|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САП** | | Тема | оцінка | підпис |
| СПКм-12 |  | **Jenes 2.0** |  |  |
| Сухацький Р. В. | |
| № залікової: 1508516 | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р.З. | |

1. **Мета роботи**

Ознайомитись з можливостями Jenes 2.0. Здобути навики використання програмного продукту для вирішення поставлених задач.

1. **Короткі теоретичні відомості**

Jenes 2.0 (читається як «гени» або «дженс») – це новий, оптимізований реліз бібліотеки для генетичних алгоритмів для Java. Бібліотека покликана бути швидкою і з чітким використанням ресурсів, але як і раніше дуже простою у використанні. Бібліотека є проектом з відкритим вихідним кодом, що розроблений Intelligentia S.r.l. у співпраці з обчислювально-інженерною лабораторією CISELab університету Sannio і розміщена на sourceforge.com.

*Особливості Jenes:*

* Оптимізована архітектура і використання пам'яті
* Модульні й високо реконфігураційні алгоритми
* Строга типізація
* Доступно для Java 1.5 і вище
* Можливість експорту результатів в MS Excel і CSV файли
* Різні типи алгоритмів підтримуються:
* Багатоцільова оптимізація NSGA2
* Багатопоточна підтримка паралельної оцінки придатності на багатоядерних процесорах.
* Кілька генетичних операторів
* Покращене управління пам'яттю

Нижче наведено деякі з особливостей даного фреймворку.

**Модульні й високо реконфігуровані алгоритми**

Алгоритм Jenes структуровані в компоненти, які здатні взаємодіяти один з одним. Дані мають змогу проходити через декілька алгоритмів і там трансформуватися. Кожен етап Приймає вхідні дані і продукує вихідні. Всі етапи є взаємозамінними. Це робить можливим побудувати алгоритм, який найкращим чином відповідає потребі користувача, а його структура є простою, та зрозумілою. Етапи щодо strucuture є sequenxe і паралельні, wilst оператори елементарні стадії виконання таких завдань, як вибір, схрещування і мутації. Користувач може збірки свій власний алгоритм створивши власну структури і те які оператори використовувати, наприклад, обрати між такими вибірками як Wheel  або Tournament, або обрати між однією чи двома точками кросовера, і так далі.

**Оптимізована архітектура і пам'ять**

Особливу увагу було приділено оптимізації коду в Jenes. Основні покращення стосується використання пам'яті. Об'єкти безперервно переробляються. Це дозволяє зменшити витрати пам’яті, накопичені шляхом збору “сміття”. Якщо чисельність популяції фіксована протягом виконання алгоритму, то затрати пам'яті відбуваються на початку виконання і не змінюється в процесі виконання. Це дозволяє скоротити час для створення об'єктів.

**Строгий контроль типу**

Типи даних Jenes строго типізовані, тому населення і окремі індевіди можуть працювати тільки з сумісними типами. Це дозволяє зменшити кількість перевірок під час виконання, так як використання параметричних класів гарантується на етапі компіляції. Це також тягне за собою той факт, що немає ніякої необхідності приведення типів об'єктів. Це призводить до більш високої надійності програмного забезпечення і швидкодії.

**Написана в Java 1.5**

Нарешті, Jenes сумісна з Java 1.5 або наступними версіями. Це було зроблено для того, щоб отримати ширший набір інструментів для перевірки типів і скорочення накладних витрат, за рахунок відсутності перевірки типів під час виконання.

**Можливості Jenes які будуть незабаром реалізовані**

* Графічний інтерфейс для більш швидкого налаштування алгоритму
* Багатопоточність для багатоядерних процесорів
* Підтримка розподілених і грід обчислень
* Більш багатий набір генетичних операторів
* Параметри автонастройки

**Анатомія генетичного алгоритму в Jenes**

Програмування в Jenes легке, інтуїтивно зрозуміле та цікаве. У цьому розділі ми надамо декілька концепцій, які потрібно мати на увазі. Ці поняття допоможуть вам ефективно переміщатися серед класів, реалізованих у Jenes API. Ви будете здивовані тим, як швидко можна придумати генетичний алгоритм, а також реалізувати та протестувати.

**Хромосоми, індивіди та популяція.**

Перше, що потрібно мати на увазі відмінність між хромосомами, індивідів і популяцій.

Їх реалізації в просторі бібліотеки представлені окремими класами. Вони утворені з хромосоми і фітнес-значення:

INDIVIDUAL = CHROMOSOME + FITNESS.

Реалізація хромосом. У Jenes, вони розглядаються як масив генів. Ось список класів хромосом, передбачених Jenes.

