МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

**Звіт до розрахунково-графічної роботи**

на тему: «Jenes 2.0 »

з курсу:

«Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні»

Варіант - 16

Виконала:

ст. гр. КНм-14

Євтух Олеся

Перевірив:

Кривий Р. З.

Львів 2017

1. **Опис та основні особливості фреймворка Jenes 2.0**

**Jenes 2.0** (читається як genes" або "jeans) це новий випуск оптимізованої бібліотеки для генетичних алгоритмів на Java. Бібліотека призначена для швидкої і легкої пам'яті, але як і раніше дуже проста у використанні. Бібліотека є проектом з відкритим вихідним кодом, розробленим і Intelligentia S.r.l. у співпраці з обчислювальною та інженерною лабораторією Intelligent System (CISELab) в Університеті Sannio і розміщення на sourceforge.com.

Jenes це програмне забезпечення з відкритим вихідним кодом. Ви можете розповсюджувати та / або змінювати його відповідно до умов GNU General Public License v.3, виданої Free Software Foundation.

Фреймворк володіє такими основними можливостями:

1. **Модульні й високо перебудовувані алгоритми**

Алгоритм Jenes структурований із компонованих компонент, так звані етапи. Популяція проходить через етапи і там вона трансформується. Кожен етап приймає вхідну популяцію і випускає вихідну популяцію. Всі етапи є взаємозамінними. Це робить можливим структурувати алгоритм, який найкращим чином відповідає проблемній характеристиці і потребам користувачів, просто вирішити, як організувати потік через етапи. Етапи щодо структури є послідовність і паралельний, в той час як оператори є елементарними етапами виконання таких завдань, як селекція, схрещування і мутація. Користувач може зібрати свій власний алгоритм обравши структуру і потрібні оператори використання, наприклад, як вибір між пропорційним або турнірним відбором, або між одноточковим або двохточковим кросовером, і так далі.

1. **Оптимізована архітектура і пам'ять**

Особливу увагу було приділено оптимізації коду в Jenes. Найважливіше стосується використання пам'яті. Об'єкти безперервно перероблені. Це дозволяє знизити накладні витрати, представлені в процесі складання сміття. Якщо чисельність популяції фіксується за алгоритмом еволюції, окупація пам'яті зростає на початку і потім не змінюється. Це дозволяє скоротити загальне програмне забезпечення і час для будівництва і збору об'єктів. Хромосоми слід зменшується, наприклад, в разі BitwiseChromosome, здатного працювати на рівні окремих бітів, таким чином, випереджаючи інші рішення. Крім того, люди вже оцінили не переоцінені int наступних ітерацій.

1. **Строгий контроль типу**

Типи даних Jenes сильно набрані, тому популяція і окремі особи можуть працювати тільки з сумісними типами. Це дозволяє зменшити кількість перевірок під час виконання, так як використання параметричних класів забезпечується при компіляції, що використовує відповідні дані. Це також має на увазі, що немає ніякої необхідності лиття об'єктів. Це призводить до підвищення надійності програмного забезпечення і швидкості.

1. **Написана для Java 1.5+**

Нарешті, Jenes написана на Java 1.5 або наступних версіях. Цей вибір був для того, щоб забезпечити більш багатий набір перевірки типу і зниження накладних витрат, набравши перевірку під час виконання.

1. **Журнал експериментів у MS Excel і CSV-файлах**
2. **Різні типи алгоритмів підтримують:**

* Simple
* Crowding
* Island
* Steady State

1. **Багатоцільова оптимізація NSGA2**
2. **Багатопотокова підтримка паралельної оцінки придатності на багатоядерних процесорах**
3. **Деякі генетичні оператори**
4. **Індивідуальні та популяційні об'єднання для поліпшення управління пам'яттю**

**Можливості, які заплановані в майбутній версіях фреймворка:**

* Графічний інтерфейс користувача GUI для більш швидкого налаштування алгоритму
* Багатопотоковість для багатоядерних процесорів
* Підтримка розподілених обчислень
* Більш багатий набір хромосом і генетичних операторів
* Параметри автонастройки

1. **Анатомія генетичного алгоритму в Jenes 2.0**

Програмування в Jenes легко, інтуїтивно і веселе. У цьому розділі надамо кілька концепцій, розробник повинні мати на увазі, з формую викладення рішень в Jenes. Ці поняття допоможуть розробнику ефективно переміщатися між класами, передбаченими в Jenes API. Розробник буде здивований, як швидко можна визначити генетичний алгоритм, реалізувати і протестувати.

