Contemporary Intelligent Intellectual Technology (CIIT)
Practice #2 (21/09/2016)
Siarhei Savaniuk (AI-10)

Task №1 (20-Dimensional Schwefel's function)

The task is to minimise $y = x1 \sin(|x1|) + x2 \sin(|x2|) + \cdots + x20 \sin(|x20|)$ in the following way:

(1) Represent each of $x_i (i = 1, \cdots, 20)$ by a chromosome with 20 genes.

(2) Create a population of 20 chromosomes at random, with fitness being y.

(3) Evolve this population till fitness dosen't change.

Source code (write in Java):

File Individual.java

```
import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

class Individual implements Comparable<Individual> {

    public static final int GENE_LENGTH = 20;

    private double[] genes;
    private double fitnessValue;

    public Individual(boolean initialize) {
        genes = new double[GENE_LENGTH];

        if (initialize) {
            generateIndividual();
            fitnessValue = getFitness();
        }
    }

    public Individual(double[] genes) {
        this.genes = genes;
        fitnessValue = getFitness();
    }

    public double getFitnessValue() {
        return fitnessValue;
    }

    public void setFitnessValue(int fitnessValue) {
        this.fitnessValue = fitnessValue;
    }

    public void generateIndividual() {
        for (int i = 0; i < GENE_LENGTH; ++i) {
            genes[i] = ThreadLocalRandom.current().nextDouble(6.0) - 5.0;
        }
    }

    public double getFitness() {
```

```
        double fitness = 0.0;

        for (int i = 0; i < GENE_LENGTH; ++i) {
            fitness += genes[i] * Math.sin(Math.abs(genes[i]));
        }

        return fitness;
    }

    public double[] getGenesBeforeCutPoint(int cutPoint) {
        double[] genes = new double[cutPoint];

        System.arraycopy(this.genes, 0, genes, 0, cutPoint);

        return genes;
    }

    public double[] getGenesAfterCutPoint(int cutPoint) {
        double[] genes = new double[GENE_LENGTH - cutPoint];

        System.arraycopy(this.genes, cutPoint, genes, 0, genes.length);

        return genes;
    }

    @Override
    public String toString() {
        return "Individual{" +
                "fitnessValue=" + fitnessValue +
                ", genes=" + Arrays.toString(genes) +
                '}' + '\n';
    }

    @Override
    public int compareTo(Individual o) {
        return (o.getFitnessValue() < fitnessValue ? 1 : (o.getFitnessValue()
== fitnessValue) ? 0 : -1);
    }
}
```

File Population.java:

```
import java.util.Arrays;

class Population {

    public static final int POPULATION_SIZE = 20;

    private Individual[] individuals;

    public Population(boolean initialize) {
        individuals = new Individual[POPULATION_SIZE];

        if (initialize) {
            for (int i = 0; i < POPULATION_SIZE; ++i) {
                individuals[i] = new Individual(true);
            }
        }
    }

    public Population(Individual[] individuals) {
        this.individuals = new Individual[POPULATION_SIZE];
        System.arraycopy(individuals, 0, this.individuals, 0,
individuals.length);
```

```
    }

    public Individual getIndividual(int index) {
        return individuals[index];
    }

    public void addIndividual(int index, Individual individual) {
        individuals[index] = individual;
    }

    public Individual[] getHalfFittestIndividuals() {
        Individual[] fittestIndividuals = new Individual[POPULATION_SIZE /
2];

        System.arraycopy(individuals, 0, fittestIndividuals, 0,
fittestIndividuals.length);

        return fittestIndividuals;
    }

    public double getMaxFitness() {
        return individuals[0].getFitnessValue();
    }

    public double getAverageFitness() {
        double sum = 0;

        for (int i = 0; i < POPULATION_SIZE; ++i) {
            sum += individuals[i].getFitnessValue();
        }

        return sum / POPULATION_SIZE;
    }

    public Individual[] getAllIndividuals() {
        return individuals;
    }

    @Override
    public String toString() {
        return "Population{\n" + Arrays.toString(individuals) + "}\n";
    }
}
```