* *BitwiseChromosome*: моделі хромосом бітів. Їх ген містить значення що представлено в бітових кодах.
* *BooleanChromosome*: моделі хромосом логічних значень. Кожен ген може бути істинним або хибним.
* *DoubleChromosome*: моделі хромосом подвійних значень. Кожне значення знаходиться в діапазоні [LowerBound, UpperBound]. Ці межі визначаються в момент створення екземпляра.
* *IntegerChromosome*: моделі хромосом цілочисельних значень. Кожне значення знаходиться в діапазоні [LowerBound, UpperBound]. Ці межі визначаються в момент створення екземпляра.
* *ObjectChromosome:* моделі хромосом, що містять значення об'єкта.
* *PermutationChromosome*: моделі хромосом, що містять масив цілочисельних значень. Найбільш важливою властивістю є те, що хромосома не містить дубльованих значень.

Всі класи є хромосоми реалізації інтерфейсу хромосома. Загалом, хромосоми мають фіксовану кількість генів, але їх довжина може змінюватися під час виконання алгоритм. Чисельність популяції є просто сукупність індивідів (тобто рішень).

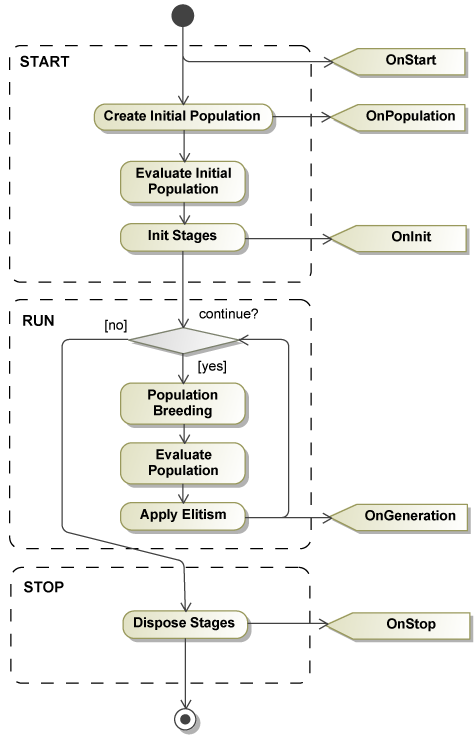
Клас Індивід та всі екземпляри класу Індивіди<T> розширюють клас Хромосома, щоб мати сильніший контроль над типами.

В процесі еволюції алгоритму, минулі популяції буферизуются в історії алгоритму. На кожній ітерації еволюційного алгоритму, найстаріші популяції і його індивіди повторно використовуються. Цей метод дозволяє уникнути виділення пам'яті для нової популяції і обмежує використання збирача сміття.

Як вже говорилося, генетичний алгоритм обробляє популяцію індивідів. У кожному поколінні є вхідні і вихідні популяції. Важливо враховувати, що вихідна популяція попередньо ініціалізована раніше. Так Jenes в цілому не виділяє нових індивідів.

Структура алгоритму

Гени забезпечують гнучку структуру для еволюції популяції індивідуумів. В цілому генетичний алгоритм структурований, як показано на рис. 1.



В основному генетичний алгоритм виконує наступний цикл:

* Випадково створюється початкова популяція
* Повторення
* Валідація придатність кожного індивідуума
* Вибір одного або кілька індивідів з популяції з вірогідністю, що спирається на придатність придатності індивіда
* Створення нових індивідів шляхом застосування генетичних операцій
* Поки потрібне рішення не буде знайдено або деяка інша умова зупинки виконається
* Повернути кращі індивіди

У Jenes генетичний алгоритм може бути реалізований за допомогою підкласів GeneticAlgorithm або реалізацію класу Fitness Цей метод використовується для оцінки придатності окремого індивіда.

У Jenes тіло генетичного алгоритму називається "труба етапів". Етапи виконуються послідовно в порядку їх додавання до послідовності. Кожен етап приймає вхідну групу популяцій (отриманий в якості вихідного сигналу по попередній стадії) і формує вихідну популяцію. Питання які етапи і в якому порядку виконувати залишається за розробником.

Посилання на поточний, так і в процесі популяції відповідно можуть бути отримані методами getCurrentPopulation () і getNextPopulation ().

Jenes також дозволяє визначати паралельні етапи. Паралельний етап формується різними гілками, кожна гілка є етап, який отримує субпопуляції відповідно до розподільника населення. Розподільник розподіляє популяції між гілками паралельної стадії і об'єднує вихід популяції кожної гілки.

Точка розриву і генетичні оператори являють собою найпростіші загальні етапи труб.

