МІНІСТЕРСТВО НАУКИ І ОСВІТИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра СТ

Звіт

З лабораторної роботи №1

з дисципліни «Нейросистеми та генетичні алгоритми»

|  |  |
| --- | --- |
| Виконав:  ст. гр. КСУАм-15-1  Міщук В. В. | Перевірила:  Доц. каф. СТ  Імангулова З. А. |

Харків 2016

1 ОПТИМІЗАЦІЯ СКЛАДНИХ ФУНКЦІЙ ЗА ДОПОМОГОЮ ГЕНЕТИЧНИХ АЛГОРИТМІВ

1.1 Мета роботи

Вивчення принципів роботи генетичних алгоритмів. Набуття навичок реалізації генетичних алгоритмів для оптимізації складних багатопараметричних функцій.

1.2 Постановка задачі

Необхідно за допомогою генетичного алгоритму знайти мінімум функції (1.1).

Таблиця 1.1 – Індивідуальне завдання

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Варіант | Функція | Кроссовер | Вибір батьків | Механізм добору |
| 4 |  | Одноточковий | Аутбридинг | Елітний |

1.3 Хід роботи

Для вирішення задачі застосовано модифікований код Java-бібліотеки Genetics Framework (Додаток А). Для цього реалізовано клас MyFitnessFunction, який буде реалізувати інтерфейс FitnessFunction. В ньому задані методи для обчислення аргумента функції за комплектом генів заданої особи та для обчислення значення функції (1.1). Також проведено обов’язкове перевизначення двох методів з інтерфейсу, згідно з одним з них кодування хромосоми буде відбуватися за допомогою 8 бітів, а інший повертатиме значення функції приналежності (значення цільової функції мінімізації, помножене на деякий коефіцієнт підсилення).

**public** **class** MyFitnessFunction **implements** FitnessFunction {

@Override

**public** **int** getArity() {

**return** 16;

}

@Override

**public** **long** run(**long**[] genom) {

**return** (**long**) (***GAIN*** \* getFunction(genom));

}

**public** **double** getXOne(**long**[] genom) {

**double** xOne = ((**double**) (genom[0] & 0xFF00)) / 256;

xOne = xOne / 256;

**double** argument = xOne \* 6.0;

**return** argument;

}

**public** **double** getXTwo(**long**[] genom) {

**double** xTwo = ((**double**) (genom[0] & 0x00FF)) / 256;

**double** argument = xTwo \* 6.0;

**return** argument;

}

**public** **double** getFunction(**long**[] genom) {

**double** z = (getXOne(genom) - 3) \* (getXOne(genom) - 3);

z += (getXTwo(genom) - 3) \* (getXTwo(genom) - 3);

z = z / 18;

**double** functionValue = 2 \* z - 0.9;

functionValue \*= (7 \* z - 1);

functionValue \*= (17 \* z - 19);

functionValue \*= (15 \* z - 2);

**return** functionValue;

}

}

Далі передаємо даний клас в об’єкт, що реалізує функціонал генного алгоритму, після чого виставили необхідну кількість особин та генерацій, способи добору членів нової популяції та проведення схрещування, використання мутації та її імовірність.

**public** **class** Main {

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

MyFitnessFunction ff = **new** MyFitnessFunction();

GeneticEngine ge = **new** GeneticEngine(ff);

ge.setIndividualCount(100);

ge.setAccuracy(0.0); //set desired accuracy as double value

ge.setGenerationCount(40);

ge.setSelectionType(GeneticEngine.SelectionType.***TOURNEY***);

ge.setCrossingType(GeneticEngine.CrossingType.***ONE\_POINT\_RECOMBINATION***);

ge.setUseMutation(**true**);

ge.setMutationPercent(0.1d);

**long** time = System.*currentTimeMillis*();

**long**[] better = ge.run();

System.***out***.println("Running:\t" + ge.getIterations() + " iterations");