File GeneticAlgorithm.java:

```
import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class GeneticAlgorithm {

    private Population population;

    public GeneticAlgorithm(Population population) {
        this.population = population;
    }

    public Population run() {
        Arrays.sort(population.getAllIndividuals());

        Population halfPopulation = new
Population(population.getHalfFittestIndividuals());
        Population nextGeneration = new
Population(halfPopulation.getAllIndividuals());
```

```
        for (int i = 0, j = Population.POPULATION_SIZE / 2; i <
Population.POPULATION_SIZE / 4; ++i, j += 2) {
            Individual[] parents = chooseParents(halfPopulation);

            int cutPoint =
ThreadLocalRandom.current().nextInt(Individual.GENE_LENGTH);

            Individual[] descendants = crossover(parents, cutPoint);

            nextGeneration.addIndividual(j, descendants[0]);
            nextGeneration.addIndividual(j + 1, descendants[1]);
        }

        population = nextGeneration;

        Arrays.sort(population.getAllIndividuals());

        return population;
    }

    private Individual[] chooseParents(Population fittestIndividuals) {
        return new Individual[]
{fittestIndividuals.getIndividual(ThreadLocalRandom.current().nextInt(10)),

fittestIndividuals.getIndividual(ThreadLocalRandom.current().nextInt(10))};
    }

    private Individual[] crossover(Individual[] parents, int curPoint) {
        Individual[] descendants = new Individual[2];

        double[] firstDescendantGenes =
concat(parents[0].getGenesBeforeCutPoint(curPoint),
                parents[1].getGenesAfterCutPoint(curPoint));
        double[] secondIndividualGenes =
concat(parents[0].getGenesAfterCutPoint(curPoint),
                parents[1].getGenesBeforeCutPoint(curPoint));

        descendants[0] = new Individual(firstDescendantGenes);
        descendants[1] = new Individual(secondIndividualGenes);

        return descendants;
    }

    private double[] concat(double[] genes1, double[] genes2) {
        double[] genes = new double[Individual.GENE_LENGTH];

        System.arraycopy(genes1, 0, genes, 0, genes1.length);
        System.arraycopy(genes2, 0, genes, genes1.length, genes2.length);

        return genes;
    }
}
```

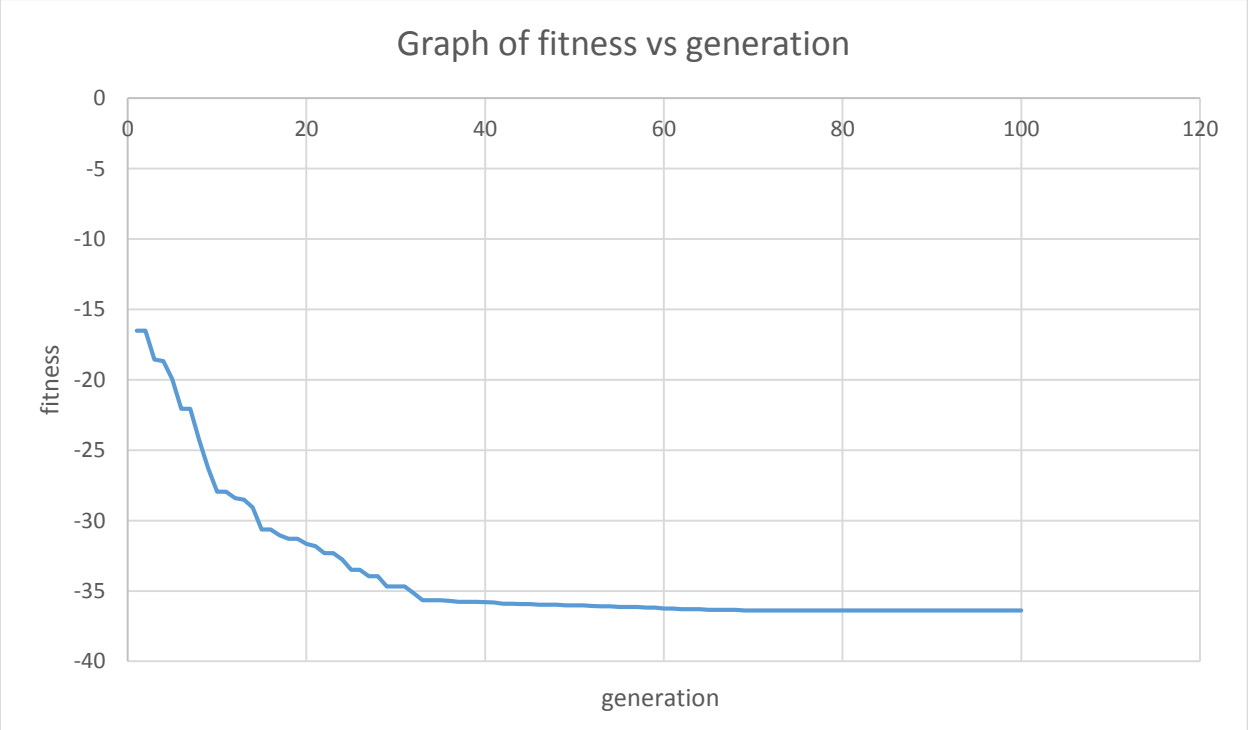
File Main.java:

```
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) throws IOException {
        Population population = new Population(true);
```