Генетичні оператори, які використовуються в генетичних алгоритмів аналогічні тим, які відбуваються в навколишньому світі:

**Вибір**: який віддає перевагу кращому індивіду, дозволяючи їм передати позитивні риси наступному поколінню.

**Кросовер**: що являє спаровування між окремими особами.

**Мутація**: який вводить випадкові зміни.

Jenes забезпечує реалізацію найбільш поширених генетичних операторів:

**TournamentSelector**: вибирає в довільному порядку ряду осіб з популяції. Вони порівнюються один з одним і оберається кращий з них, щоб бути батьком. TournamentSelector дійсно гарантує, що навіть середньої якості індивіди мають певний шанс мати нащадків. У Jenes необхідно вказати кількість спроб для TournamentSelector.

**RouletteWheelSelector**: вибирає конкретний індивід подуляції, щоб бути батьком.

**OnePointCrossover**: вибирає випадковим чином одну точку кросовера від батьківських хромосом і створює нове потомство. Одна точка кросовера може виглядати наступним чином (де "|" є точка перетину):

|  |  |
| --- | --- |
| **Chromosome 1** | **11101 | 001000** |
| Chromosome 2 | 00001 | 010101 |
| *Offspring 1* | **11101**| 010101 |
| *Offspring 2* | 00001 | **001000** |

**TwoPointsCrossover**: вибирає випадковим чином дві точки кросовера від батьківських хромосом і створює нове потомство. Два кросовера може виглядати наступним чином:

|  |  |
| --- | --- |
| **Chromosome 1** | **11 | 10100 | 1000** |
| Chromosome 2 | 00 | 00101 | 0101 |
| *Offspring 1* | **11**| 00101 | **1000** |
| *Offspring 2* | 00 | **10100**| 0101 |

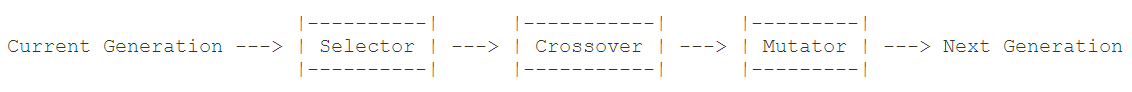
У Jenes необхідно вказати ймовірність, за допомогою якого буде використовуватися оператор кросовера. У загальному випадку для оператора кросовера ймовірність встановлений на 80%

**Simple Mutator:** випадковим чином вибирає точку мутації і рандомізують його ген. Оператор Мутація також потребує ряд ймовірностей, яка визначає його використання.

У загальному випадку для оператора мутації ймовірність встановлена на 20%.

Jenes надає також простий інтерфейс для GeneticAlgorithm. Це клас SimpleGA, який реалізує три етапи генетичний алгоритм, що складається з одного селектора, одного кросовера і одного мутатора в підряд.

Він надає простий спосіб для реалізації свого власного генетичного алгоритму, що дозволяє нехтувати проблемами, пов'язаними з побудовою етапів.



Клас надає набір конструкторів за допомогою яких можна створити екземпляр алгоритму SimpleGA. Конструктори дозволяє вирішити, який метод (тобто турнір або колеса рулетки) або метод кросовера (тобто одна точка або дві) буде обрано. Крім того, кросовер і мутації можуть бути задані під час будівництва. SimpleGA підклас GeneticAlgorithm.

Jenes включає в себе підтримку елітарності, тобто кращі особини в кожному поколінні будуть представлені в наступному поколінні. Число елітних особин встановлюється методом setElitism (int).

Ці особи замінюються на деяких осіб до обробки популяції у відповідності з наступними стратегіями:

**Random Elitism Strategy**: наступний індивід популяції випадковим чином вибирається і заміщується елітним.

**Worst Elitism Strategy**: наступний індивід популяції гірших індивідів замінюються елітним.

Ви можете встановити бажану стратегію елітарності наступним чином:

sga.setElitismStrategy(ElitismStrategy.RANDOM);

або

sga.setElitismStrategy(ElitismStrategy.WORST);

Перша стратегія є більш ефективною. Недолік полягає в тому, хороші індивіди можуть бути замінені елітою. Друга стратегія повільніша, але гарантує, що тільки гірші особини замінюються.

Jenes використовує клас **MersenneTwisterRandom** для генерації псевдовипадкових чисел і значень. Цей оптимізований клас, який забезпечує швидку генерацію дуже високоякісних випадкових чисел. З цієї причини він використовується замість java.util.Random. Генерація псевдовипадкових чисел дуже важлива на всіх етапах еволюційних алгоритмів (наприклад, ініціалізація населення, використання операторів вибору, визначення кросовера і мутації точок, і так далі).

1. **Процес встановлення Jenes 2.0**

Фреймворк Jenes написано на Java, що не вимагає будь-якого додаткового програмного забезпечення. Все що необхідно, це мати інстальованим Java JDK 1.5 або більш пізню версію. Вихідний код поставляється в пакеті tar.gz, який можна завантажити з наступного поситання <https://sourceforge.net/projects/jenes/files/Jenes/>.

Нижче буде представлено спосіб інсталяції Jenes використовуючи Eclipse IDE.

