Optimization Exam Definition:
Enhancing the simple evolutionary algorithm

In this exam, we are going to enhance the simple evolutionary algorithm from Lesson 5, to be able to find the global optima of the previously given continuous test-problems, that is, P1, P2 and RevAckley from the mini-project. In order to do this, the following assignments shows how to improve the given simple evolutionary algorithm gradually. All files needed are included in the SimpleGeneticAlgorithm.jar file on Canvas. The following assumes that we implement the changes using Java.

Assignment 0 (General clean-up):

The original SimpleGeneticAlgorithm takes as input a binary string, which is the solution to the current problem. As we are now moving on to using real, continuous problems, we need to remove everything related to the use of the **solution** input string. For this, do the following clean – up:

1. Remove the argument **String solution** from the **runAlgorithm** method in the SimpleGeneticAlgorithm class. Further, remove its usage in the MainApp class.
2. Remove the corresponding field **byte[] solution.**
3. Remove the if – sentence on the first line of the **runAlgorithm** method.
4. Remove the **setSolution** method, as well as its usage.
5. Remove the **getMaxFitness** method.
6. Remove the for – loop in the **getFitness** method.

Assignment 1 (Using the Problem class):

In order to use the Problem class, we must take a Problem object as input to our evolutionary algorithm, as we did with the iterated hill – climber. We further need to propagate the input problem to our population and our individuals, in order to create genomes of the correct size (the number of dimensions of the problem). Perform the following changes in the SimpleGeneticAlgorithm class:

1. The method **runAlgorithm** must take a Problem object as argument. Keep the **populationSize** argument.
2. The method **getFitness** must return a double, and take a Problem object as second argument, the first being an Individual object. The method must return the **Eval** function of the Problem, given the Individuals genes (using the **getGenes** method, see below).

In the Individual class, we must also do a few changes, to go from a binary string representation, to an ArrayList of doubles as genes. Perform the following changes in the Individual class:

1. Set the field **defaultGeneLength** to be 2.
2. Set the **genes** field to be an ArrayList of doubles, and initialize it with an empty list.
3. Introduce a new field for holding a Problem reference called **problem**.
4. The Individual constructor must now take a Problem object as argument, set the **problem** field to be the input Problem object, set the defaultGeneLength to be the number of dimensions, as well as initializing the genes with values between the min and max, of the **problem**.

Peter Justesen

Exam Definition

5. Set the field **fitness** to be a double, and initialize it with the value **Double.MIN_VALUE**. This is in order to facilitate lazy evaluation, ensuring that an Individual is only evaluated once. Notice how this is performed in the original **getFitness,** mimic this, and further change the **getFitness** method to include the **problem** as a second argument to the SimpleGeneticAlgorithm's **getFitness** method.
6. A **getGenes** method must be created, that has an ArrayList of doubles as the return type, and returns **genes.** The **getSingleGene** and **setSingleGene** must now use doubles, as return type and argument type, respectively. **setSingleGene** must now set the fitness to **Double.MIN_VALUE**.

Lastly, we must make a few changes in the Population class, in order to use the Problem class. Perform the following changes in the Population class:

1. Introduce a new field for holding a Problem reference called **problem**.
2. The constructor for the Population class must now include a Problem object. This must set the **problem** field.
3. When creating new Individuals in the **createNewPopulation** method, include the **problem** as an argument to the Individual constructor. Change usage accordingly.

Assignment 2 (Introducing control parameters):

The original SimpleGeneticAlgorithm had the outermost while loop controlled by the input solution, and ran until the best individual matched the solution. This must be changed to be a number of generations instead. For this, perform the following changes in the SimpleGeneticAlgorithm class:

1) Introduce a field called **generations**, which must be initialized through an extra argument in the constructor. Change the condition of the outermost while loop to run **generations** number of times.
2) The original SimpleGeneticAlgorithm class has hardcoded the **uniformRate**, which is the crossover rate, as well as the **mutationRate**. Include these as arguments in the constructor, and set them through this instead.
3) Further, remove the field **elitism** from the original SimpleGeneticAlgorithm class, and introduce a new field called **elite** of integer type, for later use. Remove the if – sentence in the **evolvePopulation** method, which was using the **elitism** Boolean.
4) It is ***optional***, if you want to include the **tournamentSize** as an argument to the constructor, as the hardcoded value of 5 is well – chosen. If you do, include this in the experiments.

