|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САП** | | Тема | оцінка | підпис |
| СПКм-12 | РГР | ФРЕЙМВОРК  “JGAP” |  |  |
| Переговський. В.Р. | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р.З. | |

**Мета роботи:**

Ознайомитися з особливостями та прикладами використання заданого фреймворка.

**ІНДИВІДУАЛЬНЕ ЗАВДАННЯ**

**Варіант – 8**

1. Описати заданий фреймворк.
2. Вказати його особливості.
3. Описати процес встановлення.
4. Показати приклад з використанням заданого фреймворка.

Фреймворк: JGAP.

**Огляд фреймворка JGAP**

JGAP (вимовляється як "джей-ґап") є компонентом генетичних алгоритмів та генетичного програмування, що подається як фреймворк Java. Він забезпечує основні генетичні механізми, які можуть бути легко використані для застосування еволюційних принципів при вирішення завдань. Генетичний алгоритм є потужним засобом для вирішення проблем з великим простором рішень, що має обмежену кількість часу і потужність процесора. JGAP був розроблений, щоб бути простим у використанні "з коробки", а також має високий ступінь модульності, так що користувачі можуть легко встановлювати специфічні генетичні оператори, такі як мутації або кросинговери та інших під компоненти.

**Процес встановлення фреймворкa JGAP**

Фреймворк JGAP написано на Java, це значить, що він не вимагає будь-якого додаткового програмного забезпечення. Вихідний код поставляється в пакеті zip, який можна завантажити з jgap.sourceforge.net при цьому є вибір завантаження повної збірки чи окремих компонентів(jar, src, 3rdpartylib).

Нижче буде представлено спосіб інсталяції JGAP використовуючи Eclipse IDE.

Послідовність кроків при встановленні буде наступна:

1. Завантажити архів з сайту jgap.sourceforge.net. Повний архів матиме назву формату jgap\_xxx\_full.zip.
2. Видобути файли з архіву в папку з довільною назвою, наприклад jgap-current. Вміст папки буде як на рис.1

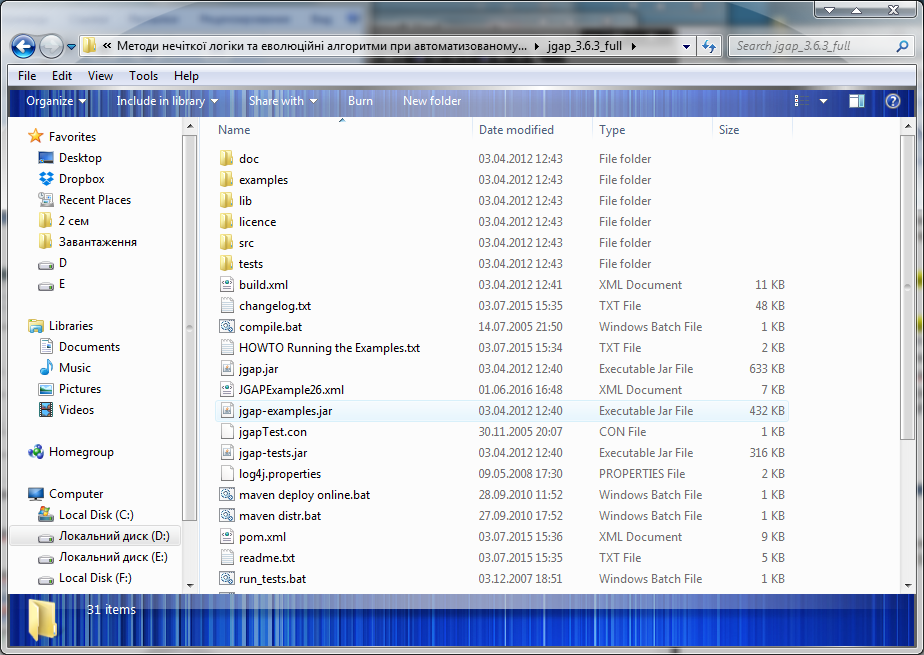


Рис.1 Вміст архіву JGAP

1. Підключення JGAP в Eclipse IDE.