Після завантаження фреймворк матиме виляд Jar бібліотеки рис. 2.

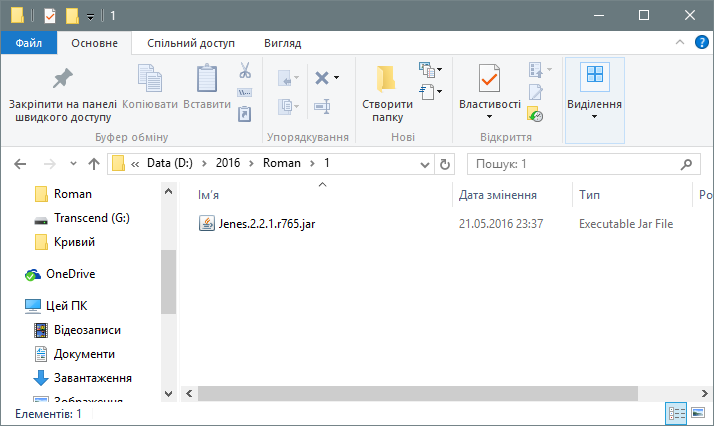


Рис. 2. JAR бібліотека Jenes після завантаження

Наступним кроком є підключення Jenes до середовища Eclipse IDE. Для цього необхідно виконати наступні кроки:

* Виберіть *File* → *Create* → *New Java Project*.
* Введіть ім'я проекту (наприклад, Jenes) і натисніть на кнопку *Finish*.
* Натисніть правою кнопкою миші на папці з проектом (в провіднику Eclipse) та оберіть *Properties* (Рис. 3.).
* Оберіть розділ *Java Build Pat*.
* В меню праворуч натисніть кнопку *Add External JARs..* (Рис. 4.)
* Вкажіть шлях до папки, що містить JAR бібліотеку Jenes та натисніть відкрити (Рис. 5.).

