|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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

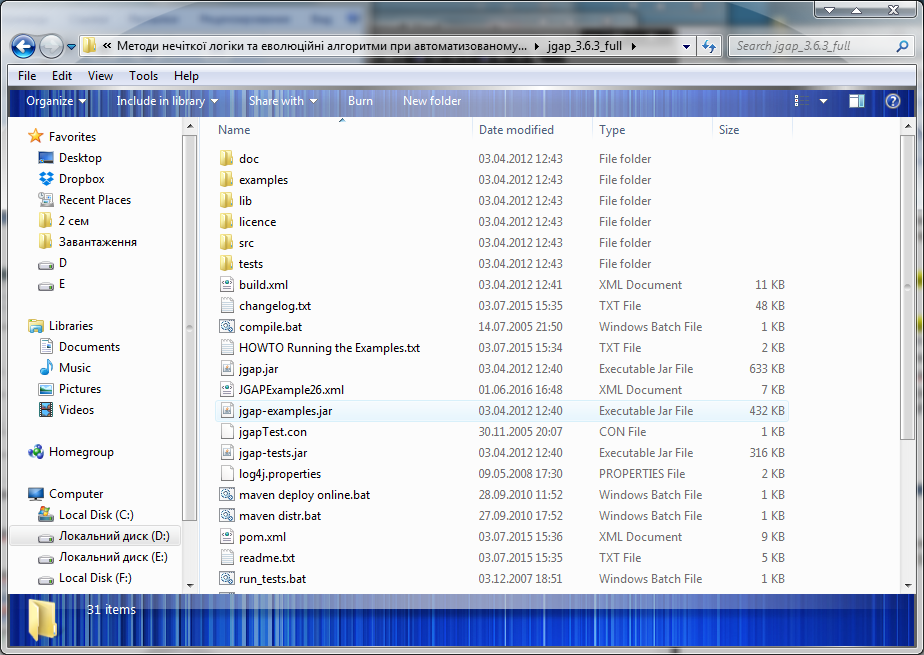


Рис.1 Вміст архіву JGAP

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

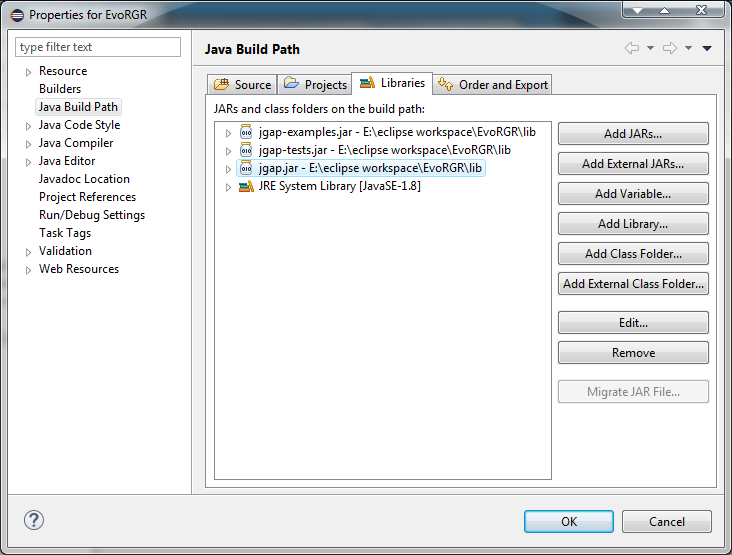


Рис.2 Додавання бібліотек до проекту

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).

