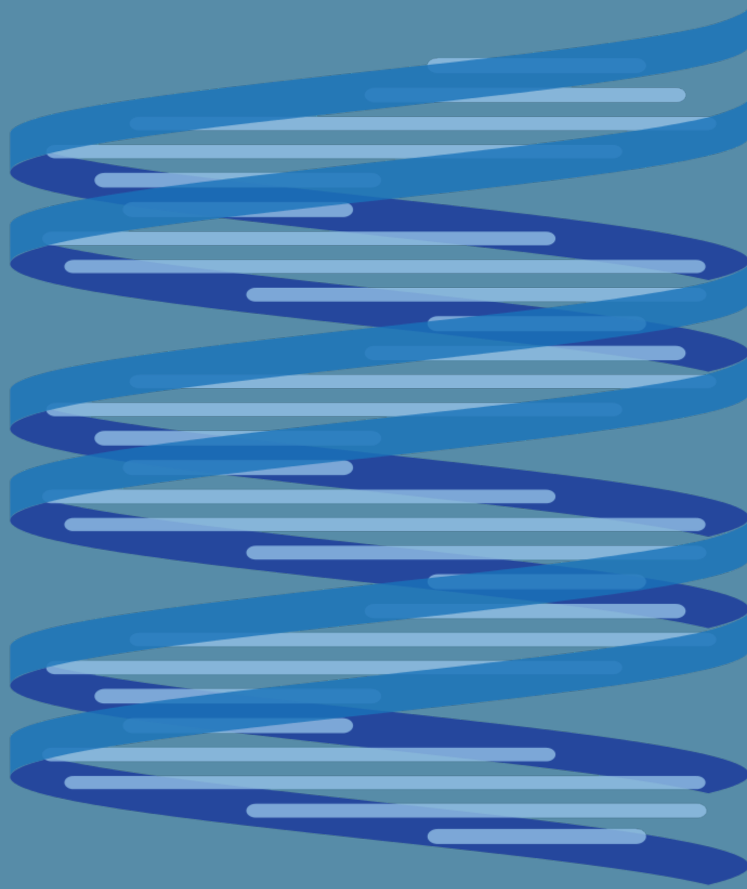
Jenetics

Java Genetic Algorithm Library Manual

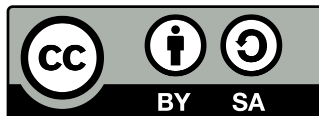


Franz Wilhelmstötter

Franz Wilhelmstötter
franz.wilhelmstoetter@gmx.at

http://jenetics.sourceforge.net/

2.0.2—2014/10/03



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Austria License. To view a copy of this license, visithttp://creativecommons.org/licenses/by-sa/3.0/at/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Abstract

Jenetics is an **Genetic Algorithm**, respectively an **Evolutionary Algorithm**, library written in Java. It is designed with a clear separation of the several algorithm concepts, e. g. `Gene`, `Chromosome`, `Genotype`, `Phenotype`, `Population` and fitness `Function`. **Jenetics** allows you to minimize or maximize the given fitness function without tweaking it. This manual describes the concepts implemented in the **Jenetics** project and gives you examples and best practice tips.

Contents

1 Introduction **1**

2 Getting started **2**

3 Base classes **3**

4 Genetic operators **5**
4.1 Selectors 5
4.2 Alterers 9
4.2.1 Mutation 10
4.2.2 Recombination 11

5 Nuts and bolts **13**
5.1 Concurrency 13
5.2 Statistics 13
5.3 Termination 15
5.4 Randomness 16
5.5 Serialization 19
5.6 Utility classes 20

6 Extending Jenetics **23**
6.1 Genes 23
6.2 Chromosomes 23
6.3 Selectors 24
6.4 Alterers 24
6.5 Statistics 25

7 Examples **27**
7.1 Ones counting 27
7.2 Real function 28
7.3 0/1 Knapsack 30
7.4 Traveling salesman 32

8 Build **35**

9 License **37**

References **38**

1 Introduction

The **Jenetics** project is a Java[1] library which provides an genetic algorithm (GA) implementation. The project has no dependencies to other libraries. For building the library from the sources you only have the JDK 1.7 and Gradle[2] to be installed, all other dependencies are available in the package which you can download from `https://sourceforge.net/projects/jenetics/files/jenetics/` or `https://bitbucket.org/fwilhelm/jenetics/downloads`. For additional informations revere to the build instructions at section 8 on page 35.

This manual is not an introduction or a tutorial for genetic algorithms in general. It is assumed that the reader has a knowledge about the structure and the functionality of genetic algorithms. A good GA introduction can be found in[7],[5],[6] or[8].

The order of the single execution steps of genetic algorithm may slightly differ from implementation to implementation. Listing1 shows the pseudo-code of the **Jenetics** genetic algorithm steps.

1. $P_0 \leftarrow P_{initial}$
2. $F(P_0)$
3. **while** $!finished$ **do**
4. $\quad g \leftarrow g + 1$
5. $\quad S_g \leftarrow select_S(P_{g-1})$
6. $\quad O_g \leftarrow select_O(P_{g-1})$
7. $\quad O_g \leftarrow alter(O_g)$
8. $\quad P_g \leftarrow filter[g_i \geq g_{max}](S_g) + filter[g_i \geq g_{max}](O_g)$
9. $\quad F(P_g)$

Listing 1: Genetic algorithm

Line (1) creates the initial population and line (2) calculates the fitness value of the individuals. (This is done by the `GeneticAlgorithm.setup()` method.) Line (4) increases the generation number and line (5) and (6) selects the survivor and the offspring population. The offspring/survivor fraction is determined by the `offspringFraction` property of the GA. The selected offspring are altered in line (7). The next line combines the survivor population and the altered offspring population—after removing the *died* individuals—to the new population. The steps from line (4) to (9) are repeated until a given termination criterion is fulfilled.

Figure 1.1 shows the class-diagram of the main structures of the GA. The `Gene` is the base of the class structure.`Genes` are aggregated in `Chromosomes`. One to n `Chromosomes` are aggregated in `Genotypes`. A `Genotype` and a fitness `Function` form the `Phenotype, which` are collected into a `Population`.

[1]The library is build with and depends on Java SE 7:`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

[2]The used Gradle version is 1.10 and it is downloaded automatically if you build the library with the wrapper script `./gradlew`:`http://www.gradle.org/`

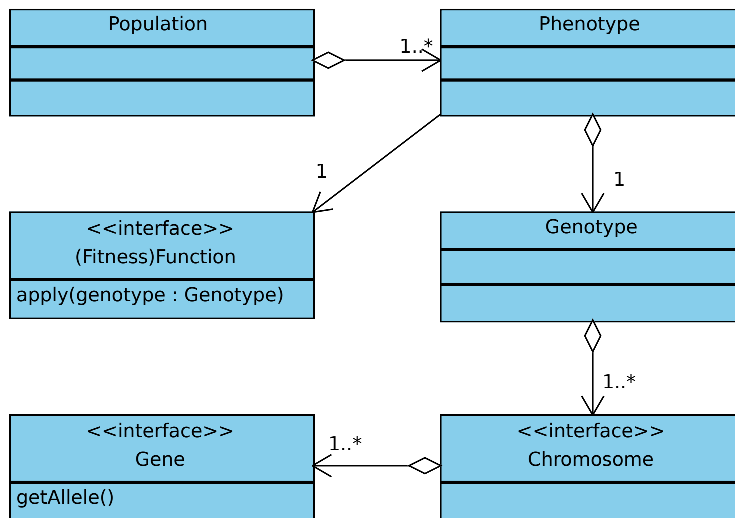


Figure 1.1: Structure diagram

2 Getting started

The minimum GA setup needs a genotype factory, `Factory<Genotype<?>>`, and a fitness `Function`. The `Genotype` implements the `Factory` interface and can therefore be used as *prototype* for creating the initial `Population` and for creating new random `Genotypes`.

```
public static void main(String[] args) {
    Factory<Genotype<BitGene>> gtf = Genotype.of(
        BitChromosome.of(10, 0.5)
    );
    Function<Genotype<BitGene>, Double> ff = ...
    GeneticAlgorithm<BitGene, Double>
    ga = new GeneticAlgorithm<>(gtf, ff, Optimize.MAXIMUM);

    ga.setup();
    ga.evolve(100);
    System.out.println(ga.getBestPhenotype());
}
```

Listing 2: Simple GA setup

The genotype factory, `gtf`, in the example in listing 2, will create genotypes which consists of one `BitChromosome` with length 10. The one to zero probability of the newly created genotypes is set to 0.5. The fitness function is parameterized with a `BitGene` and a `Double`. That means that the fitness function is calculating the fitness value as `Double` values. The return type of the fitness function must be at least of the type `Comparable`. The `GeneticAlgorithm` object is then created with the genotype factory and the fitness function. In this example the GA tries to maximize the fitness function. If you want to find the minimal value you have to change the `optimize` parameter from `Optimize.MAXIMUM` to `Optimize.MINIMUM`. The `ga.setup()` call creates the initial population and calculates its fitness values. Then the GA evolves 100

generations (`ga.evolve(100)`) an prints the best phenotype found so far onto the console.