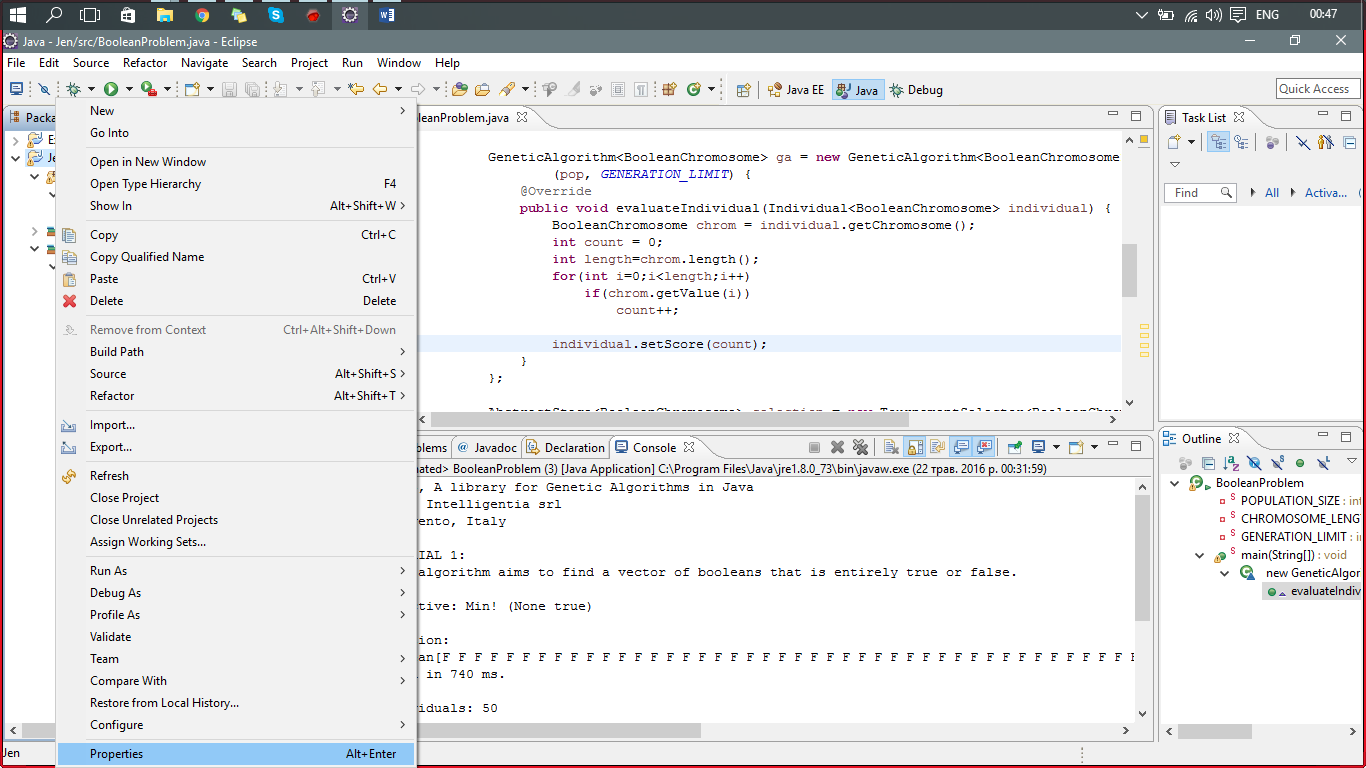


Рис. 3. Вікно *Project Properties*

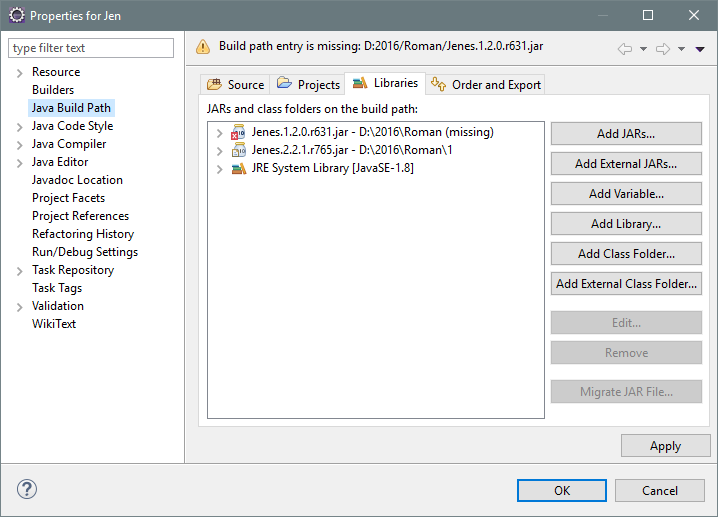


Рис. 4. Вікно *Java Build Pat*

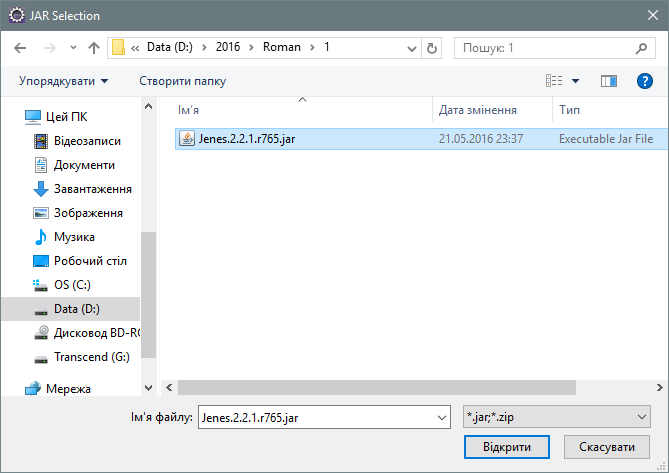


Рис. 5. *JAR Selection*

Мною було обране тестове завдання **BooleanProblem (jenes.intelligentia.it/tutorials/tutorial1)** в якості прикладу. Для того щоб налаштувати програму, відкрийте файл **BooleanProblem** вибравши його з **jenes.tutorials.problem1.BooleanProblem** (Рис. 5). Після цього ви можете внести будь які зміни, чи удосконалення, в залежності від поставленої задачі.