System.***out***.println((System.*currentTimeMillis*() - time) + " milliseconds");

System.***out***.println("Optimized value of x1: " + ff.getXOne(better) + " x2: " + ff.getXTwo(better));

System.***out***.println("Optimized function value: ");

System.***out***.println(ff.getFunction(better));

}

}

Результат роботи програми зображений на рисунку 1.1:

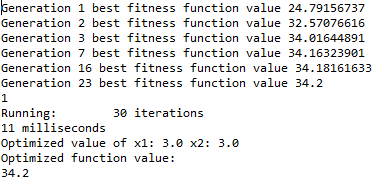


Рисунок 1.1 – Результати роботи генетичного алгоритму

ВИСНОВКИ

В ході лабораторної роботи вивчили принципи роботи генетичних алгоритмів. Вивчили схеми побудови генетичних алгоритмів. Набули практичних навичок реалізації генетичних алгоритмів для розв’язанні задач оптимізації.

ДОДАТОК А

Текст бібліотеки GeneticsFramework

**public** **interface** FitnessFunction {

**final** **double** ***GAIN*** = 100000000.0;

**int** getArity();

**long** run(**long**[] genom);

}

**public** **class** GeneticEngine {

**public** **long** timeToSelection = 0;

**public** **long** timeToCrossing = 0;

**public** **long** timeToMutate = 0;

**public** **long** timeToFF = 0;

**public** **static** **final** **double** ***CHANCE\_TO\_FULLNESS*** = 0.999d;

**public** **static** **final** SelectionType ***DEFAULT\_SELECTION\_TYPE*** = SelectionType.***TOURNEY***;

**public** **static** **final** CrossingType ***DEFAULT\_CROSSING\_TYPE*** = CrossingType.***ONE\_POINT\_RECOMBINATION***;

**public** **static** **final** **boolean** ***DEFAULT\_USE\_MUTATION*** = **true**;

**public** **static** **final** **long** ***DEFAULT\_GENERATION\_COUNT*** = 10000L;

**public** **static** **final** **int** ***OCTET\_LENGTH*** = 64; // for long

**public** **static** **final** **int** ***MASK\_FOR\_MOD*** = ***OCTET\_LENGTH*** - 1;

**public** **static** **final** **int** ***SHIFT\_FOR\_DIVISION***;

**static** {

**int** shiftForDivision = 0;

**int** tmp = ***OCTET\_LENGTH***;

**while** (tmp > 1) {

tmp >>= 1;

shiftForDivision++;

}

***SHIFT\_FOR\_DIVISION*** = shiftForDivision;

}

**public** **enum** SelectionType {

***TOURNEY***, ***ROULETTE\_WHEEL***

}

**public** **enum** CrossingType {

***ONE\_POINT\_RECOMBINATION***, ***TWO\_POINT\_RECOMBINATION***, ***ELEMENTWISE\_RECOMBINATION***, ***ONE\_ELEMENT\_EXCHANGE***

}

**private** FitnessFunction fitnessFunction;

**private** **int** genomLength;

**private** **int** sizeOfArray;

**private** **long** accuracy;

**private** **long** maxGenerationCount;

**private** **int** individualCount;

**private** SelectionType selectionType;

**private** CrossingType crossingType;

**private** **boolean** useMutation;

**private** **double** mutationPercent;

**private** **long**[][] genomListParents;

**private** **long**[][] genomListOffsprings;

**private** **long**[] actual;

**private** **long**[] fitnessFunctionResult;

**private** **long** currentGeneration = 0;

**private** **int** iterations = 0;

**public** **int** getIterations() {

**return** iterations;

}

**private** Random random = **new** Random(System.*currentTimeMillis*());

**public** GeneticEngine(FitnessFunction fitnessFunction) {

**this**.fitnessFunction = fitnessFunction;

**this**.genomLength = fitnessFunction.getArity();

**this**.sizeOfArray = (**int**) Math.*ceil*((**double**) **this**.genomLength / ***OCTET\_LENGTH***);

**this**.maxGenerationCount = ***DEFAULT\_GENERATION\_COUNT***;

**this**.individualCount = (**int**) (1

+ Math.*log*(1 / Math.*pow*(1 - ***CHANCE\_TO\_FULLNESS***, 1 / genomLength)) / Math.*log*(2));

