|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САП** | | Тема | оцінка | підпис |
| СПКс-11 | РГР | JGAP |  |  |
| Карпляк Р. В. | |
| № залікової: | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р.З. | |

**Мета роботи:**

Ознайомитись зі можливостями JGAP. Навчитись використовувати цей програмний продукт для вирішення певних задач.

**ОГЛЯД ПРОГРАМНОГО ПРОДУКТУ JGAP**

**JGAP** (вимовляється "jay-gap") – це Java-фреймворк для програмування генетичних алгоритми. Він забезпечує основні генетичні механізми, які можуть бути легко використані для застосування еволюційних принципів до вирішення завдань.

JGAP був розроблений, щоб бути дуже простим у використанні "з коробки", в той же час він є модульним, тому більш підприємливі користувачі можуть легко підключити свої власні генетичні оператори та інші суб-компоненти.

**Що таке генетичні алгоритми і генетичне програмування?**

*Генетичні алгоритми (ГА)* – це еволюційні алгоритми, що використовують принцип природного відбору для знаходження набору оптимальних рішень. ГА не тільки досить потужні, але також дуже прості у використанні, як більша частина роботи може бути втілена в одному компоненті, вимагаючи від користувачів тільки визначити фітнес функцію, яка використовується для визначення "кращого" рішення щодо інших рішень. Генетичні програми (ГП) розширюють генетичні алгоритми. Вони дозволяють розводити динамічні програми замість статичних хромосом.

**Встановлення JGAP**

JGAP це фреймворк, написаний на Java. Це не додаток, який можна запустити як виконуваний файл. Але є приклад, наведений у JGAP, який можна запустити за допомогою командного рядка (або пакетного файлу, переважно). JGAP це частина Java і тому запускається на різних системах!

*Щоб встановити JGAP, потрібно:*

1.Завантажити пакет JGAP з SourceForge сайту. Можна використовувати це посилання, щоб потрапити в список доступних JGAP файлів. Перед завантаженням потрібно переконатися, що ви отримаєте повну установку (jgap\_xxx\_full.zip)Якщо ви знаєте, як компілювати Java проети, то потрібно просто це зробити.

2.Після цього, витягти файли в директорію за вибором, краще в каталог з ім'ям jgap десь на жорсткому диску.

3.Відкрити каталог у файловому менеджері, напрклад Windows Explorer, для систем Windows, або інший для \* NIX систем.

4.У каталозі, в який завантажено JGAP і який відкрито в провіднику, повинні бути деякі файли, що закінчуються .jar. Ці файли містять скомпільований JGAP-дистрибутив і це все, що нам потрібно.

*Запуск тестового прикладу*

5.Запустіть приклад, виконавши пакетний файл start\_example.bat.

6.У випадку, якщо ви використовуєте класи, які покладаються на зовнішніх бібліотек, ви повинні додати ці файли до класів на попередньому кроці п'ять. Наприклад, якщо ви запустите клас, який використовує клас WeightedRouletteSelector з пакету org.jgap.impl вам доведеться додати файл Lib / клад-2.0.2.jar до класів.

На консолі повинно з'явитися щось на зразок (приклад, ваш вихід може змінюватися в залежності від генетичного алгоритму):

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.

**Дослідження прикладів**

JGAP може бути скомпільований з Java версії 5 або пізнішої версії. Якщо хочете використовувати Java 1.4, ви повинні внести декілька змін.  
Якщо ви хочете закодувати свій генетичний алгоритм за допомогою JGAP, то хорошою відправною точка в цьому будуть приклади, забезпечені JGAP.  
Дослідження тестових шаблонів дійсно може допомогти зрозуміти, як JGAP повинен працювати.

**Початок роботи з JGAP**

Щоб використовувати JGAP потрібно виконати п'ять основних речей:  
  
     1. Спланувати свої хромосоми.  
     2. Реалізувати "фітнес функцію".  
     3. Налаштувати конфігурації об'єкта.  
     4. Створити популяцію можливих рішень.  
     5. Розвивати популяцію!