* Створити новий Java проект *File* → *Create* → *New Java Project*
* Натиснувши правою кнопкою миші на проекті в контекстному меню вибрати Properties, щоб перейти в налаштування проекту
* В налаштуваннях проекту вибрати Java Build Path
* Вибрати Add External JARs (рис.2) та вказати розташування бібліотеки, що дадається, тобто файлу jgap.jar
* За потреби можна також додати бібліотеки jgap-tests.jar та jgap-examples.jar

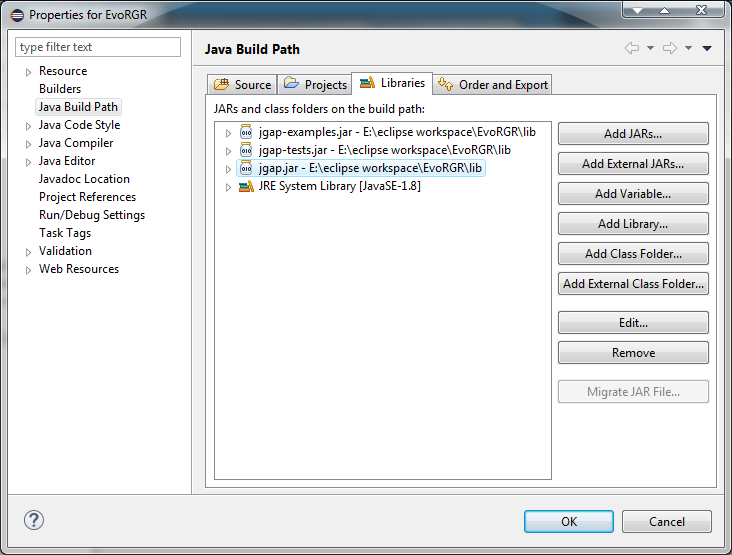


Рис.2 Додавання бібліотек до проекту

Для демонстрації роботи мною вибране тестове завдання *examples.knapsack* . Для того щоб налаштувати програму, відкрийте файл *KnapsackMain* вибравши його в *Package Explorer* бібліотеки *jgap-examples* (рис.3).

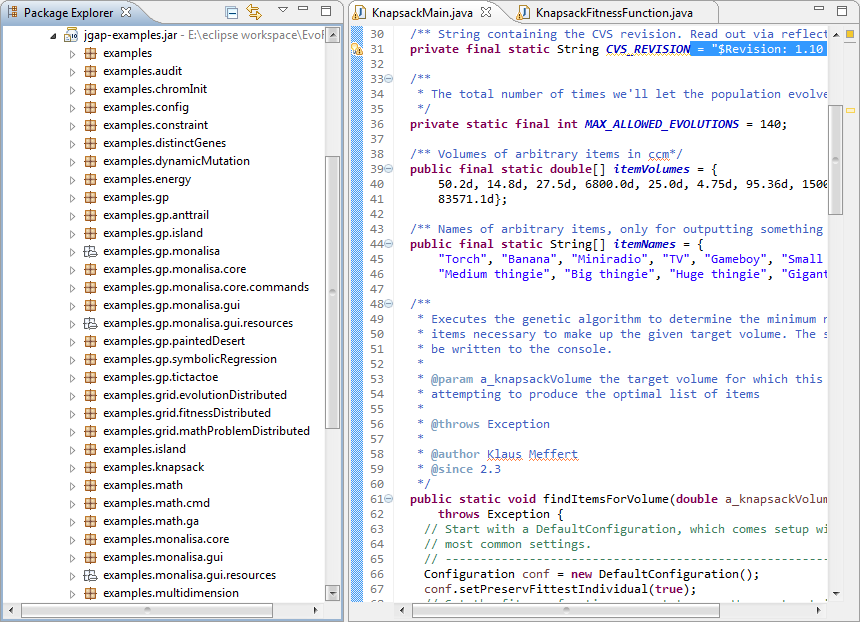


Рис.3 Приклади з бібліотеки jgap-examples.

Для запуску прикладу, клацніть правою кнопкою миші на KnapsackMain.class в дереві проекту. Виберіть *Run As* → *Run Configuration*. У вкладці *Arguments* задайте довільне число(на рис.4 я задав 25). В результаті ви отримаєте вивід на консоль результату виконання програми рис. 5.

