|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САПР** | | Тема | оцінка | підпис |
| КНм-14 | РГР | **Фреймворк JGAP** |  |  |
| Малоїд Т.В. | |
| № залікової: | |
| **Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні** | | Викладач: | |
| к.т.н. Кривий Р.З. | |

Зміст

[Вступ 3](#_Toc482745720)

[Встановлення JGAP 3](#_Toc482745721)

[Покрокове налаштування ГА 4](#_Toc482745722)

[Крок 1. Проектування хромосоми 5](#_Toc482745723)

[Крок 2. Реалізація фітнес функції 5](#_Toc482745724)

[Крок 3. Налаштування конфігурації об’єкта 7](#_Toc482745725)

[Крок 4. Створення популяції 8](#_Toc482745726)

[Крок 5. Еволюція 8](#_Toc482745727)

[Стандартні налаштування 9](#_Toc482745728)

[Приклади 10](#_Toc482745729)

[Приклад 1 10](#_Toc482745730)

[Приклад 2 10](#_Toc482745731)

[Приклад 3 11](#_Toc482745732)

[Приклад 4 13](#_Toc482745733)

# Вступ

JGAP – генетичний алгоритм та Java фреймворк для програмування генетичних алгоритмів. Він забезпечує основні генетичні механізми, які можуть бути легко використані для застосування еволюційних принципів до вирішення задач.

JGAP був розроблений, щоб бути дуже простим у використанні «з коробки», а також був спроектований з високим ступенем модульності, так що більш прогресивні користувачі можуть легко написати та використовувати свої кастомні генетичні оператори, такі як мутації або кросинговери чи інші компоненти.

Документація, якість і стабільність коду є найбільш важливими аспектами при розробці JGAP. Користувач з легкістю може знайти різні юніт тести JGAP (> 1400), розширені Java-доки і досить багато прикладів.

# Встановлення JGAP

JGAP це фреймворк написаний на Java, це не програма, яка може працювати як виконуваний файл. Проте є приклади, які можна запустити з командного рядка або за допомогою batch-файлу. Так як фреймворк є частиною Java, то його можна запускати на будь-яких операційних системах.

Встановлення:

1. Спочатку необхідно завантажити пакет JGAP з сайту SourceForge. Бажано завантажувати повний пакет, з прикладами та іншою необхідною інформацією, а не лише вихідний код. Приклад назви архіву «jgap\_xxx\_full.zip». На даний момент остання версія фреймворку jgap\_3.6.3, останній раз модифікована 2015-07-03.

2. Після цього потрібно витягнути архів у директорію на жорсткому диску.

3. В директорії куди було розархівовано пакет знаходяться файли з розширенням .jar. Ці файли містять скомпільований JGAP код. Також у папці Lib знаходяться деякі залежності (dependencied) для зовнішніх бібліотек.

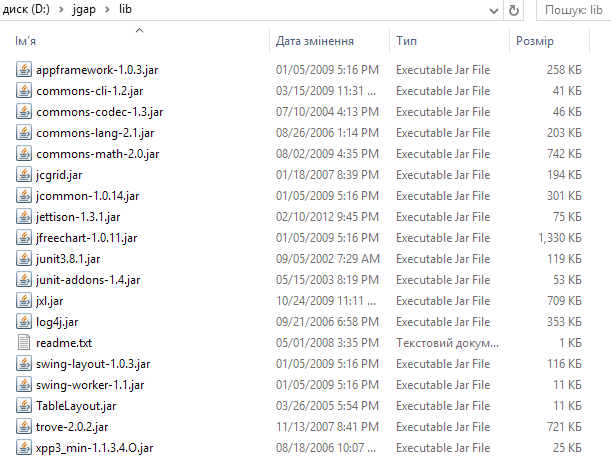


Рис.1 Бібліотеки в папці Lib

5. Запуск прикладу.

Windows: Запустити приклад можна виконавши файл start\_example.bat.

