|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САПР** | | Тема | оцінка | підпис |
| СПКс-11 | РГР | JGAP Framework |  |  |
| Тураш Ю.Ю. | |
| № залікової: 1508500 | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р.З. | |

ЗМІСТ

[Вступ 2](#_Toc452679867)

[Що таке JGAP? 2](#_Toc452679868)

[Інсталяція JGAP 4](#_Toc452679869)

[Встановлення бібліотеки 4](#_Toc452679870)

[Запуск прикладів 4](#_Toc452679871)

[Приклад вирішення задачі комівояжера за допомогою JGAP 5](#_Toc452679872)

[Приклад використання JGap у платформі Robocode 8](#_Toc452679873)

[Як Robocode запускає роботів? 12](#_Toc452679874)

[Висновок 16](#_Toc452679875)

# Вступ

Для розробки генетичних алгоритмів для цілей розподілених обчислень було створено безліч бібліотек на різних мовах програмування, в Java вони представлені бібліотеками JGAP (Java Genetic Algorithms and genetic programming package), ECJ (одна з найпоширеніших бібліотек, однак не підтримує простого кодування різнотипних параметрів в одній хромосомі ), Watchmaker та EvoJ.

Найгнучкішою є бібліотека JGAP, яка дозволяє адаптувати алгоритм під конкретну задачу оптимізації. Одним з найбільш зручних інструментів бібліотеки є спеціальний генотип MapGene. Він дозволяє визначити фіксований набір параметрів, з якого при схрещуванні може бути обраний якийсь один. Також присутній механізм додавання функцій в гени, що дозволяє зіставляти значення генів параметрам воркфлоу. Інша особливість фреймворка проявляється в способі проведення схрещування: популяція підзавдань воркфлоу-проекту спочатку оцінюється по функції якості з метою визначення найбільш вигідної хромосоми, потім відбувається формування нової популяції на основі заданого способу селекції.

Далі до батьків застосовуються оператори схрещування і мутації і вони додаються в нову популяцію. Для заповнення популяції до необхідного розміру використовується стратегія додавання випадкових хромосом (random-immigrant strategy), що дозволяє збалансувати співвідношення перевірених і нових елементів для досягнення максимальної ефективності.

Перевагою генетичних алгоритмів є висока оптимальність отримуваного рішення і відносно висока швидкість роботи, завдяки можливості розпаралелювання процесу.

# Що таке JGAP?

Бібліотека JGAP призначена для моделювання основних генетичних механізмів і може використовуватися для вирішення завдань еволюційними методами.

Ключовою ланкою ГА в JGAP є поняття «хромосома» (тип «Chromosome»). Хромосома кодує можливе рішення і складається з безлічі генів (тип «Gene»). У бібліотеці JGAP різні аспекти вирішення завдання видаються як єдине ціле (приблизно так само, як людські гени представляють різні ознаки людини, такі як колір волосся або очей). В процесі еволюції, що імітується в JGAP, хромосоми піддаються впливу генетичних операторів (схрещування, мутації і т.д.), а потім відбираються для створення наступного покоління відповідно до значень їх функцій пристосування (ФП). Процес еволюції в бібліотеці JGAP був змодельований так, щоб максимально нагадувати процеси, що відбуваються в живій природі, з метою отримання найкращого рішення задачі.

Для вирішення задачі за допомогою JGAP необхідно виконати ряд кроків:

**Крок 1.** Визначити, як буде представлятися рішення за допомогою послідовності генів, а також задати тип значення, яке можуть приймати об'єкти «Gene», тобто вибрати відповідний тип гена. Допускається використання різних типів генів в рамках однієї хромосоми.

**Крок 2.** Реалізувати ФП у вигляді програмного коду. Значення ФП має бути тим вище, чим вище якість виконання завдання. Для цього необхідно реалізувати підклас класу «FitnessFunction». У підкласі необхідно перевизначити метод «evaluate», що приймає в якості аргументу хромосому і повертає в якості результату число з плаваючою точкою (тип «double»).