**Хромосоми, окремі особи і популяція**

Перше, що потрібно мати на увазі це відмінність між хромосомами, окремими особами і популяцією.

Рішення в просторі пошуку представлені приватними особами. Вони зроблені з хромосом і фітнес значення: INDIVIDUAL = CHROMOSOME + FITNESS.

Хромосоми є вирішенням для кодувань. У Jenes, вони розглядаються як масив генів. Ось список класів хромосом, передбачених Jenes.

* BitwiseChromosome: моделі хромосома бітів. Його гени містять значення що кодуються відповідно до зазначеного двійкового кодуванням;
* BooleanChromosome: моделі хромосом булевих 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 хромосома цілочисельних значень. Кожне значення знаходиться в діапазоні [LowerBound, UpperBound]. Ці межі визначаються в момент створення екземпляра;
* ObjectChromosome: являє собою хромосому з генами, що містять значення аллели об'єкта;
* PermutationChromosome: забезпечує хромосому, здатну моделювати перестановки. Масив цілочисельних значень є його геном. Найбільш важливою властивістю є те, що гени не містить дубльовані значення.

Всі класи хромосом є реалізацією інтерфейсу Chromosome. Загалом, хромосоми мають фіксовану кількість генів, але їх довжина може змінюватися під час виконання алгоритму з метою реалізації кодування змінної довжини розчину. Популяція це просто набір індивідуумів (тобто розчини).

Індивіди всі екземпляри параметричного класу Individual<T extends Chromosome>, для того, щоб мати сильніший контроль над типами. Особа може бути законна чи не законна. Можна встановити lagality індивідуума шляхом виклику методу setLegal (boolen) на своєму екземплярі. За замовчуванням кожен індивідуум встановлюється спочатку як законний.

В процесі еволюції алгоритму, минулі популяції буферні в історії алгоритму. Минула популяція, замість того, щоб бути вивільнені, підтримуються в пам'яті і використовується повторно. Цей метод дозволяє уникнути виділення в пам'яті нової популяції і обмежує використання збирача сміття.

Як вже говорилося, генетичний алгоритм обробляє популяцію особин. У кожному поколінні є вхідні і вихідні популяції. Важливо враховувати, що вихід популяції попередньо ініціалізується з «переробленими» особами, взятими з історії (з міркувань продуктивності). Так Jenes взагалі не виділяє нових особин.

**Структура алгоритму**

Jenes забезпечує гнучку структуру для еволюції популяції індивідуумів. В цілому генетичний алгоритм структуровано, як показано на рис. 1:

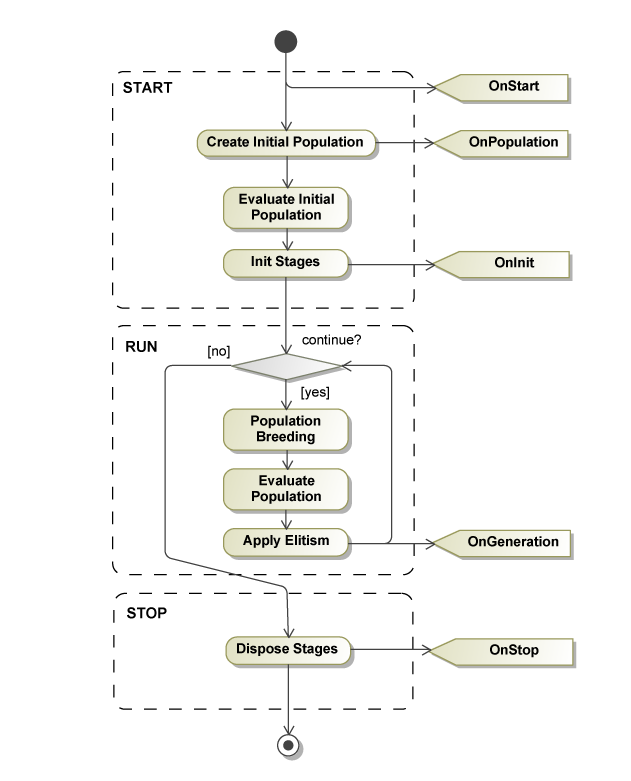


Рис. 1 Структура генетичного алгоритму в Jenes

В основному генетичний алгоритм виконує наступний цикл:

* Випадкове створення початкової популяцію
* repeate
  + Визначити пристосованість кожного індивідуума
  + Виберіть один або кілька особи з популяції з ймовірністю на основі придатності до генетичних операцій
  + Створення нових осіб шляхом застосування генетичних операцій з заданими ймовірностями
* until прийнятне рішення не буде знайдено або будь-яка інша умова зупинки виконується
* returne кращі так даного індивідууму

У Jenes генетичний алгоритм може бути реалізований за допомогою підкласів GeneticAlgorithm і визначення придатності реалізації популяції шляхом створення підкласів Fitness і реалізацій абстрактного методу evaluate(Individual). Цей метод використовується для оцінки придатності одної особини. Реалізація цього методу є те, що пов'язано з проблемою вирішити.

У Jenes генетичного алгоритму тіло є «трубою» етапів. Етапи виконуються послідовно в порядку їх додавання до послідовності. Кожен етап приймає вхідні популяцію (вироблену в якості вихідного сигналу на попередній стадії) і формує вихідну популяцію. Які етапи і в якому порядку розгляду залишено на потреби конкретного алгоритму.

Посилання на поточну, так і в процесі популяції може бути отримано посилання на шялхи за допомогою методів getCurrentPopulation () і getNextPopulation ().

Jenes також дозволяє визначати паралельні етапи. Паралельні етапи формуються різними гілками, кожна гілка є етапом, який отримує субпопуляції відповідно до вказаної популяції, що використовує. Дозуючий пристрій розподіляє населення між гілками паралельної стадії і об'єднує вихід кожної гілки в вихідній популяції паралельно.

Зупинка і генетичні оператори являють собою найпростіші загальні етапи процесу.

Точка зупину це етап, що повідомляє своїх слухачів, коли його метод зупиняється. Немає альтернативних вхідних популяцій.

Генетичні оператори, які використовуються в генетичних алгоритмах аналогічні тим, які відбуваються в природі:

* Вибір: який віддає перевагу особам, що дозволяють передавати наступному поколінню
* Схрещування: що являє схрещування між особинами
* Мутація: який вводить випадкові зміни

1. **Встановлення та приклади використання**

Даний фреймворк встановлюється простим способом. Для встановлення фреймворку потрібно його скачати та добавити jar файл, як бібліотеку до проекту Java.

**Приклад 1**

Завдання полягає в знаходженні вектора повного нулів або одиниць.

**Програма :**

**Клас BooleanProblem:**

**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;  
  
**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);  
    }  
      
}

**Результат виконання:**

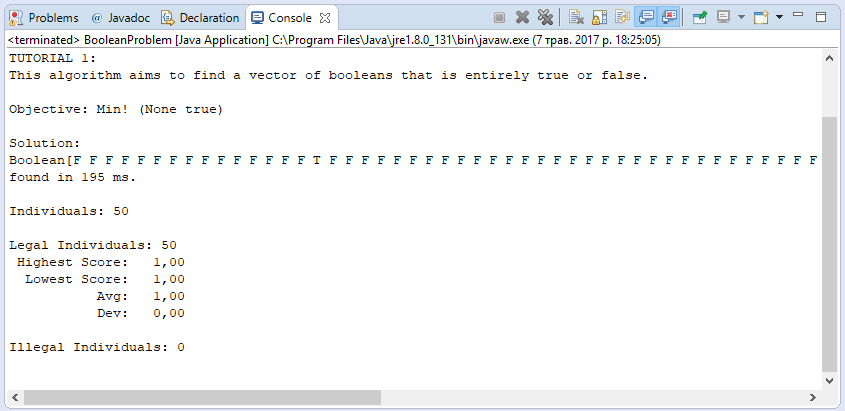


Рис. 2 Результат запуску програми до Прикладу 1

**Приклад 2**

У другому прикладі показано, як створити паралельний стан і покласти його в послідовність алгоритму, як розподіляти і об'єднувати паралельні стратегії, як встановити визначені користувачем умова завершення і як використовувати слухачів щоб зловити події алгоритму. Мета навчального завдання це знайти найближчий цілочисельний масив (представлений з заданим числом десяткових шифрів) до цільового з кожним номером в діапазоні [0,49].