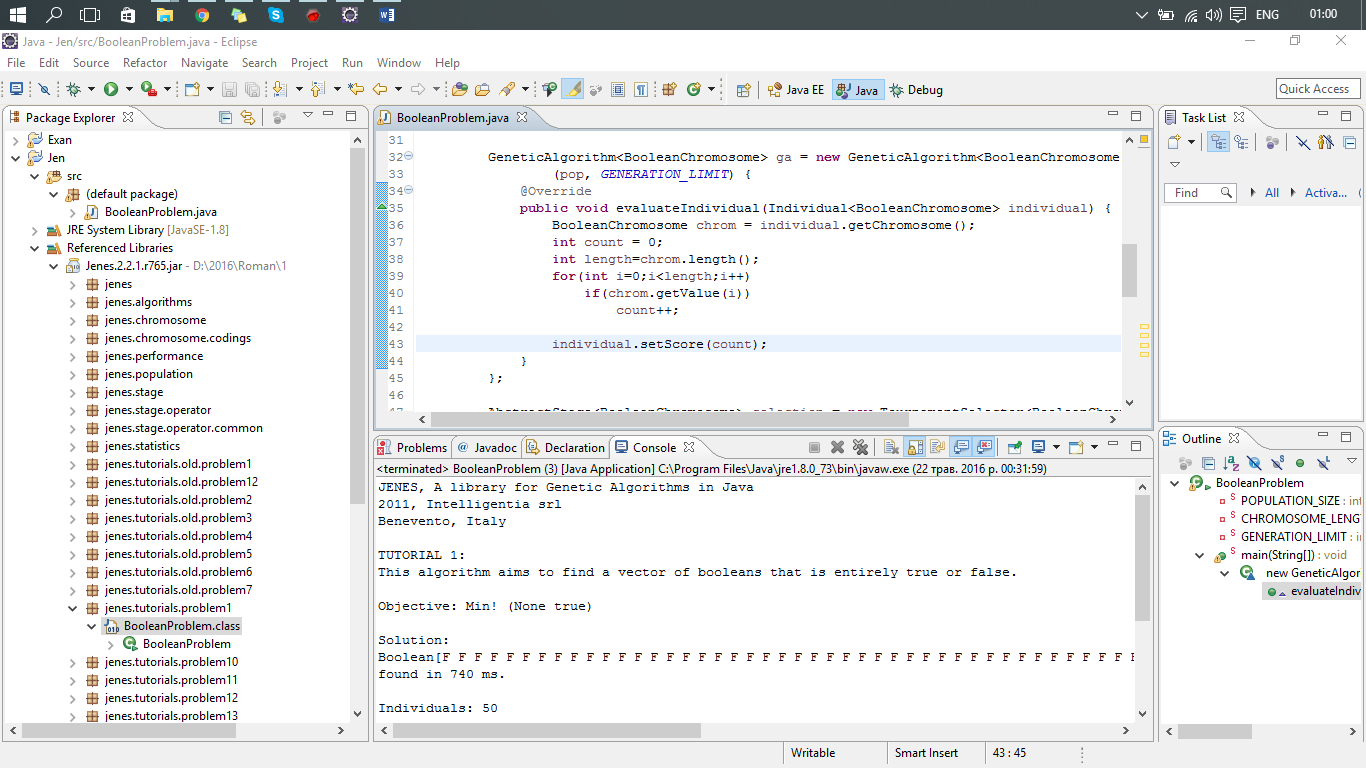


Рис. 6. Представлення реалізації програми BooleanProblem в Jenes

Для запуску алгоритму, клацніть правою кнопкою миші на **BooleanProblem.cls** в дереві проекту. Виберіть *Run As* → *Java Application*. В результаті ви отримаєте вивід на консоль результату виконання програми рис. 7.

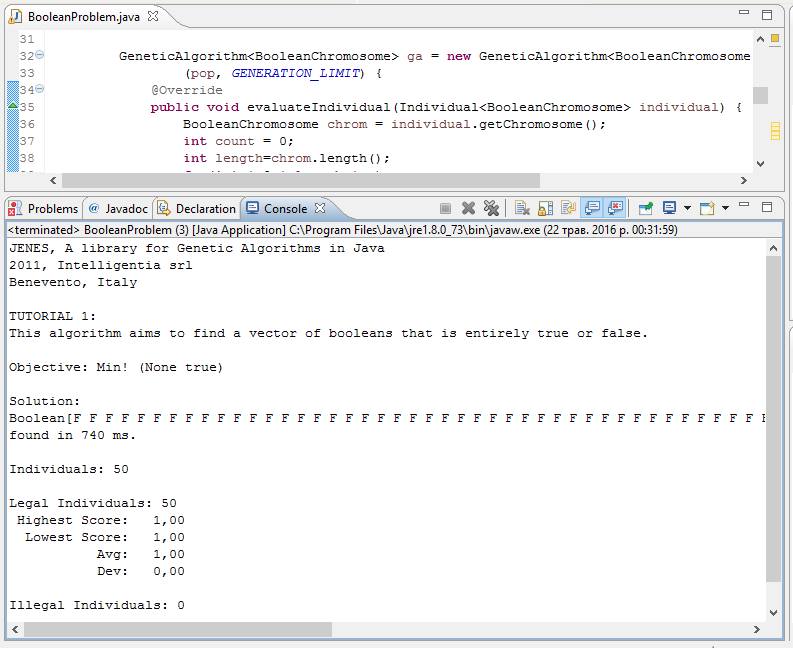


Рис. 7. Вигляд консолі Eclipseз результатом виконання програми

А тепер розглянемо детальніше приклад простої логічної проблеми.

1. **Виберіть підходящу проблему хромосому і створити початкову популяцію.**

Вибір відповідного подання хромосоми є найбільш важливим завданням зробити перед запуском генетичного алгоритму. Яке уявлення для використання залежить від структури проблемних рішень. У нашому випадку рішення виконані з логічних масивів. Таким чином, BooleanChromosome виглядає бути approppriate вибір. Хромосоми являють собою ключовий компонент рішень (тобто фізичних осіб). Для побудови початкової популяції нам потрібен прототип рішень (зразка), як показано в наступному коді.