In an more advanced setup you may want to change the default mutation and/or selection strategies.

```
public static void main(String[] args) {
    ...
    ga.setSelectors(new RouletteWheelSelector<BitGene>());
    ga.setAlterers(
        new SinglePointCrossover<BitGene>(0.1),
        new Mutator<BitGene>(0.01)
    );

    ga.setup();
    ga.evolve(100);
    System.out.println(ga.getBestPhenotype());
}
```

The selection strategy for offspring and survivors in the given example are set to the roulette-wheel selector. It is also possible to set the selector for offspring and survivors independently with the `setOffspringSelector` and `setSurvivorSelector` methods. The alterers are concatenated, at first the crossover (with probability 0.1) is performed and then the chromosomes are mutated (with probability 0.01). For a detailed description of the available genetic operators revere to chapter 4 on page 5.

3 Base classes

This chapter describes the classes which are used to transform the actual problem into a structure that can be used by **Jenetics**.[3]

Genotype The central class, the GA is working with, is the `Genotype`. It is the *structural* representative of an individual. The `Phenotype` class is the *actual* representative of an individual, but only consists of the genotype and the fitness function and doesn't change the basic structure. The phenotype is *only* a container which forms the environment of the genotype.

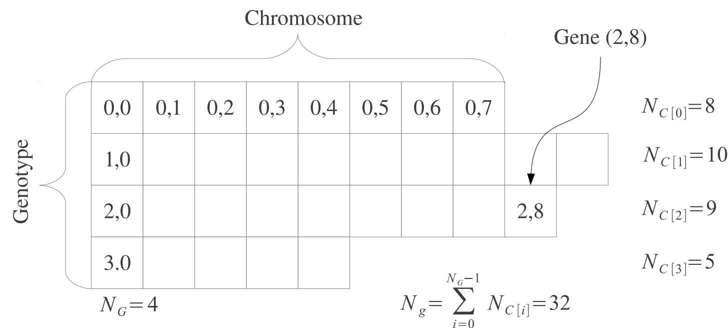


Figure 3.1: Genotype structure

[3]The documentation of the whole API is part of the download package or can be viewed online: `http://jenetics.sourceforge.net/javadoc/index.html`.

Figure 3.1 shows the genotype structure. A genotype consists of N_G chromosomes and a chromosome consists of $N_{C[i]}$ genes (depending on the chromosome). The overall number of genes of a genotype is given by the sum of the chromosome's genes, which can be accessed via the `Genotype.getNumberOfGenes()` method:

$$N_g = \sum_{i=0}^{N_G-1} N_{C[i]} \tag{3.1}$$

The chromosomes of a genotype doesn't have to have necessarily the same size. It is only required that all genes are from the same type and the genes within a chromosome have the same constraints; e. g. the same min- and max values for numerical genes.

```
1 Genotype<DoubleGene> genotype = Genotype.of(
2     DoubleChromosome.of(0.0, 1.0, 8),
3     DoubleChromosome.of(1.0, 2.0, 10),
4     DoubleChromosome.of(0.0, 10.0, 9),
5     DoubleChromosome.of(0.1, 0.9, 5)
6 );
```

The code snippet in the listing above creates a genotype with the same structure as shown in figure 3.1. In this example the `DoubleGene` has been chosen as gene type.

Fitness function The fitness function is also an important part when modeling the GA. It takes a genotype as argument and returns, at least, a `Comparable` object as result—the fitness value. This allows the GA, respectively the selection operators, to select the offspring- and survivor population. Some selectors have stronger requirements to the fitness value than a `Comparable`, but this constraints is checked by the Java type system at compile time.

The following example shows the simplest possible fitness function. It's the identity function and returns the allele of an1x1 *float* genotype.

```
1 class Id implements Function<Genotype<DoubleGene>, Double> {
2     @Override
3     public Double apply(Genotype<DoubleGene> genotype) {
4         return genotype.getGene().getAllele();
5     }
6 }
```

The first type parameter of the `Function` defines the kind of genotype from which the fitness value is calculated and the second type parameter determines the return type.

Fitness scaler The fitness value, calculated by the fitness function, is treated as the *raw*-fitness of an individual. The **Jenetics** library allows you to apply an additional scaling function on the raw-fitness to form the fitness value which is used by the selectors. This can be useful when using probability selectors (see chapter 4.1 on page 7), where the actual amount of the fitness value influences the selection probability. In such cases, the fitness scaler gives you additional flexibility when selecting offspring and survivors. In the default configuration the raw-fitness is equal to the actual fitness value, that means, the used fitness scaler is the identity function.

```java
class Sqrt extends Function<Double, Double> {
    @Override
    public Double apply(final Double value) {
        return sqrt(value);
    }
}
```

The given listing shows a fitness scaler which reduces the the raw-fitness to its square root. This gives weaker individuals a greater changes being selected and weakens the influence of *super*-individuals.

When using a fitness scaler you have to take care that your scaler doesn't *destroy* your fitness value. This can be the case when your fitness value is negative and your fitness scaler squares the value. Trying to find the minimum will not work in this configuration.

Genes Genes are the basic building blocks of the **Jenetics** library. They contain the actual information (alleles) of the encoded solution. The available Gene implementations should be sufficient for a very wide range of problem domains. Refer to chapter 6.1 on page 23 for how to implement your own gene types.

4 Genetic operators

Genetic operators are used for creating *genetic* diversity (`Alterer`) and select potentially useful solutions for recombination (`Selector`). This section gives you an overview about the genetic operators available in the **Jenetics** library. It also contains some theoretical information, which should help you to choose the right combination of operators and parameters, for the problem to be solved.

4.1 Selectors

Selectors are responsible for selecting a given number of individuals from the population. The selectors are used to divide the population into *survivors* and *offspring*. The selectors for offspring and for the survivors can be chosen independently.

```java
final GeneticAlgorithm<DoubleGene, Double> ga = ...
ga.setOffspringFraction(0.7);
ga.setSurvivorSelector(
    new RouletteWheelSelector<DoubleGene, Double>()
);
ga.setOffspringSelector(
    new TournamentSelector<DoubleGene, Double>()
);
```

The `offspringFraction` property, $f_O \in [0, 1]$, determines the number of selected offspring

$$N_{O_g} = \|O_g\| = \text{rint}\left(\|P_g\| \cdot f_O\right) \tag{4.1}$$

and the number of selected survivors

$$N_{S_g} = \|S_g\| = \|P_g\| - \|O_g\| \,. \tag{4.2}$$

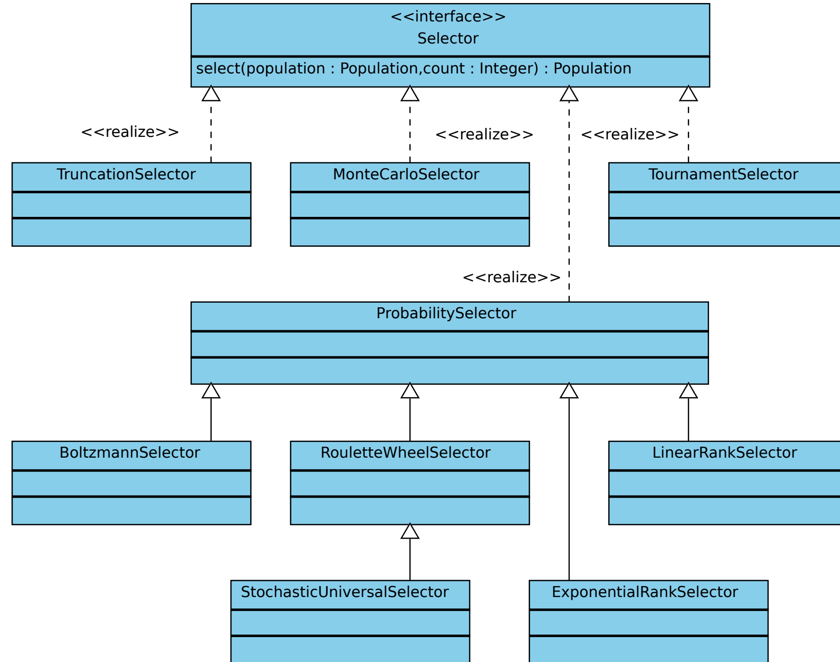


Figure 4.1: `Selector` class diagram

Figure 4.1 shows the whole class hierarchy of the currently available selectors. On the top the hierarchy is the `Selector` interface with the `select` method. Beside the well known standard selector implementation the `ProbabilitySelector` is the base of a set of fitness proportional selectors.