Інші операційні системи : Перейти до командного рядка і змінити свій робочий каталог на jgap. Тоді ввести наступну команду:

Java -classpath ";. Jgap-examples.jar; jgap.jar" examples.MinimizingMakeChange 89

Ubuntu Linux : Як і раніше, але з двокрапками замість ком для розділення компонентів шляху до класів):

Java -classpath ":. Jgap-examples.jar: jgap.jar" examples.MinimizingMakeChange 89

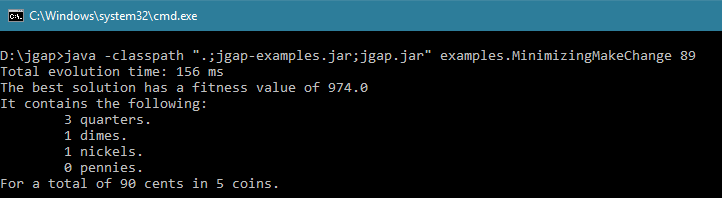


Рис.2 Результат запуску програми

6. Якщо потрібно використовувати класи зовнішніх бібліотек потрібно додати ці бібліотеки у команду кроку 5.

7. JGAP може бути скомпільований з версією Java 5 або пізнішої.

# Покрокове налаштування ГА

Щоб використовувати JGAP в програмах, є п'ять основних речей, які потрібно зробити:

1. Спроектувати хромосому
2. Реалізувати фітнес функції
3. Налаштувати об’єкт конфігурації
4. Створити популяцію потенційних рішень
5. Еволюціонувати популяцію

Далі буде покроково створено та представлено простий приклад програми, що використовує JGAP.

## Крок 1. Проектування хромосоми

В основі генетичного алгоритму є Хромосома. Хромосома являє собою потенційне рішення і розділена на кілька генів. Гени в JGAP представляють різні аспекти вирішення в цілому, так само, як людські гени являють собою різні аспекти окремих людей, наприклад, статі або кольору очей. В процесі еволюції JGAP, хромосоми піддаються кільком генетичних операторам, які представляють собою схрещування, мутацію і т.д., а потім вибираються для наступного покоління під час природної фази відбору на основі їх «придатності», яка є мірою того, наскільки оптимальним рішення є відносно інших можливих рішень. Вся мета генетичного алгоритму, полягає у тому щоб імітувати природний процес еволюції з метою отримання кращих рішень.

Крок 1 полягає у прийнятті рішення про склад хромосом, який включає в себе, скільки генів потрібно і що ці гени будуть представляти. Програма, що розглядається полягає в пошуку найменшої кількості монет різного номіналу сума яких буде рівною числу, що задасть користувач. Таким чином хромосомою є набір монет, а генами різні номінали монет, наприклад, номінали американських монет (рис.1).

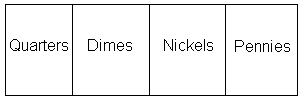


Рис.3 Хромосома з чотирьома генами

## Крок 2. Реалізація фітнес функції

JGAP розроблений, щоб зробити майже всю еволюційну роботу для користувача у відносно загальному вигляді. Тим не менш, немає загального шляху вирішення, для будь якої проблеми. Ось де фітнес функція є необхідною: це єдиний метод, який необхідно реалізувати, який приймає потенційне рішення задачі і повертає ціле значення, яке вказує, наскільки кращим є рішення щодо інших можливих рішень. Чим вище число, тим краще рішення. Чим менше число (1 є найнижчим допустимим значенням придатності), тим гірше рішення.

Оскільки мета цієї програми, щоб знайти найменший набір американських монет при якому сума рівна заданій, зрозуміло, що міра придатності для конкретного рішення буде являти собою поєднання суми грошей та загальної кількості монет.