Assignment 3 (Introducing parameterized elitism):

In Assignment 2, we introduced a new parameter, controlling the elite size, that is, the number of Individuals, who survives unchanged from one generation to the next. The elite is the best individuals of our current population, which is why this must be sorted (using e.g. Collections.sort), before we include them in the next generation. Perform the following changes in the Individual class:

1) Make sure the Individual class implements the **Comparable** interface, so that Individuals can be compared to each other, to facilitate sorting. Thus, the class declaration must now include: implements Comparable<Individual>
2) The **compareTo** method must be implemented and overridden, so that it fulfils the conditions for this:

"Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object." This is with respect to the fitness value in a maximization setting.

The header of the method should look like this:

```
@Override
public int compareTo(Individual o)
```

3) In the **evolvePopulation** method, set the **elitismOffset** to the **elite** value**.** After having created the **newPopulation**, now sort the input population **pop**. Copy from index 0 to index **elitismOffset** - 1 of the input population **pop** to the **newPopulation.**

Assignment 4 (Altering the variation operators):

Lastly, change the crossover and mutation operators to work on Individuals with genes consisting of real values (Slide 10, Lesson 5). Perform the following changes in the SimpleGeneticAlgorithm class:

1) Change the **crossover** method, such that for each gene, if **(Math.random() <= uniformRate),** the gene of the output Individual (create a new) will be the average of the genes of the two parent Individuals. Create an else part, that sets the gene to be that of one of the two parents with a 50/50 probability, if the condition is not fulfilled. Return the Individual.
2) Change the **mutation** method, such that 1/10 of a Gaussion distributed value (mean 0, standard deviation 1) is added to each gene of the input Individual, if **(Math.random() <= mutationRate)**. For this, you may use the following snippet:

```
Random r = new Random();
double gene = 0.1*r.nextGaussian() + indiv.getSingleGene(dim);
```

Remember to check if the new value violates the min or max values, and set it to the corresponding border value, in that case. Using **setSingleGene** ensures evaluation, when the **getFittest()** method of the Population is called just before **evolvePopulation** is called.

Assignment 5 (Testing and reporting):

The three concrete problem classes, P1, P2 and RevAckley, are the subject of testing with your final version of the evolutionary algorithm. The overall goal is to find the global optima of the three problems, by varying the parameters of the evolutionary algorithm.

Hand – in your final version of the evolutionary algorithm as a .jar or .zip file (if anything other than Java is used) as well as a small report (max twenty pages, including figures, e.g. graphs) on Canvas.

The testing consist of varying the parameters of the evolutionary algorithm in order to find the global maximum of the test problems. Conduct the following experiments:

1. Vary the *generations* parameter. You can use a default value of 100.
2. Vary the *population size* parameter. You can use a default value of 100.
3. Vary the *elite size*. You can use a default value of 5.
4. Vary the *mutation rate.* You can use a default value of 0.025.
5. Vary the *crossover rate.* You can use a default value of 0.5.

Peter Justesen

Exam Definition

For each of your experiments, include the maximum number of evaluations, that is, the population size times the number of generations (assuming we evaluate the population once per iteration). It is ***optional*** to count the actual number of evaluations using the lazy evaluation approach. If you do, include this instead.

Further, during experimentation, print or log the function value for the best individual found in each generation, to see when there is an improvement.

Report and reflect on the following:

1. Your experiments in general.
2. The effectiveness in terms of number of evaluations, compared to the iterated hill – climber?
3. Where do you think improvement happens in the evolutionary algorithm?
4. What was the closest you got to the optimal point for each problem, during all your runs?
5. What was the best parameter setting for your evolutionary algorithm for each problem? That is, the setting that gave you the best overall closeness to the optimal point with the smallest number of evaluations.
6. What was the average closeness to the optimal points for each problem, over 20 runs?
    a. The 20 runs must be with the same parameter settings, e.g. the best found.

It is ***optional*** to experiment with the two extra test problems also included in the SimpleGeneticAlgorithm.jar file, called RevSphere and RevRosenbrock. These are the reversed (for maximization) Sphere and Rosenbrock functions from https://en.wikipedia.org/wiki/Test_functions_for_optimization with 20 and 3 dimensions, respectively (default values, easily changed). Note, that Rosenbrock is a bit hard (you might want to try a very high mutation rate, e.g. 75%, as well as high numbers of generations, population size and elite), and that Sphere is very smooth, making it ideal for e.g. the iterated hill – climber.

It is further ***optional*** to experiment with other kinds of variation and selection operators – in this case, give the intent and the description of these in the report.

If any other programming language than Java has been used, include compilation and running instructions, as well as a pre-compiled test file, e.g. an .exe file. In any case, remember to check, that you can properly import your exported files.

Deadline for the exam is Friday, October 11 at 16.00 on Wiseflow.