Tournament selector In tournament selection the best individual from a random sample of s individuals is chosen from the population Pg. The samples are drawn with replacement. An individual will win a tournament only if the fitness is greater than the fitness of the other $s-1$ competitors. Note that the worst individual never survives, and the best individual wins in all the tournaments it participates. The selection pressure can be varied by changing the tournament size$s$. For large values of s, weak individuals have less chance of being selected.

Truncation selector In truncation selection individuals are sorted according to their fitness. Only the n best individuals are selected. The truncation selection is a very basic selection algorithm. It has it's strength in fast selecting individuals in large populations, but is not very often used in practice.

Monte Carlo selector The Monte Carlo selector selects the individuals from a given population randomly. This selector can be used to measure the performance of a other selectors. In general, the performance of a selector should be better than the selection performance of the Monte Carlo selector.

Probability selectors Probability selectors are a variation of *fitness proportional* selectors and selects individuals from a given population based on it's *selection* probability $P(i)$. Fitness proportional selection works as shown in

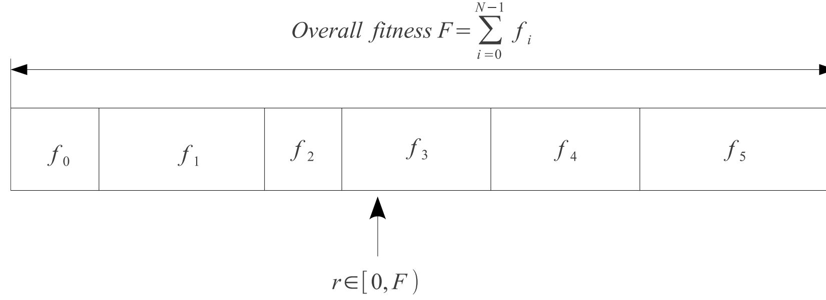


Figure 4.2: Fitness proportional selection

figure 4.2. An uniform distributed random number $r \in [0, F)$ specifies which individual is selected, by argument minimization:

$$i \leftarrow \underset{n\in[0,N)}{\text{minarg}} \left\{ r < \sum_{i=0}^{n} f_i \right\}, \tag{4.3}$$

where N is the number of individuals and f_i the fitness value of the ith individual. The probability selector works the same way, only the fitness value f_i is replaced by the individual's selection probability $P(i)$. It is not necessary to sort the population. The selection probability of an individual i follows a binomial distribution

$$P(i,k) = \left(\begin{array}{c} n \\ k \end{array} \right) P(i)^k \left(1 - P(i)\right)^{n-k} \tag{4.4}$$

where n is the overall number of selected individuals and k the number of individual$i$ in the set of selected individuals. The runtime complexity of the implemented probability selectors is $O(n + \log(n))$ instead of $O(n^2)$ as for the naive approach: *A binary (index) search is performed on the summed probability array.*

Roulette-wheel selector The roulette-wheel selector is also known as fitness proportional selector. In the **Jenetics** library it is implemented as *probability* selector. The fitness value f_i is used to calculate the selection probability of individual i.

$$P(i) = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{4.5}$$

Selecting n individuals from a given population is equivalent to play n times on the roulette-wheel. The population don't have to be sorted before selecting the individuals. Roulette-wheel selection is one of the traditional selection strategies.

Linear-rank selector In linear-ranking selection the individuals are sorted according to their fitness values. The rank N is assigned to the best individual and the rank 1 to the worst individual. The selection probability $P(i)$ of individual$i$ is linearly assigned to the individuals according to their rank.

$$P(i) = \frac{1}{N}\left(n^- + \left(n^+ - n^-\right)\frac{i-1}{N-1}\right). \tag{4.6}$$

Here $\frac{n^-}{N}$ is the probability of the worst individual to be selected and $\frac{n^+}{N}$ the probability of the best individual to be selected. As the population size is held constant, the condition $n^+ = 2 - n^-$ and $n^- \geq 0$ must be fulfilled. Note that all individuals get a different rank, respectively a different selection probability, even if they have the same fitness value.[4]

Exponential-rank selector An alternative to the *weak* linear-rank selector is to assign survival probabilities to the sorted individuals using an exponential function:

$$P(i) = (c-1)\frac{c^{i-1}}{c^N - 1}, \tag{4.7}$$

where$c$ must within the range $[0 \dots 1)$. A small value of c increases the probability of the best individual to be selected. If c is set to zero, the selection probability of the best individual is set to one. The selection probability of all other individuals is zero. A value near one equalizes the selection probabilities. This selector sorts the population in descending order before calculating the selection probabilities.

Boltzmann selector The selection probability of the Boltzmann selector is defined as

$$P(i) = \frac{\mathrm{e}^{b \cdot f_i}}{Z}, \tag{4.8}$$

where b is a parameter which controls the selection intensity and Z is defined as

$$Z = \sum_{i=1}^{n} \mathrm{e}^{f_i}. \tag{4.9}$$

Positive values of b increases the selection probability of individuals with high fitness values and negative values of b decreases it. If b is zero, the selection probability of all individuals is set to $\frac{1}{N}$.

Stochastic-universal selector Stochastic-universal selection [1] (SUS) is a method for selecting individuals according to some given probability in a way that minimizes the chance of fluctuations. It can be viewed as a type of roulette game where we now have p equally spaced points which we spin. SUS uses a single random value for selecting individuals by choosing them at equally spaced

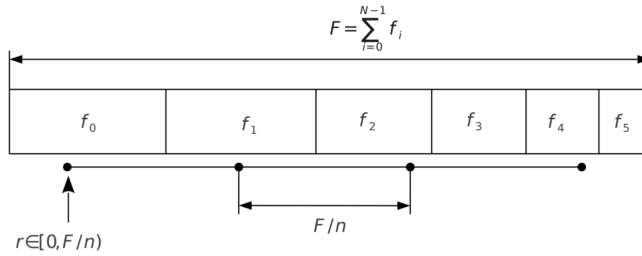


Figure 4.3: Stochastic-universal selection

intervals. The selection method was introduced by James Baker. [2] Figure 4.3 shows the function of the stochastic-universal selection, where n is the number of individuals to select. Stochastic universal sampling ensures a selection of offspring, which is closer to what is deserved than roulette wheel selection.[7]

4.2 Alterers

Alterers are responsible for the genetic diversity of the genetic algorithm. The alterer types used in **Jenetics** are

1. mutation and
2. recombination (e. g. crossover).

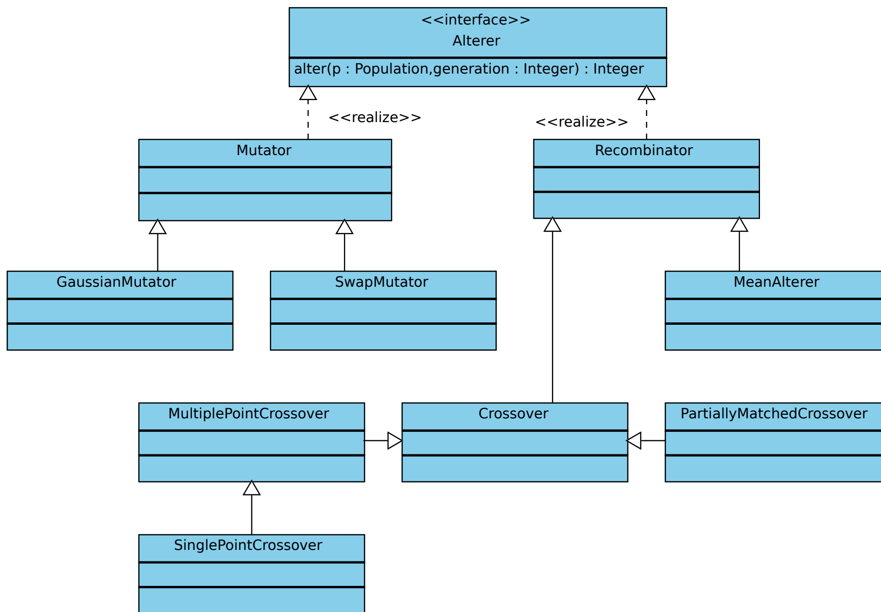


Figure 4.4: `Alterer` class diagram

4.2.1 Mutation

There are two distinct roles *mutation* plays in an Genetic algorithm:

1. Exploring the search space: By making small moves, mutation allows a population to explore the search space. This exploration is often slow compared to crossover, but in problems where crossover is disruptive this can be an important way to explore the landscape.
2. Maintaining diversity: Mutation prevents a population from correlating. Even if most of the search is being performed by crossover, mutation can be vital to provide the diversity which crossover needs.

