CSG2341 Intelligent Systems

Using the Evolution package: a tutorial

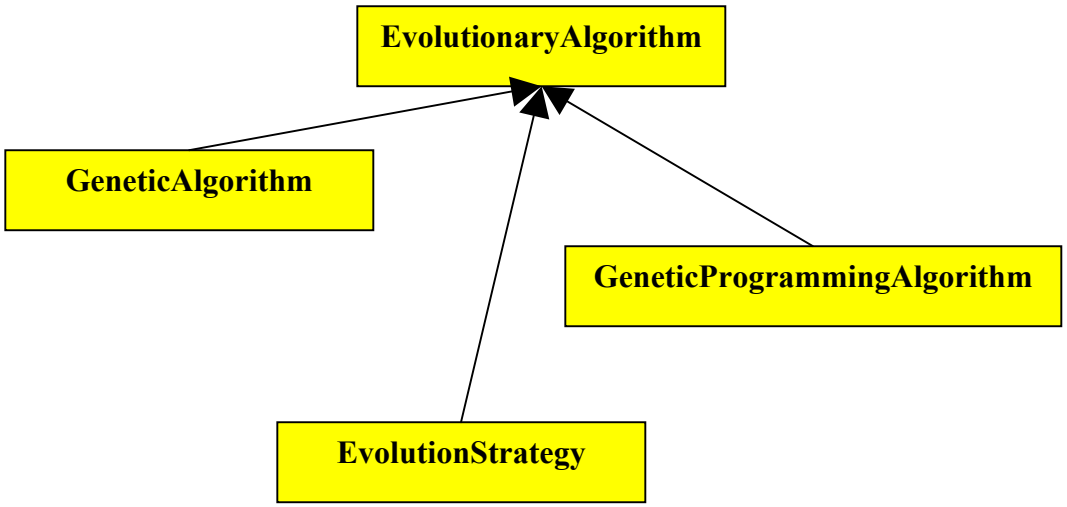
Intelligent Systems includes a number of modules on Evolutionary Algorithms. Workshop and lecture activities for these modules use the Evolution package from the Intelligent Systems Java library, IS.jar.

This tutorial shows how to develop a simple Evolutionary Algorithm using the Evolution package. We use the example from the text book, Section 7.3, as a case study. While it all looks rather complicated, it is usually only necessary to write 3 or 4 simple classes for a particular application. The Evolution package takes care of all the messy details for you.

Class design:

Class EvolutionaryAlgorithm

The most important class is the `EvolutionaryAlgorithm` class. This is an abstract class with three concrete subclasses:



You are free to develop your own subclasses of any of these if you wish, but usually you will just use an instance of one of the three concrete subclasses of `EvolutionaryAlgorithm`. In our example, we will use the `GeneticAlgorithm` class.

But first, we will take a look at the `EvolutionaryAlgorithm` class to understand the common features of the three subclasses.

We'll start with the constructor:

```
public EvolutionaryAlgorithm
```

```
    (
      Class evolvable,
      Object[] constructorParams,
      int popSize,
      Random random,
      double mutationProbability,
      double crossoverProbability,
      Evaluator evaluator,
      Terminator terminator
    )
    throws EvolutionException;
```

and discuss each of its parameters in turn.

`evolvable` – this must be a subclass of another abstract class, `Evolvable`. An `Evolvable` is something that can be evolved by an evolutionary algorithm. It has methods for cloning, mutating etc. This class is discussed in more detail later.

`constructorParams` – this is an array of objects that are passed to the constructor of the class `Evolvable` in order to create instances of that class. Because of this, the parameters of the constructor of an `Evolvable` must be `Objects` (not integers or other primitive types.)

`random` – this is an object of class `java.util.Random` that is used by the evolutionary algorithm whenever randomness is needed.

`mutationProbability` – this is a double between 0.0 and 1.0. It is passed as a parameter when an `Evolvable` object needs to be mutated.

`crossoverProbability` – this is a double between 0.0 and 1.0. It is used by the evolutionary algorithm to decide whether or not to perform crossover on a pair of `Evolvable` objects.

`evaluator` – this is an object that knows how to calculate and set the fitness values for the Evolvable objects in a population.

`terminator` – no, it's not Arnold Schwartzeneger, it is an object that decides when to stop a run of the evolutionary algorithm.

