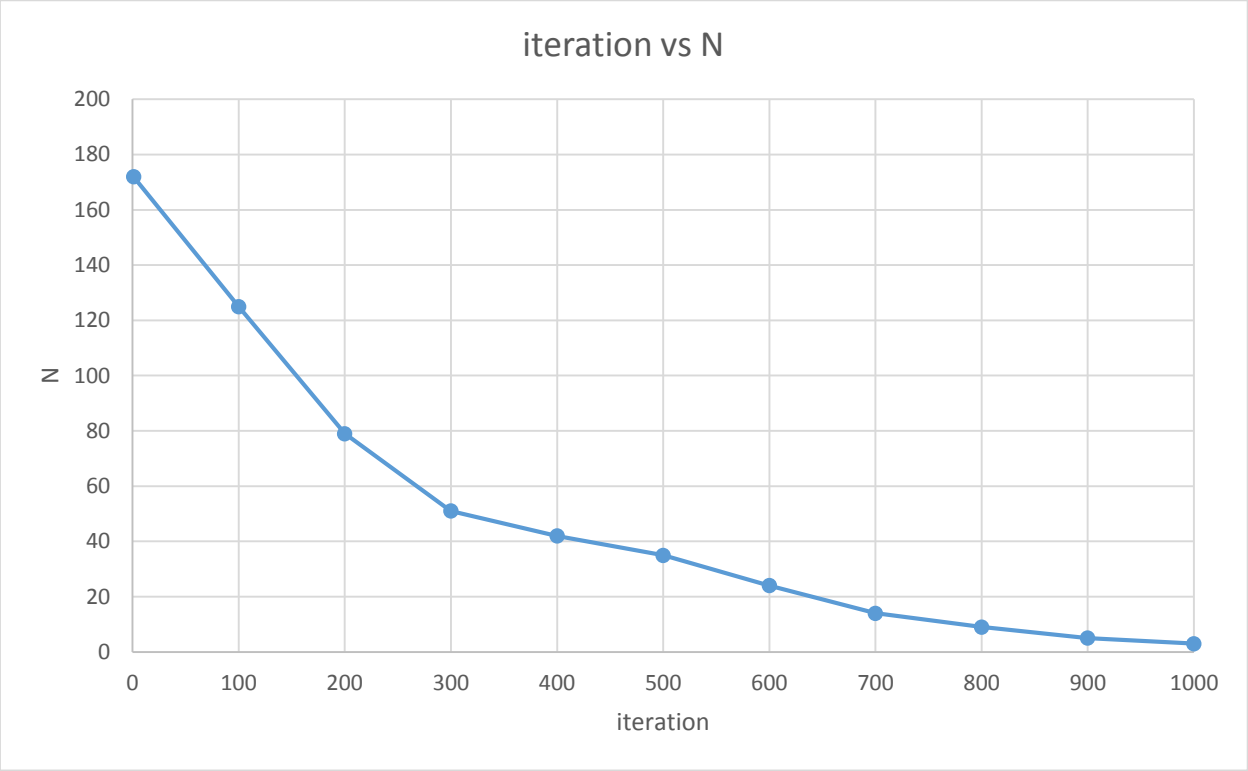
Contemporary Intelligent Information Techniques (CIIT)
Practice #6 (14/11/2016)
Siarhei Savaniuk (AI-10)

LabWork #6 Hinton and Nowlan's Simulation of Life Time Learning

An algorithm:
1. Create a population of 1000 chromosomes with 20 genes either of 1, 0, ? with probability being 0.25, 0.25, 0.5, respectively.
2. Make each chromosome a maximum of 1000 learnings by assigning "1" or "0" to each of "?" * each gene feels happy or unhappy and if happy the gene is _xed and otherwise remain "?" * if all "?" are happy tnen $N = 1000 - so.far.trials$ else keep learning till $N = 0$.
3. Select two parents at random with the probability of $1 - 19N/1000$.
4. Create next generation by repeating 3. 1000 times
5. Repeat from 2. to 4. untill solution found



Source code (written in Java)

File Individual.java:

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

public class Individual {

    public static final int GENE_LENGTH = 10;

    private int[] genes;
    private double x;
    private double y1;
    private double y2;
    private int rank;

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

        if (initialize) {
```

```
            generateIndividual();
            this.x = calculateX();
            this.y1 = calculateY1();
            this.y2 = calculateY2();
            this.rank = 0;
        }
    }

    public Individual(int[] genes) {
        this.genes = genes;
        this.x = calculateX();
        this.y1 = calculateY1();
        this.y2 = calculateY2();
        this.rank = 0;
    }

    public double getX() {
        return x;
    }

    public double getY1() {
        return y1;
    }

    public double getY2() {
        return y2;
    }

    public int getRank() {
        return rank;
    }

    public int updateRank() {
        return ++this.rank;
    }

    public void resetRank() {
        this.rank = 0;
    }

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

    public double calculateX() {
        return Integer.parseInt(Arrays.toString(genes).replaceAll("[,\\[\\]
]", ""), 2) / 170.5;
    }

    public double calculateY1() {
        return Math.pow(x - 2, 2);
    }

    public double calculateY2() {
        return Math.pow(x - 4, 2);
    }

    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 boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Individual that = (Individual) o;

        if (Double.compare(that.y1, y1) != 0) return false;
        if (Double.compare(that.y2, y2) != 0) return false;
        if (Double.compare(that.x, x) != 0) return false;
        if (rank != that.rank) return false;
        return Arrays.equals(genes, that.genes);