The mutation probability, $P(m)$, is the parameter that must be optimized. The optimal value of the mutation rate depends on the role mutation plays. If mutation is the only source of exploration (if there is no crossover), the mutation rate should be set to a value that ensures that a reasonable neighborhood of solutions is explored.

The mutation probability, $P(m)$, is defined as the probability that a specific gene, over the whole population, is mutated. That means, the (average) number of genes mutated by a mutator is

$$\hat{\mu} = N_P \cdot N_g \cdot P(m) \tag{4.10}$$

where N_g is the number of available genes of a genotype and N_P the population size (revere to equation 3.1 on page 4).

Mutator The mutator has to deal with the problem, that the genes are arranged in a $3D$ structure (see chapter 3). The mutator selects the gene which will be mutated in three steps:

1. Select a genotype $G[i]$ from the population with probability $P_G(m)$,
2. select a chromosome $C[j]$ from the selected genotype $G[i]$ with probability$P_C(m)$ and
3. select a gene $g[k]$ from the selected chromosome $C[j]$ with probability $P_g(m)$.

The needed *sub*-selection probabilities are set to

$$P_G(m) = P_C(m) = P_g(m) = \sqrt[3]{P(m)}. \tag{4.11}$$

Gaussian mutator The Gaussian mutator performs the mutation of number genes. This mutator picks a new value based on a Gaussian distribution around the current value of the gene. The variance of the new value (before clipping to the allowed gene range) will be

$$\hat{\sigma}^2 = \left(\frac{g_{max} - g_{min}}{4}\right)^2 \tag{4.12}$$

where g_{min} and g_{max} are the valid minimum and maximum values of the number gene. The new value will be cropped to the gene's boundaries.

Swap mutator The swap mutator changes the order of genes in a chromosome, with the hope of bringing related genes closer together, thereby facilitating the production of building blocks. This mutation operator can also be used for combinatorial problems, where no duplicated genes within a chromosome are allowed, e. g. for the TSP.

4.2.2 Recombination

An enhanced genetic algorithm (EGA) combine elements of existing solutions in order to create a new solution, with some of the properties of each parents. Recombination creates a new chromosome by combining parts of two (or more) parent chromosomes. This combination of chromosomes can be made by selecting one or more crossover points, splitting these chromosomes on the selected points, and merge those portions of different chromosomes to form new ones.

Because of the possible different chromosome length and/or chromosome constraints within a genotype, only chromosomes with the same genotype position are recombined.

The recombination probability, $P(r)$, determines the probability that a given individual (genotype) of a population is selected for recombination. The (mean) number of changed individuals depend on the concrete implementation and can be vary from $P(r) \cdot N_G$ to $P(r) \cdot N_G \cdot O_R$, where O_R is the order of the recombination, which is the number of individuals involved in the`re combine` method.

Single-point crossover The single-point crossover changes two children chromosomes by taking two chromosomes and cutting them at some, randomly chosen, site. If we create a child and its complement we preserve the total number of genes in the population, preventing any genetic drift. Single-point crossover is the classic form of crossover. However, it produces very slow mixing compared with multi-point crossover or uniform crossover. For problems where the site position has some intrinsic meaning to the problem single-point crossover can lead to smaller disruption than multiple-point or uniform crossover.

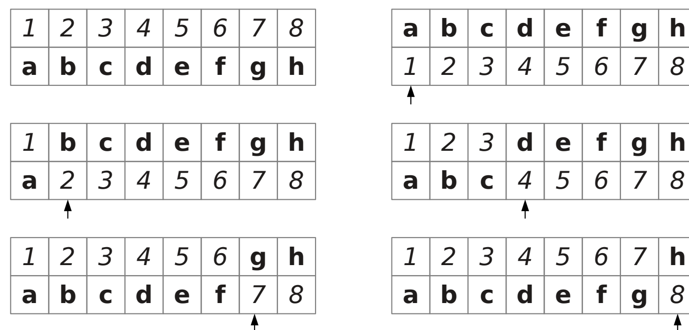


Figure 4.5: Single-point crossover

Figure 4.5 shows how the `SinglePointCrossover` class is performing the crossover for different crossover points—in the given example for the chromosome indexes 0, 1, 3, 6 and 7.

Multi-point crossover If the `MultiPointCrossover` class is created with one crossover point, it behaves exactly like the single-point crossover. The following picture shows how the Multi-point crossover works with two crossover points, defined at index 1 and 4.

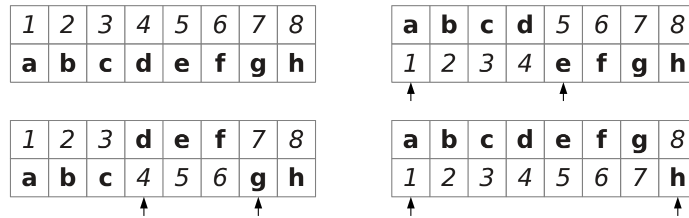


Figure 4.6: 2-point crossover

Figure 4.7 you can see how the crossover works for an odd number of crossover points.

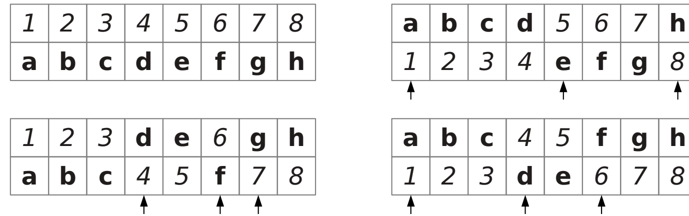


Figure 4.7: 3-point crossover

Partially-matched crossover The partially-matched crossover guarantees that all genes are found exactly once in each chromosome. No gene is duplicated by this crossover strategy. The partially-matched crossover (PMX) can be applied usefully in the TSP or other permutation problem encodings. Permutation encoding is useful for all problems where the fitness only depends on the ordering of the genes within the chromosome. This is the case in many combinatorial optimization problems. Other crossover operators for combinatorial optimization are:

- order crossover
- cycle crossover
- edge recombination crossover
- edge assembly crossover

The PMX is similar to the two-point crossover. A crossing region is chosen by selecting two crossing points (see figure 4.8 *a*)). After performing the crossover we–normally–got two invalid chromosomes (figure 4.8 *b*)). Chromosome1 contains the value 6 twice and misses the value 3. On the other side chromosome 2 contains the value 3 twice and misses the value 6. We can observe that this

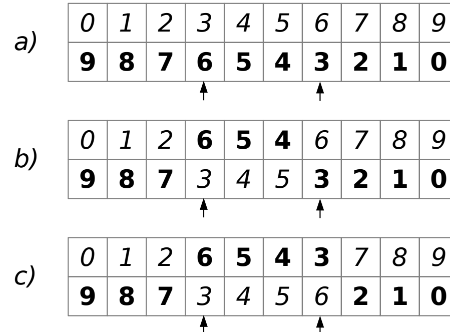


Figure 4.8: Partially-matched crossover. *a)* shows the original chromosomes, *b)* the (invalid) chromosomes after crossover and *c)* the repaired chromosomes.

crossover is equivalent to the exchange of the values 3→6, 4→5 and 5→4. To repair the two chromosomes we have to apply this exchange outside the crossing region (figure 4.8 *b*)).

5 Nuts and bolts

5.1 Concurrency

The **Jenetics** library parallelizes independent task whenever possible. Especially the evaluation of the fitness function is done concurrently. That means that the GA's fitness-function must be thread safe, because it is shared by all phenotypes of a population. The easiest way for achieving thread-safety is to make the fitness function immutable and re-entrant. The used `Executor` can be defined when creating the `GeneticAlgorithm` object.