**package** examples;

**import** org.jgap.Chromosome;

**import** org.jgap.FitnessFunction;

**public** **class** MinimizingMakeChangeFitnessFunction **extends** FitnessFunction

{

**private** **final** **int** m\_targetAmount;

**public** MinimizingMakeChangeFitnessFunction( **int** a\_targetAmount )

{

**if**( a\_targetAmount < 1 || a\_targetAmount > 99 )

{

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

"Change amount must be between 1 and 99 cents." );

}

m\_targetAmount = a\_targetAmount;

}

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

{

**int** changeAmount = *amountOfChange*( a\_subject );

**int** totalCoins = *getTotalNumberOfCoins*( a\_subject );

**int** changeDifference = Math.*abs*( m\_targetAmount - changeAmount );

**double** fitness = ( 99 - changeDifference );

**if**( changeAmount == m\_targetAmount )

{

fitness += 100 - ( 10 \* totalCoins );

}

**return** fitness;

}

**public** **static** **int** amountOfChange( IChromosome a\_potentialSolution )

{

**int** numQuarters = *getNumberOfCoinsAtGene*( a\_potentialSolution, 0 );

**int** numDimes = *getNumberOfCoinsAtGene*( a\_potentialSolution, 1 );

**int** numNickels = *getNumberOfCoinsAtGene*( a\_potentialSolution, 2 );

**int** numPennies = *getNumberOfCoinsAtGene*( a\_potentialSolution, 3 );

**return** ( numQuarters \* 25 ) + ( numDimes \* 10 ) + ( numNickels \* 5 ) +

numPennies;

}

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

**int** a\_position )

{

Integer numCoins =

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

**return** numCoins.intValue();

}

**public** **static** **int** getTotalNumberOfCoins( IChromosome a\_potentialsolution )

{

**int** totalCoins = 0;

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

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

{

totalCoins += *getNumberOfCoinsAtGene*( a\_potentialsolution, i );

}

**return** totalCoins;

}

}

Для початку, визначаємо власний клас і наслідуємо клас org.jgap.FitnessFunction. Всі фітнес-функції повинні наслідувати клас FitnessFunction. Потім визначаємо конструктор і метод оцінки. Метод оцінки evaluate() є стандартним методом, що всі фітнес функції повинні виконувати. Тобто метод, який буде викликатися генетичним двигуном, коли він повинен знайти, фітнес значення хромосоми.

Конструктор є не дуже цікавим: він просто приймає величину зміни цільового значення, що бажає користувач, перевіряє, чи сума відповідає обмеженню від 1 до 99 центів, а потім зберігає введену суму в змінному екземплярі для подальшого використання.

Цікава частина цілого класу є метод оцінки evaluate(), в якому і відбувається основна робота. Метод оцінки завжди передається в хромосомі, яка представляє потенційне рішення. Хромосома складається з генів, кожен з яких представляє собою відповідну частину рішення. У прикладі, Хромосома являє собою суму монет, в той час як гени являють собою конкретні види монет: 25 копійок для першого гена, 10 для другого, 5 для третього, і 1 для четвертого.

Зрештою, ми хочемо повернути високі значення придатності для рішень, які відповідають цільовій сумі з дуже малою кількістю монет, і повернути більш низькі значення придатності для рішень, які знаходяться далеко очікуваної суми або представляють велику кількість монет.

## Крок 3. Налаштування конфігурації об’єкта

JGAP призначений бути дуже гнучким і легко налаштовуваним. Він дозволяє створювати свої власні генетичні оператори, генератори випадкових чисел, натуральні селектори і так далі. Для того, щоб підтримати все це, JGAP використовує об'єкт конфігурації, який повинен бути встановлений з усіма параметрами, які користувач хоче застосовувати. Для зручності наявний стандартний клас DefaultConfiguration, який поставляється вже налаштований з найбільш поширеними параметрами. Потрібно лише надати три додаткові частини інформації: які фітнес функції використовувати, як Хромосоми мають бути налаштовані, і скільки Хромосом має бути у популяції. Розглянемо приклад:

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

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

**int** targetAmount = Integer.parseInt(args[0]);

FitnessFunction myFunc =

**new** MinimizingMakeChangeFitnessFunction( targetAmount );

conf.setFitnessFunction( myFunc );