*Крок 1: планування хромосом*

У центрі генетичного алгоритму міститься хромосома. Хромосома представляє потенційне рішення і розділена на декілька генів. Гени в JGAP представляють різні аспекти рішення в цілому, як гени людини представляють різні аспекти окремих людей, таких, як їх статі або кольору очей. У процесі еволюції JGAP, хромосоми піддаються численним генетичним операціям, які представляють спарювання, мутації і т.д., а потім вибирають для наступного покоління під час стихійного етапу відбору, заснованого на їх «придатності», яка є мірою того, наскільки оптимальне це рішення по відношенню до інших потенційних рішень. Вся мета генетичного алгоритму, щоб імітувати природний процес еволюції з метою отримання кращих рішень.

Отже, крок 1 - прийняти рішення про склад ваших хромосом, які включають в себе, скільки генів, скільки ви хочете і що ці гени представляють. У нашому прикладі програми ми хочемо створити купу змін, що містять найменшу кількість можливих американських монет, які є в сумі, зазначеній користувачем. Так хромосома являє собою потенційне рішення, у нашій вибірці вона буде являти собою купу змін. Наші гени представляють різні номінали монет, так що ми будемо мати в цілому чотири гени на хромосому.

### *Крок 2. реалізація* *Fitness – функції*

JGAP побудований так , щоб всю роботу користувач проводив у відносно загальному вигляді. Тим не менш, фреймворк не знає про конкретну проблему, яку ви насправді намагаєтеся вирішити. Тому fitness-функція - це єдиний метод, який необхідно реалізувати, який приймає потенційне рішення проблеми, і повертає ціле значення, яке вказує, наскільки краще це рішення (або "відповідає")щодо інших можливих рішень. Чим більше число, тим краще рішення.

Розглянемо повністю робочий приклад фітнес-функції, деталі якого будуть описані нижче.

**package** examples;

**import** org.jgap.Chromosome;

**import** org.jgap.FitnessFunction;

*/\*\**

*\* This class provides an implementation of the classic "Make change" problem*

*\* using a genetic algorithm. The goal of the problem is to provide a*

*\* specified amount of change (from a cash purchase) in the fewest coins*

*\* possible. This example implementation uses American currency (quarters,*

*\* dimes, nickels, and pennies).*

*\**

*\* This example may be seen as somewhat significant because it demonstrates*

*\* the use of a genetic algorithm in a less-than-optimal problem space.*

*\* The genetic algorithm does best when there is a smooth slope of fitness*

*\* over the problem space towards the optimum solution. This problem exhibits*

*\* a more choppy space with more local optima. However, as can be seen from*

*\* running this example, the genetic algorithm still will get the correct*

*\* answer virtually everytime.*

*\*/*

**public** **class** MinimizingMakeChangeFitnessFunction **extends** FitnessFunction

{

**private** **final** **int** m\_targetAmount;

*/\*\**

*\* Constructs this MinimizingMakeChangeFitnessFunction with the desired*

*\* amount of change to make.*

*\**

*\* @param a\_targetAmount The desired amount of change, in cents. This*

*\* value must be between 1 and 99 cents.*

*\*/*

**public** MinimizingMakeChangeFitnessFunction( **int** a\_targetAmount )

{

**if**( a\_targetAmount < 1 || a\_targetAmount > 99 )

{

**throw** **new** IllegalArgumentException(

**"Change amount must be between 1 and 99 cents."** );

}

m\_targetAmount = a\_targetAmount;