The three subclasses of `EvolutionaryAlgorithm` have constructors that share most of these parameters. We will illustrate their use in the case study later. To solve a particular problem with an evolutionary algorithm, you usually need to write three classes: a subclass of `Evolvable` whose instances represent your solutions, a subclass of `Evaluator` whose instances know how to evaluate fitness, and a driver class to create and start the evolutionary algorithm.

Another important method of the `EvolutionaryAlgorithm` class is the `run()` method, which looks something like this (simplified):

```
    public void run()
    {
        generation = 0;
        evaluator.evaluate(population);
        initListeners();
        try
```

```
        {
            while(!terminator.terminate(
                        population,
                        population[0].getFitness(),
                        generation))
            {
                  evolve();

                  generation++;
                  evaluator.evaluate(population);
                  updateListeners();
            }
        }
        catch(Exception e)
        {
            e.printStackTrace(System.out);
        }
        doneListeners();
    }
```

As you can see, it is basically a loop, in which a new generation is created by `evolve()`, fitnesses are set by the evaluator, and the terminator checks to see whether the run is completed. The details of the `evolve()` method vary depending on the specific algorithm. If you are interested, the source for the `GeneticAlgorithm` class is available on eCourse.

Also, notice the calls to the `xxxListener()` methods. These calls allow other objects, possibly running in other threads, to keep track of the algorithm as it runs. You will probably never need to worry about this, or even to use the `run()` method, as it is called for you by the graphical interface provided by `EvolutionaryAlgorithm`:

```
    public void display();
```

This method is the normal way to run an evolutionary algorithm using the `Evolution` package. It creates a graphical control and display that you can use to start, stop, pause and single step through runs of an evolutionary algorithm. The display provides a performance graph as well as a customisable area that can be used to show the current best solution found by the algorithm.

Other useful methods provided by `EvolutionaryAlgorithm` are:

```
    // returns the best Evolvable object in the
    // current generation
    public Evolvable getBest();

    // returns the mean fitness of Evolvable objects
    // in the current population
    public double getMeanFitness();

    // returns an array containing the current population
    public Evolvable[] getPopulation();
```

```
    // returns the best Evolvable object found so far
    public Evolvable getBestEver();
```

Class GeneticAlgorithm

The constructor is

```
    public GeneticAlgorithm
        (
            Class evolvable,
            Object[] constructorParams,
            int popSize,
            Random random,
            double mutationProbability,
            double crossoverProbability,
            Evaluator evaluator,
            Terminator terminator,
            Selector selector,
            boolean elitist
        )
        throws EvolutionException;
```

This is the same as the constructor for `EvolutionaryAlgorithm`, except for the extra parameters `selector` and `elitist`.

`selector` – this is an object that is used to select `Evolvable` objects from the parent population to take part in reproduction when the evolutionary algorithm runs.

`elitist` - if this `boolean` is set to true, then the algorithm ensures that the best `Evolvable` object from each generation is copied unchanged to the next generation.

Class Evolvable

This is the class whose instances represent solutions to your problem. You must write a concrete subclass of this abstract class, and implement the following methods:

```
    // this method is really a "clone" method
    // it should copy the genome of the parent into
    // this object
    public void copy(Evolvable parent)
      throws EvolutionException;

    // this method determines how mutation works for
    // your solutions. You can use the parameter probability
    // in any way you choose.
    public void mutate(double probability)
      throws EvolutionException;

    // this method determines how crossover works for
    // your solutions. It should cross over the genetic material
```

```
// of this object and the mate.
// If you don't want crossover, leave the method empty.
public void crossover(Evolvable mate)
   throws EvolutionException;

// in this method, set any variables etc in this solution
// that are not part of the genome
public void develop()
    throws EvolutionException;
```

The class also provides the following methods, which you shouldn't override:

```
// get the fitness of this solution
public double getFitness();

// set the fitness of this solution
public void setFitness(double fitness);
```

And the following method, which you may override if you like:

```
// this method is called to draw the graphics in the top
// panel of the display created by EvolutionaryAlgorithm's
// display() method.
// The default uses toString() to print the solution.
public void draw
    (
         Graphics g,            // graphics context for drawing
         Rectangle viewport,    // rectangular area to draw into
     );
```

Abstract class Evaluator

You will need to write a class that knows how to evaluate solutions, which must be a subclass of the following:

```
public abstract class Evaluator
{
    public abstract double evaluate(Evolvable evolvable);

    public void evaluate(Evolvable[] population)
    {
        for(int i = 0; i < population.length; i++)
        {
            population[i].setFitness(evaluate(population[i]));
        }
    }
}
```

Interface Terminator

You will also need to supply an instance of a class that implements this interface:

```
public interface Terminator
{
```

```
    public boolean terminate
        (
            Evolvable[] population,
            double bestFitness,
            int generations
         );
}
```

Two suitable classes are provided in the package. Their constructors are described below:

```
public GenerationsTerminator(int maxGenerations);

public FitnessTerminator(double targetFitness);
```

The first returns true when the right number of generations have passed, and the second returns true when the best fitness reaches the target value.

Interface Selector

You will need to supply an instance of a class that implements this interface:

```
public interface Selector
{
    public void initSelection(double[] fitnesses);
    public int nextSelection();
}
```

Three suitable classes are provided in the package. Their constructors are described below:

```
public RouletteWheelSelector
    (
        Random random
    );

public StochasticUniformSelector
    (
        Random random
    );

public TournamentSelector
    (
        Random random,
        int tournamentSize
    );
```

The first one implements roulette wheel selection as in the text book. The second implements an improved version of roulette wheel selection. The last implements a scheme where `tournamentSize` solutions are chosen at random, and the fittest individual is allowed to reproduce.

The case study: Maximum value of $15x - x^2$

This is the example in the text book, Section 7.3. The first thing we need to do is to create a class that describes the parameters of the problem to be solved. This is not required, but is good software engineering practice. Here is a possible implementation:

```
public class NegExampleProblem
{
    public double evaluate(int arg)
    {
       return 15.0*arg - arg*arg;
    }

    public int getNBits()
    {
        return nBits;
    }

    private static final int nBits = 4;
}
```

Next, we need to write a subclass of `Evolvable`, whose instances represent solutions to the problem. As outlined in the text book, for this problem a 4-bit binary genome has been chosen. The Evolution package provides an abstract class `BinaryEvolvable` that we can subclass. This class provides implementations for `copy()`, `mutate()` and `crossover()` that we can use. The constructor is

```
    public BinaryEvolvable
        (
             Random random,
             int chromosomeLength, // the number of bits
             int crossoverType // choose from
                               // EvolutionaryAlgorithm.ONEPOINT and
                               // EvolutionaryAlgorithm.TWOPOINT
        );
```

The class also provides a protected attribute, which is the genome, a `boolean` array representing a binary string:

```
    protected boolean[] chromosome;
```

Here is a possible implementation of an Evolvable class for this problem:

```
public class NegExampleEvolvable extends BinaryEvolvable
{
    public NegExampleEvolvable
        (
             Random random,
             NegExampleProblem problem,
             Integer crossoverType
        )
    {
        super(random, problem.getNBits(), crossoverType.intValue());
    }

    public void develop()
    {
        value = 0;

        int pow2 = 1;
        for(int i = 0; i < chromosome.length; i++)
        {
            if(chromosome[chromosome.length-1-i]) value += pow2;
            pow2 *= 2;
        }
    }

    public String toString()
    {
        String result = "";
        for(int i = 0; i < chromosome.length; i++)
        {
            result += chromosome[i]? "1 ": "0 ";
        }
        result += "= " + value;

        return result;
    }

    public int getValue()
    {
        return value;
    }

    private int value;
}
```

The constructor just calls the superclass constructor. Note that the parameters to the constructor need to be `Objects`, so `crossoverType` is an `Integer`, not an `int`.

The `copy()`, `mutate()` and `crossover()` methods are inherited from the superclass. The `develop()` method has been added to translate the binary string into an “x” value. The `toString()` method has been added so that we will be able to easily see what the best solution is when it is displayed.

Now we need to write an Evaluator class. Here is one possible implementation:

```
public class NegExampleEvaluator implements Evaluator
{
    public NegExampleEvaluator(NegExampleProblem problem)
    {
        this.problem = problem;
    }

    public double evaluate(Evolvable evolvable)
    {
        NegExampleEvolvable example = (NegExampleEvolvable)evolvable;

        int value = example.getValue();

        return problem.evaluate(value);
    }

    private NegExampleProblem problem;
}
```

All we need now is a driver class like this:

```
public class EvolveNegExample
{
    public static void main(String[] args)
    {
        EvolveNegExample evolver = new EvolveNegExample();

        try
        {
            evolver.go();
        }
        catch(Exception e)
        {
            e.printStackTrace(System.out);
        }
    }

    public void go() throws EvolutionException
    {
        Random random = new Random();

        NegExampleProblem problem = new NegExampleProblem();

        Object[] constructorParams =
            {
                random,
                problem,
                new Integer(EvolutionaryAlgorithm.ONEPOINT)
            };

        GeneticAlgorithm ga = new GeneticAlgorithm
            (
                NegExampleEvolvable.class,
                constructorParams,
```