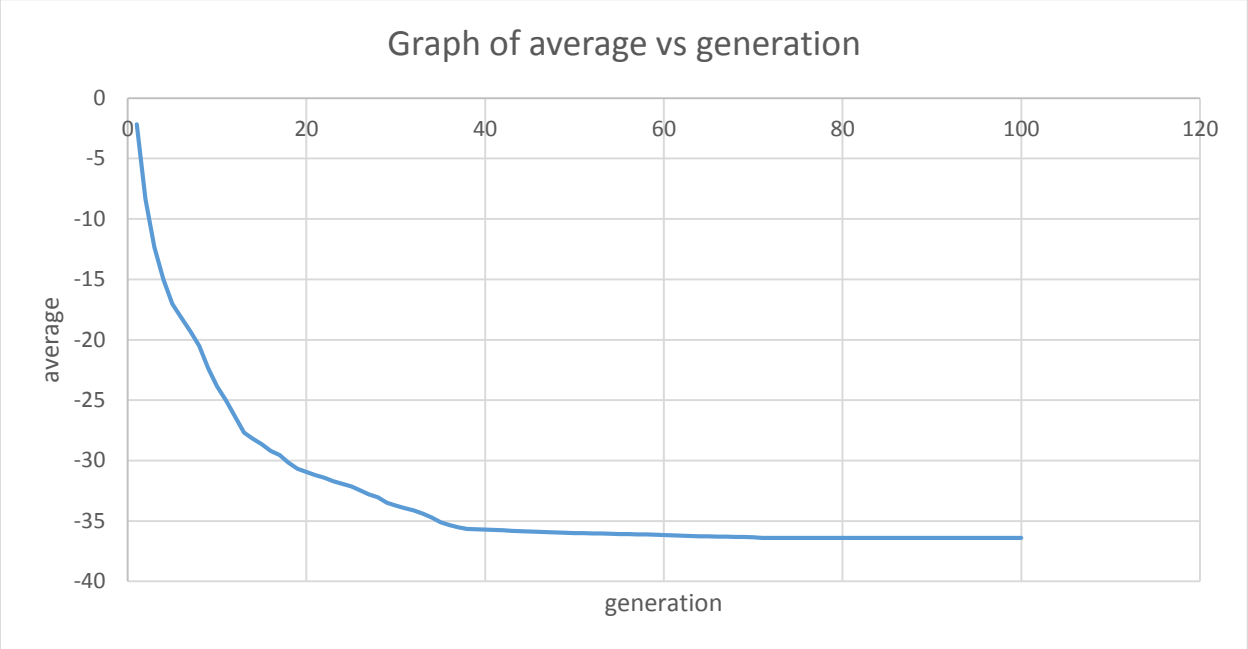
```
        GeneticAlgorithm geneticAlgorithm = new GeneticAlgorithm(population);

        List<Double> max = new ArrayList<>();
        List<Double> average = new ArrayList<>();

        double min = 0.0, copyMin = 0.0;

        for (int i = 0; i < 100; ++i) {
            Population newPopulation = geneticAlgorithm.run();
            min = newPopulation.getMaxFitness();
            System.out.println("Min = " + min + ", Average = " +
newPopulation.getAverageFitness());
            max.add(newPopulation.getMaxFitness());
            average.add(newPopulation.getAverageFitness());

        }

        System.out.println("MAX");
        for (Double val : max) {
            System.out.printf("%.2f\n", val);
        }

        System.out.println("\n\nAVERAGE");
        for (Double val : average) {
            System.out.printf("%.2f\n", val);
        }
    }
}
```