}

*/\*\**

*\* Determine the fitness of the given Chromosome instance. The higher the*

*\* return value, the more fit the instance. This method should always*

*\* return the same fitness value for two equivalent Chromosome instances.*

*\**

*\* @param a\_subject: The Chromosome instance to evaluate.*

*\**

*\* @return A positive integer reflecting the fitness rating of the given*

*\* Chromosome.*

*\*/*

**public** **double** evaluate( IChromosome a\_subject )

{

*// The fitness value measures both how close the value is to the*

*// target amount supplied by the user and the total number of coins*

*// represented by the solution. We do this in two steps: first,*

*// we consider only the represented amount of change vs. the target*

*// amount of change and calculate higher fitness values for amounts*

*// closer to the target, and lower fitness values for amounts further*

*// away from the target. If the amount equals the target, then we go*

*// to step 2, which adjusts the fitness to a higher value for*

*// solutions representing fewer total coins, and lower fitness*

*// values for solutions representing a larger total number of coins.*

*// ------------------------------------------------------------------*

**int** changeAmount = amountOfChange( a\_subject );

**int** totalCoins = getTotalNumberOfCoins( a\_subject );

**int** changeDifference = Math.abs( m\_targetAmount - changeAmount );

*// Step 1: Determine the distance of the amount represented by the*

*// solution from the target amount. Since we know the maximum amount*

*// of change is 99 cents, we'll subtract from that the difference*

*// between the solution amount and the target amount. That will give*

*// the desired effect of returning higher values for amounts close*

*// to the target amount and lower values for amounts further away*

*// from the target amount.*

*// ------------------------------------------------------------------*

**double** fitness = ( 99 - changeDifference );

*// Step 2: If the solution amount equals the target amount, then*

*// we add additional fitness points for solutions representing fewer*

*// total coins.*

*// -----------------------------------------------------------------*

**if**( changeAmount == m\_targetAmount )

{

fitness += 100 - ( 10 \* totalCoins );

}

**return** fitness;

}

*/\*\**

*\* Calculates the total amount of change (in cents) represented by*

*\* the given chromosome and returns that amount.*

*\**

*\* @param a\_potentialSolution The potential solution to evaluate.*

*\* @return The total amount of change (in cents) represented by the*

*\* given solution.*

*\*/*

**public** **static** **int** amountOfChange( IChromosome a\_potentialSolution )

{

**int** numQuarters = getNumberOfCoinsAtGene( a\_potentialSolution, 0 );

**int** numDimes = getNumberOfCoinsAtGene( a\_potentialSolution, 1 );

**int** numNickels = getNumberOfCoinsAtGene( a\_potentialSolution, 2 );

**int** numPennies = getNumberOfCoinsAtGene( a\_potentialSolution, 3 );

**return** ( numQuarters \* 25 ) + ( numDimes \* 10 ) + ( numNickels \* 5 ) +

numPennies;

}

*/\*\**

*\* Retrieves the number of coins represented by the given potential*

*\* solution at the given gene position.*

*\**

*\* @param a\_potentialSolution The potential solution to evaluate.*

*\* @param a\_position The gene position to evaluate.*

*\* @return the number of coins represented by the potential solution*

*\* at the given gene position.*

*\*/*

**public** **static** **int** getNumberOfCoinsAtGene( IChromosome a\_potentialSolution,

**int** a\_position )

{

Integer numCoins =

(Integer) a\_potentialSolution.getGene(a\_position).getAllele();

**return** numCoins.intValue();

}

*/\*\**

*\* Returns the total number of coins represented by all of the genes in*

*\* the given chromosome.*

*\**

*\* @param a\_potentialsolution The potential solution to evaluate.*

*\* @return The total number of coins represented by the given Chromosome.*

*\*/*

**public** **static** **int** getTotalNumberOfCoins( IChromosome a\_potentialsolution )

{

**int** totalCoins = 0;

**int** numberOfGenes = a\_potentialsolution.size();

**for**( **int** i = 0; i < numberOfGenes; i++ )

{

totalCoins += getNumberOfCoinsAtGene( a\_potentialsolution, i );

}

**return** totalCoins;

}

}

Розглянемо наш приклад фітнес-функції по кроках Для початку, ми визначаємо наш власний клас і розширюємо клас org.jgap.FitnessFunction. Всі фітнес -функції повинні розширити клас FitnessFunction. Потім ми визначаємо конструктор і метод evaluate (). Метод evaluate () є стандартним методом, що всі фітнес-функції повинні бути реалізованими. Це метод, який буде викликаний генетичним двигуном, коли це необхідно, щоб знати значення хромосоми.

*Крок 3: налаштування конфігурації об’єкта*

JGAP є дуже гнучким до змін. Якщо ви хочете, ви можете створити свої власний генетичні оператори, генератори випадкових чисел, природні селектори, і так далі. Для підтримки всього цього, JGAP використовує об'єкт конфігурації, який повинен бути встановлений з усіма потрібними установками перед використанням генетичного двигуна. Вам просто потрібно вказати три додаткових порції інформації: які функції придатності ви хочете використовувати, як ви хочете, щоб ваші хромосоми були налаштовані, і скільки хромосом ви хочете у вашій популяції. Давайте подивимося на деякі приклади коду.

