CSC3423 – Biocomputing
Practical1: Genetic Algorithms

The aim of this practical is to familiarise yourself with the working of a basic genetic algorithm. The practical has two parts. In the first one you will use a standalone evolutionary computation program called ECJ. In the second part you will programmatically build a simple genetic algorithm using a library called JGAP.

Part 1: ECJ

The software you will use is an open-source evolutionary computation package called **ECJ** (Evolutionary Computation in Java ; http://cs.gmu.edu/~eclab/projects/ecj/).

Downloading the files for the practical

A compiled version of ECJ plus other relevant files for this tutorial is in Blackboard in Teaching Materials > Practicals > Practical1 files - ECJ. Download and unzip the file.

Within you will see four files:

- ecj.22.jar → The jar file contains the whole ECJ framework
- MaxOnes.java, MaxOnes.class → specify the fitness function to use in this particular case (the OneMax problem described in Lecture2)
- tutorial1.params → text file that specifies how to run the framework. Within it will select what type of evolutionary algorithm to run, what operators to use in each stage of the evolutionary cycle, the length of the chromosome, the values for the paramters, etc.

Running the basic example

To run ECJ, for this example, from the command line simply run:

```
java -jar ecj.22.jar ec.Evolve -file tutorial1.params

| ECJ
| An evolutionary computation system (version 22)
| By Sean Luke
| Contributors: L. Panait, G. Balan, S. Paus, Z. Skolicki, R. Kicinger,
|               E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R.
Hubley,
|               A. Desai, A. Chircop, J. Compton, W. Haddon, S. Donnelly,
|               B. Jamil, J. Zelibor, E. Kangas, F. Abidi, H. Mooers,
|               J. O'Beirne, L. Manzoni, K. Talukder, and J. McDermott
| URL: http://cs.gmu.edu/~eclab/projects/ecj/
| Mail: ecj-help@cs.gmu.edu
|       (better: join ECJ-INTEREST at URL above)
| Date: August 1, 2014
| Current Java: 1.7.0_12-ea / Java HotSpot(TM) 64-Bit Server VM-24.0-b28
| Required Minimum Java: 1.5


Threads:  breed/1 eval/1
Seed: 1337
```

```
Job: 0
Setting up
WARNING:
No statistics file specified, printing to stdout at end.
PARAMETER: stat.file
Initializing Generation 0

Generation: 0
Best Individual:
Subpopulation 0:
Evaluated: T
Fitness: 0.68
01111001101011110111011110100110111101110101001111
Subpop 0 best fitness of generation Fitness: 0.68
Generation 1

Generation: 1
Best Individual:
Subpopulation 0:
Evaluated: T
Fitness: 0.68
10001110111001111111001110010111111101110011011101
Subpop 0 best fitness of generation Fitness: 0.68
Generation 2

Generation: 2
Best Individual:
Subpopulation 0:
Evaluated: T
Fitness: 0.74
01111111111111110110100011101111010011111101001111
Subpop 0 best fitness of generation Fitness: 0.74
Generation 3

.
.
.

Generation: 19
Best Individual:
Subpopulation 0:
Evaluated: T
Fitness: 0.96
11111110110111111111111111111111111111111111111111
Subpop 0 best fitness of generation Fitness: 0.96
Generation 20

Generation: 20
Best Individual:
Subpopulation 0:
Evaluated: T
Fitness: 1.0
11111111111111111111111111111111111111111111111111
Subpop 0 best fitness of generation Fitness: 1.0
Found Ideal Individual

Best Individual of Run:
Subpopulation 0:
Evaluated: T
Fitness: 1.0
11111111111111111111111111111111111111111111111111
Subpop 0 best fitness of run: Fitness: 1.0
```

As you can see, the algorithm found the optimal solution (a chromosome with all 1's) at iteration 20.

Changing the random seed, collecting data from multiple runs of the algorithm

If you run the program again if will produce the same exact output. This is because a seed for the random number generator is specified (and constant). In order to tell ECJ to use the current time as random seed, change the line

```
seed.0          = 1337
```

to

```
seed.0          = time
```

If you run the algorithm a few times, you will observe that the iteration at which it finds the optimal solution now is different each time. Run the algorithm a large number of times and compute the average finish iteration (e.g. using Excel or writing some simple program).

You can also extract from the output of the program the fitness of the best individual at each iteration, and plot the learning curve (e.g. in Excel or gnuplot).