Gene[] sampleGenes = **new** Gene[ 4 ];

sampleGenes[0] = **new** IntegerGene(conf, 0, 3 ); // Quarters

sampleGenes[1] = **new** IntegerGene(conf, 0, 2 ); // Dimes

sampleGenes[2] = **new** IntegerGene(conf, 0, 1 ); // Nickels

sampleGenes[3] = **new** IntegerGene(conf, 0, 4 ); // Pennies

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

conf.setSampleChromosome( sampleChromosome );

conf.setPopulationSize( 500 );

}

Для початку створюється об’єкт класу Configuration як DefaultConfiguration, тобто конфігурація по замовчуванню. Тоді оголошується фітнес функція, розроблена раніше, з вхідним параметром – сумою грошей.

Після цього за допомогою метода setFitnessFunction() оголошена фітнес функція підключається до конфігурації. Тоді оголошується масив генів, який є одним з вхідних параметрів Хромосоми, що оголошується після.

Останнім кроком налаштування є додавання до конфігурації оголошеної Хромосоми та розміру популяції.

## Крок 4. Створення популяції

Популяція хромосом називається генотипом, оголошення об’єкта цього класу і є наступним пунктом налаштування ГА. Можна побудувати кожну хромосому окремо, а потім створити з них генотип, так як створювалась хромосома з генів у кроці 3. Але в JGAP є простіший та легший спосіб для створення випадкової популяції:

Genotype population = Genotype.randomInitialGenotype( conf );

Метод randomInitialGenotype () приймає об’єкт конфігурації (який був налаштований на кроці 3) і повертає генотип з правильною кількістю хромосом, кожна з яких має свої випадкові гени. Іншими словами, він генерує випадкову популяцію. Для більшості програм, це все, що необхідно для створення початкової популяції потенційних рішень.

## Крок 5. Еволюція

Наступним кроком після всіх налаштувань є еволюція. Найпростішим способом зробити один цикл еволюції є:

population.evolve();

Для визначення найкращої хромосоми з популяції застосовується метод:

IChromosome bestSolutionSoFar = population.getFittestChromosome();

Якщо отримане рішення не влаштовує можна провести еволюцію знову та знову, або задати кількість еволюцій та провести їх у циклі.

IChromosome bestSolutionSoFar;

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

{

population.evolve();

}

System.out.println( "The best solution contained the following: " );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 0 ) + " quarters." );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 1 ) + " dimes." );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 2 ) + " nickels." );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 3 ) + " pennies." );

System.out.println( "For a total of " +

MinimizingMakeChangeFitnessFunction.amountOfChange(

bestSolutionSoFar ) + " cents in " +

MinimizingMakeChangeFitnessFunction.getTotalNumberOfCoins(

bestSolutionSoFar ) + " coins." );

# Стандартні налаштування

Частота мутацій. По замочуванню частота мутацій встановлюється динамічно, в середньому один раз на 10 хромосом. Але її також можна задати і вручну використавши клас MutationOperator.

Тип селекції. За замовчуванням в якості селектора використовується клас WeightedRouleteSelector, тобто метод рулетки. Існує також можливість створення власної селекції.

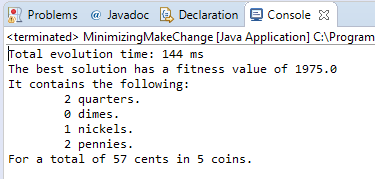
Тип кросоверу. Відсоток кількості кросоверів рівний 35%. Тобто (РозмірПопуляції\*0,35) операцій кросоверу на популяцію. Після кожного кросоверу отримуємо 2 особини. Точка кросоверу є випадковою.

# Приклади

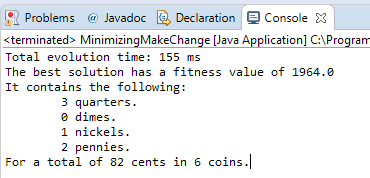
## Приклад 1

Результат роботи зібраної вище програми для пошуку набору найменшої кількості монет певної заданої суми.