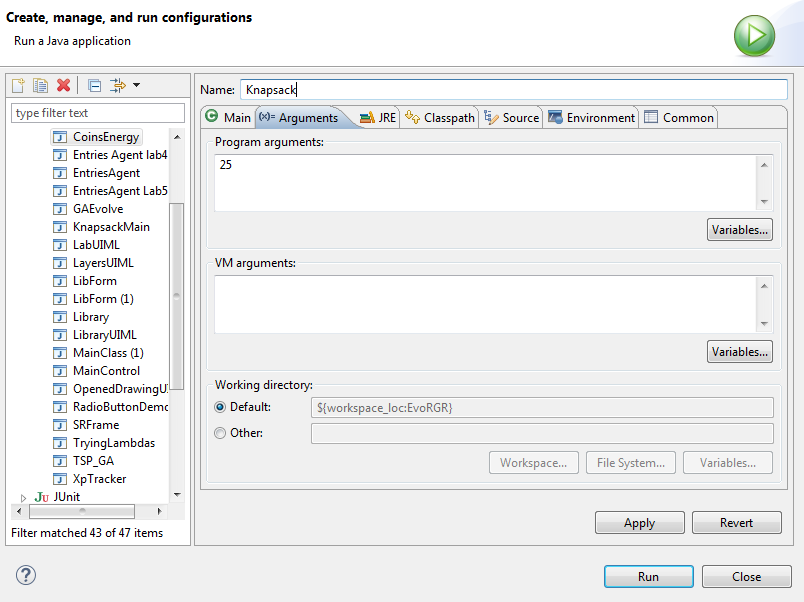


Рис.4 Задання параметру запуску прикладу

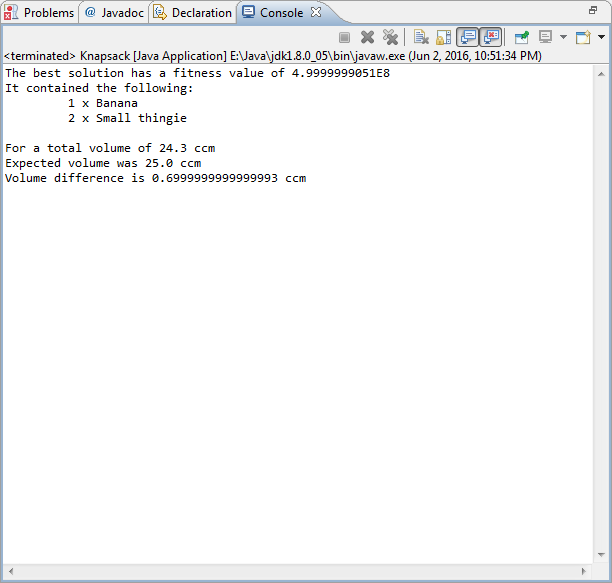


Рис.5 Результат запуску прикладу Knapsack в консолі

**Генетичне програмування з JGAP**

Про програми генетичного програмування (ГП)

Програма ГП представлена у вигляді дерева з одним кореневим вузлом . Кореневий вузол має дочірні вузли кожен з яких в свою чергу може мати власні дочірні вузли. На зображені нижче виражено цю ситуацію.