060         Individual<BooleanChromosome> sample = **new**Individual<BooleanChromosome>(**new**BooleanChromosome(CHROMOSOME\_LENGTH));  
061         Population<BooleanChromosome> pop = **new**Population<BooleanChromosome>(sample, POPULATION\_SIZE);

where

045     **private static int**POPULATION\_SIZE=50;  
046     **private static int**CHROMOSOME\_LENGTH=100;  
047     **private static int**GENERATION\_LIMIT=1000;

1. **Встановити вгору генетичний алгоритм.**

063         Fitness<BooleanChromosome> fit = **new**Fitness<BooleanChromosome>(**false**) {  
064   
065             @Override  
066             **public void**evaluate(Individual<BooleanChromosome> individual) {  
067                 BooleanChromosome chrom = individual.getChromosome();  
068                 **int**count = 0;  
069                 **int**length=chrom.length();  
070                 **for**(**int**i=0;i<length;i++)  
071                     **if**(chrom.getValue(i))  
072                         count++;  
073                   
074                 individual.setScore(count);  
075             }  
076               
077         };

а потім передати цей об'єкт в якості першого параметра генетичного алгоритму (або будь-якого заздалегідь заданого підкласу)

079         GeneticAlgorithm<BooleanChromosome> ga = **new**GeneticAlgorithm<BooleanChromosome>  
080                 (fit, pop, GENERATION\_LIMIT);

Клас Фітнес пропонує два різних конструкторів для того, щоб визначити кількість завдань, щоб розглянути в оцінці окремих осіб, так і для кожної цілі прапор, який вказує, якщо це функція розгортання чи ні. У коді, перерахованих вище, ми створюємо придатність для однієї об'єктивної завдання (тільки один прапор в якості параметрів).

1. **Виберіть операторів, які будуть використовуватися генетичного алгоритму і додати їх як етапи га.**

Після того, як генетичний алгоритм визначено, нам необхідно визначити послідовність операторів населення буде проходити через нього. Найпростіша схема містить тільки три оператора в наступній послідовності: один селектор, один кросовер і один мутатор. Проте, можна створити більш складну трубу, що має paralleles і послідовності. Для цього уроку ми приймемо наступну просту структуру (див API, утворюють більш детальну інформацію про вжиті операторами):

082   AbstractStage<BooleanChromosome> selection = **new**TournamentSelector<BooleanChromosome>(3);  
083   AbstractStage<BooleanChromosome> crossover = **new**OnePointCrossover<BooleanChromosome>(0.8);  
084   AbstractStage<BooleanChromosome> mutation = **new**SimpleMutator<BooleanChromosome>(0.02);  
085   ga.addStage(selection);  
086   ga.addStage(crossover);  
087   ga.addStage(mutation);

1. **Встановіть параметри алгоритму і запустити еволюцію.**

Можна налаштувати генетичну установку значення елітарності і цілі оптимізації перед тим, щоб запустити алгоритм еволюції. Елітарність це число кращих людей, щоб провести в наступному поколінні (1 в нашому випадку).

089         ga.setElitism(1);

І, нарешті, ми можемо зробити алгоритм працює.

091         ga.evolve();

1. **Отримання результату еволюції.**

Jenes надає статистичні дані як для алгоритму і населення. Перший відносяться до статистики щодо роботи алгоритму, а саме раз ініціалізації, починаючи, еволюції та поколінь. Другий до розподілу рішень і пов'язаних з ними значень придатності, наприклад, осіб, що були замовлені за рахунок зменшення функції придатності, середнє максимальне і мінімальне значення придатності. Вони можуть бути отримані в будь-який момент. Ми будемо використовувати їх, коли алгоритм завершений.

093         Population.Statistics stats = ga.getCurrentPopulation().getStatistics();  
094         GeneticAlgorithm.Statistics algostats = ga.getStatistics();  
095           
096         System.out.println("Objective: " + (fit.getBiggerIsBetter()[0] ? "Max! (All true)" : "Min! (None true)"));  
097         System.out.println();  
098           
099         Group legals = stats.getGroup(Population.LEGALS);  
100           
101         Individual solution = legals.get(0);  
102                   
103         System.out.println("Solution: ");  
104         System.out.println( solution );  
105         System.out.format("found in %d ms.\n", algostats.getExecutionTime() );  
106         System.out.println();  
107           
108         Utils.printStatistics(stats);

**Висновок:** під час виконання розрахункової роботи я ознайомивсь з бібліотекою для розробки генетичних алгоритмів Jenes 2.0. 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 Jenes 2.0.