Сума 57 центів:



Сума 82 центи:



## Приклад 2

Програма, що вирішує задачу Комівояжера (TravellingSalesman.java):

Задаємо кількість міст та їх координати

**public** **static** **final** **int** ***CITIES*** = 7;

**public** **static** **final** **int**[][] ***CITYARRAY*** = **new** **int**[][] { {2, 4}, {7, 5}, {7, 11},

{8, 1}, {1, 6}, {5, 9}, {0, 11}

};

Створюємо хромосому:

**public** IChromosome createSampleChromosome(Object a\_initial\_data) {

**try** {

Gene[] genes = **new** Gene[***CITIES***];

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

genes[i] = **new** IntegerGene(getConfiguration(), 0, ***CITIES*** - 1);

genes[i].setAllele(**new** Integer(i));

}

IChromosome sample = **new** Chromosome(getConfiguration(), genes);

**return** sample;

}

**catch** (InvalidConfigurationException iex) {

**throw** **new** IllegalStateException(iex.getMessage());

}

}

Відстань між містами:

**public** **double** distance(Gene a\_from, Gene a\_to) {

IntegerGene geneA = (IntegerGene) a\_from;

IntegerGene geneB = (IntegerGene) a\_to;

**int** a = geneA.intValue();

**int** b = geneB.intValue();

**int** x1 = ***CITYARRAY***[a][0];

**int** y1 = ***CITYARRAY***[a][1];

**int** x2 = ***CITYARRAY***[b][0];

**int** y2 = ***CITYARRAY***[b][1];

**return** Math.*sqrt*( (x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2));

}

Запускаємо програму:

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

**try** {

TravellingSalesman t = **new** TravellingSalesman();

IChromosome optimal = t.findOptimalPath(**null**);

System.***out***.println("Solution: ");

System.***out***.println(optimal);

System.***out***.println("Score " +

(Integer.***MAX\_VALUE*** / 2 - optimal.getFitnessValue()));

}

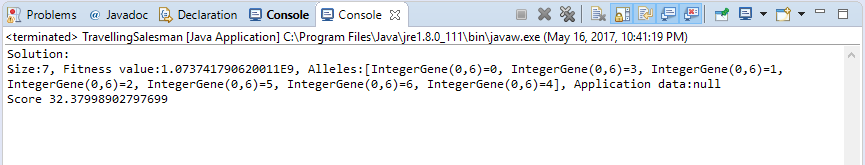
**catch** (Exception ex) {

ex.printStackTrace();

}

}

Результат роботи:



## Приклад 3

Програма пошуку максимального натурального числа, що кодується заданою кількістю бітів.

Фітнес функція пошуку максимуму:

**public** **class** MaxFunction

**extends** FitnessFunction {

**private** **final** **static** String ***CVS\_REVISION*** = "$Revision: 1.6 $";

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

**int** total = 0;

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

BooleanGene value = (BooleanGene) a\_subject.getGene(a\_subject.size() -

(i + 1));

**if** (value.booleanValue()) {

total += Math.*pow*(2.0, (**double**) i);

}

}

**return** total;

}

}

Налаштування ГА та запуск:

**public** **class** SimpleExample {

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

**private** **static** **final** String ***CVS\_REVISION*** = "$Revision: 1.9 $";

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

**int** numEvolutions = 500;

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

gaConf.setPreservFittestIndividual(**true**);

gaConf.setKeepPopulationSizeConstant(**false**);

Genotype genotype = **null**;

**int** chromeSize=16;

**double** maxFitness = Math.*pow*(2.0, (**double**) chromeSize) - 1;

**try** {

IChromosome sampleChromosome = **new** Chromosome(gaConf,

**new** BooleanGene(gaConf), chromeSize);

gaConf.setSampleChromosome(sampleChromosome);

gaConf.setPopulationSize(20);

gaConf.setFitnessFunction(**new** MaxFunction());

genotype = Genotype.*randomInitialGenotype*(gaConf);

}

**catch** (InvalidConfigurationException e) {

e.printStackTrace();

System.*exit*( -2);

}

**int** progress = 0;

**int** percentEvolution = numEvolutions / 100;

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

genotype.evolve();

// Print progress.