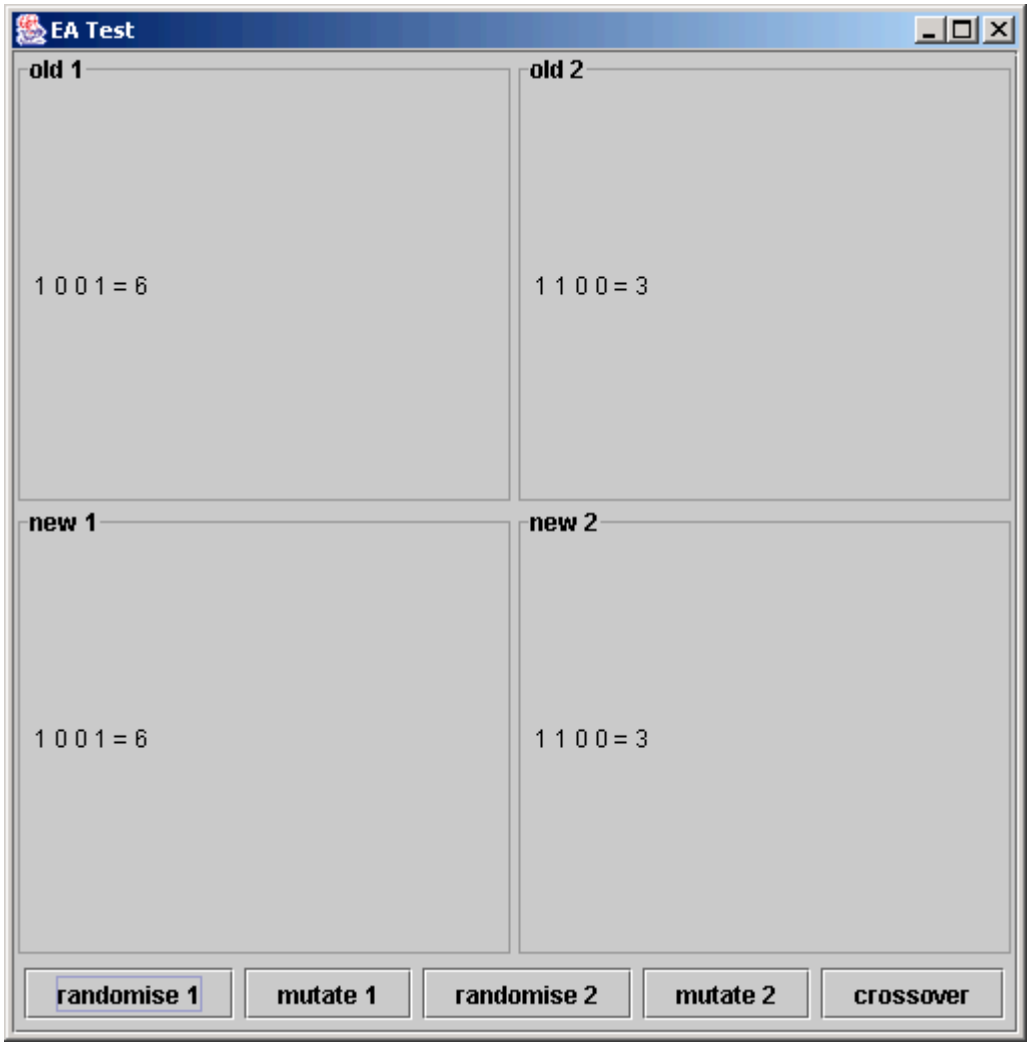
```
                POP,
                random,
                1.0/problem.getNBits(),
                CROSSOVERPROB,
                new NegExampleEvaluator(problem),
                new GenerationsTerminator(GENERATIONS),
                new StochasticUniformSelector(random),
                true
            );

        ga.test();
        //ga.display();
    }

    private static final int GENERATIONS = 20;
    private static final int POP = 20;
    private static final double CROSSOVERPROB = 0.7;
}
```

The tricky bit is creating the `constructorParams` array. The items in this array correspond to the parameters of the constructor of the `NegExampleEvolvable` class. This must be passed as the second parameter to the `GeneticAlgorithm` constructor. The first parameter is the `NegExampleEvolvable` class itself, which you can get in Java using `NegExampleEvolvable.class`.

