|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САП** | | Тема | оцінка | підпис |
| КНC-13 | РГР | Фреймворк Opt4J |  |  |
| Ракочий Т.І. | |
|  | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р. З. | |

**Загальний опис фреймворку**

Opt4J - це заснований на Java фреймворк для еволюційних обчислень. Він містить набір (багатоцільових) алгоритмів оптимізації, таких як еволюційні алгоритми (включаючи SPEA2 і NSGA2), диференціальну еволюцію, оптимізацію рою частинок і модельований отжиг. У число еталонних тестів входять ZDT, DTLZ, WFG і проблема рюкзака. Метою Opt4J є спрощення еволюційної оптимізації користувальницьких задач, а також реалізація довільних мета-евристичних алгоритмів оптимізації. Для цього Opt4J спирається на модульну реалізацію і пропонує графічний користувальницький інтерфейс для конфігурації, а також візуалізацію процесу оптимізації.

**Можливості фреймворку**

* *Відмінності фенотип-генотип*

МУ запропонованій структурі поділ генетичного уявлення,Генотип і одне рішення задачі оптимізації,Фенотип. Декодер [21] використовується для трансляції Генотип в фенотип. Поділ дозволяє скласти Генотипу з набору визначених генотипів без повторної реалізації Цих або відповідних операторів для кожного завдання оптимізації, відповідно. Таким чином, тільки залежний від проблем фенотип, а також Необхідно вказати декодер. Один індивід i визначається як 3-кортеж, що складається з генотипу G, фенотип p цілі y. Як правило, цілі

Визначено в R м. Однак рамки не накладають цього Обмеження, щоб дозволити також інші типи цілей, наприклад, розглянути Невизначеність [18]. Набір всіх генотипів визначається як простір пошуку . Безліч всіх фенотипів визначається як простір рішень P.

Сукупність усіх цілей визначається як об'єктивне простір Y. Тут функція Create: ∅ → G генерує випадковий генотип g ∈ G, функція Декодування: G → P переводить генотип g ∈ G в фенотип P ∈ P, і оцінюємо функцію: P → Y визначає цілі Фенотипу. Огляд цієї концепції показаний на рис. 1. Різниця генотипу і фенотипу дозволяє оптимізувати Складних проблем, в яких повинен бути оброблений генотип Стати можливим рішенням. Це справедливо і для алгоритмів відновлення Які використовують локальний метод пошуку для поліпшення отриманих рішень. Подальша перевага поділу генотипу і фенотипу Покращена ремонтопридатність і розширюваність оптимізації роблема. Зокрема, модифікація генетичного уявлення Стає можливим без впливу на функцію оцінки оцінити.

* *Коммпозитні генотипи*

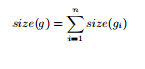
Багато складні завдання оптимізації можуть бути розкладені на підзадачі. В цілому, ці підзадачі не можуть бути оптимізовані окремо За їх взаємної кореляції. Щоб впоратися з такою високою проблемою Складність, композиційний підхід. Тут, генотип, Функція створення і функція декодування можуть бути скомпоновані в Деревоподібна структура. Це значно зменшує зчеплення і покращує Гнучкість, така, що підзадачі задачі оптимізації Можуть бути розроблені і протестовані окремо. На останньому етапі Окремо розроблені частини об'єднані без проблем у всій оптимізації задача.

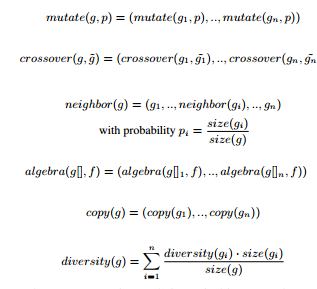
* *Оператори*

Щоб використовувати складний генотип, оператори стають Необхідні для обробки цих складових генотипів. У разі, якщо г не є складовим, диспетчер застосовує відповідна підстава оператор. Диспетчерська функція застосовується до кожного лист композит

Дерево генотипів для визначення конкретного оператора. Надалі кілька стандартних операторів для складових генотипів

коротко:





Оператор розміру в рівнянні потребує многий інші оператори.Тут розмір генотип визначається сума Розміри генотипів листя. ОПЕРАТОРИ мутувати і кросовер У формулі необхідні для еволюційних алгоритмів (EAS). Тут оператор мутації змінює існуючий Генотип з частотою мутацій р. Оператор кросовер створює Новий генотип з двох існуючих генотипів і може бути узагальнено Для обробки довільного числа генотипів. Сусід Оператор в рівнянні. (6) потрібно алгоритми оптимізація, такі як Модельований отжиг (SA) або Табу-пошук (TS). тут оператор Застосовується до однієї дитини складеного генотипу з імовірністю Що визначається відношенням розміру дитини до розмір складеного генотипу. Оператор алгебри в рівнянні (7) представляє собою Використовується оптимізаційні алгоритми, такі як оптимізація еся частки (ПСО) або диференціальної еволюції (DE).

**Встановлення фреймворку**

Для встановлення фреймворку можна скористатися Maven (рис. 1-2).

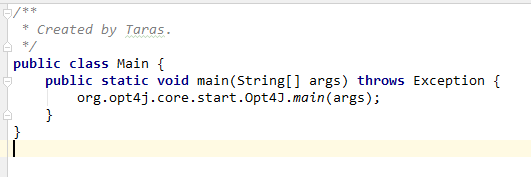


**Рис.1 Структура файлу для ддоавання бібілотеки opt4j**



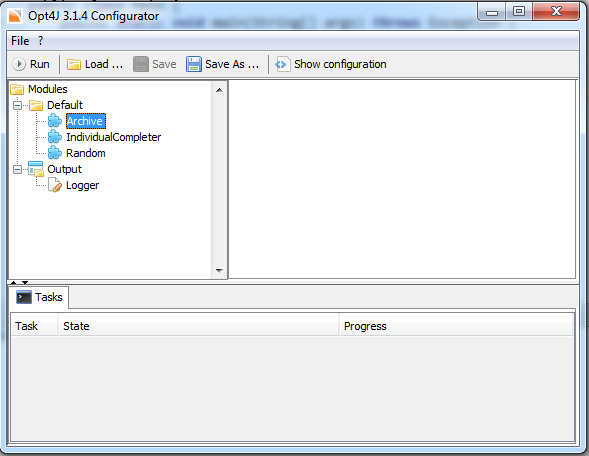
**Рис.2 Maven репозиторій**

Запуск на Windows за допомого opt4j не вдався через помилки, тому фреймворк запускався в IntelijIDE наступним чином:



**Рис.3 Код для запуску фреймворку**

Після його запуску з’являється наступне вікно:



**Рис.4 Конфігуратор**

**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.

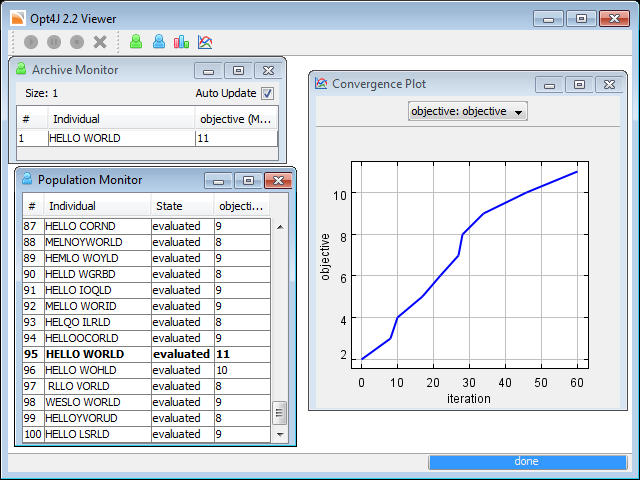
На третьому етапі ми визначаємо оцінювач:

**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"**, 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 творця, декодера і оцінювача. На рис4. показано результат виконання даного завдання.****

### Рис.5 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<SalesmanProblem.City> {  
}

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

**public class** SalesmanCreator **implements** Creator<PermutationGenotype<SalesmanProblem.City>> {  
 **protected final** SalesmanProblem **problem**;  
 @Inject  
 **public** SalesmanCreator(SalesmanProblem problem) {  
 **this**.**problem** = problem;  
 }  
 **public** PermutationGenotype<SalesmanProblem.City> create() {  
 PermutationGenotype<SalesmanProblem.City> genotype = **new** PermutationGenotype<SalesmanProblem.City>();  
 **for** (SalesmanProblem.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<SalesmanProblem.City>, SalesmanRoute> {  
 **public** SalesmanRoute decode(PermutationGenotype<SalesmanProblem.City> genotype) {  
 SalesmanRoute salesmanRoute = **new** SalesmanRoute();  
 **for** (SalesmanProblem.City city : genotype) {  
 salesmanRoute.add(city);  
 }  
 **return** salesmanRoute;  
 }  
}

Оцінювач виглядає наступним чином:

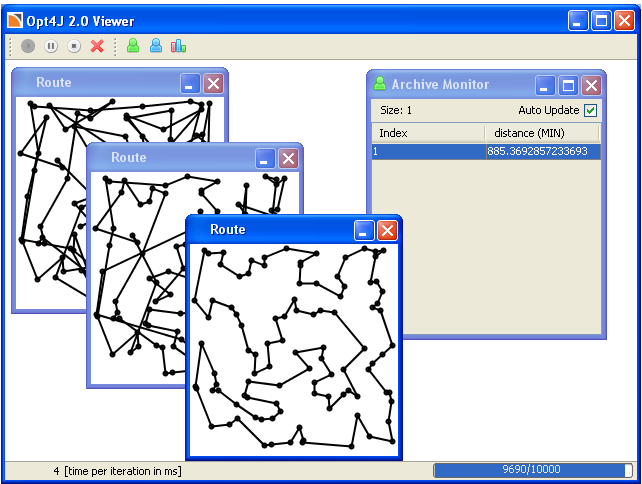
**public class** SalesmanEvaluator **implements** Evaluator<SalesmanRoute> {  
 **public** Objectives evaluate(SalesmanRoute salesmanRoute) {  
 **double** dist = 0;  
 **for** (**int** i = 0; i < salesmanRoute.size(); i++) {  
 SalesmanProblem.City one = salesmanRoute.get(i);  
 SalesmanProblem.City two = salesmanRoute.get((i + 1) % salesmanRoute.size());  
 dist += getEuclideanDistance(one, two);  
 }  
 Objectives objectives = **new** Objectives();  
 objectives.add(**"distance"**, Objective.Sign.***MIN***, dist);  
 **return** objectives;  
 }  
 **private double** getEuclideanDistance(SalesmanProblem.City one, SalesmanProblem.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.6 Вирішення завдання «Travelling salesman» за допомогою фреймворку Opt4J

**Висновок**

В дані розрахунковій роботі здійснено огляд фреймворку Opt4J. Продемонстровано спосіб його встановлення. Присутній огляд фреймворку. Також розглянуті приклади роботи з фреймворком.