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class Main {
    public static void main(final String[] args) {
        // Using 10 threads for evolving.
        Executor executor = Executors.newFixedThreadPool(10);
        Factory<Genotype<DoubleGene>> factory = ...
        Function<Genotype<DoubleGene>, Double> ff = ...
        GeneticAlgorithm<DoubleGene, Double> ga =
            new GeneticAlgorithm<>(factory, ff, executor);
        ...
    }
}
```

If no `Executor` is given, **Jenetics** uses a default `ForkJoinPool` for concurrency.

5.2 Statistics

The `GeneticAlgorithm` class offers population- and timing-statistics after every evolve step. This information can be used to measure the performance

of the GA, or to implement a more sophisticated termination strategy than `evolve(100)`. Have a look at chapter 5.3 for more information about GA termination.

```
GeneticAlgorithm<DoubleGene, Double> ga = ...
ga.setup();
for (int i = 0; i < 100; ++i) {
    ga.evolve();
    System.out.println(ga.getStatistics());
}
```

The statistics object returned by the GA stores information—among other things—about the best- and the worst phenotype and some timing information, for performance analysis. The following listing shows the console output of an actual GA statistics.

```
+---------------------------------------------------------+
|  Population Statistics                                  |
+---------------------------------------------------------+
|                      Age mean: 0.70000000000            |
|                  Age variance: 0.45555555556            |
|                       Samples: 10                       |
|                  Best fitness: 0.9999419432876977       |
|                 Worst fitness: 0.9955246169910878       |
+---------------------------------------------------------+
```

If the fitness value is a number type you can change the statistics calculator of the GA to get more specific statistics information. The listing shows how to set the number-statistics calculator for the GA.

```
final GeneticAlgorithm<DoubleGene, Double> ga = ...
ga.setStatisticsCalculator(
    new NumberStatistics.Calculator<DoubleGene, Double>()
);
ga.setup();
for (int i = 0; i < 100; ++i) {
    ga.evolve();
    System.out.println(ga.getStatistics());
}
```

With the new statistics calculator the console output for an statistics object will look like the following listing. This is because instead of an `Statistics` object an `NumberStatistics` object is returned by the `getStatistics()` method.

```
+---------------------------------------------------------+
|  Population Statistics                                  |
+---------------------------------------------------------+
|                      Age mean: 0.70000000000            |
|                  Age variance: 0.90000000000            |
|                       Samples: 10                       |
|                  Best fitness: 0.996850999047305        |
|                 Worst fitness: 0.77027602016712726      |
+---------------------------------------------------------+
+---------------------------------------------------------+
|  Fitness Statistics                                     |
+---------------------------------------------------------+
|                  Fitness mean: 0.95754066252            |
|              Fitness variance: 0.00505962871            |
|         Fitness error of mean: 0.30280094458            |
+---------------------------------------------------------+
```

For computation-performance analysis it might be interesting in which processing steps the computation time is spent. On *generation* basis this can be accessed via the `ga.getStatistics().getTime()` property. The overall calculation time statistics is available via the `ga.getTimeStatistics()` method.

```
+---------------------------------------------------------+
| Time Statistics                                         |
+---------------------------------------------------------+
|                  Select time: 0.00301738300             |
|                   Alter time: 0.00385119900             |
|                 Combine time: 0.00200538900             |
|      Fitness calculation time: 0.01767134600            |
|   Statistics calculation time: 0.02445853400            |
|        Overall execution time: 0.05306906200            |
+---------------------------------------------------------+
```

The console output of the time statistics looks like listing above and has the same format for the overall- and the generation time statistics.

5.3 Termination

The easiest way to terminate an GA is to evolve a specific number of generations. This works well for most problems and termination is guaranteed. With the statistics object, which is available for every generation, you can define a more advanced termination strategy.

```
GeneticAlgorithm<DoubleGene, Double> ga = ...
// Defining your termination function.
Function<Statistics<DoubleGene, Double>, Boolean> until = ...

ga.setup();
ga.evolve(until);

// Using the 'termination.SteadyFitness' terminator.
ga.evolve(termination.SteadyFitness(5));
```

The GA terminates when the termination function returns false. **Jenetics** comes with some default terminations functions implemented in the `termination` class in the `org.jenetics` package. The steady-fitness terminator for example finishes the *evolution* if the best fitness value doesn't increase for 5 consecutive generations.

```
class SteadyFitness<C extends Comparable<? super C>>
    implements Function<Statistics<?, C>, Boolean>
{
    private final int _genenerations;
    private C _fitness;
    private int _stableGenerations = 0;

    public SteadyFitness(final int generations) {
        _genenerations = generations;
    }

    @Override
    public Boolean apply(Statistics<?, C> statistics) {
        boolean proceed = true;
        if (_fitness == null) {
            _fitness = stat.getBestFitness();
            _stableGenerations = 1;
        } else {
            final Optimize opt = stat.getOptimize();
            if (opt.compare(_fitness, stat.getBestFitness()) >= 0) {
                proceed = ++_stableGenerations <= _genenerations;
            } else {
                _fitness = stat.getBestFitness();
                _stableGenerations = 1;
            }
```

```
26            }
27
28            return proceed ? Boolean.TRUE : Boolean.FALSE;
29        }
30    }
```

Listing 3: Steady state termination

Listing 3 shows the (shortened) code for the steady-state termination function which you can find in the `termination` class and should give you an idea how to implement your own termination function.

5.4 Randomness

In general, GAs heavily depends on *pseudo* random number generators (PRNG) for creating new individuals and for the selection- and mutation-algorithms. **Jenetics** uses the Java `Random` object, respectively sub-types from it, for generating random numbers. To make the random engine pluggable, the `Random` object is always fetched from the `RandomRegistry`. This makes it possible to change the implementation of the random engine without changing the client code. The random engine can easily changed, even for specific parts of the code.

The following example shows how to change and restore the `Random` object. When entering the `Scoped<Random>` context, changes to the `RandomRegistry` are only visible within this context. When leaving the context, the original `Random` object is restored.

```
1  Factory<Genotype<DoubleGene>> factory = Genotype.valueOf(
2      new DoubleChromosome(0.0, 100.0, 10)
3  );
4  List<Genotype<DoubleGene>> genotypes = new ArrayList<>();
5
6  try (Scoped<Random> s = RandomRegistry.scope(new Random(123)) {
7      for (int i = 0; i < 100; ++i) {
8          genotypes.add(factory.newInstance());
9      }
10 }
```

With the previous listing, a random, but reproducible, list of genotypes is created. This might be useful while testing your application or when you want to run the GA several times with the same initial population.