**Крок 3.** Створити, конфігурувати і ініціалізувати об'єкти бібліотеки JGAP, а саме:

- Об'єкт класу Configuration - містить основні настройки ГА; базова реалізація, яка містить настройки за замовчуванням, надається класом DefaultConfiguration;

- Об'єкт, який реалізує ФП (підклас класу «FitnessFunction»);

- Зразок хромосоми (він повинен містити гени потрібних типів в правильному порядку).

Необхідно встановити і розмір популяції.

**Крок 4.** Створити початкову популяцію з хромосом, ініціалізованих випадковим чином. Для більшості завдань це оптимальний варіант створення початкової популяції.

**Крок 5.** Запустити процес еволюції. На кожній ітерації процесу можна контролювати, чи знайдено оптимальне рішення.

Особливістю бібліотеки JGAP є можливість розширення її функціональності. Існують наступні можливості для розширення: визначення власних типів генів і генетичних операторів, реалізація користувацького механізму відбору та ін.

Важлива особливість бібліотек JOONE і JGAP полягає в тому, що вони є вільними і можуть використовуватися без ліцензійних відрахувань. Їх вихідний код відкритий, що дозволяє краще зрозуміти роботу бібліотек.

# Інсталяція JGAP

JGAP це фреймворк написаний на Java. Це не виконуваний додаток, який можна просто запустити. Але є приклади що йдуть разом з JGAP, які можна виконати за допомогою командної строки (або batch-файла). Інструкція нижче призначена для Windows системи. Але так як JGAP написаний на Java, то його можна використати на будь-якій системі.

## Встановлення бібліотеки

1. Спочатку завантажуємо пакет JGAP з сайту sourceforge. Потрібно впевнетись що скачаний файл повної інсталяції (jgap\_xxx\_full.zip), а не просто вихідний кодя, який ще потрібно компілювати перед використанням.
2. Далі необхідно розпакувати файли в директорію на жосткому диску, наприклад C:\\jgap-current
3. Відкриваємо створену папку з роспакованими файлами
4. В даній папці повинні міститись файли з розширанням .jar. Ці файли містять скомпільований код JGAP і це все що нам потрібно. Потрібно зауважити що є також деякі залежності від зовнішніх бібліотек, що використовуються в JGAP. Усі ці jar-файли можна знайти в підпапці lib.

## Запуск прикладів

Windows: Запустити приклад виконавши batch-файл start\_example.bat.

Інші ОС: Йдемо в командний рядок і переходимо до робочої директорії (в нашому випадку jgap-current). Далі запускаємо приклад наступною командою:

java -classpath ".;jgap-examples.jar;jgap.jar" examples.MinimizingMakeChange 89

Ubuntu Linux: Як і попередньому випадку заходимо в командний рядок, міняємо директорію і виконуємо наступне (використовуємо двокрапку замість крапки з комою для розділення класів):

java -classpath ".:jgap-examples.jar:jgap.jar" examples.MinimizingMakeChange 89

У випадку використання класів що покладаються на зовнішні бібліотеки, вам необхідно додати ці jar-файли до шляху класу. Наприклад, якщо ми запускаємо клас, що використовує WeightedRouletteSelector клас з пакету org.jgap.impl то необхідно додати файл lib/trove-2.0.2.jar до шляху класу. Для прикоаду вище цього робити не потрібно!

От і все! В консолі повинно з’явитись щось таке:

The best solution has a fitness value of 1599.0

It contained the following:

3 quarters.

1 dimes.

0 nickels.

4 pennies.

For a total of 89 cents in 8 coins.

JGAP може бути скомпільований з Java версії 5 та вище.

Якщо потрібно написати свій власний генетичний алгоритм, використовуючи JGAP, хорошою стартовою очкою є приклади що поставляються з JGAP.

Пропонується поглянути на клас examples.simpleBoolean.SimpleExample. Спробуйте зрозуміти що в ньому відбувається, модифікувати та розширити код прикладу, запустити та побачити що відбудеться.

Щоб побачити навіть більше, що відбувається за сценою можна дослідити тестові випадки. Знайти їх можна в підпапці tests. Вивчення тестового коду дійсно допомагає краще зроуміти як працює JGAP.