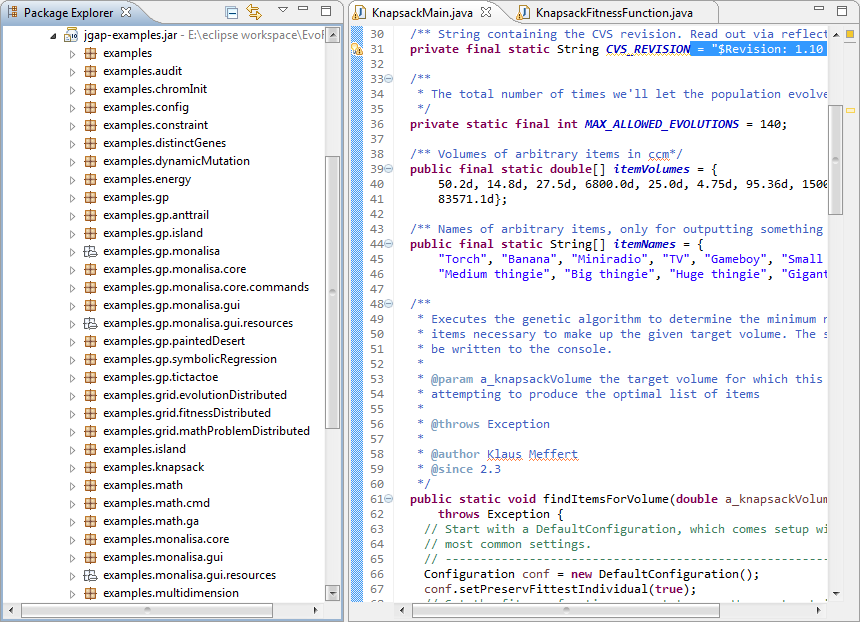


Рис.3 Приклади з бібліотеки jgap-examples.

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.