**Додаток А. Код програми**

/\*  
 \* JENES  
 \* A time and memory efficient Java library for genetic algorithms and more   
 \* Copyright (C) 2011 Intelligentia srl  
 \*  
 \* This program is free software: you can redistribute it and/or modify  
 \* it under the terms of the GNU General Public License as published by  
 \* the Free Software Foundation, either version 3 of the License, or  
 \* (at your option) any later version.  
 \*  
 \*  This program is distributed in the hope that it will be useful,  
 \*  but WITHOUT ANY WARRANTY; without even the implied warranty of  
 \*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the  
 \*  GNU General Public License for more details.  
 \*  
 \*  You should have received a copy of the GNU General Public License  
 \*  along with this program.  If not, see <http://www.gnu.org/licenses/>.   
 \*/  
**package**jenes.tutorials.problem1;  
  
**import**jenes.Fitness;  
**import**jenes.GeneticAlgorithm;  
**import**jenes.chromosome.BooleanChromosome;  
**import**jenes.population.Individual;  
**import**jenes.population.Population;  
**import**jenes.population.Population.Statistics.Group;  
**import**jenes.stage.AbstractStage;  
**import**jenes.stage.operator.common.OnePointCrossover;  
**import**jenes.stage.operator.common.SimpleMutator;  
**import**jenes.stage.operator.common.TournamentSelector;  
**import**jenes.tutorials.utils.Utils;  
  
/\*\*  
 \* Tutorial implementing a basic genetic algorithm.  
 \* The problem consists in finding a vector full of zeros or ones.  
 \*   
 \* @author Luigi Troiano  
 \*  
 \* @version 2.0  
 \*  
 \* @since 1.0  
 \*/  
**public class**BooleanProblem {  
      
    **private static int**POPULATION\_SIZE=50;  
    **private static int**CHROMOSOME\_LENGTH=100;  
    **private static int**GENERATION\_LIMIT=1000;  
      
    **public static void**main(String[] args) **throws**Exception {  
          
        Utils.printHeader();  
        System.out.println();  
          
        System.out.println("TUTORIAL 1:");  
        System.out.println("This algorithm aims to find a vector of booleans that is entirely true or false.");  
        System.out.println();  
          
        // Random.getInstance().setStandardSeed();  
          
        Individual<BooleanChromosome> sample = **new**Individual<BooleanChromosome>(**new**BooleanChromosome(CHROMOSOME\_LENGTH));  
        Population<BooleanChromosome> pop = **new**Population<BooleanChromosome>(sample, POPULATION\_SIZE);  
          
        Fitness<BooleanChromosome> fit = **new**Fitness<BooleanChromosome>(**false**) {  
  
            @Override  
            **public void**evaluate(Individual<BooleanChromosome> individual) {  
                BooleanChromosome chrom = individual.getChromosome();  
                **int**count = 0;  
                **int**length=chrom.length();  
                **for**(**int**i=0;i<length;i++)  
                    **if**(chrom.getValue(i))  
                        count++;  
                  
                individual.setScore(count);  
            }  
              
        };  
          
        GeneticAlgorithm<BooleanChromosome> ga = **new**GeneticAlgorithm<BooleanChromosome>  
                (fit, pop, GENERATION\_LIMIT);  
          
        AbstractStage<BooleanChromosome> selection = **new**TournamentSelector<BooleanChromosome>(3);  
        AbstractStage<BooleanChromosome> crossover = **new**OnePointCrossover<BooleanChromosome>(0.8);  
        AbstractStage<BooleanChromosome> mutation = **new**SimpleMutator<BooleanChromosome>(0.02);  
        ga.addStage(selection);  
        ga.addStage(crossover);  
        ga.addStage(mutation);  
          
        ga.setElitism(1);  
          
        ga.evolve();  
          
        Population.Statistics stats = ga.getCurrentPopulation().getStatistics();  
        GeneticAlgorithm.Statistics algostats = ga.getStatistics();  
          
        System.out.println("Objective: " + (fit.getBiggerIsBetter()[0] ? "Max! (All true)" : "Min! (None true)"));  
        System.out.println();  
          
        Group legals = stats.getGroup(Population.LEGALS);  
          
        Individual solution = legals.get(0);  
                  
        System.out.println("Solution: ");  
        System.out.println( solution );  
        System.out.format("found in %d ms.\n", algostats.getExecutionTime() );  
        System.out.println();  
          
        Utils.printStatistics(stats);  
    }  
      
}

**Додаток Б. Результат виконання програми**

JENES, A library for Genetic Algorithms in Java

2011, Intelligentia srl

Benevento, Italy

TUTORIAL 1:

This algorithm aims to find a vector of booleans that is entirely true or false.

Objective: Min! (None true)

Solution:

Boolean[F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F T F F F F F F F F F F F F F F F F F]: scored [1.0]

found in 740 ms.

Individuals: 50

Legal Individuals: 50

Highest Score: 1,00

Lowest Score: 1,00

Avg: 1,00

Dev: 0,00

Illegal Individuals: 0