Results:

The graph of fitness vs generation:



The graph of average vs generation:



Task #2 (2-D version of Schwefel's function)

The task is to minimise $y = x \sin(|x|)$ in the following way:

(1) Represent value of x by a 10-bit binary chromosome.

(2) Create a population of 20 chromosomes at random, with fitness being y.

(3) Evolve this population till fitness doesn't change.

Source code (written in Java)

File Individual.java:

```
import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

class Individual implements Comparable<Individual> {

    public static final int GENE_LENGTH = 10;

    private int[] genes;
    private double fitnessValue;
    private double x;

    public Individual(boolean initialize) {
        genes = new int[GENE_LENGTH];

        if (initialize) {
            generateIndividual();
            fitnessValue = getFitness();
        }
```

```
}

public Individual(int[] genes) {
    this.genes = genes;
    fitnessValue = getFitness();
}

public double getFitnessValue() {
    return fitnessValue;
}

public void setFitnessValue(int fitnessValue) {
    this.fitnessValue = fitnessValue;
}

public double getX() {
    return x;
}

public void generateIndividual() {
    for (int i = 0; i < GENE_LENGTH; ++i) {
        genes[i] = ThreadLocalRandom.current().nextInt(2);
    }
}

public double getFitness() {
    x = convertToDecimal();
    return x * Math.sin(Math.abs((x)));
}

private double convertToDecimal() {
    int dec = 0;
    for (int i = 0; i < 10; ++i) {
        if (genes[i] == 1) {
            dec += 1 * Math.pow(2, i);
        }
    }
    dec = (genes[0] == 1) ? -dec: dec;
    return dec * 5.0 / 1023;
}

public int[] getGenesBeforeCutPoint(int cutPoint) {
    int[] genes = new int[cutPoint];

    System.arraycopy(this.genes, 0, genes, 0, cutPoint);

    return genes;
}

public int[] getGenesAfterCutPoint(int cutPoint) {
    int[] genes = new int[GENE_LENGTH - cutPoint];

    System.arraycopy(this.genes, cutPoint, genes, 0, genes.length);

    return genes;
}

@Override
public String toString() {
    return "Individual{" +
            "fitnessValue=" + fitnessValue +
            ", genes=" + Arrays.toString(genes) +
            '}' + '\n';
}
```

```
    @Override
    public int compareTo(Individual o) {
        return (o.getFitnessValue() < fitnessValue ? 1 : (o.getFitnessValue()
== fitnessValue) ? 0 : -1);
    }
}
```