**Програма:**

**Клас PatternProblem:**

**package** jenes.tutorials.problem2;

**import** jenes.GenerationEventListener;

**import** jenes.GeneticAlgorithm;

**import** jenes.utils.Random;

**import** jenes.chromosome.IntegerChromosome;

**import** jenes.population.Individual;

**import** jenes.population.Population;

**import** jenes.population.Population.Statistics;

**import** jenes.population.Population.Statistics.Group;

**import** jenes.stage.AbstractStage;

**import** jenes.stage.Parallel;

**import** jenes.stage.operator.common.OnePointCrossover;

**import** jenes.stage.operator.common.SimpleMutator;

**import** jenes.stage.operator.common.TournamentSelector;

**import** jenes.stage.operator.common.TwoPointsCrossover;

**import** jenes.tutorials.utils.Utils;

**public** **class** PatternProblem **implements** GenerationEventListener<IntegerChromosome> {

**private** **static** **int** *POPULATION\_SIZE* = 100;

**private** **static** **int** *CHROMOSOME\_LENGTH* = 10;

**private** **static** **int** *GENERATION\_LIMIT* = 1000;

**private** **static** **int** *MAX\_INT* = 49;

**private** PatternGA algorithm = **null**;

**public** PatternProblem() {

IntegerChromosome chrom = **new** IntegerChromosome(*CHROMOSOME\_LENGTH*,0,*MAX\_INT*);

Individual<IntegerChromosome> ind = **new** Individual<IntegerChromosome>(chrom);

Population<IntegerChromosome> pop = **new** Population<IntegerChromosome>(ind, *POPULATION\_SIZE*);

algorithm = **new** PatternGA(pop, *GENERATION\_LIMIT*);

algorithm.setElitism(5);

AbstractStage<IntegerChromosome> selection = **new** TournamentSelector<IntegerChromosome>(2);

Parallel<IntegerChromosome> parallel = **new** Parallel<IntegerChromosome>(**new** SimpleDispenser<IntegerChromosome>(2));

AbstractStage<IntegerChromosome> crossover1p = **new** OnePointCrossover<IntegerChromosome>(0.8);

parallel.add(crossover1p);

AbstractStage<IntegerChromosome> crossover2p = **new** TwoPointsCrossover<IntegerChromosome>(0.5);

parallel.add(crossover2p);

AbstractStage<IntegerChromosome> mutation = **new** SimpleMutator<IntegerChromosome>(0.02);

algorithm.addStage(selection);

algorithm.addStage(parallel);

algorithm.addStage(mutation);

algorithm.addGenerationEventListener(**this**);

}

**public** **void** run(**int**[] target, **int** precision) {

((PatternGA.PatternFitness) algorithm.getFitness()).setTarget(target);

((PatternGA.PatternFitness) algorithm.getFitness()).setPrecision(precision);

algorithm.evolve();

Population.Statistics stats = algorithm.~~getCurrentPopulation~~().getStatistics();

GeneticAlgorithm.Statistics algostats = algorithm.getStatistics();

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

System.***out***.print("Target:[");

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

System.***out***.print(target[i]+ ( i < target.length - 1 ? " " : ""));

}

System.***out***.println("]");

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

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

System.***out***.println(stats.getGroup(Population.***LEGALS***).get(0));

System.***out***.format("found in %d ms and %d generations.\n", algostats.getExecutionTime(), algostats.getGenerations() );

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

}

**public** **void** onGeneration(GeneticAlgorithm ga, **long** time) {

Statistics stat = ga.~~getCurrentPopulation~~().getStatistics();

Group legals = stat.getGroup(Population.***LEGALS***);

System.***out***.println("Current generation: " + ga.getGeneration());

System.***out***.println("\tBest score: " + legals.getMin()[0]);

System.***out***.println("\tAvg score : " + legals.getMean()[0]);

