|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САП** | | Тема | оцінка | підпис |
| КНC-13 | РГР | Фреймворк Opt4J |  |  |
| Абрам Б.Я. | |
| № залікової: 1608401 | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р. З. | |

**Загальний опис фреймворку**

Opt4J – фреймворк з відкритим вихідним кодом на основі Java для еволюційних обчислень. Він містить набір алгоритмів (багатоцільової) оптимізації, таких як еволюційні алгоритми (в тому числі SPEA2 і NSGA2), диференціальної еволюції, оптимізації рою частинок і імітації відпалу.

Метою Opt4J є спрощення еволюційної оптимізації користувальницьких задач, а також реалізації довільних алгоритмів мета-евристичної оптимізації. Для цієї мети Opt4J спирається на реалізацію модульної основи і надає графічний користувальницький інтерфейс для конфігурації, а також візуалізації процесу оптимізації.

**Можливості фреймворку**

* *Розробка*

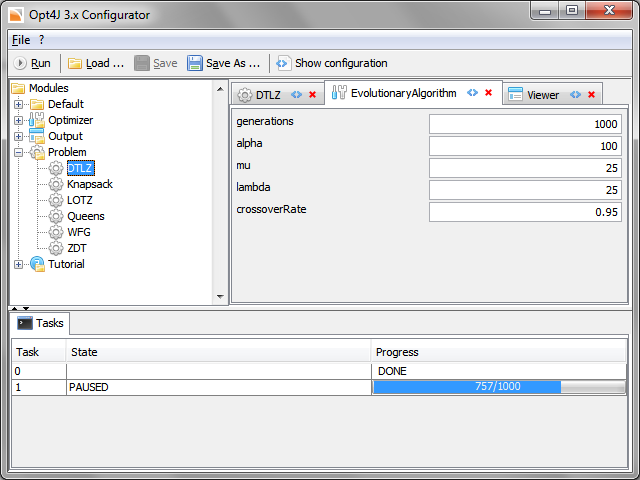
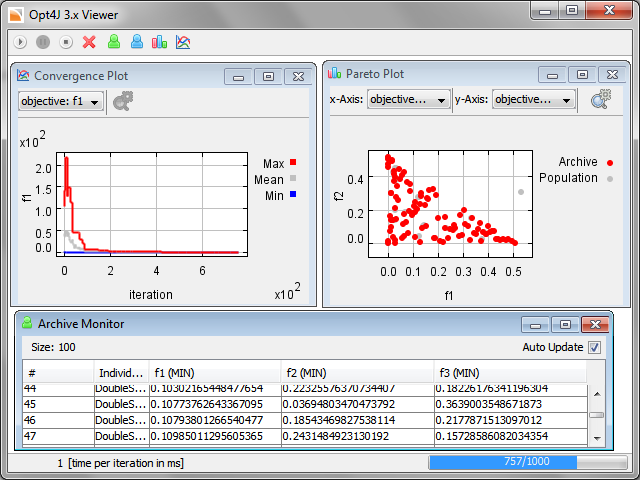
Мета Opt4J - зробити розробку і використання користувальницьких завдань і нових алгоритмів оптимізації якомога простішим. Для цієї мети підхід інтерфейсу на основі, що використовується, дозволяє модульний розвиток. В результаті код є чистим Java і без будь-яких текстових файлів конфігурації або шаблонного коду.

* *Оптимізатори і орієнтири*

Opt4J містить великий набір алгоритмів (багатоцільової) оптимізації, таких як еволюційні алгоритми (в тому числі і SPEA2 NSGA2), диференціальної еволюції, оптимізація рою частинок і імітації відпалу.

* *Графічний інтерфейс користувача*

Opt4J містить графічні інтерфейси (GUI) для зміни властивостей оптимізації, а також для візуалізації процесу оптимізації. GUI конфігурації дозволяють зберігати і завантажувати конфігурацій в файл XML, а також безпосередній запуск і моніторинг завдань оптимізації. GUI містить панель візуалізації для ілюстрації процесу оптимізації. Приклади GUI Opt4J показано на рис.1. Використання графічного інтерфейсу користувача для завдань або алгоритмів оптимізації не потребує будь-якого додаткового коду оскільки модулі встановлені автоматично.