Changing other parameters in the algorithm

In tutorial1.params there are several lines that you can modify and observe how they affect the number of iterations before convergence:

- breed.elite.0=0
  This changes the elitism behavior: The number of top individuals from the previous population that will be copied directly to the next one
- select.tournament.size = 2
  This changes the number of individuals involved in each Tournament of the selection algorithm
- pop.subpop.0.species.mutation-prob = 0.01
  This changes the (gene-wise) mutation probability. This parameter is very influential
- pop.subpop.0.species.genome-size = 50
  This changes the size of the problem (number of bits in the individuals). The larger the genome size, the more iterations that will be needed to find the optimal solution.

Modify all these four parameters and see the effect they create. First modify them individually, and then play with modifying pairs (triplets) of parameters at once.

Part 2: JGAP

JGAP (http://jgap.sourceforge.net/) is a Java-based library for genetic algorithms and genetic programming that can be integrated into your own code, similarly to what you will have to do in the module's coursework. A javadoc documentation of the library is available at http://jgap.sourceforge.net/javadoc/3.6/.

Setting up a genetic algorithm

The code below creates a genetic algorithm to optimise the OneMax Problem (same as in part 1 of the practical). This example shows how JGAP has been designed so that users can get a GA up and running with the minimal amount of hassle. The aspects that need to be specified are the following:

1. Build a fitness evaluator
2. Create a configuration object
3. Specify the structure of a chromosome
4. Set up the population size

```
import org.jgap.*;
import org.jgap.impl.*;

public class OneMaxFitnessFunction extends FitnessFunction {

     public OneMaxFitnessFunction() {
     }

     public double evaluate( IChromosome ind ) {
       int numBits = ind.size();
       int countOnes = 0;

       for(int i=0;i<numBits;i++) {
              countOnes+=((Integer)ind.getGene(i).getAllele()).intValue();
       }

       return (double)countOnes/(double)numBits;
     }

     public static String getPhenotype(IChromosome ind) {
       int numBits = ind.size();
       String phenotype="";

       for(int i=0;i<numBits;i++) {
              phenotype+=((Integer)ind.getGene(i).getAllele()).intValue();
       }

       return phenotype;
     }


     public static void main(String[] args) throws Exception {
       // Start with a DefaultConfiguration, which comes setup with the
       // most common settings.
       // -------------------------------------------------------------
       Configuration conf = new DefaultConfiguration();

       // Creates an object of the fitness function class
       FitnessFunction myFunc = new OneMaxFitnessFunction();
       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.
       //
```

```
        // The number of bits of the OneMax problem is specified as
        // a command line argument. We have to create an array of the
        // Gene object of such size, and create objects of the class
        // IntegerGene which is the suitable class for specifying bits
        // The parameters of the IntegerGene class are the configuration
        // object and the lower and upper bound for the integer value
        // -------------------------------------------------------------
        int problemSize = Integer.parseInt(args[0]);
        Gene[] sampleGenes = new Gene[ problemSize ];
        for(int i=0;i<problemSize;i++) {
               sampleGenes[i] = new IntegerGene(conf, 0, 1);
        }

        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( 100 );

        Genotype population = Genotype.randomInitialGenotype( conf );
        IChromosome bestSolutionSoFar;

        int numIterations = Integer.parseInt(args[1]);
        for( int i = 0; i < numIterations; i++ ) {
               population.evolve();
               bestSolutionSoFar = population.getFittestChromosome();

               System.out.println("Iteration "+i+". The best individual is "
                      +OneMaxFitnessFunction.getPhenotype(bestSolutionSoFar)
                      +" with fitness "+bestSolutionSoFar.getFitnessValue());
        }
      }
}
```

1. Building a fitness evaluator

In order to evaluate individuals in JGAP, users need to create a class that extends from org.jgap.FitnessFunction. This function needs to implement the method "public double evaluate(IChromosome ind)". IChromosome is the class that holds the data structure of each individual of a GA. In the fitness evaluator we are just calling two of its functions: size() to know the number of genes in a individual, and getGene(i) to retrieve the value for gene *i* of the individual. The evaluate function will compute the fraction of genes set to 1 and return that as fitness value.

The class has a second function, called getPhenotype. This function returns a string representation of the individual.

2. Creating a configuration object

The first line of code in the main function creates an object of the class *DefaultConfiguration*. This class is designed precisely for a "minimal-overhead" configuration by automatically setting up lots of options and mechanisms. If you would like to know exactly what it does, download the source code of JGAP and look at the file src/org/jgap/impl/DefaultConfiguration.java.