```
1  Function<Genotype<DoubleGene>> ff = ...
2  GeneticAlgorithm<DoubleGene, Double> ga = new GeneticAlgorithm<>(
3      genotypes.get(0), ff
4  );
5  ga.setup(genotypes);
```

This example uses the generated genotypes to setup the initial population of the GA. The GA is created with the first element of the genotypes, which is used as genotype factory. This guarantees that the same kind of genotypes are created while evolving. Calling the `setup(Collection)` method, the given collection of genotypes is used as initial population. This method also automatically sets the GA's population size, that means the `populationSize` property of the GA is changed to `genotypes.size()`.

Setting the PRNG to a Random object with a defined seed has the effect, that every run of the GA produces the same result—in a single threaded environment.

The parallel nature of the GA implementation requires the creation of streams $t_{i,j}$ of random numbers which are statistically independent, where the streams are numbered with $j = 1, 2, 3, ..., p$. p denotes the number of processes. We expect statistical independence between the streams as well. The used PRNG should enable the GA to *play fair*, which means that the outcome of the GA is strictly independent from the underlying hardware and the number of parallel processes. This is essential for reproducing results in parallel environments where the number of parallel tasks may vary from run to run.

The *Fair Play* property of a PRNG guarantees that the quality of the GA does not depend on the degree of parallelization.

There are essentially four different parallelizations techniques used in practice: *Random seeding*, *Parameterization*, *Block splitting* and *Leapfrogging*.

Random seeding Every thread uses the same kind of PRNG but with a different seed. This is the default strategy used by the **Jenetics** library. The `RandomRegistry` is initialized with the `ThreadLocalRandom` class from the `java.util.concurrent` package. Random seeding works well for the most problems but without theoretical foundation.[4] If you assume that this strategy is responsible for some *non*-reproducible results, consider using the `LCG64ShiftRandom` PRNG instead, which uses *block splitting* as parallelization strategy.

Parameterization All threads uses the same kind of PRNG but with different parameters. This requires the PRNG to be parameterizable, which is not the case for the Random object of the JDK. You can use the `LCG64ShiftRandom` class if you want to use this strategy. The theoretical foundation for these method is weak. In a massive parallel environment you will need a reliable set of parameters for every random stream, which are not trivial to find.

Block splitting With this method each thread will be assigned a block of random numbers, which should be enough for the whole runtime of the process. This strategy is used when using the `LCG64ShiftRandom.ThreadLocal` class. This class assigns every thread a block of $2^{56} \approx 7,2 \cdot 10^{16}$ random numbers. After 128 threads, the blocks are recycled, but with changed seed.

Leapfrog Each thread $t \in [0, P)$ only consumes the Pth random number. Figure 5.2 shows the concept of the *leapfrog* method.

[4] This is also expressed by Donald Knuth's advice: »Random number generators should not be chosen at random.«

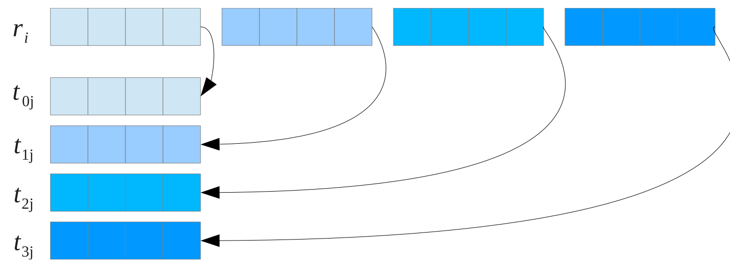


Figure 5.1: Parallelization via *block splitting*

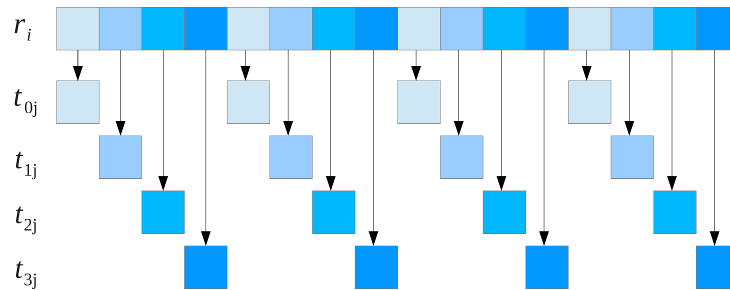


Figure 5.2: Parallelization via *leapfrogging*

`org.jenetics.util.LCG64ShiftRandom` The `LCG64ShiftRandom` class is a Java port of the `trng::lcg64_shift` PRNG class of the TRNG[5] library, implemented in C++.[3] It implements additional methods, which allows to implement the *block splitting*–and also the *leapfrog*–method.

```java
public class LCG64ShiftRandom extends Random {
    public void split(final int p, final int s);
    public void jump(final long step);
    public void jump2(final int s);
    ...
}
```

Listing 4: LCG64ShiftRandom class

Listing 4 shows the interface used for implementing the block splitting and leapfrog parallelizations technique. This methods have the following meaning:

`split` Changes the internal state of the PRNG in a way that future calls to `nextLong()` will generated the sth sub-stream of pth sub-streams. s must be within the range of $[0, p-1)$. This method is used for parallelization via *leapfrogging*.

`jump` Changes the internal state of the PRNG in such a way that the engine jumps$s$ steps ahead. This method is used for parallelization via *block splitting*.

`jump2` Changes the internal state of the PRNG in such a way that the engine jumps2^s steps ahead. This method is used for parallelization via *block splitting*.

[5] `http://numbercrunch.de/trng/`

Runtime performance Table 1 shows the random number (`int`, `long`, `float` and `double`) generation speed for the different PRNG implementations.[6]

	`int`/s	`long`/s	`float`/s	`double`/s
`Random`	$70 \cdot 10^6$	$35 \cdot 10^6$	$70 \cdot 10^6$	$35 \cdot 10^6$
`ThreadLocalRandom`	$272 \cdot 10^6$	$182 \cdot 10^6$	$272 \cdot 10^6$	$180 \cdot 10^6$
`LCG64ShiftRandom`	$230 \cdot 10^6$	$247 \cdot 10^6$	$213 \cdot 10^6$	$209 \cdot 10^6$

Table 1: Performance of various PRNG implementations.

The default PRNG used by the **Jenetics** has the best runtime performance behavior (for generating `int` values)[7].

5.5 Serialization

Jenetics supports serialization for a number of classes, most of them are located in the `org.jenetics` package:

- `BitGene`
- `BitChromosome`
- `CharacterGene`
- `CharacterChromosome`
- IntegerGene
- IntegerChromosome
- LongGene
- `LongChomosome`
- `DoubleGene`
- `DoubleChromosome`
- `EnumGene`
- `PermutationChromosome`
- `Genotype`
- `Phenotype`
- `Population`

With the serialization mechanism you can write a population to disk and load it into an GA at a later time. It can also be used to transfer populations to GAs, running on different hosts, over a network link. The `IO` class, located in the `org.jenetics.util` package, supports native Java serialization and JAXB XML serialization[8].

```
// Writing the population to disk.
final File file = new File("population.xml");
IO.jaxb.write(ga.getPopulation(), file);

// Reading the population from disk.
Population<DoubleGene, Double> population =
    (Population<DoubleGene, Double>)IO.jaxb.read(file);
ga.setPopulation(population);
```

[6] Measured on a Intel(R) Core(TM) i5-3427U CPU @ 1.80GHz with Java(TM) SE Runtime Environment (build 1.7.0_51-b13)—Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)

[7] The *random* `IndexStream` implementation uses random `int` values for creating the *random* indexes and this `IndexStream` is used for selecting the genes, chromosomes and genotypes.

[8] The serialization through the XML support from the *Javolution* project has been deprecated and will be removed in the next major version.

The following listing shows the XML serialization of a population which consists of genotypes as shown in figure 3.1 on page 3; only the first phenotype is shown.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<org.jenetics.Population size="5">
    <phenotype
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="org.jenetics.Phenotype" generation="294"
    >
        <genotype length="5" ngenes="100">
            <chromosome
                xsi:type="org.jenetics.DoubleChromosome"
                length="20" min="0.0" max="1.0"
            >
                <allele>0.27251556008507416</allele>
                <allele>0.003140816229067145</allele>
                <allele>0.43947528327497376</allele>
                <allele>0.10654807463069327</allele>
                <allele>0.19696530915810317</allele>
                <allele>0.7450003838065538</allele>
                <allele>0.5594416969271359</allele>
                <allele>0.02823782430152355</allele>
                <allele>0.5741102315010789</allele>
                <allele>0.4533651041367144</allele>
                <allele>0.811148141800367</allele>
                <allele>0.5710456351848858</allele>
                <allele>0.30166768355230955</allele>
                <allele>0.5455492865240272</allele>
                <allele>0.21068427527733102</allele>
                <allele>0.5265067943902246</allele>
                <allele>0.273549098065591</allele>
                <allele>0.2648197379297126</allele>
                <allele>0.8732775776362911</allele>
                <allele>0.9498003919007005</allele>
            </chromosome>
            ...
        </genotype>
        <fitness
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xsi:type="xs:double"
        >234.23443</fitness>
        <raw-fitness
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xsi:type="xs:double"
        >34.2498</raw-fitness>
    </phenotype>
    ...
</org.jenetics.Population>
```

When serializing a whole population the fitness function and fitness scaler are not serialized. If a GA is initialized with a previously serialized population, the GA's current fitness function and fitness scaler are used for *re*-calculating the fitness values.

The `ga.setPopulation(Collection)` method doesn't perform a recalculation of the fitness values. This is done on demand, when evolving the next generation. Setting the population can be done whenever desired. In contrast to the `ga.setup(Collection)` method, which can only be called before starting evaluation, as replacement for the `ga.setup()` call.

5.6 Utility classes

The `org.jenetics.util` package of the library contains utility classes which are also very important for the GA implementation.

`org.jenetics.util.Seq` Most notable are the `Seq` interfaces and its implementation. They are used, among others, in the `Chromosome` and `Genotype` classes and holds the `Genes` and `Chromosomes`, respectively. The Seq interface itself represents a fixed-sized, ordered sequence of elements. It is an abstraction over the Java build-in *array*-type, but much safer to use for *generic* elements, because there are no casts needed when using *nested* generic types.