}

**private** **static** **void** randomize(**int**[] sample) {

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

sample[i] = Random.*getInstance*().nextInt(0, *MAX\_INT*+1);

}

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

Utils.*printHeader*();

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

System.***out***.println("TUTORIAL 2:");

System.***out***.println("This algorithm aims to autonomously find a vector of integers that best matches with a target vector.");

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

Random.*getInstance*().setStandardSeed();

PatternProblem problem = **new** PatternProblem();

**int**[] target = **new** **int**[*CHROMOSOME\_LENGTH*];

*randomize*(target);

problem.run(target, 2);

*randomize*(target);

problem.run(target, 0);

}

}

**Клас PatternGA:**

**package** jenes.tutorials.problem2;

**import** jenes.Fitness;

**import** jenes.GeneticAlgorithm;

**import** jenes.chromosome.IntegerChromosome;

**import** jenes.population.Individual;

**import** jenes.population.Population;

**public** **class** PatternGA **extends** GeneticAlgorithm<IntegerChromosome> {

**private** PatternFitness fitness = **new** PatternFitness();

**public** PatternGA(Population<IntegerChromosome> pop, **int** numGen) {

**super**(pop, numGen);

**this**.setFitness(fitness);

}

@Override

**protected** **boolean** end() {

jenes.population.Population.Statistics stat = **this**.~~getCurrentPopulation~~().getStatistics();

**return** stat.getGroup(Population.***LEGALS***).getMin()[0] <= **this**.fitness.precision;

}

**public** **class** PatternFitness **extends** Fitness<IntegerChromosome> {

**private** **int**[] target = **null**;

**private** **int** precision = 0;

**private** PatternFitness() {

**super**(**false**);

}

@Override

**public** **void** evaluate(Individual<IntegerChromosome> individual) {

IntegerChromosome chrom = individual.getChromosome();

**int** diff = 0;

**int** length = chrom.length();

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

diff += Math.*abs*(chrom.getValue(i) - target[i]);

}

individual.setScore(diff);