# Приклад вирішення задачі комівояжера за допомогою JGAP

Нагадаємо, задача комівояжера (комівояжер — бродячий торговець; англ. Travelling Salesman Problem, TSP; нім. Problem des Handlungsreisenden) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів.

Фрейморк JGAP дозволяє швидко та просто вирішити цю задачу, використовуючи клас Salesman:

**public** **class** **TravellingSalesman** **extends** Salesman {

*/\*\* Чило міст по замовчуванню. \*/*

**public** **static** **final** int CITIES = 7;

*/\*\* Створюємо масив з заданим числом генів типу integer.*

*\* Перший ген завжди = 0*

*\* Тобто вказує на місце звідки наш комівояжер починає подорож*

*\*/*

**public** Chromosome createSampleChromosome(Object initial\_data) {

Gene [] genes = **new** Gene [CITIES];

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

genes [i] = **new** IntegerGene(0, CITIES-1);

genes [i] .setAllele( **new** Integer (i));

}

Chromosome sample = **new** Chromosome (genes);

System.out.println("Optimal way "+sample);

System.out.println("Score "+

(Integer.MAX\_VALUE/2-m\_conf.getFitnessFunction()

.getFitnessValue(sample)));

RandomGenerator g = **new** StockRandomGenerator();

shuffle (genes);

System.out.println("Sample chromosome "+sample);

System.out.println("Score "+

(Integer.MAX\_VALUE/2-m\_conf.getFitnessFunction()

.getFitnessValue(sample)));

**return** sample;

}

*/\*\* Довжина між містами дорівнює різниці їх номерівs,*

*\* окрім довжини між першим та останнім містом, яка*

*\* приймається з 1. Так ми можемо легко перевірити рішення*

*\* на оптимальність, бо оптимальним буде - 0 1 2 3 .. n.*

*\*/*

**public** double distance(Gene a\_from, Gene a\_to) {

IntegerGene a = (IntegerGene) a\_from;

IntegerGene b = (IntegerGene) a\_to;

int A = a.intValue();

int B = b.intValue();

**if** ( A == 0 && B == CITIES-1) **return** 1;

**if** ( B == 0 && A == CITIES-1) **return** 1;

**return** Math.abs( A - B );

}

*/\*\* Вирішуємо приклад з числом міст вказаним*

*\* в константі CITIES. Потім виводимо оптимальний маршрут,*

*\* а також довжину для йього маршруту*

*\*/*

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

**try** {

TravellingSalesman t = **new** TravellingSalesman ();

Chromosome optimal = t.findOptimalPath (**null**);

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

System.out.println(optimal);

System.out.println("Score "+

(Integer.MAX\_VALUE/2-optimal.getFitnessValue()));

}

**catch** (Exception ex) {

ex.printStackTrace();

}

}

}

# Приклад використання JGap у платформі Robocode

Robocode це платформа, а також гра для програмістів, створена з використанням фреймворка JGAP. Суть гри полягає у створенні роботів, які змагаються між собою, постійно еволюціонують та стають розумнішими!.

Створюємо робота та зберігаємо у директорію robots збілджений java-файл:

**import** **robocode.HitRobotEvent**;

**import** **robocode.Robot**;

**import** **robocode.ScannedRobotEvent**;

**import** **robocode.WinEvent**;

**import** **static** robocode.util.Utils.normalRelativeAngleDegrees;

**import** **java.awt.\***;

**public** **class** **Tracker** **extends** Robot {

int count = 0; *// просто лічильник, який рахує скільки «ходів»*

*// ми зробили*

double gunTurnAmt; *// коефіцієнт попороту гармати танка при пошуку цілі*

String trackName; *// ім’я цілі за якою ми слідкуємо*

*/\*\**

*\* run: основна функція, що керує поведінкою робота після запуску бою*

*\*/*

**public** void run() {

*// встановлюємо кольори робота, зброї, пуль тощо.*

setBodyColor(**new** Color(128, 128, 50));

setGunColor(**new** Color(50, 50, 20));

setRadarColor(**new** Color(200, 200, 70));

setScanColor(Color.white);

setBulletColor(Color.blue);