Figure 5.3 shows the `Seq` class diagram with their most important methods. The interfaces `MSeq` and `ISeq` are mutable, respectively immutable specializa-

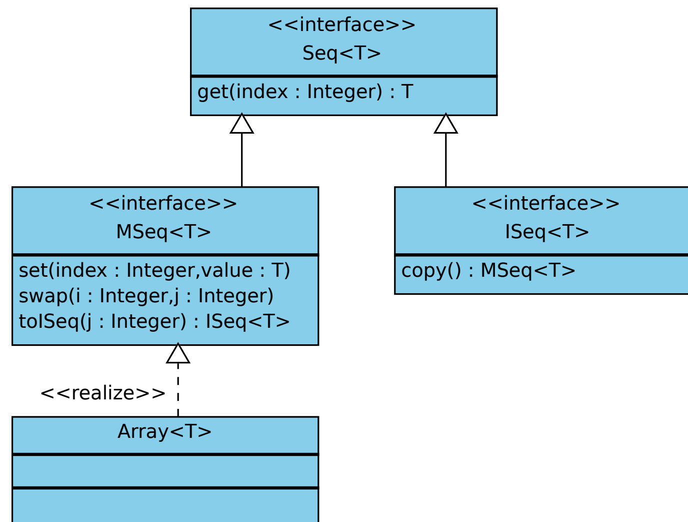


Figure 5.3: `Seq` class diagram

tions of the basis interface. Creating instances of the `Seq` interfaces is possible via the `Array` class.

```java
// Create 'different' sequences.
final Seq<Integer> a1 = Array.box(1, 2, 3);
final MSeq<Integer> a2 = Array.box(1, 2, 3);
final ISeq<Integer> a3 = Array.box(1, 2, 3).toISeq();
final MSeq<Integer> a4 = a3.copy();

// The 'equals' method performs element-wise comparison.
assert(a1.equals(a2) && a1 != a2);
assert(a2.equals(a3) && a2 != a3);
assert(a3.equals(a4) && a3 != a4);
```

How to create instances of the three `Seq` types is shown in the listing above. The `Seq` classes also allows a more *functional* programming style. For a full method description refer to the Javadoc.

`org.jenetics.util.IndexStream` The abstract `IndexStream` class is used to generate a stream of positive `int` values by calling the `next()` method. The end of the stream is reached, when `next()` returns `-1`.

```java
public abstract class IndexStream {
    public abstract int next();
    public static IndexStream Random(int n, double p, Random r) {
        ...
    }
}
```

Listing 5: `IndexStream` class

An *random* `IndexStream`, which is created with the `IndexStream.Random(int, double, Random)` factory method, is used in the `Selector` and `Alterer` classes

for choosing random genes and/or chromosomes (indices).[9]

The following listing shows the typical usage of the `IndexStream`.

```java
final Seq<Integer> array = ...
final IndexStream indices = IndexStream.Random(
    array.length(), 0.5, new Random()
);

for (int i = indices.next(); i != -1; i = indices.next()) {
    System.out.println(array.get(i));
}
```

An important property of the random `IndexStream` is, that the number of selected items n_s follows a Binomial distribution with $\mu = n \cdot p$ and may differ from run to run. The random `IndexStream` produces unique values in ascending order.

`org.jenetics.util.Accumulator` The accumulator classes are mainly used for (incrementally) calculating statistic values in the `Statistics` object.

```java
public interface Accumulator<T> {
    public void accumulate(final T value);
}
```

Listing 6: `Accumulator` interface

Implementations of the Accumulator interface have an internal state which is updated when the accumulate method is called. If the values for accumulation are *stored* in an `Iterable` object, the `accumulate` helper methods in the `accumulator` object can be used for accumulation. The accumulation of two or more accumulators is parallelized.

```java
final Seq<String> data = Array.of("-10","1","2","3","4","5");
final accumulators.Max<Integer> max = new accumulators.Max<>();
final accumulators.Min<Integer> min = new accumulators.Min<>();
accumulators.accumulate(
    data,
    max.map(functions.StringToInteger),
    min.map(functions.StringLength)
);
System.out.println(String.format(
    "Max value: %s, min length: %s.", max.getMax(), min.getMin()
));
```

The given usage example *calculates* the maximum value and the minimum string length of the given data values. It also shows how the `MappedAccumulator.map` method can be used to operate on different data-type then the one given in the data array. However the example snippet will print

```
$ Max value: 5, min length: 1.
```

onto the console.

[9]The elements returned by the *random* `IndexStream` are strictly increasing, except the termination element., which is -1.

6 Extending Jenetics

The **Jenetics** library was designed to give you a great flexibility in transforming your problem into a structure that can be solved by an GA. It also comes with different implementations for the base data-types (genes and chromosomes) and genetic operators (alterers and selectors). If it is still some functionality missing, this section describes how you can extend the existing classes. Most of the *extensible* classes are defined by an interface and have an abstract implementation which makes it easier to extend it.

6.1 Genes

Genes are the starting point in the class hierarchy. They hold the actual information, the alleles, of your problem domain. Beside the *classical* bit-gene, **Jenetics** comes with gene implementations for numbers (double- and long values), characters and enumeration types.

```
public interface Gene<A, G extends Gene<A, G>>
    extends Factory<G>, Serializable, ValueType, Verifiable
{
    public A getAllele();
    public G newInstance();
    public G newInstance(A allele);
    public boolean isValid();
}
```

Listing 7: `Gene` interface

For implementing your own gene type you have to implement the `Gene` interface with three methods: (1) the `getAllele()` method which will return the wrapped data, (2) the `newInstance` method for creating new, random instances of the gene—must be of the same type and have the same constraint—and (3) the `isValid()` method which checks if the gene fulfill the expected constraints. The gene constraint might be violated after mutation and/or recombination. If you want to implement a new number-gene, e. g. a gene which holds complex values, you may want extend it from the abstract `NumericGene` class. Every gene extends the `Serializable` interface. For *normal* genes there is no more work to do for using the Java serialization mechanism.

6.2 Chromosomes

A new gene type normally needs a corresponding chromosome implementation. The following listing shows the `Chromosome` interface and the methods that must be implemented.

```
public interface Chromosome<G extends Gene<?, G>>
    extends Factory<Chromosome<G>>, Iterable<G>, Verifiable,
            Immutable, Serializable
{
    public Chromosome<G> newInstance(ISeq<G> genes);
    public G getGene();
    public G getGene(int index);
    public ISeq<G> toSeq();
    public int length();
}
```

Listing 8: `Chromosome` interface

The most important part of a chromosome is the factory method `newInstance`, which lets the GA create a new chromosome instance from a sequence of genes. This method is used by the alterers when creating new, combined chromosomes. The other methods should be self-explanatory. The chromosome has the same serialization mechanism as the gene. For the minimal case it extends the `Serializable` interface.

6.3 Selectors

If you want to implement your own selection strategy you only have to implement the `Selector` interface with the `select` method.

```
public interface Selector<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
{
    public Population<G, C> select(
        Population<G, C> population,
        int count,
        Optimize opt
    );
}
```

Listing 9: `Selector` interface

The first parameter is the original `population` from which the *sub*-population is selected. The second parameter,`count`, is the number of individuals of the returned sub-population. Depending on the selection algorithm, it is possible that the sub-population contains more elements than the original one. The last parameter, `opt`, determines the optimization strategy which must be used by the selector. This is exactly the point where it is decided whether the GA minimizes or maximizes the fitness function.

Before implementing a selector from scratch, consider to extend your selector from the `ProbabilitySelector` (or any other available selector implementation). It is worth the effort to try to express your selection strategy in terms of selection property $P(i)$.

6.4 Alterers

For implementing a new alterer class it is necessary to implement the `Alterer` interface. You might do this if your new gene type needs a special kind of alterer not available in the **Jenetics** project.

```
public interface Alterer<G extends Gene<?, G>> {
    public <C extends Comparable<? super C>> int alter(
        Population<G, C> population,
        int generation
    );
}
```

Listing 10: `Alterer` interface

The first parameter of the `alter` method is the `population` which has to be altered. Since the the `Population` class is mutable, the altering is performed in place. The second parameter is the `generation` of the newly created individuals and the return value is the number of genes that has been altered.

6.5 Statistics

The GA statistics is the only object which doesn't define an interface which must be implemented. For extending the GA statistics you have to implement three classes.

1. **`Statistics`**: Contains the actual statistics information and is an immutable *value* class. It is not required that derived classes are also immutable, but strongly recommended.

2. **`Statistics.Builder`**: Is a classical object builder[10] which is building `Statistics` objects.

3. **`Statistics.Calculator`**: Changing the statistics calculator lets the GA create an instance of your statistics object. The statistics object is not created directly but via the builder indirection. This is necessary because the GA also adds some statistical information to the statistics object.