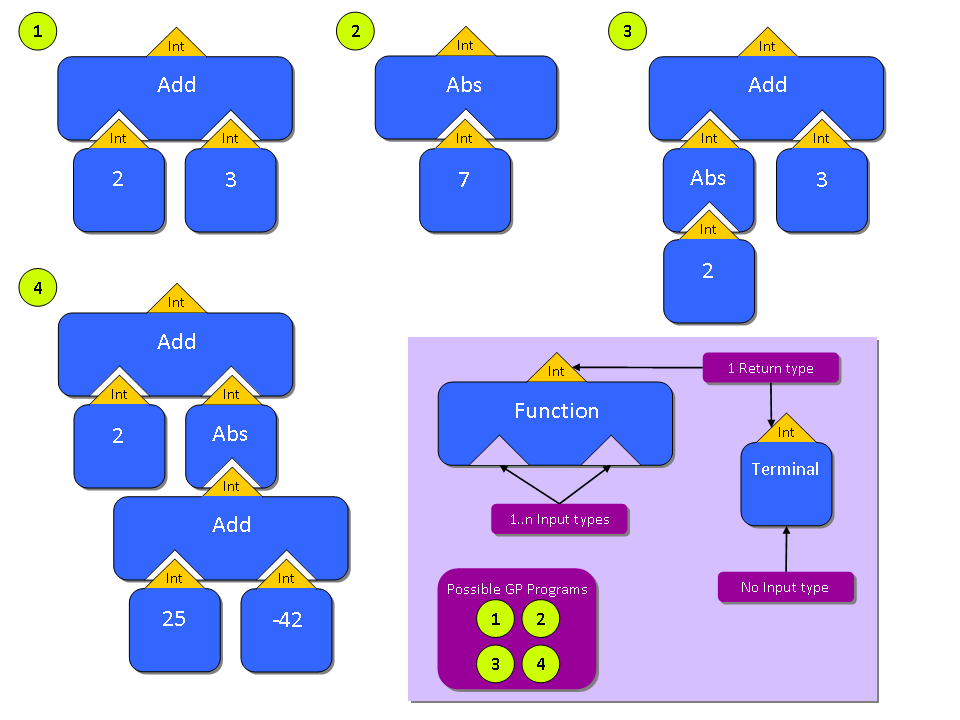


Рис.6 Ілюстрація до Генетичного Програмування

На зображені можна побачити дві функції Add і Abs. Add має два дочірні вузли цілочисельного типу і повертає цілочисельне значення. Abs має один дочірній вузол цілочисельного типу і повертає значення цілочисельного типу. Варто зазначити, що тип кореневого вузла дорівнює типу який повертає вся програма. На зображені додатково показано різні розстановки деяких функцій і терміналів.

**Імплементація в JGAP**

Щоб побачити як саме можна імплементувати вашу ГП-програму з JGAP, поглянемо на приклад розташований в класі examples.gp.MathProblem. Цей приклад намагається знайти формулу до заданої таблиці істинності і заданого набору операторів. Згаданий приклад також містить фітнес функцію названу FormulaFitnessFunction. Більш складний приклад можна знайти в класах examples.gp.Fibonacci та examples.gp.anttrail.AntTrailProblem.

Математичний приклад

1. **Створити конфігурацію**

/\*\*  
 \* Starts the example  
 \* @param args ignored  
 \* @throws Exception  
 \*/  
public static void main(String[] args) throws Exception {  
  GPConfiguration config = new GPConfiguration();  
  config.setGPFitnessEvaluator(new DeltaGPFitnessEvaluator());  
  config.setMaxInitDepth(6);  
  config.setPopulationSize(100);  
  config.setFitnessFunction(new MathProblem.FormulaFitnessFunction());  
  ... // continued below

Параметри використані тут:

* Fitness evaluator: дельта фітнес евалюатор дає нижчим значенням функції пристосованості кращий ранг, тому що значення пристосованості розглядається як оцінку дефективності. При бажанні можна використовувати значення пристосованості як оцінку успіху тоді використовуйте DefaultGPFitnessEvaluator.
* Максимальна глибина ГП-програми при ініціалізації встановлюється рівною 6. ГП-програма представлена як дерево. Максимальна кількість дітей кореневого вузла 6, що приводить до максимальної глибини 6. Завеликі значення можуть привести до погіршення продуктивності чи навіть переповнення пам’яті. 6 є підходящим значенням для нашого прикладу. В більш складних прикладах, намагайтесь використовувати максимум 12.
* Розмір популяції, тобто кількість ГП-програм, що існують одночасно, встановлено 100. Це вже достатньо велике значення так як кожна програма забирає час і ресурси. Якщо програми стають більшими, час і споживання ресурсів росте відповідно.
* Далі встановлюється фітнес функція. Вона оцінює кожну ГП-програму, що ще називається індивідуумом. Зверніть увагу, що фітнес функція ЗАВЖДИ специфічна до проблеми яка розглядається.

1. **Створити початковий генотип**

Ініціалізація і створення генотипу проводиться в методі create, що викликається з методу main.

 GPGenotype gp = create(config);  
}   
  
  
public static GPGenotype create(GPConfiguration a\_conf) throws InvalidConfigurationException {  
  Class[] types = {  
    CommandGene.FloatClass};  
  Class[][] argTypes = {  
    {}  
  };

Перший масив types, визначає тип повернення підпрограм. В нашому прикладі, ми маємо лише одну підпрограму, тому довжина масиву 1. Параметр CommandGene.FloatClass показує, що тип повернення Float.Class.