After creating the configuration object, we pass it an instance of the fitness evaluation class.

3. Specifying the structure of a chromosome

If you go to the middle part of the main function you will see the following lines of code:

```
        int problemSize = Integer.parseInt(args[0]);
        Gene[] sampleGenes = new Gene[ problemSize ];
        for(int i=0;i<problemSize;i++) {
                sampleGenes[i] = new IntegerGene(conf, 0, 1);
        }

        Chromosome sampleChromosome = new Chromosome(conf, sampleGenes );
        conf.setSampleChromosome( sampleChromosome );
```

These lines tell JGAP about: (1) the number of genes in an individual, via the size of the Gene[] array, and (2) the definition of each Gene object in the array. In this case all genes are initialised equally, as integer variables with a lower bound of 0 and an upper bound of 1. Hence, they become Boolean variables.

4. Setting up the population size

Finally, we specify the size of the GA population via the conf.setPopulationSize() function using the args[0] command line argument.

Running the GA

The last part of the main function runs the GA.

```
        Genotype population = Genotype.randomInitialGenotype( conf );
        IChromosome bestSolutionSoFar;

        int numIterations = Integer.parseInt(args[1]);
        for( int i = 0; i < numIterations; i++ ) {
                population.evolve();
                bestSolutionSoFar = population.getFittestChromosome();

                System.out.println("Iteration "+i+". The best individual is "
                        +OneMaxFitnessFunction.getPhenotype(bestSolutionSoFar)
                        +" with fitness "+bestSolutionSoFar.getFitnessValue());
        }
```

It initialises the population and then runs the GA cycle for the number of iterations specified in the args[1] command line argument. After each iteration it retrieves the best individual found so far and prints its phenotype and fitness value.

Compiling and running the example

The files for this example are available in Blackboard in Teaching Materials > Practicals > Practical1 > Practical1 files - JGAP. Download and unzip the file.

This compiles the code and runs the example with a chromosome of 50 genes and 25 iterations of the genetic algorithm.

```
javac -cp jgap.jar OneMaxFitnessFunction.java

java -cp jgap.jar:. OneMaxFitnessFunction 50 25
Iteration 0. The best individual is 11011101111011111000011100111111111001001111101110 with fitness 0.7
Iteration 1. The best individual is 01011011111111011001110011110111110011001111110011 with fitness 0.7
Iteration 2. The best individual is 11111011110111011011111100111111110010011111101110 with fitness 0.76
Iteration 3. The best individual is 11111011110111010011111100111101100101111111111111 with fitness 0.78
Iteration 4. The best individual is 11011101111111111111111011110111110011001111111100 with fitness 0.8
Iteration 5. The best individual is 11111011110111111110110111111111110010011111101111 with fitness 0.82
Iteration 6. The best individual is 11011101111111111111111011110111110011001111111111 with fitness 0.84
Iteration 7. The best individual is 11111011110111111011111100111111110010111111111111 with fitness 0.84
Iteration 8. The best individual is 11011101111111111111111111110111110111011111101111 with fitness 0.88
Iteration 9. The best individual is 11011101111111111111111111111111110010011111111111 with fitness 0.88
Iteration 10. The best individual is 11111011111111111111111111111111110010111111111111 with fitness 0.92
Iteration 11. The best individual is 11011101111111111111111111111111111011110111111111 with fitness 0.92
Iteration 12. The best individual is 11111011111111111111111111111111111011101111111111 with fitness 0.94
Iteration 13. The best individual is 11111111111111111111111111111111111011101111111111 with fitness 0.96
Iteration 14. The best individual is 11111111111111111111111111111111111101111111111111 with fitness 0.98
Iteration 15. The best individual is 11111111111111111111111111111111111111110111111111 with fitness 0.98
Iteration 16. The best individual is 11111111111111111111111111111111111101111111111111 with fitness 0.98
Iteration 17. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
Iteration 18. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
Iteration 19. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
Iteration 20. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
Iteration 21. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
Iteration 22. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
Iteration 23. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
Iteration 24. The best individual is 11111111111111111111111111111111111111111111111111 with fitness 1.0
```

Further notes

In the example above we did not tune many of the parameters of a GA, such as the probabilities of crossover and mutation. The DefaultConfiguration object did that automatically. If you would like to play with these parameters, you need to create a new configuration object. Make a copy of DefaultConfiguration and start to change some of its contents.