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

**if** (percentEvolution > 0 && i % percentEvolution == 0) {

progress++;

IChromosome fittest = genotype.getFittestChromosome();

**double** fitness = fittest.getFitnessValue();

System.***out***.println("Currently fittest Chromosome has fitness " +

fitness);

**if** (fitness >= maxFitness) {

**break**;

}

}

}

// Print summary.

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

IChromosome fittest = genotype.getFittestChromosome();

System.***out***.println("Fittest Chromosome has fitness " +

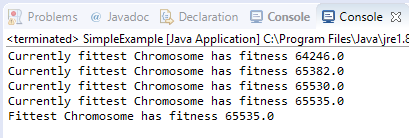
fittest.getFitnessValue());

}

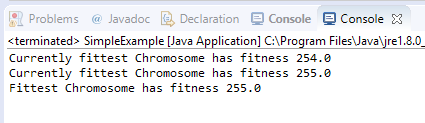
}

Результат роботи:

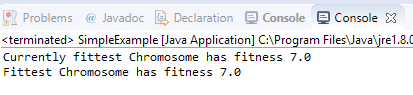
16 біт:



8 біт:

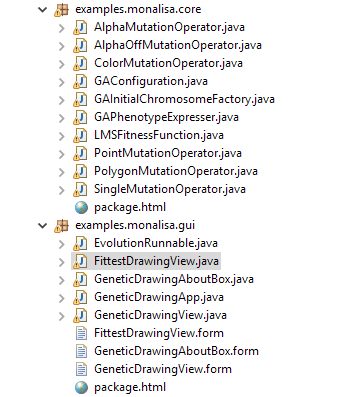


3 біти:



## Приклад 4

Приклад промальовування зображення за допомогою ГА. Дерево проекту:



Налаштування ГА:

**public** **class** GAConfiguration

**extends** Configuration **implements** ICloneable {

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

**private** **final** **static** String ***CVS\_REVISION*** = "$Revision: 1.3 $";

**final** **private** GAPhenotypeExpresser m\_phenotypeExpresser;

**final** **private** BufferedImage m\_target;

**final** **private** **int** m\_maxPolygons;

**public** GAConfiguration(BufferedImage a\_target, **int** a\_maxPolygons)

**throws** InvalidConfigurationException {

**super**();

m\_maxPolygons = a\_maxPolygons;

m\_target = a\_target;

m\_phenotypeExpresser = **new** GAPhenotypeExpresser(**this**);

setBreeder(**new** GABreeder());

setRandomGenerator(**new** StockRandomGenerator());

setEventManager(**new** EventManager());

BestChromosomesSelector bestChromsSelector = **new** BestChromosomesSelector(

**this**, 0.50d);

bestChromsSelector.setDoubletteChromosomesAllowed(**true**);

addNaturalSelector(bestChromsSelector, !**false**);

setMinimumPopSizePercent(0);

setPreservFittestIndividual(**true**);

setPopulationSize(5);

setKeepPopulationSizeConstant(**false**);

setFitnessEvaluator(**new** DeltaFitnessEvaluator());

setFitnessFunction(**new** LMSFitnessFunction(**this**));

setChromosomePool(**new** ChromosomePool());

/\* addGeneticOperator(new CrossoverOperator(this, 0.35d));\*/

addGeneticOperator(**new** SingleMutationOperator(**this**, 2));

addGeneticOperator(**new** PointMutationOperator(**this**, 8));

addGeneticOperator(**new** ColorMutationOperator(**this**, 10));

//addGeneticOperator(new AlphaMutationOperator(this, 200));

//addGeneticOperator(new AlphaOffMutationOperator(this, 150));

addGeneticOperator(**new** PolygonMutationOperator(**this**, 5));

setSampleChromosome(GAInitialChromosomeFactory.*create*(**this**));

}

Результат роботи після 3500 генерацій:

