|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САП** | | Тема | оцінка | підпис |
| СПКм-12 | ргр | The Watchmaker Framework |  |  |
| Левицька Х.В. | |
| № залікової: 1508519 | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р.З. | |

**Мета роботи:**

Ознайомитись з особливостями та можливостями The Watchmaker Framework. Навчитись використовувати даний фреймворк, визначити процес встановлення (підключення) заданого фреймворка.

**ОПИС THE WATCHMAKER FRAMEWORK**

The Watchmaker Framework є розширеним, високої продуктивності, об'єктно-орієнтованим середовищем для реалізації незалежних від платформи еволюційних / генетичних алгоритмів у Java. Фреймворк забезпечує типову еволюцію для довільних типів за допомогою неінвазивного API. The Watchmaker Framework є відкритим вихідним кодом, безкоштовний для скачування та використання при дотриманні умов Apache Software Licence, версії 2.0.

Центральним об'єктом еволюційної програми, побудованої з The Watchmaker Framework є *evolution engine*.

Інфраструктура забезпечує кілька реалізацій інтерфейсу *EvolutionEngine*, але той, який зазвичай використовується – *GenerationalEvolutionEngine*. Це реалізація загального призначення контуру еволюційного алгоритму.

*EvolutionEngine* має один параметр універсального типу, який вказує тип об'єкта, який він може еволюціонувати. Код, який створює двигун і може еволюціонувати рядки буде виглядати приблизно так:

EvolutionEngine<String> engine

= **new** GenerationalEvolutionEngine<String>(candidateFactory,

evolutionaryOperator,

fitnessEvaluator,

selectionStrategy,

rng);

Після того як створили *EvolutionEngine*, програма буде настільки проста, як виклик методу *evolve* відповідними аргументами. Проте, як видно з фрагмента коду вище, в першу чергу потрібно багато попрацювати над створенням *EvolutionEngine*, який налаштований відповідним чином для даної задачі. Для конструктора класу *GenerationalEvolutionEngine* потрібно п'ять об'єктів. До них відносяться:

* Candidate Factory;
* Еволюційний Оператор;
* Fitness Оцінювач;
* Стратегія Вибору;
* Генератор Випадкових Чисел.

Модуль The Watchmaker Swing включає в себе загальний компонент *EvolutionMonitor*. Він може бути доданий до будь-якого *EvolutionEngine*, щоб забезпечити зворотний зв'язок про хід еволюції. Інформація, яка відображається на моніторі включає в себе візуальне уявлення найсильнішого кандидата і графік пристосованості популяції з плином часу. *EvolutionMonitor* додають до *EvolutionEngine* через метод *addEvolutionObserver*.

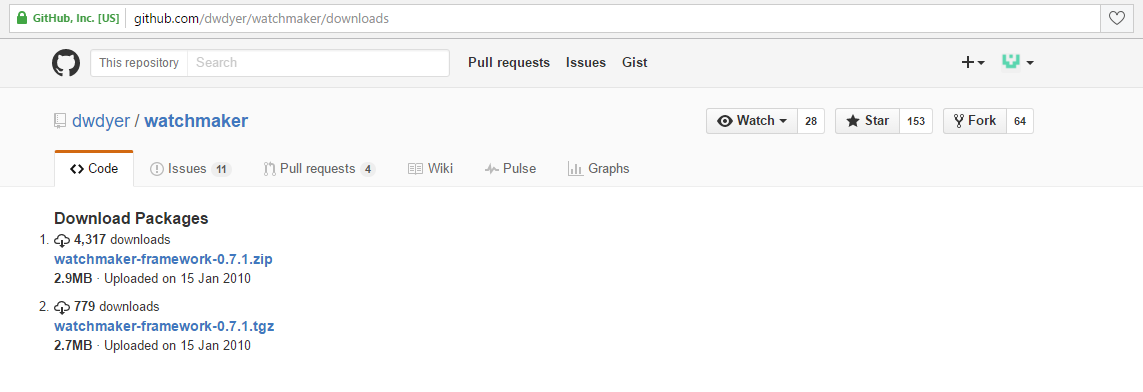
**ОСОБЛИВОСТІ THE WATCHMAKER FRAMEWORK**

* **Багатопоточна еволюція двигуна** **(*Multi-Threaded Evolution Engine*)** - використовує переваги паралелізму для підвищення продуктивності на багатоядерних і багатопроцесорних машинах.
* **Неінвазивність** (***Non-Invasive***) - об'єкти будь-якого типу можуть бути розвинені без Evolvable класу, що має реалізовувати певний інтерфейс або походити від загального базового класу. Це означає, що не існує ніяких обмежень для реалізації типу Evolvable і ніяких залежностей від будь-яких рамкових класів. Типи Evolvable повністю відокремлені.
* **Змінні стратегії вибору** **(*Pluggable Selection Strategies*)**  - надаються: вибір рулетки, вибір турніру і кілька інших стратегій вибору. В якості альтернативи, можна реалізувати свою власну стратегію вибору швидко і легко.
* **Гнучкі схеми еволюції** (***Flexible Evolution Schemes***) - процес еволюції може бути простим або складним. Один крок або кілька операторів об'єднані в послідовності і / або з розгалуженням. Можна використовувати надані оператори, реалізовувати свій власний, або використовувати комбінацію обох.
* **Багаторазові оператори для загальних типів** (***Re-usable Operators for Common Types***) - передбачені готовий до використання перехресного типу і реалізації мутації для декількох типів даних, у тому числі бітових рядків, символьних рядків, масивів і списків.
* **Island Model Evolution** (нове у версії 0.7.0) - розвиток кількох популяцій паралельно з періодичною міграцією особин між островами.
* **Steady-State Evolution** (нове у версії 0.7.0) - розвиток одного члена населення в часі.
* **Еволюційні стратегії** (***Evolution Strategies***) (нове у версії 0.7.1) – підтримка обох стратегії еволюції: (М + А) і (μ, λ).
* **Інтерактивні еволюційні алгоритми** (***Interactive Evolutionary Algorithms***) - підтримка вибору управління користувачем робить структуру такою, яка підходить для додатків, в яких послуга Applications визначенна адекватною функцією придатності.
* **Розподілена обробка** (***Distributed Processing***) - підтримка розподілених оцінок придатності з використанням або Hadoop (через проект Apache Mahout) або Terracotta.
* **Бібліотека компонентів Swing** (***Swing Component Library***) – повторне використання компонентів, щоб спростити побудову графічних користувацьких інтерфейсів для еволюційних програм. Включає в себе загальний компонент Evolution Monitor, який надає інформацію про запущену еволюційну програму.

**ПРОЦЕС ВСТАНОВЛЕННЯ (ПІДКЛЮЧЕННЯ) ФРЕЙМВОРКА**

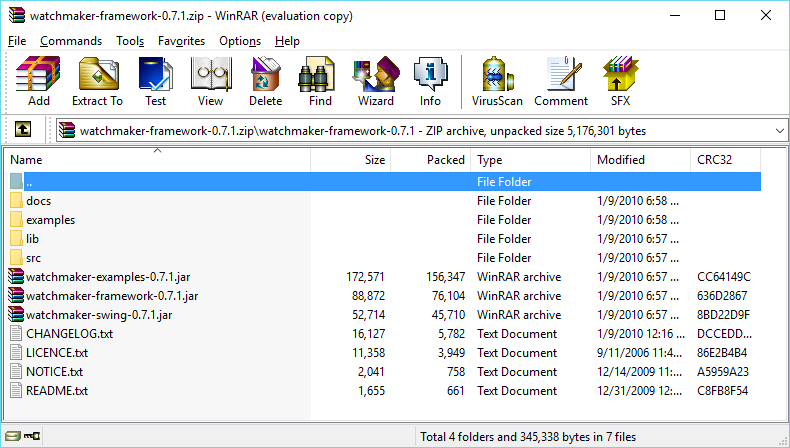
***Поточний реліз***

Останній стабільний реліз The Watchmaker Framework доступний зі сторінки завантаження на GitHub (рис.1).



*Рис.1. Завантаження The Watchmaker Framework зі сторінки на GitHub.*

Завантаження доступне як архів ZIP або TGZ. Вміст завантаженого архіву зображено на рис.2. Вміст однаковий для кожного файлу і включає в себе API документацію і вихідний код для декількох прикладів програм.



*Рис.2. Вміст завантаженого архіву The Watchmaker Framework.*

The Watchmaker Framework вимагає Java 5 або пізнішої версії.

***Архівні релізи***

Можна завантажити всі попередні версії The Watchmaker Framework зі сторінки завантаження GitHub. Необхідно звернутися до журналу змін, щоб дізнатися відмінності між версіями.

***Maven Користувачі***

Останні версії The Watchmaker Framework доступні з репозиторію Java.net Maven 2.

***Вихідний код***

Останній вихідний код доступний з репозиторію Git на GitHub. Якщо потрібно представити зміни, розгалузити репозиторій Git, необхідно внести зміни, а потім відправити запит *pull request*, щоб внесені зміни відносились до об'єднання в головний репозиторій.

Якщо користувач вже знайомий з еволюційними алгоритмами, рекомендується завантажити останню версію The Watchmaker Framework. Якщо користувач являється початківцем, потрібно почати з розділу однієї з інструкції або звернутися до ресурсів, перерахованих в розділі "Background Reading"  на сайті http://watchmaker.uncommons.org.

Основним компонентом The Watchmaker Framework є *EvolutionEngine*. Для того, щоб написати еволюційну програму за допомогою The Watchmaker Framework потрібно створити екземпляр *EvolutionEngine* і викликати одну зі своїх *evolve* або *evolvePopulation* методів.

Для того, щоб створити *еvolution еngine* потрібно надати кілька інших об'єктів. Єдиний клас, який завжди потрібний, є реалізація інтерфейсу *FitnessEvaluator*. Це відноситься тільки до проблеми, яку намагаєтеся вирішити користувач.

Користувач також потребуватиме *CandidateFactory* і один або більше *EvolutionaryOperators* (тобто мутації і / або перехресний тип). Залежно від типу об'єкта, який еволюціонується, можна знайти необхідні класи вже включені в рамки. Якщо їх немає, то можна просто написати необхідні реалізації для будь-якого типу, який вибере користувач. І, нарешті, потрібно вибрати одну з включених стратегій відбору і генератор випадкових чисел (ГВЧ). Можна використовувати стандартний java.util.Random ГВЧ, але є більш швидкі та більш випадкові альтернативи. The Watchmaker Framework підтримує бібліотеку [*Uncommons Maths*](http://maths.uncommons.org/) , яка включає в себе кілька ГВЧ. *MersenneTwisterRNG* є хорошим вибором в більшості випадків.

Завантаження The Watchmaker Framework включає в себе вихідний код для декількох прикладів програм, в тому числі аплетів. Цей код може допомогти в розумінні того, як різні компоненти об'єднуються в робочі програми.

**ПРИКЛАДИ ВИКОРИСТАННЯ THE WATCHMAKER FRAMEWORK**

***Приклад №1***

Даний приклад буде реалізовано з використанням The Watchmaker Framework. Мета полягає в тому, щоб варіювати рядок з 11 символів, таких, щоб знайти фразу HELLO WORLD. Вводиться основна структура каркаса OPT4J з використанням класів *Creator*, *Decoder*, і *Evaluator*, а також класів *Genotype* і *Phenotype*.

Генотипом є список з 11 символів. Тут випадкова *SelectGenotype* створений за допомогою *HelloWordCreator*:

***HelloWordCreator.java***

public class HelloWorldCreator implements Creator<SelectGenotype<Character>> {    
    
  List<Character> chars = new ArrayList<Character>();    
  {    
    for (char c = ’A’; c <= ’Z’; c++) {    
      chars.add(c);    
    }    
    chars.add(’ ’);    
  }    
  Random random = new Random();    
    
  public SelectGenotype<Character> create() {    
    SelectGenotype<Character> genotype = new SelectGenotype<Character>(chars);    
    genotype.init(random, 11);    
    return genotype;    
  }    
}

Так як інтерфейс *Phenotype* не може бути доданий до класу *String*, використовується *PhenotypeWrapper*. *HelloWordDecoder* декодує *Genotype* в *Phenotype*:

***HelloWordDecoder.java***

public class HelloWorldDecoder implementsDecoder<SelectGenotype<Character>, PhenotypeWrapper<String>> {    
    
  public PhenotypeWrapper<String> decode(SelectGenotype<Character> genotype) {    
    String s = "";    
    for(int i=0; i<genotype.size(); i++){    
      s += genotype.getValue(i);    
    }    
    return new PhenotypeWrapper<String>(s);    
  }    
}

Нарешті, *Phenotype* повинен бути оцінений. Тут кількість матчів в порівнянні з HELLO WORLD підраховуються і збільшуються в *HelloWordEvaluator*:

***HelloWordEvaluator*.*java***

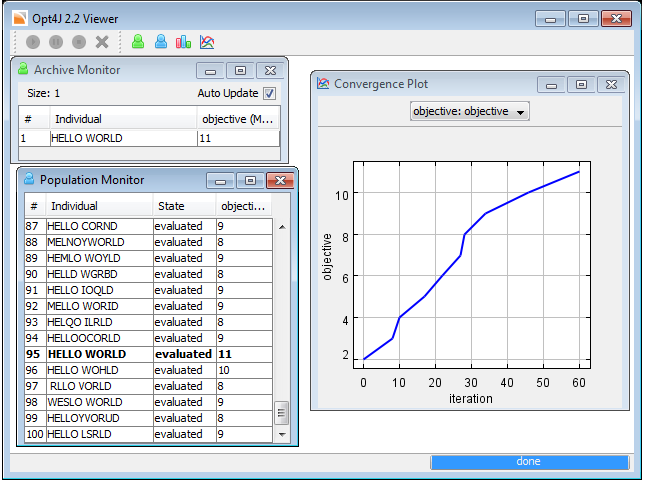
public class HelloWorldEvaluator implements Evaluator<PhenotypeWrapper<String>> {    
    
  Objective objective = new Objective("objective", Sign.MAX);    
    
  String target = "HELLO WORLD";    
    
  public Objectives evaluate(PhenotypeWrapper<String> phenotype) {    
    String s = phenotype.get();    
    
    int value = 0;    
    for (int i = 0; i < s.length(); i++) {    
      if (s.charAt(i) == target.charAt(i)) {    
        value++;    
      }    
    }    
    
    Objectives objectives = new Objectives();    
    objectives.add(objective, value);    
    return objectives;    
  }    
    
  public Collection<Objective> getObjectives() {    
    return Arrays.asList(objective);    
  }    
}

*ProblemModule* для цього завдання визначається в *HelloWorldModule*:

**HelloWordModule.java**

public class HelloWorldModule extends ProblemModule {    
    
  @Override    
  protected void config() {    
    bindProblem(HelloWorldCreator.class, HelloWorldDecoder.class,    
        HelloWorldEvaluator.class);    
  }    
}

Результати виглядають наступним чином (рис.3):

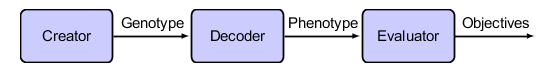


*Рис.3. Варіювання рядка з 11 символів для пошуку фрази HELLO WORLD за допомогою The Watchmaker Framework.*

***Приклад №2***

У даному прикладі показано, як реалізувати задачу комівояжера (англ.–TSP) як задачу оптимізації для OPT4J, використовуючи The Watchmaker Framework.

Спочатку, завдання оптимізації визначається з використанням *Creator*, *Decoder* і *Evaluator*. *Creator* створює новий *Genotype* і передає його до *Decoder*. *Decoder* переводить *Genotype* в *Phenotype*. *Evaluator* оцінює *Objectives*:

**

У задачі комівояжера, *Genotype* є *PermutationGenotype* міст, що представляють одну поїздку туди і назад (маршрут). *Phenotype* являє собою маршрут. Інтерфейси *Creator*, *Decoder* і *Evaluator* реалізуються через *SalesmanCreator,* *SalesmanDecoder*, і *SalesmanEvaluator*.

*SalesmanProblem* містить опис проблеми. Визначається як сукупність *n* міст з площею в 100 на 100. Клас пропонує відкритий метод *(getCities ())* для вилучення цих міст:

**SalesmanProblem.java**

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;    
  }    
    
}

Число міст передається в конструктор. Конструктор повинен бути анотований з *@Inject*. Розмір параметр конструктора позначається *@Constant* (value="size").

*SalesmanCreator* повинен створити *PermutationGenotypes*. Кожен з них містить всі міста в випадковому порядку. Рандомізації здійснюється за допомогою методу *shuffle()* класу *Collections*:

**SalesmanCreator.java**

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;    
  }    
}

Тепер, перетворення *Genotype* в *Phenotype* повинно бути зроблено за допомогою *Decoder*. В цьому випадку *Phenotype* є маршрутом, який представляє подорож туди і назад. Маршрут задається як упорядкований список значень *City*.

**SalesmanRoute.java**

public class SalesmanRoute extends ArrayList<City> implements Phenotype{}

Оскільки *SalesmanGenotype* вже дано як перестановку всіх міст, *SalesmanDecoder* є простою операцією копіювання елементів перестановки *Genotype* на маршрут.

**SalesmanDecoder.java**

public class SalesmanDecoder implements    
    Decoder<PermutationGenotype<City>, Route> {    
    
  public Route decode(PermutationGenotype<City> genotype) {    
    Route route = new Route();    
    for (City city : genotype) {    
      route.add(city);    
    }    
    return route;    
  }    
}

Наступним завданням є оцінити відстань в *SalesmanRoute*.

**SalesmanEvaluator.java**

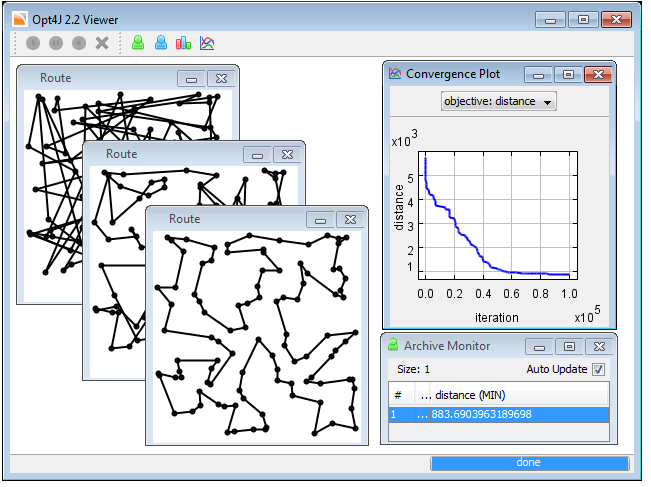
public class SalesmanEvaluator implements Evaluator<Route> {    
    
  Objective distance = new Objective("distance", Sign.MIN);    
    
  public Objectives evaluate(Route route) {    
    double dist = 0;    
    for (int i = 0; i < route.size(); i++) {    
      City one = route.get(i);    
      City two = route.get((i + 1) % route.size());    
      dist += getEuclideanDistance(one, two);    
    }    
    Objectives objectives = new Objectives();    
    objectives.add(distance, 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);    
  }    
    
  public List<Objective> getObjectives() {    
    return Arrays.asList(distance);    
  }    
}

Ця TSP має одновимірну функцію, яка повинна бути зведена до мінімуму. Таким чином, відстань *Objective* єдина функція з ім'ям «*distance*», яку потрібно звести до мінімуму (Sign.MIN).

**TravelingSalesmanModule.java**

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);    
  }    
}

Результати виглядають наступним чином (рис.4):

****

*Рис.4. Задача комівояжера за допомогою The Watchmaker Framework.*

**Висновок:** під час виконання розрахунково-графічної роботи я ознайомилась з особливостями та можливостями The Watchmaker Framework. Навчилась використовувати даний фреймворк, визначила процес встановлення (підключення) заданого фреймворка.

Основними перевагами даного фреймворка можна вважати те, що The Watchmaker Framework є розширеним, високої продуктивності, об'єктно-орієнтованим середовищем для реалізації незалежних від платформи еволюційних / генетичних алгоритмів у Java. Фреймворк забезпечує типову еволюцію для довільних типів за допомогою неінвазивного API. The Watchmaker Framework є відкритим вихідним кодом, безкоштовний для скачування та використання.