*// Підготовлюємо зброю*

trackName = **null**; *// Спочатку ми не маємо цілі, тому null*

setAdjustGunForRobotTurn(**true**); *// Keep the gun still when we turn*

gunTurnAmt = 10; *// будемо крутити гарматою кожен раз на 10 градусів*

*// бескінечний цикл життя робота*

**while** (**true**) {

*// повертаємо гармату при кожній ітерації (пошук супротивника)*

turnGunRight(gunTurnAmt);

*// зберігаємо число що показує як довго ми шукаємо*

count++;

*// якщо ми не побачили ціль після двох поворотів вправо – рухаємося вліво*

**if** (count > 2) {

gunTurnAmt = -10;

}

*// якщо ми досі не побачили ціль за 5 кроків, то знову повертаємося*

**if** (count > 5) {

gunTurnAmt = 10;

}

*// якщо ми досі не побачили ціль після 11 кроків то захоплюємо іншу ціль*

**if** (count > 11) {

trackName = **null**;

}

}

}

*/\*\**

*\* onScannedRobot: подія що відбувається постійно – сканування цілей*

*\*/*

**public** void onScannedRobot(ScannedRobotEvent e) {

*// If we have a target, and this isn't it, return immediately*

*// so we can get more ScannedRobotEvents.*

**if** (trackName != **null** && !e.getName().equals(trackName)) {

**return**;

}

*// Якщо в нас немає цілі, то берем випадкову*

**if** (trackName == **null**) {

trackName = e.getName();

out.println("Tracking " + trackName);

}

*// Після цього збиваємо лічильник часу*

count = 0;

*// Якщо ціль задалеко, то рухаємося до неї.*

**if** (e.getDistance() > 150) {

gunTurnAmt = normalRelativeAngleDegrees(e.getBearing() + (getHeading() - getRadarHeading()));

turnGunRight(gunTurnAmt);

turnRight(e.getBearing());

ahead(e.getDistance() - 140);

**return**;

}

*// Наша ціль близько – атакуєм!*

gunTurnAmt = normalRelativeAngleDegrees(e.getBearing() + (getHeading() - getRadarHeading()));

turnGunRight(gunTurnAmt);

fire(3);

*// Наша ціль занадто близька, відступаєм!*

**if** (e.getDistance() < 100) {

**if** (e.getBearing() > -90 && e.getBearing() <= 90) {

back(40);

} **else** {

ahead(40);

}

}

scan();

}

*/\*\**

*\* onHitRobot: якщо нас атакують – встановлюємо його як нову ціль*

*\*/*

**public** void onHitRobot(HitRobotEvent e) {

*// просто виведем чи це і так наша ціль.*

**if** (trackName != **null** && !trackName.equals(e.getName())) {

out.println("Tracking " + e.getName() + " due to collision");

}

*// берем як ціль*

trackName = e.getName();

*// трохи відступаємо і стріляємо.*

gunTurnAmt = normalRelativeAngleDegrees(e.getBearing() + (getHeading() - getRadarHeading()));

turnGunRight(gunTurnAmt);

fire(3);

back(50);

}

*/\*\**

*\* onWin: якщо перемогли – танцюємо(крутимося)!*

*\*/*

**public** void onWin(WinEvent e) {

**for** (int i = 0; i < 50; i++) {

turnRight(30);

