntelliJ, Visual Studio Code и тн.), като това е по-лесният и удобен начин.

Програмата работи с текстови файлове, като текстовите файлове трябва да са написани във формат:
Ред №1: **name#value#weight**, като програмата чете ред по ред, и може да бъдат въведени **N на брой** реда без ограничение. **name** е името на продукта, **value** е неговата стойност, **weight** е неговото тегло.
Пример за текстовия файл:

```
Silk#500#2200
Cars#650#5000
Gold#1200#1000
Microchips#300#900
Petrol#1250#3400
Gas#400#1230
Laptops#230#1230
Cobalt#350#4000
Medicine#2000#1000
```

4. Примерни резултати

Пример 1:

Вход:

Нека имаме следните контейнери с товари:

Silk#500#2200
Cars#650#5000
Gold#1200#1000
Microchips#300#900
Petrol#1250#3400
Gas#400#1230
Laptops#230#1230
Cobalt#350#4000
Medicine#2000#1000
weightLimit 5000

Отговор, показан на конзолата:
Generation:
[[1, 1, 1, 0, 1, 0, 1, 0, 1], [1, 0, 0, 1, 1, 0, 0, 1, 1], [1, 1, 1, 1, 1, 0, 1, 0, 1], [1, 0, 0, 1, 1, 1, 0, 1, 1], [1, 0, 1, 0, 0, 1, 0, 0, 1], [1, 1, 1, 0, 1, 0, 0, 1, 1], [1, 0, 1, 0, 0, 1, 0, 0, 1], [0, 0, 0, 1, 0, 1, 0, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1, 1], [1, 1, 1, 1, 1, 1, 0, 0, 1]]
[0, 0, 0, 0, 0, 0, 0, 2700, 0, 0]

Answer: [0, 0, 0, 1, 0, 1, 0, 0, 1]
[[Silk, Cars, Gold, Microchips, Petrol, Gas, Laptops, Cobalt, Medicine]]

Обяснение: Показва ни, че трябва да вземем Microchips, Medicine, Gas, като тяхното тегло е 3130, което не надвишава weightLimit.

Пример 2:

Вход:
Нека имаме следните товари:
Coffe#500#220
iPhones#450#560
Cobalt#400#3500
Platinium#700#3333
Weapons#150#1200
Gas#400#1230
Laptops#230#1230
Sugar#100#1000
weightLimit = 8000

Изход:
Generation:

[[1, 1, 1, 0, 1, 1, 1, 0], [1, 0, 1, 0, 0, 1, 1, 1], [1, 1, 1, 0, 0, 1, 1, 1], [1, 1, 0, 1, 1, 0, 1, 0], [1, 1, 0, 1, 0, 1, 1, 1], [1, 1, 1, 0, 1, 0, 1, 0], [1, 1, 1, 0, 1, 1, 1, 0], [1, 0, 1, 0, 0, 1, 0, 1], [1, 1, 1, 0, 0, 1, 1, 1], [1, 1, 1, 0, 0, 1, 1, 0], [1, 1, 1, 0, 1, 1, 0, 1], [1, 1, 1, 0, 0, 1, 1, 0], [1, 1, 1, 0, 1, 1, 0, 1], [1, 1, 1, 0, 0, 1, 1, 0]]
[2130, 1630, 2080, 2030, 2380, 1730, 2130, 1400, 2080, 1980, 2000, 1980, 2000, 1980]

Answer: [1, 1, 0, 1, 0, 1, 1, 1]
[[Coffee, iPhones, Cobalt, Platinium, Weapons, Gas, Laptops, Sugar]]

Обяснение: Алгоритъмът ни казва, че трябва да качим Coffee, iPhones, Platinium, Gas, Laptops, Sugar, като тяхното тегло е 7573, следователно алгоритъма не е нарушил weightLimit.

***При въвеждане на максимално тегло, което ръчно сме пресметнали с цел проверка, алгоритъма изкарва само 1ци, което показва, че той функционира правилно.**