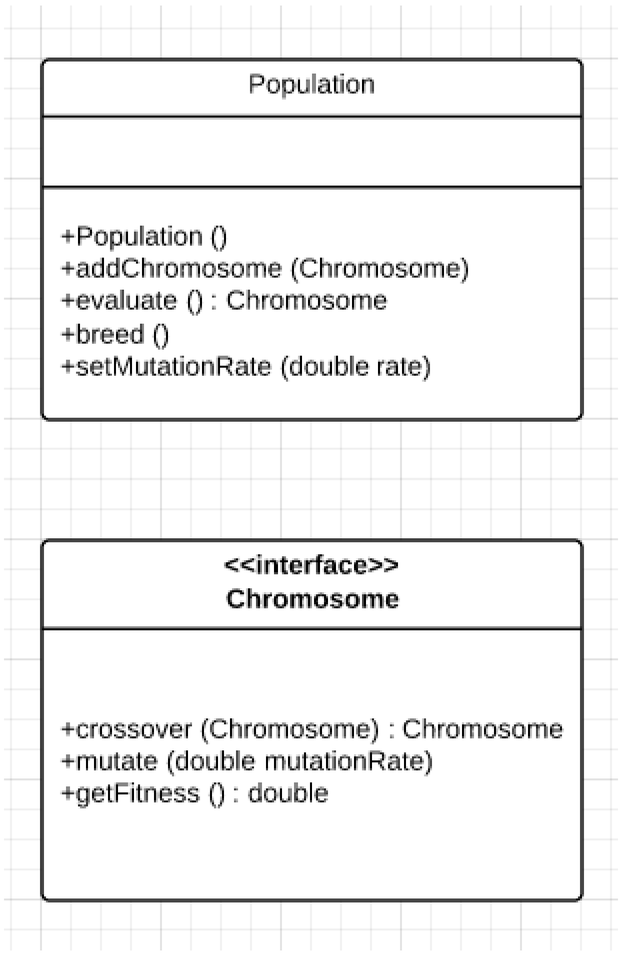
Genetic Algorithms Lab

Overview

The goal of this lab is in two parts. Part one is to write a framework for training genetic algorithms. Part two is to use that framework to solve line fitting problems.

Part One

The framework must be designed with inheritance and polymorphism, to provide the ability to use the framework for a wide variety of different problems with as few code changes as possible. You must implement the following:


Population
+Population ()
+addChromosome (Chromosome)
+evaluate () : Chromosome
+breed ()
+setMutationRate (double rate)
<<interface>>
Chromosome
+crossover (Chromosome) : Chromosome
+mutate (double mutationRate)
+getFitness () : double

The Population class will hold a collection of Chromosomes. Each chromosome represents an individual in the population.

The addChromosome method on the population is used to add one chromosome to the population. The addChromosome method must be called once per individual in the initial population.

The evaluate method on the population is called to calculate the fitness of every individual and returns the chromosome with the highest fitness.

The breed method on the population is called to create a new population by breeding the most fit individuals. Use a lottery system for selecting which individuals are most likely to be chosen for reproduction. See below for details. Note that if a child chromosome is created that has a fitness of 0, it should be discarded.

The setMutationRate is called to set the mutation rate...this should be checked during breeding after an offspring has been created, to see if that offspring needs mutated.

The Chromosome interface represents the methods a chromosome has to implement. The actual way a chromosome stores data depends on the problem being solved, so none of these methods can be implemented here. They must be implemented in a class that implements the Chromosome interface.

The getFitness method should return a number that represents how fit the chromosome is. Higher values are more fit. A chromosome that represents an illegal solution should return 0 for fitness. A chromosome that represents a perfect solution should return 1 for fitness.

The mutate method alters one part of the chromosome based on the mutation rate (what this means depends on what sort of data is in the chromosome).

The crossover method will take in a chromosome, cross it over with the current chromosome, and return a new offspring created.

Lottery System

The lottery system allows any chromosome to be selected to reproduce for the next generation, but weights the selection toward more fit individuals.

Start by identifying the chromosome that is *least* fit (but with a fitness greater than 0). Let's say that the least fit chromosome has a fitness of 0.01. All chromosomes get a number of tickets based on comparing its fitness to the least fit chromosome's fitness. So if a chromosome had a fitness of .25, and the least fit chromosome had a fitness of .01, we'd divide .25 by .01 to get 25 tickets. The least fit chromosome will, by definition, get only 1 ticket. Allow natural rounding on the result (so a result of 1.25 tickets will get 1 ticket).

At the end, generate two random numbers from 1 to the number of tickets available, and use those two chromosomes to generate a chromosome for the new population. Do that over and over until the population is back to its initial size.

It's okay for the same chromosome to be picked as both parents. The net effect is that the chromosome passed into the next generation unchanged (except possibly for mutation).

<u>Part Two</u>

Write a class that implements Chromosome that will use the framework designed in Part One to find the equation of a line given a series of points. Recall that the equation of a line is this:

```
    Ax + By + C = 0
```

The chromosome for an individual should provide values for a, b, and c, any of which may be fractional (e.g. ½, 4¾, etc) and/or negative. You will need to decide how to encode these values into the chromosome. Assume that A, B, and C need only be represented by 8-bit numerators and an 8-bit denominators.

The fitness of each chromosome will be tested against a set of points to which we are trying to fit a line. You can plug each point into the equation to see how far off the result is from 0. Values closer to 0 are more fit than values farther from 0.

That hardest part about this calculation of fitness is scaling the fitness into a 0 to 1 range. When you run a chromosome through the equation of a line given a particular set of x and y values, you will get a set of distances from 0 to some maximum value. The value of 0 coming out of the equation means that the chromosome is a perfect match (which should be a fitness of 1). To flip these values into a 0 to 1 range, use:

```
    1 / (1 + distance)
```

This gives you a set of values that are essentially flipped (better solutions have bigger values instead of smaller distances) and in the 0 to 1 range. That's your fitness for that one x and y point.

The fitness for the chromosome is the average of all those fitnesses.

Also write a main class that adds an initial population of 100 chromosomes and calls the Population class repeatedly until either 100,000 generations have passed or the best fitness is 99.999% or more. Play around with different mutation rates to see what happens.

<u>Extra Credit (5 points)</u>

Provide a graphical user interface that will display the testing points in an x-y coordinate system.

Then display the lines for each individual of each generation. Start out displaying the initial population, and then each generation draw the lines for each individual (erasing all lines from the previous generation). This visualization should show the lines getting closer and closer to a best fit for the testing points.