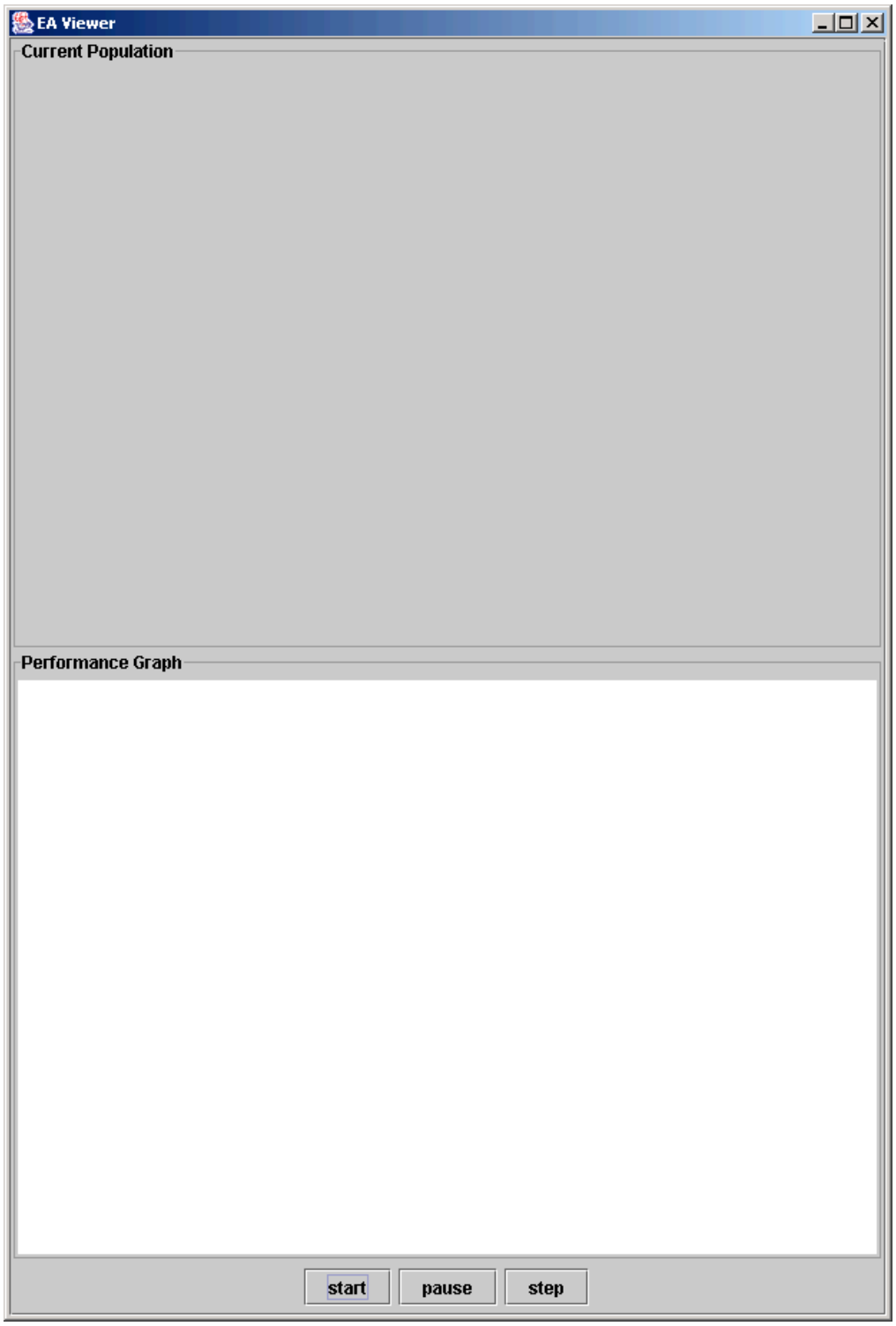
In the example above, the method `ga.test()` is called instead of `ga.display()`. This is a method you can use to check that your representation and genetic operators are working the way you expect. Compile these classes, and type in the command:

```
java EvolveNegExample
```