File Population.java:

```
import java.util.Arrays;

class Population {

    public static final int POPULATION_SIZE = 20;

    private Individual[] individuals;

    public Population(boolean initialize) {
        individuals = new Individual[POPULATION_SIZE];

        if (initialize) {
            for (int i = 0; i < POPULATION_SIZE; ++i) {
                individuals[i] = new Individual(true);
            }
        }
    }

    public Population(Individual[] individuals) {
        this.individuals = new Individual[POPULATION_SIZE];
        System.arraycopy(individuals, 0, this.individuals, 0,
individuals.length);
    }

    public Individual getIndividual(int index) {
        return individuals[index];
    }

    public void addIndividual(int index, Individual individual) {
        individuals[index] = individual;
    }

    public Individual[] getHalfFittestIndividuals() {
        Individual[] fittestIndividuals = new Individual[POPULATION_SIZE /
2];

        System.arraycopy(individuals, 0, fittestIndividuals, 0,
fittestIndividuals.length);

        return fittestIndividuals;
    }

    public Individual[] getFiveIndividuals() {
        Individual[] fittestIndividuals = new Individual[POPULATION_SIZE /
4];

        System.arraycopy(individuals, 0, fittestIndividuals, 0,
fittestIndividuals.length);

        return fittestIndividuals;
    }

    public double getMaxFitness() {
        return individuals[0].getFitnessValue();
```

```
    }

    public double getAverageFitness() {
        double sum = 0;

        for (int i = 0; i < POPULATION_SIZE; ++i) {
            sum += individuals[i].getFitnessValue();
        }

        return sum / POPULATION_SIZE;
    }

    public Individual[] getAllIndividuals() {
        return individuals;
    }

    @Override
    public String toString() {
        return "Population{\n" + Arrays.toString(individuals) + "}\n";
    }
}
```

File GeneticAlgorithm.java:

```
import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class GeneticAlgorithm {

    private Population population;

    public GeneticAlgorithm(Population population) {
        this.population = population;
    }

    public Population run() {
        Arrays.sort(population.getAllIndividuals());

        Population halfPopulation = new
Population(population.getHalfFittestIndividuals());
        Population nextGeneration = new
Population(halfPopulation.getAllIndividuals());

        for (int i = 0, j = Population.POPULATION_SIZE / 2; i <
Population.POPULATION_SIZE / 4; ++i, j += 2) {
            Individual[] parents = chooseParents(halfPopulation);

            int cutPoint =
ThreadLocalRandom.current().nextInt(Individual.GENE_LENGTH);

            Individual[] descendants = crossover(parents, cutPoint);

            nextGeneration.addIndividual(j, descendants[0]);
            nextGeneration.addIndividual(j + 1, descendants[1]);
        }

        population = nextGeneration;

        Arrays.sort(population.getAllIndividuals());
//
//        population = population.getHalfFittestIndividuals();

        return population;
    }
```

```
    private Individual[] chooseParents(Population fittestIndividuals) {
        return new Individual[]
{fittestIndividuals.getIndividual(ThreadLocalRandom.current().nextInt(10)),

fittestIndividuals.getIndividual(ThreadLocalRandom.current().nextInt(10))};
    }

    private Individual[] crossover(Individual[] parents, int curPoint) {
        Individual[] descendants = new Individual[2];

        int[] firstDescendantGenes =
concat(parents[0].getGenesBeforeCutPoint(curPoint),
                parents[1].getGenesAfterCutPoint(curPoint));
        int[] secondIndividualGenes =
concat(parents[0].getGenesAfterCutPoint(curPoint),
                parents[1].getGenesBeforeCutPoint(curPoint));

        descendants[0] = new Individual(firstDescendantGenes);
        descendants[1] = new Individual(secondIndividualGenes);

        return descendants;
    }

    private int[] concat(int[] genes1, int[] genes2) {
        int[] genes = new int[Individual.GENE_LENGTH];

        System.arraycopy(genes1, 0, genes, 0, genes1.length);
        System.arraycopy(genes2, 0, genes, genes1.length, genes2.length);

        return genes;
    }
}
```

File Main.java:

```
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class Main {

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

        Population population = new Population(true);
        GeneticAlgorithm geneticAlgorithm = new GeneticAlgorithm(population);

        List<Double> max = new ArrayList<>();
        List<Double> average = new ArrayList<>();

        double min = 0.0, copyMin = 0.0;

        for (int i = 0; i < 10; ++i) {
            System.out.println("Iteration #" + (i + 1));
            Population newPopulation = geneticAlgorithm.run();
            min = newPopulation.getMaxFitness();
            System.out.println("Min = " + min + ", Average = " +
newPopulation.getAverageFitness());
            for (Individual individuals : newPopulation.getAllIndividuals())
{
                System.out.println(individuals.getX());
            }
            max.add(newPopulation.getMaxFitness());
            average.add(newPopulation.getAverageFitness());
```

```
            }

    }
```