**Рис.1 Приклади інтерфейсу користувача фреймоврку Opt4J**

* *Відкритий вихідний код*

Фреймворк є відкритим вихідним кодом випущеним під ліцензією MIT, що дозволяє без обмежень використовувати, копіювати, змінювати, об'єднувати, публікувати, поширювати, субліцензувати і / або продавати копії програмного забезпечення.

**Встановлення фреймворку**

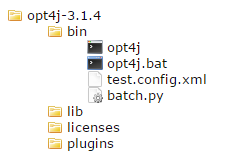
Для нормальної роботи Opt4J вимагає Java 6, тому знадобиться встановлений JDK, який можна завантажити по посиланню:

[*http://www.oracle.com/technetwork/java/javase/downloads/jdk6u37-downloads-1859587.html*](http://www.oracle.com/technetwork/java/javase/downloads/jdk6u37-downloads-1859587.html)*.*

Opt4J поширюється як єдиний .zip файл, який можна завантажити по посиланню:

[*https://sourceforge.net/projects/opt4j/*](https://sourceforge.net/projects/opt4j/)

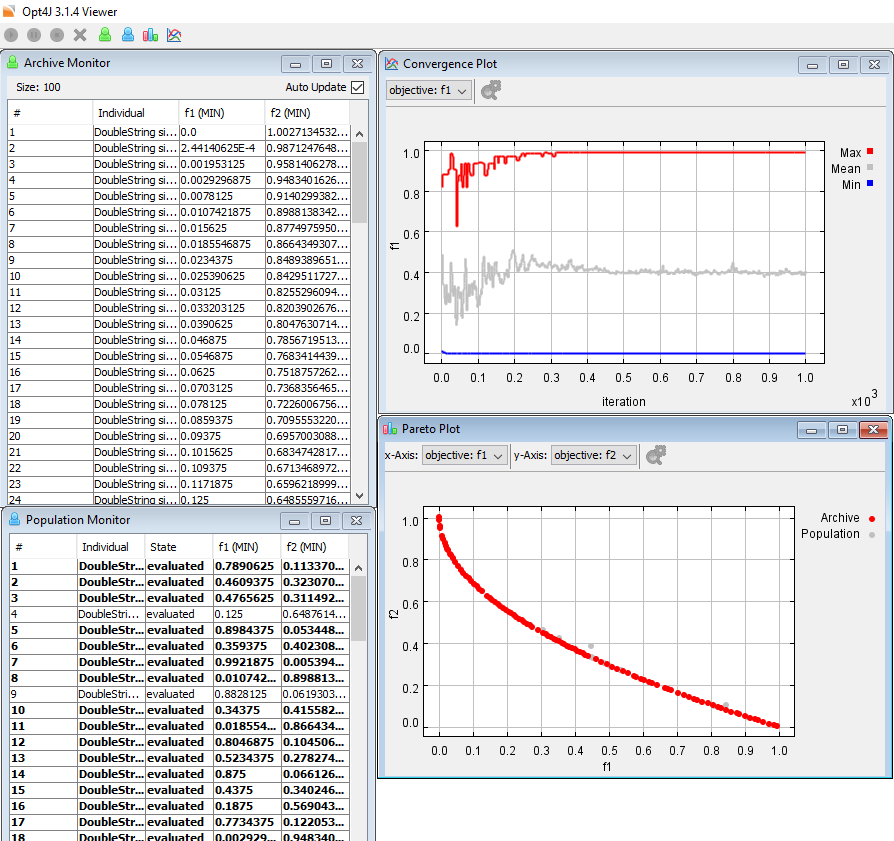
Після вилучення файлів, структура папок виглядає наступним чином:

****

**Рис.2 Структура папок після видобування**

Фреймворк запускається з opt4j.bat під Windows, і під Linux opt4j / Unix / OSX, відповідно. Після його запуску, з'явиться модуль конфигуратор, який дозволяє вибрати оптимізатора і проблеми, а також налаштування їх параметрів. Для тестування необхідно завантажити test.config.xml і почати оптимізацію кнопкою «Run».

Для розробки призначених для користувача завдань оптимізації та алгоритмів Opt4J, файли в папці Lib повинні бути на шляху до класів для компіляції і запуску коду. Для запуску модуля конфігуратора вручну, необхідно запустити клас Opt4J, який містить відповідний основний метод.