Другий масив argTypes визначає вхідний тип так званих Автоматично Визначених Функцій. Тут вони не важливі.

// Define the commands and terminals the GP is allowed to use.  
  // -----------------------------------------------------------  
  CommandGene[][] nodeSets = {  
    {  
    vx = Variable.create(a\_conf, "X", CommandGene.FloatClass),  
    new AddCommand(a\_conf, CommandGene.FloatClass),  
    new SubtractCommand(a\_conf, CommandGene.FloatClass),  
    new MultiplyCommand(a\_conf, CommandGene.FloatClass),  
    new DivideCommand(a\_conf, CommandGene.FloatClass),  
    new SinCommand(a\_conf, CommandGene.FloatClass),  
    new CosCommand(a\_conf, CommandGene.FloatClass),  
    new ExpCommand(a\_conf, CommandGene.FloatClass),  
    // Use terminal with possible value from 2.0 to 10.0 decimal  
    new Terminal(conf, CommandGene.FloatClass, 2.0d, 10.0d, false),  
    }  
  };

Кожна використана команда може бути або функцією або терміналом. Функція має принаймні один вхідний параметр, в той час як термінал ніколи не має вхідного параметру. Термінал це константа, як число чи логічна змінна. Можна знайти багато попередньо визначених функцій та терміналів в пакетах org.jgap.gp.function або org.jgap.gp.terminal.

  Random random = new Random();  
  // Randomly initialize function data (X-Y table) for x^4+x^3+x^2+x.  
  // This is problem-specific and not necessary in other cases.  
  // ----------------------------------------------------------------  
  for (int i = 0; i < 20; i++) {  
    float f = 2.0f \* (random.nextFloat() - 0.5f);  
    x[i] = new Float(f);  
    y[i] = f \* f \* f \* f + f \* f \* f + f \* f - f;  
    System.out.println(i + ") " + x[i] + " " + y[i]);  
  }  
  // Create genotype with initial population.  
  // Allow max. 100 nodes within one program.  
  // ----------------------------------------

Наступний код повертає ініціалізований гентип, використовуючи вищеописані параметри ініціалізації. Останні два параметри тут: число 100 – максимальна кількість вузлів для яких резервується пам’ять для кожної ГП-програми, останній параметр – true контролює виведення системою багатослівних повідомлень.

return randomInitialGenotype(a\_conf, types, argTypes, nodeSets, 100, true);  
}

1. **Почати еволюцію та вивести результат**

// Do 100 evolutions in a row.  
// ---------------------------  
  gp.evolve(100);  
// Output best solution found.  
// ---------------------------  
gp.outputSolution(gp.getAllTimeBest());  
  
} // end of method "main"

**Висновок:** під час виконання розрахункової роботи я ознайомивсь з бібліотекою для розробки генетичних алгоритмів JGAP. А саме дізнавсь про особливості, які нам пропонує дана бібліотека. Розібравсь, як підключити даний фреймворк до середовища розробки програмного забезпечення Eclipse. Навів приклад розробки ПЗ в JGAP.

Додаток А. Код програми

**KnapsackMain.java**

/\*

\* This file is part of JGAP.

\*

\* JGAP offers a dual license model containing the LGPL as well as the MPL.

\*

\* For licensing information please see the file license.txt included with JGAP

\* or have a look at the top of class org.jgap.Chromosome which representatively

\* includes the JGAP license policy applicable for any file delivered with JGAP.

\*/

**import** java.io.\*;

**import** org.jgap.\*;

**import** org.jgap.data.\*;

**import** org.jgap.impl.\*;

**import** org.jgap.xml.\*;

**import** org.w3c.dom.\*;

/\*\*

\* This class provides an implementation of the classic knapsack problem

\* using a genetic algorithm. The goal of the problem is to reach a given

\* volume (of a knapsack) by putting a number of items into the knapsack.

\* The closer the sum of the item volumes to the given volume the better.

\* <p>

\* For further descriptions, compare the "coins" example also provided.

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **class** KnapsackMain {

/\*\* String containing the CVS revision. Read out via reflection!\*/

**private** **final** **static** String ***CVS\_REVISION*** = "$Revision: 1.10 $";

/\*\*

\* The total number of times we'll let the population evolve.