**this**.selectionType = ***DEFAULT\_SELECTION\_TYPE***;

**this**.crossingType = ***DEFAULT\_CROSSING\_TYPE***;

**this**.useMutation = ***DEFAULT\_USE\_MUTATION***;

**this**.mutationPercent = genomLength

\* (1 - Math.*pow*((1 - 10 \* Math.*pow*((1 / 2),

(genomLength - 1))), (1 / genomLength)));

}

// Main loop

**public** **long**[] run() {

iterations = 0;

**long** prevBestFitnessFunction = Long.***MAX\_VALUE***;

// Preparing structuress

**this**.genomListParents = **new** **long**[**this**.individualCount][];

**this**.genomListOffsprings = **new** **long**[**this**.individualCount][];

**this**.fitnessFunctionResult = **new** **long**[**this**.individualCount];

**this**.actual = **new** **long**[**this**.individualCount];

**for** (**int** i = 0; i < **this**.individualCount; i++) {

**this**.actual[i] = -1;

}

**long**[] bestGenom = **null**;

// Generate 1st generation

**this**.generateFirstGeneration();

**while** (**this**.currentGeneration < **this**.maxGenerationCount) {

iterations++;

**this**.selection();

**this**.crossing();

**if** (**this**.useMutation) {

**this**.mutation();

}

**long**[][] tmp = **this**.genomListParents;

**this**.genomListParents = **this**.genomListOffsprings;

**this**.genomListOffsprings = tmp;

**this**.currentGeneration++;

**long** bestFitnessFunctionResult = Long.***MAX\_VALUE***;

**for** (**long**[] genom : **this**.genomListParents) {

**long** fitnessFunctionResult = **this**.fitnessFunction.run(genom);

**if** (bestFitnessFunctionResult >= fitnessFunctionResult) {

bestGenom = genom;

bestFitnessFunctionResult = fitnessFunctionResult;

}

}

**if** (Math.*abs*(bestFitnessFunctionResult - prevBestFitnessFunction) < accuracy){

**break**;

}

prevBestFitnessFunction = bestFitnessFunctionResult;

}

**return** bestGenom;

}

// Generate First Generation

**private** **void** generateFirstGeneration() {

**for** (**int** i = 0; i < **this**.individualCount; i++) {

**this**.genomListParents[i] = **this**.generateGenom();

}

}

// Generate Genom - Generate 1 genom

**private** **long**[] generateGenom() {

**long**[] result = **new** **long**[**this**.sizeOfArray];

**for** (**int** i = 0; i < **this**.sizeOfArray; i++) {

result[i] = **this**.random.nextLong();

}

**return** result;

}

// Selection - Select genoms for crossing

**private** **void** selection() {

**long** old = System.*currentTimeMillis*(); // time

**switch** (selectionType) {

**case** ***ROULETTE\_WHEEL***: {

**float**[] wheel = **new** **float**[**this**.individualCount];

wheel[0] = **this**.getFitnessFunctionResult(0);

**for** (**int** i = 1; i < **this**.individualCount; i++) {

wheel[i] = wheel[i - 1] + **this**.getFitnessFunctionResult(i);

}

**float** all = wheel[**this**.individualCount - 1];

**for** (**int** i = 0; i < **this**.individualCount; i++) {

**float** index = Math.*abs*(**this**.random.nextFloat()) \* all;

**int** l = 0;

**int** r = individualCount - 1;

**int** c = 0;

**while** (l != r) {

c = (l + r) >> 1;

**if** (wheel[c] < index) {

l = c;

} **else** {

r = c;

}

}

**this**.genomListOffsprings[i] = **this**.genomListParents[l].clone();

}

**break**;

}

**case** ***TOURNEY***: {

**for** (**int** i = 0; i < **this**.individualCount; i++) {

**int** index1 = random.nextInt(individualCount);

**int** index2 = random.nextInt(individualCount);

**long** ffTime = System.*currentTimeMillis*(); // time

**long** fr1 = **this**.getFitnessFunctionResult(index1);

**long** fr2 = **this**.getFitnessFunctionResult(index2);

**this**.timeToFF += (System.*currentTimeMillis*() - ffTime); // time

**this**.genomListOffsprings[i] = fr1 < fr2 ? **this**.genomListParents[index1].clone()

: **this**.genomListParents[index2].clone();

}

**break**;

}

**default**:

**throw** **new** UnsupportedOperationException();

}

**this**.timeToSelection += (System.*currentTimeMillis*() - old); // time

}

// Crossing - Crossing all genom in generation

**private** **void** crossing() {

**long** old = System.*currentTimeMillis*(); // time

**for** (**int** i = 0; i < individualCount / 2; i++) {

**int** index1 = i << 1;

**int** index2 = index1 | 1;

cross(**this**.genomListOffsprings[index1], **this**.genomListOffsprings[index2]);

}

**this**.timeToCrossing += (System.*currentTimeMillis*() - old); // time

}

// Get Fitness Function Result [with cache]

**private** **long** getFitnessFunctionResult(**int** genomNumber) {

**if** (**this**.actual[genomNumber] != **this**.currentGeneration) {

**this**.fitnessFunctionResult[genomNumber] = **this**.fitnessFunction.run(**this**.genomListParents[genomNumber]);

**this**.actual[genomNumber] = **this**.currentGeneration;

}

**return** **this**.fitnessFunctionResult[genomNumber];

}

// Cross - Crossing 2 genom

**private** **void** cross(**long**[] genom1, **long**[] genom2) {

**switch** (crossingType) {

**case** ***ONE\_POINT\_RECOMBINATION***: {

**int** index = **this**.random.nextInt(**this**.genomLength);

**int** outerOffset = index >> ***SHIFT\_FOR\_DIVISION***;

**int** innerOffset = ***OCTET\_LENGTH*** - (index & ***MASK\_FOR\_MOD***);

**long** tmp = 0;

**if** (innerOffset < 63) {

**long** mask = 1L << (innerOffset + 1) - 1;

**long** swapMask = (genom1[outerOffset] ^ genom2[outerOffset]) & mask;

genom1[outerOffset] ^= swapMask;

genom2[outerOffset] ^= swapMask;

outerOffset++;

}

**for** (**int** i = outerOffset; i < **this**.sizeOfArray; i++) {

tmp = genom1[i];

genom1[i] = genom2[i];

genom2[i] = tmp;

}

**break**;

}

**case** ***TWO\_POINT\_RECOMBINATION***: {

**int** index1 = **this**.random.nextInt(**this**.genomLength);

**int** index2 = **this**.random.nextInt(**this**.genomLength);

**int** startIndex = Math.*min*(index1, index2);

**int** endIndex = Math.*max*(index1, index2);

**int** startOuterOffset = startIndex >> ***SHIFT\_FOR\_DIVISION***;

**int** startInnerOffset = ***OCTET\_LENGTH*** - (startIndex & ***MASK\_FOR\_MOD***);

**int** endOuterOffset = endIndex >> ***SHIFT\_FOR\_DIVISION***;

**int** endInnerOffset = ***OCTET\_LENGTH*** - (endIndex & ***MASK\_FOR\_MOD***);

**long** tmp = 0;

**if** (startInnerOffset < ***OCTET\_LENGTH*** - 1) {

**long** mask = 1L << (startInnerOffset + 1) - 1;

**long** swapMask = (genom1[startOuterOffset] ^ genom2[startOuterOffset]) & mask;

genom1[startOuterOffset] ^= swapMask;

genom2[startOuterOffset] ^= swapMask;

startOuterOffset++;

}

**for** (**int** i = startOuterOffset; i <= endOuterOffset; i++) {

tmp = genom1[i];

genom1[i] = genom2[i];

genom2[i] = tmp;

}

**if** (endInnerOffset > 0) {

**long** mask = 1L << endInnerOffset - 1;

**long** swapMask = (genom1[endOuterOffset] ^ genom2[endOuterOffset]) & mask;

genom1[endOuterOffset] ^= swapMask;

genom2[endOuterOffset] ^= swapMask;

}

**break**;

}

**case** ***ELEMENTWISE\_RECOMBINATION***: {

**for** (**int** outerOffset = 0; outerOffset < **this**.sizeOfArray; outerOffset++) {

**long** mask = **this**.random.nextLong();

**long** swapMask = (genom1[outerOffset] ^ genom2[outerOffset]) & mask;

genom1[outerOffset] ^= swapMask;

genom2[outerOffset] ^= swapMask;

}

**break**;

}

**case** ***ONE\_ELEMENT\_EXCHANGE***: {

**int** index = **this**.random.nextInt(**this**.genomLength);

**int** outerOffset = index >> ***SHIFT\_FOR\_DIVISION***;

**int** innerOffset = ***OCTET\_LENGTH*** - (index & ***MASK\_FOR\_MOD***);

**long** mask = 1L << innerOffset;

**long** swapMask = (genom1[outerOffset] ^ genom2[outerOffset]) & mask;

genom1[outerOffset] ^= swapMask;

genom2[outerOffset] ^= swapMask;

**break**;

}

**default**:

**throw** **new** UnsupportedOperationException();

}

}

// Mutation - Mutate all genom in generation

**private** **void** mutation() {

**long** old = System.*currentTimeMillis*(); // time

**for** (**long**[] genom : **this**.genomListOffsprings) {

**if** (random.nextDouble() <= mutationPercent) {

mutate(genom);

}

}

**this**.timeToMutate += (System.*currentTimeMillis*() - old); // time

}

// Mutate - Mutate 1 genom

**private** **void** mutate(**long**[] genom) {

**int** index = **this**.random.nextInt(**this**.genomLength);

**int** outerOffset = index >> ***SHIFT\_FOR\_DIVISION***;

**int** innerOffset = (index & ***MASK\_FOR\_MOD***);

**long** mask = 1L << innerOffset;

genom[outerOffset] ^= mask;

}

// Setters and Getters

**public** **long** getGenerationCount() {

**return** maxGenerationCount;

}

**public** **void** setGenerationCount(**long** generationCount) {

**this**.maxGenerationCount = generationCount;

}

**public** **int** getIndividualCount() {

**return** individualCount;

}

**public** **void** setIndividualCount(**int** individualCount) {

**this**.individualCount = individualCount;

}

**public** SelectionType getSelectionType() {

**return** selectionType;

}

**public** **void** setSelectionType(SelectionType selectionType) {

**this**.selectionType = selectionType;

}

**public** CrossingType getCrossingType() {

**return** crossingType;

}

**public** **void** setCrossingType(CrossingType crossingType) {

**this**.crossingType = crossingType;

}

**public** **boolean** getUseMutation() {

**return** useMutation;

}

**public** **long** getAccuracy() {

**return** accuracy;

}

**public** **void** setAccuracy(**double** accuracy) {

**this**.accuracy = (**long**)(accuracy \* FitnessFunction.***GAIN***);

}

**public** **void** setUseMutation(**boolean** useMutation) {

**this**.useMutation = useMutation;

}

**public** **double** getMutationPercent() {

**return** mutationPercent;

}

**public** **void** setMutationPercent(**double** mutationPercent) {

**this**.mutationPercent = mutationPercent;

}

}