****

**Рис.3 Запуск тестової конфігурації**

**Приклади**

### Hello World

Мета полягає в тому, щоб оптимізувати рядок з одинадцяти символів для оптимального рішення «hello world». Для цього необхідно визначити відповідного «творця», «декодера», і «оцінювача».

На першому етапі ми визначаємо творця, який неявно визначає генотип:

# public class HelloWorldCreator implements Creator<SelectGenotype<Character>> {

# Character[] ALPHABET = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', ' ' };

# Random random = new Random();

# public SelectGenotype<Character> create() {

# SelectGenotype<Character> genotype = new SelectGenotype<Character>(ALPHABET);

# genotype.init(random, 11);

# return genotype;

# }

# }

Як уже зазначалося, завдання творця генерувати випадкові об'єкти генотипу. У нашому випадку, ми хочемо, щоб він генерував випадкові рядки з одинадцятьма символами. Тут ми вибираємо один з визначених класів генотипу, який дозволяє вибрати його з заданого алфавіту і форматувати у випадковому порядку з одинадцятьма символами.

На другому етапі ми визначаємо декодер, який неявно визначає фенотип:

# public class HelloWorldDecoder implements Decoder<SelectGenotype<Character>, String> {

# public String decode(SelectGenotype<Character> genotype) {

# String phenotype = "";

# for (int i = 0; i < genotype.size(); i++) {

# phenotype += genotype.getValue(i);

# }

# return phenotype;

# }

# }

Завдання декодера полягає в перетворенні нашого генотипу в фенотип, який в нашому випадку є рядком. Це робиться просто шляхом копіювання символів в рядку.

На третьому етапі ми визначаємо оцінювач:

# public class HelloWorldEvaluator implements Evaluator<String> {

# @Override

# public Objectives evaluate(String phenotype) {

# int value = 0;

# for (int i = 0; i < phenotype.length(); i++) {

# value += (phenotype.charAt(i) == "HELLO WORLD".charAt(i)) ? 1 : 0;

# }

# Objectives objectives = new Objectives();

# objectives.add("objective", Sign.MAX, value);

# return objectives;

# }

# }

Завданням оцінювача є визначення якості одного фенотипу, тобто, одного рядка. У нашому випадку, ми шукаємо кількість співпадаючих символів, коли його порівнюють з «HELLO WORLD».

Наступним етапом визначаємо модуль проблеми:

public class HelloWorldModule extends ProblemModule {

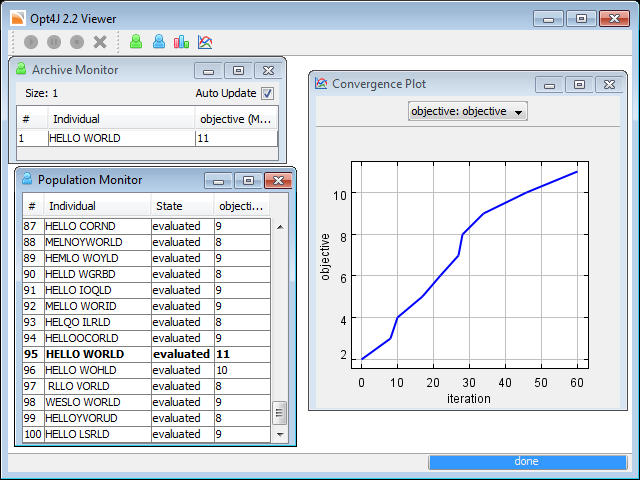
protected void config() {

bindProblem(HelloWorldCreator.class, HelloWorldDecoder.class, HelloWorldEvaluator.class);

}

}

Модуль проблеми визначає проблему, відповідного твореця, декодера і оцінювача. На рис4. показано результат виконання даного завдання.

****

### Рис.4 Вирішення завдання «Hello World» за допомогою фреймворку Opt4J

### Traveling Salesman

Завдання комівояжера є класичною задачею оптимізації. Мета полягає в тому, щоб мінімізувати довжину шляху, який необхідний, щоб відвідати безліч міст.