\*/

**private** **static** **final** **int** ***MAX\_ALLOWED\_EVOLUTIONS*** = 140;

/\*\* Volumes of arbitrary items in ccm\*/

**public** **final** **static** **double**[] ***itemVolumes*** = {

50.2d, 14.8d, 27.5d, 6800.0d, 25.0d, 4.75d, 95.36d, 1500.7d, 18365.9d,

83571.1d};

/\*\* Names of arbitrary items, only for outputting something imaginable\*/

**public** **final** **static** String[] ***itemNames*** = {

"Torch", "Banana", "Miniradio", "TV", "Gameboy", "Small thingie",

"Medium thingie", "Big thingie", "Huge thingie", "Gigantic thingie"};

/\*\*

\* Executes the genetic algorithm to determine the minimum number of

\* items necessary to make up the given target volume. The solution will then

\* be written to the console.

\*

\* **@param** a\_knapsackVolume the target volume for which this method is

\* attempting to produce the optimal list of items

\*

\* **@throws** Exception

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **static** **void** findItemsForVolume(**double** a\_knapsackVolume)

**throws** Exception {

// Start with a DefaultConfiguration, which comes setup with the

// most common settings.

// -------------------------------------------------------------

Configuration conf = **new** DefaultConfiguration();

conf.setPreservFittestIndividual(**true**);

// Set the fitness function we want to use. We construct it with

// the target volume passed in to this method.

// ---------------------------------------------------------

FitnessFunction myFunc =

**new** KnapsackFitnessFunction(a\_knapsackVolume);

conf.setFitnessFunction(myFunc);

// Now we need to tell the Configuration object how we want our

// Chromosomes to be setup. We do that by actually creating a

// sample Chromosome and then setting it on the Configuration

// object. As mentioned earlier, we want our Chromosomes to each

// have as many genes as there are different items available. We want the

// values (alleles) of those genes to be integers, which represent

// how many items of that type we have. We therefore use the

// IntegerGene class to represent each of the genes. That class

// also lets us specify a lower and upper bound, which we set

// to senseful values (i.e. maximum possible) for each item type.

// --------------------------------------------------------------

Gene[] sampleGenes = **new** Gene[***itemVolumes***.length];

**for** (**int** i = 0; i < ***itemVolumes***.length; i++) {

sampleGenes[i] = **new** IntegerGene(conf, 0,

(**int**) Math.*ceil*(a\_knapsackVolume /

***itemVolumes***[i]));

}

IChromosome sampleChromosome = **new** Chromosome(conf, sampleGenes);

conf.setSampleChromosome(sampleChromosome);

// Finally, we need to tell the Configuration object how many

// Chromosomes we want in our population. The more Chromosomes,

// the larger number of potential solutions (which is good for

// finding the answer), but the longer it will take to evolve

// the population (which could be seen as bad).

// ------------------------------------------------------------

conf.setPopulationSize(50);

// Create random initial population of Chromosomes.

// Here we try to read in a previous run via XMLManager.readFile(..)

// for demonstration purpose!

// -----------------------------------------------------------------

Genotype population;

**try** {

Document doc = XMLManager.*readFile*(**new** File("knapsackJGAP.xml"));

population = XMLManager.*getGenotypeFromDocument*(conf, doc);

}

**catch** (FileNotFoundException fex) {

population = Genotype.*randomInitialGenotype*(conf);

}

population = Genotype.*randomInitialGenotype*(conf);

// Evolve the population. Since we don't know what the best answer

// is going to be, we just evolve the max number of times.

// ---------------------------------------------------------------

**for** (**int** i = 0; i < ***MAX\_ALLOWED\_EVOLUTIONS***; i++) {

population.evolve();

}

// Save progress to file. A new run of this example will then be able to

// resume where it stopped before!

// ---------------------------------------------------------------------

// represent Genotype as tree with elements Chromomes and Genes

// ------------------------------------------------------------

DataTreeBuilder builder = DataTreeBuilder.*getInstance*();

IDataCreators doc2 = builder.representGenotypeAsDocument(population);

// create XML document from generated tree

// ---------------------------------------

XMLDocumentBuilder docbuilder = **new** XMLDocumentBuilder();

Document xmlDoc = (Document) docbuilder.buildDocument(doc2);

XMLManager.*writeFile*(xmlDoc, **new** File("knapsackJGAP.xml"));

// Display the best solution we found.