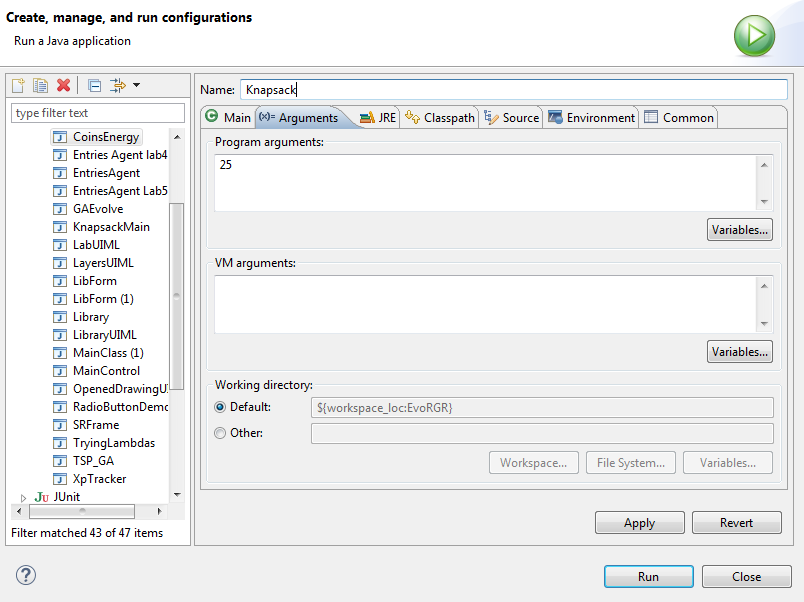


Рис.4 Задання параметру запуску прикладу

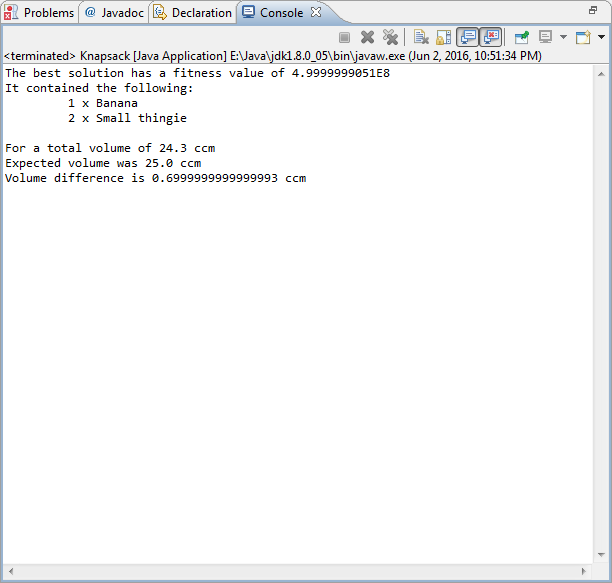


Рис.5 Результат запуску прикладу Knapsack в консолі

**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.

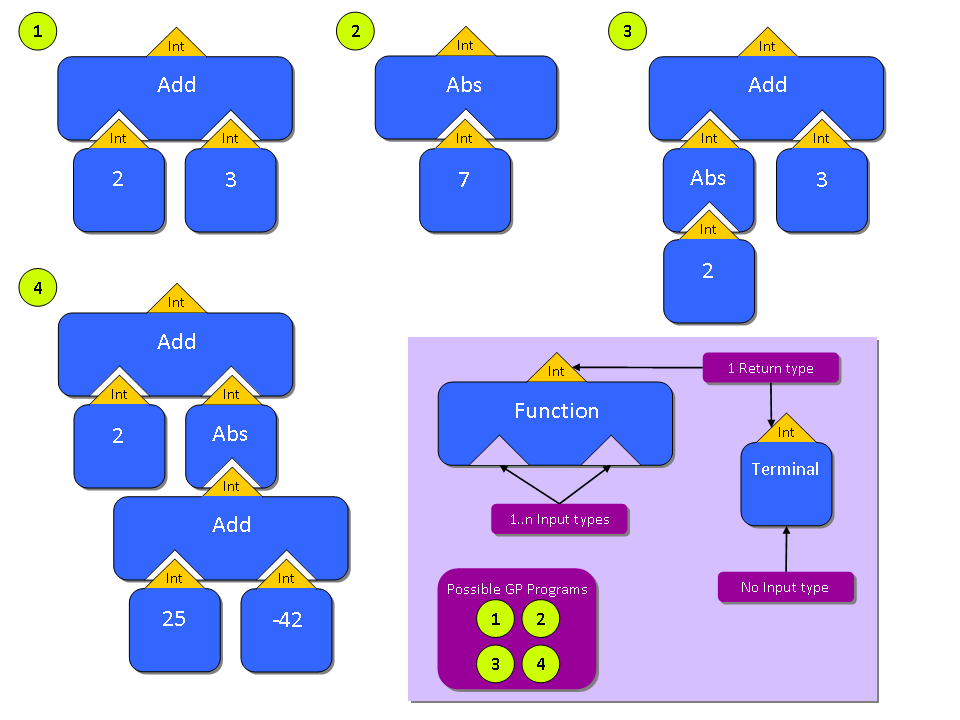


Рис.6 Ілюстрація до Генетичного Програмування

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"

**Приклад 2:**

**Створення спеціальних генів**

Гени представляють дискретні компоненти проблемного рішення(хромосоми) які можуть різнитись незалежно один від одного напротязі процесу еволюції. Подібно до того як гени контролюють різні незалежні властивості нашого власного генетичного набору, такі як стать чи колір очей, так само гени в JGAP контролюють різні аспекти рішення. JGAP має кілька стандартних реалізацій генів які є корисними для генів які представляються логічними чи цілочисельними значеннями. Але багато аплікацій можуть потребувати більш спеціалізовані представлення, в такому випадку в пригоді стають спеціальні гени. Їх легко створити, в чому ми скоро переконаємось.

**Як гени використовуються**

Просто кажучи, ген представляє деякі характеристики рішення, які можуть змінюватись незалежно від інших характеристик рішення. Наприклад, скажемо, що рішення представляє кілька монет. Тоді гени в рішенні можуть представляти різні номінали монет з цього набору (5,10,25,50). Вартість цих генів (аллели) буде кількістю монет кожного номіналу в цьому рішенні. В JGAP гени представляються класом, що імплементує інтерфейс Gene. Кожен ген в хромосомі може мати різні імплементацій Gene для керування значеннями, але ці імплементації мають бути відповідними по всій хромосомі. Наприклад, якщо ген 1 в хромосомі А використовує певну імплементацію, скажімо QuarterGene, тоді ген 1 у решти хромосом цієї популяції повинен також використовувати QuarterGene. Але ген 2 у всіх цих хромосомах може використовувати якусь іншу імплементацію. Звичайно всі гени в хромосомі можуть також використовувати ту саму імплементацію.