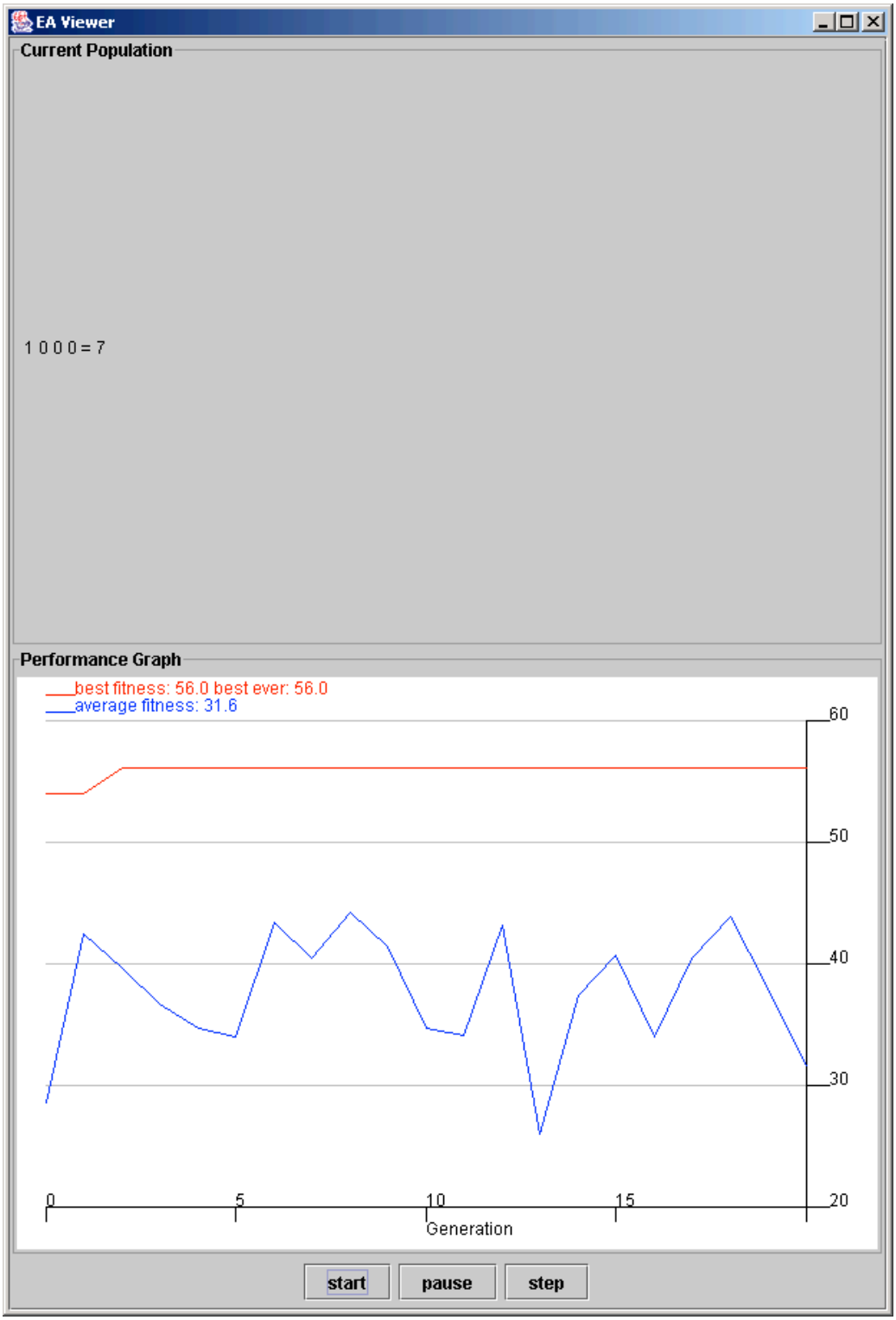
And you should see a display like the one on the next page. The display shows two evolvable solutions, one on the left and one on the right. Click on “randomise 1” to replace the solution on the left with another randomly generated one. Click on “mutate 1” to apply the mutation operator to the solution on the left. The buttons “randomise 2” and “mutate 2” operate on the solution on the right. When a solution is modified, the new version appears at the bottom, and the old version is shown on the top. Click on “crossover” to perform crossover on the two solutions.



When you are happy that your operators are working correctly, replace the call `ga.test()` with `ga.display()`, recompile and run again. You should see a display like the next one:



If you click on the “start” button, something like this should happen:



The performance graph is just like those in your text book. Here you can see it only took a couple of generations to find an optimum solution, x = 8, with fitness 56.0. Though it looks complicated, in the end, we only had to write 4 fairly small, simple classes to get this genetic algorithm going. Give it a try.