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

*// Start with a DefaultConfiguration, which comes setup with the*

*// most common settings.*

*// -------------------------------------------------------------*

Configuration conf = **new** DefaultConfiguration();

*// Set the fitness function we want to use, which is our*

*// MinimizingMakeChangeFitnessFunction that we created earlier.*

*// We construct it with the target amount of change provided*

*// by the user.*

*// ------------------------------------------------------------*

int targetAmount = Integer.parseInt(args[0]);

FitnessFunction myFunc =

**new** MinimizingMakeChangeFitnessFunction( targetAmount );

conf.setFitnessFunction( myFunc );

*// Now we need to tell the Configuration object how we want our*

*// Chromosomes to be setup. We do that by actually creating a*

*// sample Chromosome and then setting it on the Configuration*

*// object. As mentioned earlier, we want our Chromosomes to*

*// each have four genes, one for each of the coin types. We*

*// want the values of those genes to be integers, which represent*

*// how many coins of that type we have. We therefore use the*

*// IntegerGene class to represent each of the genes. That class*

*// also lets us specify a lower and upper bound, which we set*

*// to sensible values for each coin type.*

*// --------------------------------------------------------------*

Gene[] sampleGenes = **new** Gene[ 4 ];

sampleGenes[0] = **new** IntegerGene(conf, 0, 3 ); *// Quarters*

sampleGenes[1] = **new** IntegerGene(conf, 0, 2 ); *// Dimes*

sampleGenes[2] = **new** IntegerGene(conf, 0, 1 ); *// Nickels*

sampleGenes[3] = **new** IntegerGene(conf, 0, 4 ); *// Pennies*

Chromosome sampleChromosome = **new** Chromosome(conf, sampleGenes );

conf.setSampleChromosome( sampleChromosome );

*// Finally, we need to tell the Configuration object how many*

*// Chromosomes we want in our population. The more Chromosomes,*

*// the larger the number of potential solutions (which is good*

*// for finding the answer), but the longer it will take to evolve*

*// the population each round. We'll set the population size to*

*// 500 here.*

*// --------------------------------------------------------------*

conf.setPopulationSize( 500 );

// TODO: Add the code following below in this example here

}

*Крок 4: створення популяції*

Кожне з можливих рішень представлене в хромосомі. Популяція хромосом називається генотипом, а це клас, який ми повинні побудувати, щоб створити нашу популяцію. Якщо ви хочете, ви можете побудувати кожну хромосому окремо, а потім передати їх усі в new Genotype (так само, як ми побудували кожен ген і передаємо їх в зразок хромосоми на кроці 3), але JGAP забезпечує набагато швидший і більш простий спосіб створення випадкової популяції. Насправді, це займе всього один рядок коду (Додати всі наступні рядки коду в коді вище в основний метод):

Genotype population = Genotype.randomInitialGenotype( conf );

*Крок 5: розвиток популяції*

Тепер, коли ми отримали всі установки і готові до роботи, прийшов час, щоб почати розвивати популяцію поки вона не містить деякі можливі рішення, якими ми будемо задоволені. Розвиток популяції:

population.evolve();

Як правило, після кожного еволюційного циклу, ви хочете перевіряти, чи популяція містить будь-які задовільні рішення. Найпростіший спосіб зробити це - викликати метод getFittestChromosome () на популяції:

IChromosome bestSolutionSoFar = population.getFittestChromosome();

Якщо краще рішення є досить хорошим для вас, то ми отримали результат. Якщо ні, то ви можете розвивати популяцію знову.

IChromosome bestSolutionSoFar;

**for**( **int** i = 0; i < MAX\_ALLOWED\_EVOLUTIONS; i++ )

{

population.evolve();

}

System.out.println( **"The best solution contained the following: "** );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 0 ) + **" quarters."** );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 1 ) + **" dimes."** );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 2 ) + **" nickels."** );

System.out.println(

MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(

bestSolutionSoFar, 3 ) + **" pennies."** );

System.out.println( **"For a total of "** +

MinimizingMakeChangeFitnessFunction.amountOfChange(

bestSolutionSoFar ) + **" cents in "** +

MinimizingMakeChangeFitnessFunction.getTotalNumberOfCoins(

bestSolutionSoFar ) + **" coins."** );

Таким чином, тепер у нас є повноцінний генетичний додаток!.

**Висновок:** під час виконання розрахунково-графічної роботи, я ознайомився з програмним продуктом JGAP, який є дуже зручним інструментом для розв’язання різного роду задач, використовуючи генетичні алгоритми. Програма має велику базу вже готових до використання алгоритмів, а також набір задач з стандартними налаштуваннями.