**Приклад: QuarterGene**

Як приклад імплементації гену, розглянемо класс QuarterGene який може бути використаний для представлення кількості четвертаків в наборі монет. Код класу QuarterGene наведено в додатку.

**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.\*;

/\*\*

\* В цьому классы подано класичну реалізацію задачі рюкзака з використанням

\* генетичного алгоритму. Мета цієї задачі в досягнені певного об’єму кладучи в рюкзак

\* різні речі.

\* Чим ближча сума об’єму предметів до поданого числа тим краще.

\* <p>

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

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

/\*\* Стрічка містить ревізію CVS. \*/

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

/\*\*

\* Загальна кількість еволюцій популяцій.

\*/

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

/\*\* Об’єми предметів в 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};

/\*\* Назви предметів для виведення результатів\*/

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

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

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

/\*\*

\* Запускає генетичний алгоритм для визначення мінімальну кількість

\* предметів потрібних для заповнення поданого об’єму. Рішення виводиться в консолі.

\*

\* **@param** a\_knapsackVolume об’єм для якого знаходиться оптимальний об’єм придметів

\*

\* **@throws** Exception

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

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

**throws** Exception {

// Початок з DefaultConfiguration, де встановлено типові параметри.

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

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

conf.setPreservFittestIndividual(**true**);

// Встановлюєм фітнес функцію яку хочем використовувати. Ми складаємо її з цільовим //об’ємом, що подається в метод.

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

FitnessFunction myFunc =

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

conf.setFitnessFunction(myFunc);

// Тепер ми кажемо об’єкту Configuration як ми хочемо, щоб наші хромосоми були

// встановлені. Для цього створюєм хромосому і подаєм її в об’єкт конфігурації. Ми //хочем, щоб наші хромосоми мали стільки ж генів скільки в нас є предметів. Ми хочем //щоб в генів були цілочисельні типи для представлення скільки предметів цього типу ми //маєм. Тому використовуєм клас IntegerGene для представлення кожного з генів. Цей клас //також дозволяє встановити верхню і нижню межу очікуваних значень.

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

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);

// Кажемо об’єкту конфігурації скільки хромосом нам потрібно. Чим більше хромосом //тим більше число можливих рішень(що добре для знаходження рішення), але тим більше //часу займе еволюція популяції.

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

conf.setPopulationSize(50);

// Створення рандомної початкової популяції хромосом.

// Тут ми намагаємось зчитами попереднє проходження через XMLManager.readFile(..)

// для демонстрації.

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

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);

// Еволюція популяції. Так як ми не знаєм яке рішення буде найкращим

// ми просто еволюціонуємо максимальну кількість разів.

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

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

population.evolve();

}

// Збереження файлу з прогресом. Новий запуск цього прикладу зможе

// продовжити з того місця де він зупинився!

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

// представлення генотипу як дереа де елементи хромосоми і гени

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

DataTreeBuilder builder = DataTreeBuilder.*getInstance*();

IDataCreators doc2 = builder.representGenotypeAsDocument(population);

// Створити XML документ зі згенерованого дерева

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

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

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

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

// Виведення найкращого рішення.

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

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");

}

/\*\*

\* Головний метод. Очікується один аргумент, що є об’ємом який ми створюєм.

\*

\*

\* **@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.\*;

/\*\*

\* Фітнес функція для прикладу з рюкзаком.

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

**public** **class** KnapsackFitnessFunction

**extends** FitnessFunction {

**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;

}

/\*\*

\* Визначає пристосованість даної хромосоми. Чим більше

\* значення повернення, тим більше підходить приклад. Цей метод завжди повинен

\* повертати одинакове значення для двох однакових хромосом.