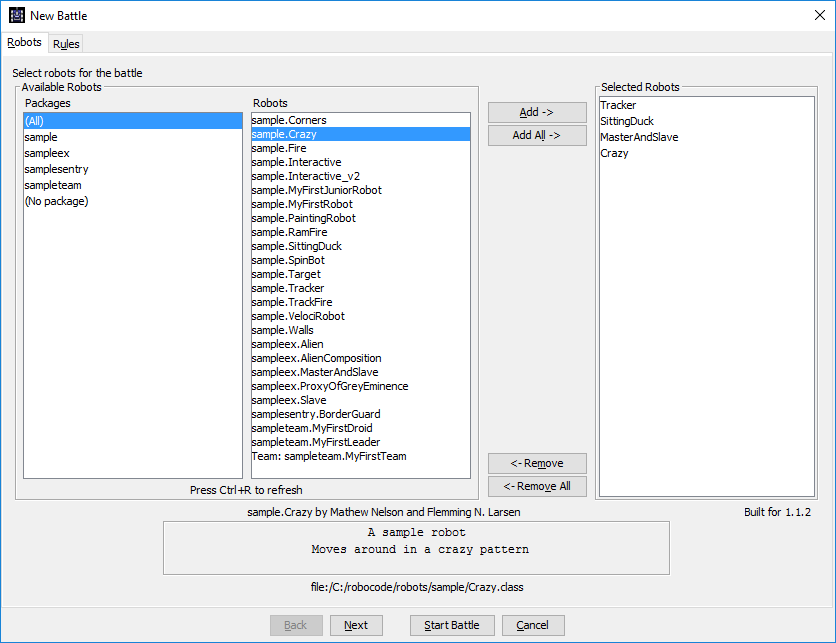
turnLeft(30);

}

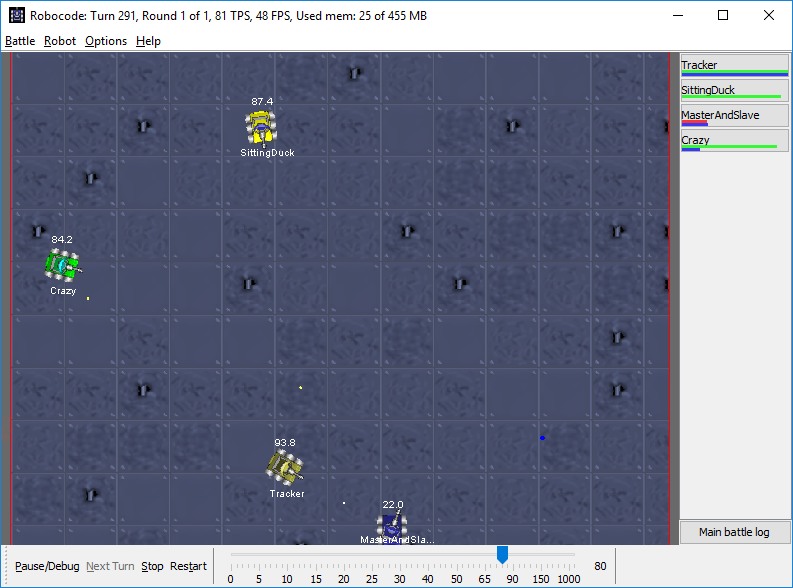
}

}

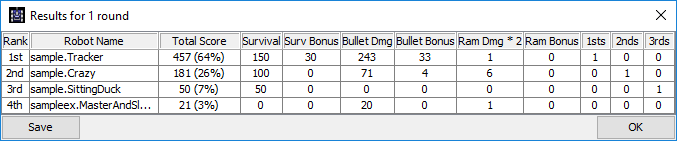
Вибираємо нашого робота та його опонентів:



Запускаємо битву:



Після запуску можна змінювати швидкість бою. Після закінчення бій починається спочатку. Роботи постійно навчаються та самовдосконалюються. В кінці кожного раунду можемо глянути статистику:



# Як Robocode запускає роботів?

Під капотом у Robocode використовується фреймворк JGAP і добре знайомі нам генетичні алгоритми з використанням селекції, мутації та схрещування:

**import** **org.jgap.\***;

**import** **org.jgap.impl.\***;

**import** **robocode.control.BattleSpecification**;

**import** **robocode.control.BattlefieldSpecification**;

**import** **robocode.control.RobocodeEngine**;

**import** **robocode.control.RobotSpecification**;

@SuppressWarnings("serial")