Першим етапом є визначення екземпляра проблеми:

public class SalesmanProblem {

protected Set<City> cities = new HashSet<City>();

public class City {

protected final double x;

protected final double y;

public City(double x, double y) {

this.x = x;

this.y = y;

}

public double getX() {

return x;

}

public double getY() {

return y;

}

}

@Inject

public SalesmanProblem(@Constant(value = "size") int size) {

Random random = new Random(0);

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

final double x = random.nextDouble() \* 100;

final double y = random.nextDouble() \* 100;

final City city = new City(x, y);

cities.add(city);

}

}

public Set<City> getCities() {

return cities;

}

}

Спочатку визначимо міста, які задаються х і у координатами. У конструкторі нашої проблеми ми випадковим чином визначаємо певну кількість міст, які можуть бути згодом отримані. Тут, конструктор з анотацією @Inject, який використовується Guice вказує, який конструктор буде використовуватися при створенні екземпляра. Аргумент розміру в конструкторі позначається як константа, яка пізніше буде встановлена в модулі проблеми.

Фенотип виглядає наступним чином:

public class SalesmanRoute extends ArrayList<City> {

}

Як і в попередньому прикладні необхідно визначити творця, декодер, і оцінювач. Визначаємо творця:

public class SalesmanCreator implements Creator<PermutationGenotype<City>> {

protected final SalesmanProblem problem;

@Inject

public SalesmanCreator(SalesmanProblem problem) {

this.problem = problem;

}

public PermutationGenotype<City> create() {

PermutationGenotype<City> genotype = new PermutationGenotype<City>();

for (City city : problem.getCities()) {

genotype.add(city);

}

Collections.shuffle(genotype);

return genotype;

}

}

Творець отримує екземпляр проблеми в конструкторі і зберігає його у змінній. Для генотипу ми використовуємо зумовлені перестановки генотипу і додаємо всі міста. Перестановка визначає порядок, в якому міста будуть відвідуються комівояжером. Кожен генотип перемішується випадковим чином, перш ніж він буде повернутий. Це має важливе значення, так як кожен генотип має спочатку бути згенерований абсолютно випадково.

На наступному кроці ми визначаємо декодер:

public class SalesmanDecoder implements Decoder<PermutationGenotype<City>, SalesmanRoute> {

public SalesmanRoute decode(PermutationGenotype<City> genotype) {

SalesmanRoute salesmanRoute = new SalesmanRoute();

for (City city : genotype) {

salesmanRoute.add(city);

}

return salesmanRoute;

}

}

Оцінювач виглядає наступним чином:

public class SalesmanEvaluator implements Evaluator<SalesmanRoute> {

public Objectives evaluate(SalesmanRoute salesmanRoute) {

double dist = 0;

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

City one = salesmanRoute.get(i);

City two = salesmanRoute.get((i + 1) % salesmanRoute.size());

dist += getEuclideanDistance(one, two);

}

Objectives objectives = new Objectives();

objectives.add("distance", Sign.MIN, dist);

return objectives;

}

private double getEuclideanDistance(City one, City two) {

final double x = one.getX() - two.getX();

final double y = one.getY() - two.getY();

return Math.sqrt(x \* x + y \* y);

}

}

Оцінювач обчислює довжину поточного маршруту і повертає його в якості мети, яка повинна бути зведена до мінімуму.

визначаємо модуль:

public class SalesmanModule extends ProblemModule {

@Constant(value = "size")

protected int size = 100;

public int getSize() {

return size;

}

public void setSize(int size) {

this.size = size;

}

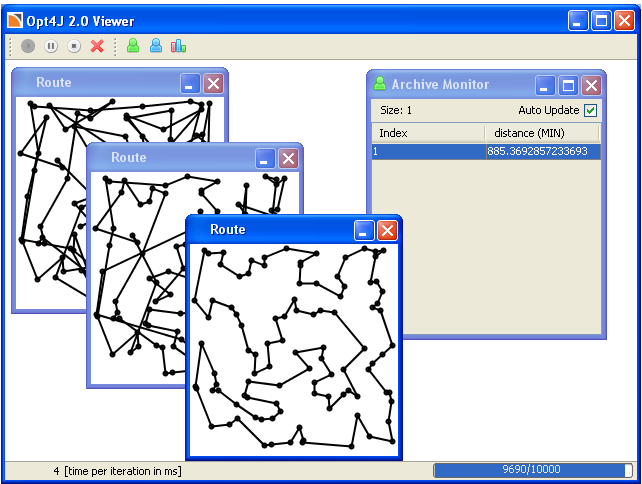
public void config() {

bindProblem(SalesmanCreator.class, SalesmanDecoder.class, SalesmanEvaluator.class);

}

}

На рис5. показано результат виконання даного завдання.