\*

\* **@param** a\_subject хромосома для розгляду

\* **@return** додатнє значення double, що показує значення пристосованості в хромосомі

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

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

// Значення пристосованості вимірює наскільки близьке значення до цільового значення яке надав користувач і загальну кількість представлену в рішені. Ми робимо це в два кроки:

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

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

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

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

**double** fitness = 0.0d;

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

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

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

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

// Переконуємось, що значення пристосованості позитивне.

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

**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** {

// ми довільно працюємо з половиною максимального значення пристосованості як

//база для не оптимального рішення

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

}

}

/\*\*

\* Обраховує штраф для застосування до значення пристосованості

\* базуючись на кількості предметів в рішенні.

\*

\* **@param** a\_maxFitness верхня межа можливого рішення

\* **@param** a\_items кількість предметів в рішенні

\* **@return** штраф

\*

\* **@author** Klaus Meffert

\* **@since** 2.3

\*/

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

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

// Ми знаємо, що в рішенні не може бути 0 предметів.

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

**return** 0;

}

**else** {

// Більше предметів більший штраф, але не більше за максимальне значення

// пристосованості. Уникнемо лінійної поведінки і використаємо

// експоненційий підрахунок штраф.

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

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

}

}

**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;

}

/\*\*

\* Дістає кількість предметів представлених даним потенційним рішенням в даній позиції гена.

\*

\* **@param** a\_potentialSolution потенційне рішення

\* **@param** a\_position позиція гена для розгляду

\*

\* **@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();

}

/\*\*

\* Повертає загальну кількість предметів.

\*

\* **@param** a\_potentialSolution потенційне рішення

\*

\* **@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;

}

}

**QuarterGene.java**

**package** hypothetical.examples;

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

**import** java.util.StringTokenizer;

*/\*\**

*\* Ген, що використовується для представлення четвертаків в наборі монет.*

*\*/*

**public** **class** QuarterGene extends BaseGene **implements** Gene, java.io.Serializable

{

**private** **static** **final** String TOKEN\_SEPARATOR = **":"**;

**private** **int** m\_maxNumberOfQuarters;

**private** Integer m\_numberOfQuarters;

*/\*\**

*\* Створює новий QuarterGene.*

*\*/*

**public** QuarterGene(Configuration a\_conf) throws InvalidConfigurationException {

this(a\_conf, Integer.MAX\_VALUE);

}

*/\*\**

*\* Конструює новий QuarterGene з обмеженням на максимальну кількість четвертаків які можуть бути представлені цим QuarterGene.*

*\**

*\* @param a\_maxNumberOfQuarters Максимальна кількість четвертаків що цей ген може представляти. Це значення не може бути відємним.*

*\**

*\* @throws IllegalArgumentException if the given maximum value is negative.*

*\*/*

**public** QuarterGene(Configuration a\_conf, **int** a\_maxNumberOfQuarters ) throws InvalidConfigurationException {

super(a\_conf);

*// Перевірка чи переданий максимум негативний.*

*// --------------------------------------------*

**if**( a\_maxNumberOfQuarters < 0 )

{

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

**"The maximum number of quarters must be non-negative."** );

}

m\_maxNumberOfQuarters = a\_maxNumberOfQuarters;

}

*/\*\**

*\* Надає засоби незалежні від імплементації для створення нового зразка гена*

*\**

*\**

*\* @return A new Gene instance of the same type and with the same*

*\* setup as this concrete Gene.*

*\*/*

**public** Gene newGeneInternal( ) {

try {

*// Ми конструюємо новий QuarterGene з такою ж максимальною кількістю четвертаків з якою цей ген був створений.*

*// -------------------------------------------------------------*

return new QuarterGene(getConfiguration(), m\_maxNumberOfQuarters);

} catch (InvalidConfigurationException ex) {

throw new IllegalStateException(ex.getMessage());

}

}

*/\*\**

*\* Встановлює значенню гена нове значення. Тип значення залежить*

*\* від імплементації.*

*\**

*\* @param a\_newValue нове значення гена.*

*\*/*

**public** **void** setAllele( Object a\_newValue )

{

m\_numberOfQuarters = (Integer) a\_newValue;