    }

    @Override
    public int hashCode() {
        int result;
        long temp;
        result = Arrays.hashCode(genes);
        temp = Double.doubleToLongBits(y1);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(y2);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(x);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        result = 31 * result + rank;
        return result;
    }

    @Override
    public String toString() {
        return "Individual{" +
                "genes=" + Arrays.toString(genes) +
                ", x=" + x +
                ", y1=" + y1 +
                ", y2=" + y2 +
                ", rank=" + rank +
                '}';
    }
}
```

File Population.java:

```
import java.util.Arrays;

public 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 replaceIndividual(Individual individual, Individual child) {
        for (int i = 0; i < individuals.length; ++i) {
            if (individuals[i].equals(individual)) {
                individuals[i] = child;
                break;
            }
        }
    }

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

    public void calculateRankForAllIndividuals() {
        resetRank();
        for (int i = 0; i < POPULATION_SIZE; ++i) {
            for (int j = 0; j < POPULATION_SIZE; ++j) {
                if (i != j && (individuals[i].getY1() <
individuals[j].getY1() &&
                                    individuals[i].getY2() <
individuals[j].getY2())) {
                    individuals[i].updateRank();
                }
            }
        }

    }

    private void resetRank() {
        for (Individual individual: individuals) {
            individual.resetRank();
        }
    }

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

File GeneticAlgorithm.java:

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

public class GeneticAlgorithm {

    private Population population;

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

    public Population run() {
        Population nextGeneration = new
Population(population.getAllIndividuals());
        nextGeneration.calculateRankForAllIndividuals();

        for (int i = 0; i < Population.POPULATION_SIZE; ++i) {
            Individual[] parents = chooseParents(nextGeneration);
            int cutPoint = ThreadLocalRandom.current().nextInt(0,
Individual.GENE_LENGTH);
```

```
            Individual[] children = crossover(parents, cutPoint);
            Individual child = getBetterIndividualFromTwoChildren(children);
            calculateIndividualRank(child);
            Individual individual =
findIndividualThatMostSimilarToChild(child);

            if (childBetterThanParent(individual, child)) {
                nextGeneration.replaceIndividual(individual, child);
                nextGeneration.calculateRankForAllIndividuals();
            }
        }

        population = nextGeneration;
        return population;
    }

    private Individual[] chooseParents(Population fittestIndividuals) {
        return new Individual[]
{fittestIndividuals.getIndividual(ThreadLocalRandom.current().nextInt(0,
Population.POPULATION_SIZE)),

fittestIndividuals.getIndividual(ThreadLocalRandom.current().nextInt(0,
Population.POPULATION_SIZE))};
    }

    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 Individual getBetterIndividualFromTwoChildren(Individual[]
children) {
        return calculateIndividualRank(children[0]) >
calculateIndividualRank(children[1]) ? children[0] : children[1];
    }

    private Individual findIndividualThatMostSimilarToChild(Individual child)
{
        Individual individual = population.getIndividual(0);
        double minX = Math.abs(population.getIndividual(0).getX() -
child.getX());
        for (int i = 1; i < Population.POPULATION_SIZE; ++i) {
            double tempMinX = Math.abs(population.getIndividual(i).getX() -
child.getX());
            if (tempMinX < minX) {
                minX = tempMinX;
                individual = population.getIndividual(i);
            }
        }
        return individual;
    }

    private boolean childBetterThanParent(Individual individual, Individual
child) {
        return child.getRank() > individual.getRank() || (child.getY1() <
individual.getY1() && child.getY2() < individual.getY2());
    }
```

```
    private int calculateIndividualRank(Individual individual) {
        for (int i = 0; i < Population.POPULATION_SIZE; ++i) {
            if (individual.getY1() < population.getIndividual(i).getY1() &&
                individual.getY2() < population.getIndividual(i).getY2()) {
                individual.updateRank();
            }
        }
        return individual.getRank();
    }

    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.FileWriter;
import java.io.IOException;

public class Main {
    public static void main(String[] args) throws IOException {
        FileWriter writer = new FileWriter("output.txt");
        Population population = new Population(true);
        GeneticAlgorithm geneticAlgorithm = new GeneticAlgorithm(population);

        for (int i = 0; i < 1000; ++i) {
            population = geneticAlgorithm.run();
            System.out.println("ITERATION #" + (i + 1));
            writer.write("ITERATION #" + (i + 1) + ":\n");
            for (Individual individual: population.getAllIndividuals()) {
                writer.write(individual.toString() + "\n");
                System.out.println(individual);
            }
        }
    }
}
```