**public** **class** **robocodeGA** **extends** FitnessFunction{

*// встановлюємо к-ть поколінь яку проживе наш робот*

**public** **static** **final** int MAX\_GENERATIONS = 10;

*// розмір популяції*

**public** **static** **final** int POPULATION\_SIZE = 10;

*// к-ть хромосом*

**public** **static** **final** int CHROMOSOME\_AMOUNT = 1;

*// слікуємо за рахунком*

**public** **static** int robotScore,enemyScore;

**public** void run() **throws** Exception {

Configuration conf = **new** DefaultConfiguration(); *// ініціалізуємо класичний ГА з налаштуваннями по замовчуванню*

conf.addGeneticOperator(**new** MutationOperator(conf, 10)); *//ця функція дозволяє додати генетичний оператор що керуєеволюцією, це може бути оператор схрещування, мутації чи селекції. Допустиме створення користувацьких операторів*

conf.setPreservFittestIndividual(**true**); *// визначає, чи слід зберігати (тримати) найбільш пристосовану особину*

conf.setFitnessFunction(**this**); *// встановлюємо fitness-функцію*

*//встановлюємо гени для прикладу*

Gene[] sampleGenes = **new** Gene[ CHROMOSOME\_AMOUNT ];

sampleGenes[0] = **new** DoubleGene(conf, 300, 600 );

*/\*sampleGenes[1] = new DoubleGene(conf,-200,200)\*/*

IChromosome sampleChromosome = **new** Chromosome(conf, sampleGenes); *// створюєм хромосому із генів*

conf.setSampleChromosome(sampleChromosome); *// set chromo to conf*

conf.setPopulationSize(POPULATION\_SIZE); *// create a population*

*//генерим випадкову популяцію*

Genotype population = Genotype.randomInitialGenotype(conf);

IChromosome fittestSolution = **null**;

*//розвиваємо її*

**for** ( int gen = 0; gen<MAX\_GENERATIONS; gen++ ) {

population.evolve(); *// розвиваємо*

fittestSolution = population.getFittestChromosome(); *// пошук розв’язку*

*//вивід*

System.out.printf("\nafter %d generations the best solution is %s \n",gen + 1,fittestSolution);

}

buildRobot(fittestSolution);

System.exit(0);

}

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

**new** robocodeGA().run(); *// запуск фреймворку*

}

//отримання результатів битви

**public** boolean battleResults(String name,int score){

String same = "custom.SamBot\*"; *// ім’я робота для якого хочемо отримати результат*

*//get results of battle*

**if**(name.equals(same)){

robotScore = score;

**return** **true**;

}

**else** {

enemyScore = score;

**return** **false**;

}

}

//побудова робота на основі хромосоми

**private** void buildRobot (IChromosome chromosome) {

int i = 0;

*//розбиваємо хромосому на масив*

double[ ] chromo = **new** double[ CHROMOSOME\_AMOUNT ];

**for** ( Gene gene : chromosome.getGenes() ) {

chromo[i] += (Double) gene.getAllele(); *// берем значення з гену*

i++;

}

createRobot.create(chromo); *// викликаємо створення робота із createRobot.java*

}

//наша фітнес функція для оцінки особини

**protected** double evaluate( IChromosome chromosome) {

int fitness,

numberOfRounds = 3;

buildRobot(chromosome); *// створюємо робота на основі даної хромосимиt*

RobocodeEngine engine = **new** RobocodeEngine(**new** java.io.File("")); *// сам Robocode який запускає роботів і керує подальшим процесом*

engine.addBattleListener(**new** battleObserver()); *// дозволяє додати кастомні класи, що реагують на івенти під час битви, в даному випадку клас, що зберігає статистику*

engine.setVisible(**true**); *// чи показувати битву в графічному вікні*

BattlefieldSpecification battlefield = **new** BattlefieldSpecification(800, 600); *// розмір поля бою*

RobotSpecification[] selectedRobots = engine.getLocalRepository("sample.VelociRobot,sample.RamFire,sample.Fire,sample.Crazy,custom.SamBot\*"); *// вказуємо типи класів роботів, що братимуть участь в битві*

BattleSpecification battleSpec = **new** BattleSpecification(numberOfRounds, battlefield, selectedRobots);

engine.runBattle(battleSpec, **true**); *// запускаємо битву із вказаними налаштуваннями; другий параметр вказує чи чекати завершення битви*

engine.close(); *// clean up engine*

fitness = robotScore; *// отримуємо оцінку нашого робота з поточного покоління*

**return** fitness > 0 ? fitness : 0; *// повертаємо оцінку якщо > 0*

}

**public** void sortScore(int roboScore,int enemScore){

robotScore = roboScore;

enemyScore = enemScore;