}

*/\*\**

*\* Дістає значення представлене геном.*

*\**

*\* @return значення гена.*

*\*/*

**public** Object getAllele()

{

**return** m\_numberOfQuarters;

}

*/\*\**

*\* Встановлює значенню гена рандомне допустиме значення*

*.*

*\**

*\*/*

**public** **void** setToRandomValue( RandomGenerator a\_numberGenerator )

{

*// Рандомне значення між 0 і максимумом четвертаків які ми можем представити.*

*// -----------------------------------------------------*

m\_numberOfQuarters = **new** Integer(

a\_numberGenerator.nextInt( m\_maxNumberOfQuarters) );

}

*/\*\**

*\* Дістає стрічкове представлення значення цього гена, що включає інформацію потрібну для його відтворення в майбутньому. Ця стрічка буде використовуватись для представлення гена в XML представлені. This*

*\**

*\* @return A string representation of this Gene's current state.*

*\* @throws UnsupportedOperationException to indicate that no implementation*

*\* is provided for this method.*

*\*/*

**public** String getPersistentRepresentation()

**throws** UnsupportedOperationException

{

*// Ми хочемо представити і максимальну кількість четвертаків які можуть бути представлені в цьому гені та теперішнє значення.*

*// ---------------------------------------------------------------*

**return** **new** Integer( m\_maxNumberOfQuarters ).toString() +

TOKEN\_SEPARATOR + m\_numberOfQuarters.toString();

}

*/\*\**

*\* Встановлює значення і внутрішній стан гену зі стрічки що повертається через метод getPersistentRepresentation().*

*\**

*\* @param a\_representation стрічка з представлення гена.*

*\**

*\*/*

**public** **void** setValueFromPersistentRepresentation( String a\_representation )

**throws** UnsupportedOperationException,

UnsupportedRepresentationException

{

StringTokenizer tokenizer = **new** StringTokenizer( a\_representation,

TOKEN\_SEPARATOR );

**if**( tokenizer.countTokens() != 2 )

{

**throw** **new** UnsupportedRepresentationException(

**"Unknown representation format: Two tokens expected."** );

}

**try**

{

m\_maxNumberOfQuarters = Integer.parseInt( tokenizer.nextToken() );

m\_numberOfQuarters = **new** Integer( tokenizer.nextToken() );

}

**catch**( NumberFormatException e )

{

**throw** **new** UnsupportedRepresentationException(

**"Unknown representation format: Expecting integer values."** );

}

}

*/\*\**

*\* Очікується генетичним движком коли зразок гена більше не потрібний.*

*\*/*

**public** **void** cleanup()

{

*// В цій імплементації очищення не потрібне.*

*// -----------------------------------------------------*

}

**public** **int** compareTo( Object a\_otherQuarterGene )

{

**if**( a\_otherQuarterGene == **null** )

{

**return** 1;

}

**if**( m\_numberOfQuarters == **null** )

{

**if** ( ((QuarterGene) a\_otherQuarterGene).m\_numberOfQuarters == **null** )

{

**return** 0;

}

**else**

{

**return** -1;

}

}

**return** m\_numberOfQuarters.compareTo(

( (QuarterGene) a\_otherQuarterGene ).m\_numberOfQuarters );

}

*/\*\**

*\* Визначає чи цей QuarterGene дорівнює даному QuarterGene.*

*\**

*\*/*

**public** **boolean** equals( Object otherQuarterGene )

{

**return** otherQuarterGene **instanceof** QuarterGene &&

compareTo( otherQuarterGene ) == 0;

}

*/\*\**

*\* Обраховує хеш-код для цього QuarterGene.*

*\**

*\* @return the hash-code of this QuarterGene*

*\*/*

**public** **int** hashCode()

{

**return** m\_numberOfQuarters;

}

**public** Object getInternalValue() {

**return** m\_numberOfQuarters;

}

public void applyMutation(int a\_index, double a\_percentage) {

setAllele(getConfiguration().getRandomGenerator().nextInt(m\_maxNumberOfQuarters));

}

}