}

**public** **void** setTarget(**int**[] target) {

**this**.target = target;

}

**public** **void** setPrecision(**int** precision) {

**this**.precision = precision;

}

}

}

**Клас SimpleDispenser:**

**package** jenes.tutorials.problem2;

**import** jenes.chromosome.Chromosome;

**import** jenes.population.Individual;

**import** jenes.population.Population;

**import** jenes.stage.ExclusiveDispenser;

**public** **class** SimpleDispenser<T **extends** Chromosome> **extends** ExclusiveDispenser<T> {

**private** **int** count;

**public** SimpleDispenser(**int** span) {

**super**(span);

}

**public** **void** preDistribute(Population<T> population){

**this**.count = 0;

}

@Override

**public** **int** distribute(Individual<T> ind) {

**return** count++ % span;

}

}

**Результат виконання:**

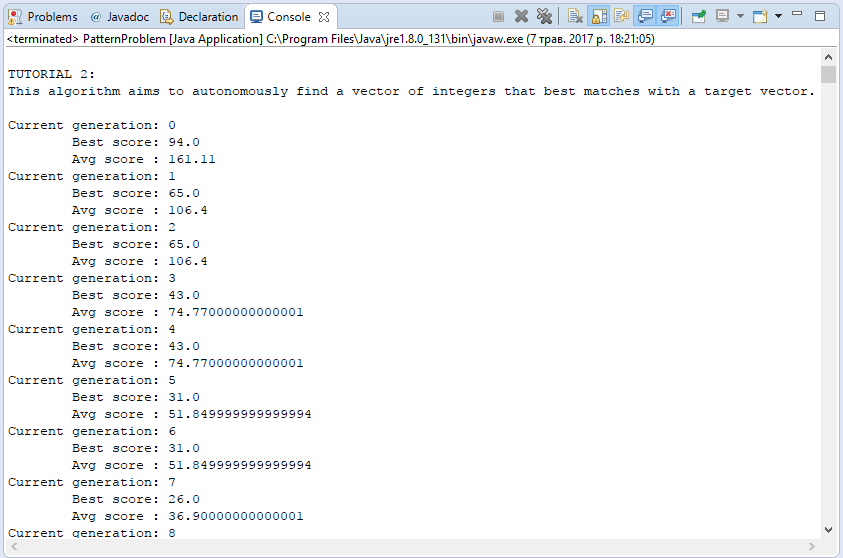


Рис. 3 Результат запуску програми до Прикладу 2

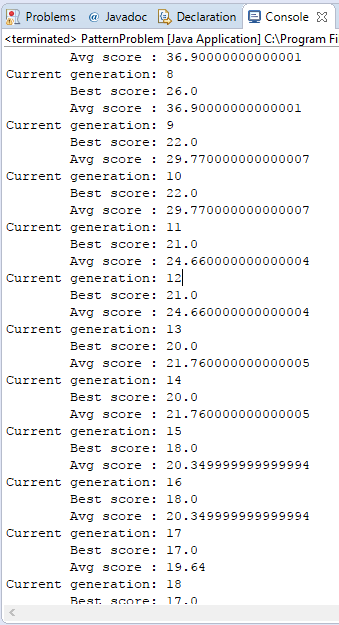
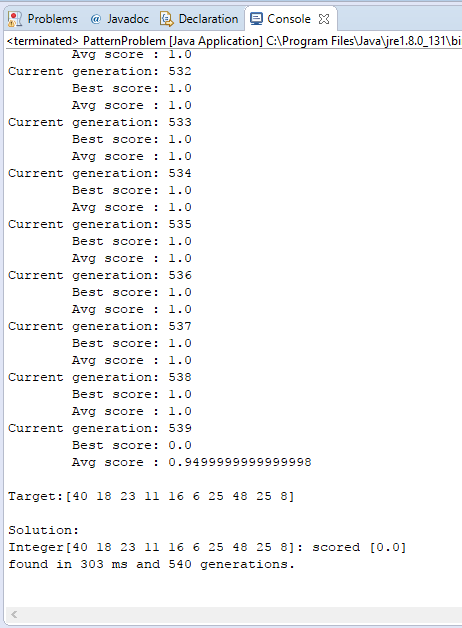
 

Рис. 4 Результат запуску програми до Прикладу 2

**Приклад 3**

Проблема завдання полягає в тому, щоб знайти вектор ентропії, після нормалізації. Мета полягає в мінімізації ентропії Шеннона, набору значень в діапазоні [0,1].

**Програма:**

**Клас EntropyProblem:**

**package** jenes.tutorials.problem3;

**import** jenes.GeneticAlgorithm;

**import** jenes.algorithms.SimpleGA;

**import** jenes.chromosome.DoubleChromosome;

**import** jenes.population.Individual;

**import** jenes.population.Population;

**import** jenes.population.Population.Statistics;

**import** jenes.population.Population.Statistics.Group;

**import** jenes.tutorials.utils.Utils;

**public** **class** EntropyProblem {

**private** **static** **int** *POPULATION\_SIZE* = 100;

**private** **static** **int** *CHROMOSOME\_LENGTH* = 5;

**private** **static** **int** *GENERATION\_LIMIT* = 100;

**private** **static** GeneticAlgorithm<DoubleChromosome> *ga*;

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

Utils.*printHeader*();

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

System.***out***.println("TUTORIAL 4:");

System.***out***.println("Find the probability distribution that maximizes (or minimize) the Shannon's entropy.");

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

Individual<DoubleChromosome> sample = **new** Individual<DoubleChromosome>(**new** DoubleChromosome(*CHROMOSOME\_LENGTH*,0,1));

Population<DoubleChromosome> pop = **new** Population<DoubleChromosome>(sample, *POPULATION\_SIZE*);

*ga* = **new** SimpleGA<DoubleChromosome>(**null**, pop, *GENERATION\_LIMIT*);

System.***out***.println("Solving Max!:");

*solve*( EntropyFitness.*MAX* );

System.***out***.println("Solving Min!:");

*solve*( EntropyFitness.*MIN* );

}

**private** **static** **void** solve(EntropyFitness fitness) {

*ga*.setFitness(fitness);

*ga*.evolve();

Statistics stats = *ga*.~~getCurrentPopulation~~().getStatistics();

GeneticAlgorithm.Statistics algostats = *ga*.getStatistics();

Group legals = stats.getGroup(Population.***LEGALS***);