****

### Рис.5 Вирішення завдання «Travelling salesman» за допомогою фреймворку Opt4J

## Defining Optimization Algorithms

Цей приклад показує, як написати дуже простий оптимізатор, заснований тільки на мутації. Представлений оптимізатор має розмір популяції 100 і створює 25 особин потомства з 25 батьківських особин кожного покоління за допомогою операції зміни. Оптимізатор реалізується в такий спосіб:

public class MutateOptimizer implements IterativeOptimizer {

protected final IndividualFactory individualFactory;

protected final Mutate<Genotype> mutate;

protected final Copy<Genotype> copy;

protected final Selector selector;

private final Population population;

public static final int POPSIZE = 100;

public static final int OFFSIZE = 25;

@Inject

public MutateOptimizer(Population population, IndividualFactory individualFactory, Selector selector,

Mutate<Genotype> mutate, Copy<Genotype> copy) {

this.individualFactory = individualFactory;

this.mutate = mutate;

this.copy = copy;

this.selector = selector;

this.population = population;

}

public void initialize() throws TerminationException {

for (int i = 0; i < OFFSIZE + POPSIZE; i++) {

population.add(individualFactory.create());

}

}

public void next() throws TerminationException {

Collection<Individual> lames = selector.getLames(OFFSIZE, population);

population.removeAll(lames);

Collection<Individual> parents = selector.getParents(OFFSIZE, population);

for (Individual parent : parents) {

Genotype genotype = copy.copy(parent.getGenotype());

mutate.mutate(genotype, 0.1);

Individual child = individualFactory.create(genotype);

population.add(child);

}

}

}

Ініціалізація створює 100 нових осіб і додає їх до населення. Основна оптимізація відбувається в наступному способі в кожній ітерації. По-перше, 25 осіб з гіршою придатністю ідентифікуються і видаляються з популяції. На наступному етапі, 25 материнських особини визначаються селектором і для кожної з цих особин, одна особина створюються шляхом копіювання генотипу і його мутації.

Модуль оптимізації виглядає наступним чином:

public class MutateOptimizerModule extends OptimizerModule {

@MaxIterations

protected int iterations = 1000;

public int getIterations() {

return iterations;

}

public void setIterations(int iterations) {

this.iterations = iterations;

}

@Override

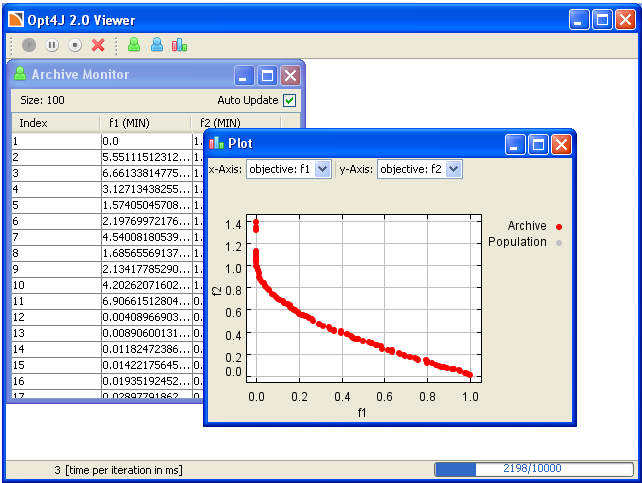
public void config() {

bindIterativeOptimizer(MutateOptimizer.class);

}

}

На рис6. показано результат виконання даного завдання.



## Рис.6 Вирішення завдання «Defining Optimization Algorithms» за допомогою фреймворку Opt4J

**Висновок**

В дані розрахунковій роботі здійснено огляд фреймворку на основі Java для еволюційних обчислень Opt4J. Проведено огляд програмного середовища, описано його основні функції та можливості. Проведений огляд наявних алгоритмів та проблем, які можна вирішувати з допомогою фреймворку. Дана програма є легкою у використанні, її не потрібно встановлювати, достатньо лише розпакувати скачаний архів та запустити відповідний файл на виконання. Фреймворк містить велику кількість різноманітних алгоритмів та задач, також наявні приклади до них. Для демонстрації роботи програми здійснив запуск кількох прикладів.