// -----------------------------------

IChromosome bestSolutionSoFar = population.getFittestChromosome();

System.***out***.println("The best solution has a fitness value of " +

bestSolutionSoFar.getFitnessValue());

System.***out***.println("It contained the following: ");

**int** count;

**double** totalVolume = 0.0d;

**for** (**int** i = 0; i < bestSolutionSoFar.size(); i++) {

count = ( (Integer) bestSolutionSoFar.getGene(i).getAllele()).intValue();

**if** (count > 0) {

System.***out***.println("\t " + count + " x " + ***itemNames***[i]);

totalVolume += ***itemVolumes***[i] \* count;

}

}

System.***out***.println("\nFor a total volume of " + totalVolume + " ccm");

System.***out***.println("Expected volume was " + a\_knapsackVolume + " ccm");

System.***out***.println("Volume difference is " +

Math.*abs*(totalVolume - a\_knapsackVolume) + " ccm");

}

/\*\*

\* Main method. A single command-line argument is expected, which is the

\* volume to create (in other words, 75 would be equal to 75 ccm).

\*

\* **@param** args first and single element in the array = volume of the knapsack

\* to fill as a double value

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **static** **void** main(String[] args) {

**if** (args.length != 1) {

System.***out***.println("Syntax: " + KnapsackMain.**class**.getName() +

" <volume>");

}

**else** {

**try** {

**double** volume = Double.*parseDouble*(args[0]);

**if** (volume < 1 ||

volume >= KnapsackFitnessFunction.***MAX\_BOUND***) {

System.***out***.println("The <volume> argument must be between 1 and "

+

(KnapsackFitnessFunction.***MAX\_BOUND*** - 1)

+ " and can be a decimal.");

}

**else** {

**try** {

*findItemsForVolume*(volume);

}

**catch** (Exception e) {

e.printStackTrace();

}

}

}

**catch** (NumberFormatException e) {

System.***out***.println(

"The <volume> argument must be a valid double value");

}

}

}

}

**KnapsackFitnessFunction.java**

/\*

\* This file is part of JGAP.

\*

\* JGAP offers a dual license model containing the LGPL as well as the MPL.

\*

\* For licensing information please see the file license.txt included with JGAP

\* or have a look at the top of class org.jgap.Chromosome which representatively

\* includes the JGAP license policy applicable for any file delivered with JGAP.

\*/

**import** org.jgap.\*;

/\*\*

\* Fitness function for the knapsack example.

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **class** KnapsackFitnessFunction

**extends** FitnessFunction {

/\*\* String containing the CVS revision. Read out via reflection!\*/

**private** **final** **static** String ***CVS\_REVISION*** = "$Revision: 1.5 $";

**private** **final** **double** m\_knapsackVolume;

**public** **static** **final** **double** ***MAX\_BOUND*** = 1000000000.0d;

**private** **static** **final** **double** ***ZERO\_DIFFERENCE\_FITNESS*** = Math.*sqrt*(***MAX\_BOUND***);

**public** KnapsackFitnessFunction(**double** a\_knapsackVolume) {

**if** (a\_knapsackVolume < 1 || a\_knapsackVolume >= ***MAX\_BOUND***) {

**throw** **new** IllegalArgumentException(

"Knapsack volumen must be between 1 and " + ***MAX\_BOUND*** + ".");

}

m\_knapsackVolume = a\_knapsackVolume;

}

/\*\*

\* Determine the fitness of the given Chromosome instance. The higher the

\* return value, the more fit the instance. This method should always

\* return the same fitness value for two equivalent Chromosome instances.

\*

\* **@param** a\_subject the Chromosome instance to evaluate

\* **@return** a positive double reflecting the fitness rating of the given

\* Chromosome

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **double** evaluate(IChromosome a\_subject) {

// The fitness value measures both how close the value is to the

// target amount supplied by the user and the total number of items

// represented by the solution. We do this in two steps: first,

// we consider only the represented amount of change vs. the target

// amount of change and return higher fitness values for amounts

// closer to the target, and lower fitness values for amounts further

// away from the target. Then we go to step 2, which returns a higher

// fitness value for solutions representing fewer total items, and

// lower fitness values for solutions representing more total items.

// ------------------------------------------------------------------

**double** totalVolume = *getTotalVolume*(a\_subject);

**int** numberOfItems = *getTotalNumberOfItems*(a\_subject);

**double** volumeDifference = Math.*abs*(m\_knapsackVolume - totalVolume);

**double** fitness = 0.0d;

// Step 1: Determine distance of amount represented by solution from

// the target amount. If the change difference is greater than zero we

// will divide one by the difference in change between the

// solution amount and the target amount. That will give the desired

// effect of returning higher values for amounts closer to the target

// amount and lower values for amounts further away from the target

// amount.

// In the case where the change difference is zero it means that we have

// the correct amount and we assign a higher fitness value

// -----------------------------------------------------------------

fitness += volumeDifferenceBonus(***MAX\_BOUND***, volumeDifference);

// Step 2: We divide the fitness value by a penalty based on the number of

// items. The higher the number of items the higher the penalty and the

// smaller the fitness value.

// And inversely the smaller number of items in the solution the higher

// the resulting fitness value.

// -----------------------------------------------------------------------

fitness -= computeItemNumberPenalty(***MAX\_BOUND***, numberOfItems);

// Make sure fitness value is always positive.

// -------------------------------------------

**return** Math.*max*(1.0d, fitness);

}

/\*\*

\* Bonus calculation of fitness value.

\* **@param** a\_maxFitness maximum fitness value appliable

\* **@param** a\_volumeDifference volume difference in ccm for the items problem

\* **@return** bonus for given volume difference

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**protected** **double** volumeDifferenceBonus(**double** a\_maxFitness,

**double** a\_volumeDifference) {

**if** (a\_volumeDifference == 0) {

**return** a\_maxFitness;

}

**else** {

// we arbitrarily work with half of the maximum fitness as basis for non-

// optimal solutions (concerning volume difference)

**return** a\_maxFitness / 2 - (a\_volumeDifference \* a\_volumeDifference);

}

}

/\*\*

\* Calculates the penalty to apply to the fitness value based on the amount

\* of items in the solution.

\*

\* **@param** a\_maxFitness upper boundary for fitness value possible

\* **@param** a\_items number of items in the solution

\* **@return** a penalty for the fitness value based on the number of items

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**protected** **double** computeItemNumberPenalty(**double** a\_maxFitness, **int** a\_items) {

**if** (a\_items == 0) {

// We know the solution cannot have less than zero items.

// ------------------------------------------------------

**return** 0;

}

**else** {

// The more items the more penalty, but not more than the maximum fitness

// value possible. Let's avoid linear behavior and use

// exponential penalty calculation instead.

// ----------------------------------------------------------------------

**return** (Math.*min*(a\_maxFitness, a\_items \* a\_items));

}

}

/\*\*

\* Calculates the total amount of change (in cents) represented by

\* the given potential solution and returns that amount.

\*

\* **@param** a\_potentialSolution the potential solution to evaluate

\* **@return** the total amount of change (in cents) represented by the

\* given solution

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **static** **double** getTotalVolume(IChromosome a\_potentialSolution) {

**double** volume = 0.0d;

**for** (**int** i = 0; i < a\_potentialSolution.size(); i++) {

volume += *getNumberOfItemsAtGene*(a\_potentialSolution, i) \*

KnapsackMain.***itemVolumes***[i];

}

**return** volume;

}

/\*\*

\* Retrieves the number of items represented by the given potential

\* solution at the given gene position.

\*

\* **@param** a\_potentialSolution the potential solution to evaluate

\* **@param** a\_position the gene position to evaluate

\* **@return** the number of items represented by the potential solution

\* at the given gene position

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **static** **int** getNumberOfItemsAtGene(IChromosome a\_potentialSolution,

**int** a\_position) {

Integer numItems =

(Integer) a\_potentialSolution.getGene(a\_position).getAllele();

**return** numItems.intValue();

}

/\*\*

\* Returns the total number of items represented by all of the genes in

\* the given potential solution.

\*

\* **@param** a\_potentialSolution the potential solution to evaluate

\* **@return** the total number of items represented by the given Chromosome

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **static** **int** getTotalNumberOfItems(IChromosome a\_potentialSolution) {

**int** totalItems = 0;

**int** numberOfGenes = a\_potentialSolution.size();

**for** (**int** i = 0; i < numberOfGenes; i++) {

totalItems += *getNumberOfItemsAtGene*(a\_potentialSolution, i);

}

**return** totalItems;

}

}