System.***out***.println(legals.get(0));

System.***out***.format("found in %d ms.\n", algostats.getExecutionTime() );

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

Utils.*printStatistics*(stats);

}

}

**Клас EntropyFitness:**

**package** jenes.tutorials.problem3;

**import** jenes.population.Fitness;

**import** jenes.chromosome.DoubleChromosome;

**import** jenes.population.Individual;

**public** **class** EntropyFitness **extends** Fitness<DoubleChromosome> {

**public** **static** EntropyFitness *MAX* = **new** EntropyFitness(**true**);

**public** **static** EntropyFitness *MIN* = **new** EntropyFitness(**false**);

**private** EntropyFitness(**boolean** maximize) {

**super**(maximize);

}

@Override

**public** **void** evaluate(Individual<DoubleChromosome> individual) {

DoubleChromosome chrom = individual.getChromosome();

**int** length = chrom.length();

**double** sum = 0.0;

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

sum += chrom.getValue(i);

}

**double** entropy = 0.0;

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

**double** pxi = chrom.getValue(i) / sum;

chrom.setValue(i, pxi);

entropy -= (pxi \* Math.*log*(pxi) / Math.*log*(2));

}

individual.setScore(entropy);

}

}

**Результат виконання:**

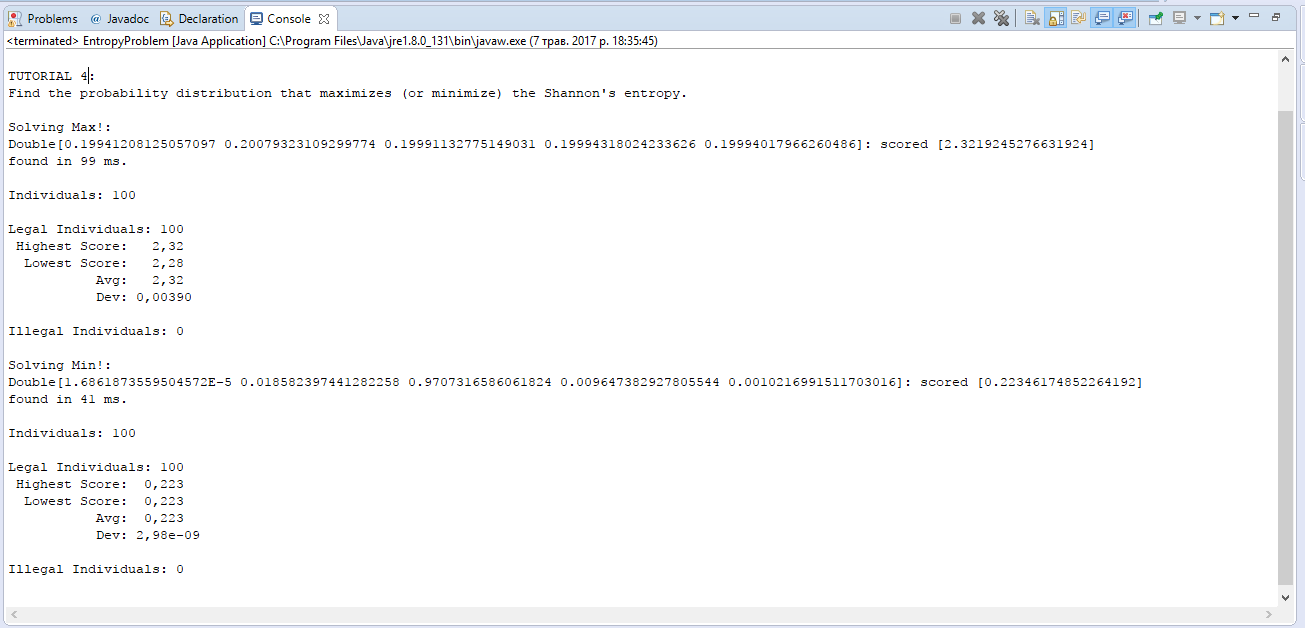


Рис. 5 Результат запуску програми до Прикладу 3

1. **Висновок**

Ознайомилася із основними поняттями при роботі з фреймворком Jenes 2.0, навчилася практично застосовувати даний фреймворк при виконанні тестових завдання.