}

}

А так відбувається створення роботів із файлів що ми програмували вище:

**import** **javax.tools.\***;

**import** **java.io.\***;

*/\**

*\* Функція створення роботів*

*\**

*\*/*

**public** **class** **createRobot** {

**public** **static** void create(double[] chromo) {

createRobotFile(chromo); *// створюємо файл робота з хромосоми*

compile(); *// компілюємо його*

}

**public** **static** void compile () {

String fileToCompile = "robots/custom/SamBot.java"; *// файл робота*

JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

compiler.run(**null**, **null**, **null**, fileToCompile); *// компілюємо*

}

**public** **static** void createRobotFile(double[] chromo){

**try** {

FileWriter fstream = **new** FileWriter("robots/custom/SamBot.java"); *// шлях до файлу для створення*

BufferedWriter out = **new** BufferedWriter(fstream);

*//початковий код*

out.write("package custom; \n " +

"import robocode.\*; \n" +

"import robocode.Robot;\n" +

"import java.awt.Color;\n" +

"public class SamBot extends Robot {\n");

*// записуємо значення генів хромосоми*

out.append("int foo = (int)" + chromo[0] + ";\n");

out.append("\n}");

out.close(); *// все*

} **catch** (Exception e){*//обробка помилки*

System.err.println("Error: " + e.getMessage());

}

}

}

І вкінці битви виводимо меседж про закінчення та статистику:

**import** **robocode.control.events.\***;

*/\**

*\* Клас для трекання подій під час битви і логування*

*\*/*

**class** **battleObserver** **extends** BattleAdaptor {

*// Викликається коли битва закінчена*

**public** void onBattleCompleted(BattleCompletedEvent e) {

int robotScore =0,

enemyScore = 0;

robocodeGA robot = **new** robocodeGA();

System.out.println("-- Battle has completed --");

*//виводимо посортовані результати та імена роботів*

System.out.println("Battle results:");

**for** (robocode.BattleResults result : e.getSortedResults()) {

**if**(robot.battleResults(result.getTeamLeaderName(),result.getScore())){

robotScore = result.getScore();

}

**else** {

enemyScore = result.getScore();

}

}

robot.sortScore(robotScore,enemyScore); *// sort score - func in robocodeGA.java*

}

*// викликається коли фреймворк посилає інформаційний меседж під час битви*

**public** void onBattleMessage(BattleMessageEvent e) {

System.out.println("Msg> " + e.getMessage());

}

*// Викликається коли під час битви сталася помилка – виводимо її*

**public** void onBattleError(BattleErrorEvent e) {

System.err.println("Err> " + e.getError());

}

}

# Висновок

Отже JGAP не створений для того щоб точно копіювати природу, а лише для використання та створення генетичних алгоритмів. Його перевагами є простота використання, можливості та продуктивність роботи (час, пам’ять, точність). Одним з прикладів чим відрізняється JGAP від інших фреймворків є Genotype.evolve() метод. В JGAP не використовується пул кандидатів хромосом так як було визначено, що він не впливає на точність, але уповільнює роботу алгоритму.

На противагу даному фреймворку можна поставити EvoJ. У jGap хромосому потрібно формувати руками описуючи дивними дескрипторами змінні і вибудовуючи їх в лінійну структуру, запам'ятовуючи при цьому на якому місці в хромосомі яка змінна знаходиться. Це дуже незручно, особливо коли ти займаєшся рефакторингом - додаєш / видаляєш змінні, від цього зміщуються всі інші і потрібно бігати по коду і правити індекси.  
У EvoJ рішення описується природним Java-інтерфейсом з невеликою допомогою анотацій. У разі рефакторінга про все подбає фреймворк і IDE.

Плюс до цього мутацією можна управляти декларативно - можна на рівні анотацій вказати області допустимих значень змінних, радіус і ймовірність мутації.