The following listing shows an excerpt of the statistics calculator which must be extended.

```
1  public class Statistics {
2      public static class Calculator<
3          G extends Gene<?, G>,
4          C extends Comparable<? super C>
5      >
6      {
7          public Statistics.Builder<G, C> evaluate(
8              Iterable<? extends Phenotype<G, C>> population,
9              int generation,
10             Optimize opt
11         ) {
12             ...
13         }
14     }
15 }
```

Listing 11: `Statistics` class

The `evaluate` method of the `Statistics.Calculator` class return a builder, pre-configured with your actual statistics values, which will create the actual statistics object. This indirection is necessary because the GA is setting additional statistical information about killed and invalid individuals to the statistics object.

The listing beneath shows the `Statistics.Builder` class with some of its properties and the `build` method.

```
1  public class Statistics {
2      public static class Builder<
3          G extends Gene<?, G>,
4          C extends Comparable<? super C>
5      >
6      {
7          public Builder<G, C> invalid(int invalid);
8          public Builder<G, C> killed(int killed);
9          ...
10         public Statistics<G, C> build();
11     }
```

[10] `https://en.wikipedia.org/wiki/Builder_pattern`

```
12 }
```

Listing 12: `Statistics.Builder` class

The type of the statistics object returned by the GA is the same type as returned by the statistics builder. If you want to access specific values from your special statistics type, you have to cast it to your type.

Appendix

7 Examples

This section contains some coding examples which should give you a feeling of how to use the **Jenetics** library. The given examples are complete, in the sense that they will compile and run and produce the given example output.

Running the examples delivered with the **Jenetics** library can be started with the `run-examples.sh` script.

```
$ ./run-examples.sh
```

Since the script uses JARs located in the build directory you have to build it with the `jar` *Gradle* target first; see section 8 on page 35.

7.1 Ones counting

Ones counting is one of the simplest model-problem. It uses a binary chromosome and forms a classic genetic algorithm[11]. The fitness of a Genotype is proportional to the number of ones.

```
import org.jenetics.BitChromosome;
import org.jenetics.BitGene;
import org.jenetics.GeneticAlgorithm;
import org.jenetics.Genotype;
import org.jenetics.Mutator;
import org.jenetics.NumberStatistics;
import org.jenetics.Optimize;
import org.jenetics.RouletteWheelSelector;
import org.jenetics.SinglePointCrossover;
import org.jenetics.util.Factory;
import org.jenetics.util.Function;

final class OneCounter
    implements Function<Genotype<BitGene>, Integer>
{
    @Override
    public Integer apply(final Genotype<BitGene> genotype) {
        final BitChromosome chromosome =
            (BitChromosome)genotype.getChromosome();
        return chromosome.bitCount();
    }
}

public class OnesCounting {
    public static void main(String[] args) {
        Factory<Genotype<BitGene>> gtf = Genotype.of(
            BitChromosome.of(20, 0.15)
        );
        Function<Genotype<BitGene>, Integer> ff = new OneCounter();
        GeneticAlgorithm<BitGene, Integer> ga =
        new GeneticAlgorithm<>(
            gtf, ff, Optimize.MAXIMUM
        );

```

[11] In the classic genetic algorithm the problem is a maximization problem and the fitness function is positive. The domain of the fitness function is a bit-chromosome.

```java
        ga.setStatisticsCalculator(
            new NumberStatistics.Calculator<BitGene, Integer>()
        );
        ga.setPopulationSize(500);
        ga.setSelectors(
            new RouletteWheelSelector<BitGene, Integer>()
        );
        ga.setAlterers(
            new Mutator<BitGene>(0.55),
            new SinglePointCrossover<BitGene>(0.06)
        );

        ga.setup();
        ga.evolve(100);
        System.out.println(ga.getBestStatistics());
        System.out.println(ga.getBestPhenotype());
    }
}
```

The genotype in this example consists of one BitChromosome with a ones probability of 0.15. The altering of the offspring population is performed by mutation, with mutation probability of 0.55, and then by a single-point crossover, with crossover probability of 0.06. After creating the initial population, with the `ga.setup()` call, 100 generations are evolved. The tournament selector is used for both, the offspring- and the survivor selection—this is the default selector.[12]

```
+---------------------------------------------------------+
|  Population Statistics                                  |
+---------------------------------------------------------+
|                      Age mean: 0.95600000000            |
|                  Age variance: 1.89385170341            |
|                       Samples: 500                      |
|                  Best fitness: 18                       |
|                 Worst fitness: 3                        |
+---------------------------------------------------------+
+---------------------------------------------------------+
|  Fitness Statistics                                     |
+---------------------------------------------------------+
|                  Fitness mean: 10.90000000000           |
|              Fitness variance: 5.50901803607            |
|         Fitness error of mean: 0.48746281909            |
+---------------------------------------------------------+
[00001101|11101111|11111111] --> 18
```

The given example will print the overall timing statistics onto the console.

7.2 Real function

In this example we try to find the minimum value of the function

$$f(x) = \cos\left(\frac{1}{2} + \sin(x)\right) \cdot \cos(x). \tag{7.1}$$

The graph of function 7.1, in the range of $[0, 2\pi]$, is shown in figure 7.1 and the listing beneath shows the GA implementation which will minimize the function.

```java
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static java.lang.Math.sin;
```

[12]For the other default values (population size, maximal age, ...) have a look at the Javadoc:`http://jenetics.sourceforge.net/javadoc/index.html`

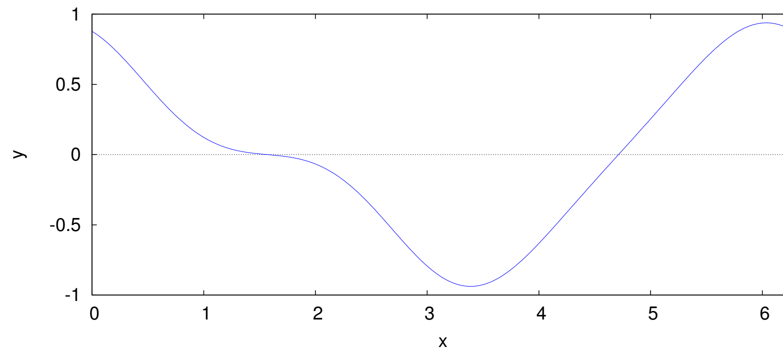


Figure 7.1: Real function 2D

```
4
5  import org.jenetics.DoubleChromosome;
6  import org.jenetics.DoubleGene;
7  import org.jenetics.GeneticAlgorithm;
8  import org.jenetics.Genotype;
9  import org.jenetics.MeanAlterer;
10 import org.jenetics.Mutator;
11 import org.jenetics.NumberStatistics;
12 import org.jenetics.Optimize;
13 import org.jenetics.util.Factory;
14 import org.jenetics.util.Function;
15
16 final class Real
17     implements Function<Genotype<DoubleGene>, Double>
18 {
19     @Override
20     public Double apply(Genotype<DoubleGene> genotype) {
21         final double x = genotype.getGene().doubleValue();
22         return cos(0.5 + sin(x)) * cos(x);
23     }
24 }
25
26 public class RealFunction {
27     public static void main(String[] args) {
28         Factory<Genotype<DoubleGene>> gtf = Genotype.of(
29             new DoubleChromosome(0.0, 2.0 * PI)
30         );
31         Function<Genotype<DoubleGene>, Double> ff = new Real();
32         GeneticAlgorithm<DoubleGene, Double> ga =
33             new GeneticAlgorithm<>(
34                 gtf, ff, Optimize.MINIMUM
35             );
36
37         ga.setStatisticsCalculator(
38             new NumberStatistics.Calculator<DoubleGene, Double>()
39         );
40         ga.setPopulationSize(500);
41         ga.setAlterers(
42             new Mutator<DoubleGene>(0.03),
43             new MeanAlterer<DoubleGene>(0.6)
44         );
45
46         ga.setup();
```

```
        ga.evolve(100);
        System.out.println(ga.getBestStatistics());
        System.out.println(ga.getBestPhenotype());
    }
}
```

The GA works with 1×1 `DoubleChromosomes` whose values are restricted to the range $[0, 2\pi]$. Without this restriction, the search space of the chromosome will be between `Double.MIN_VALUE` and `Double.MAX_VALUE`.

```
+---------------------------------------------------------+
|  Population Statistics                                  |
+---------------------------------------------------------+
|                     Age mean: 1.02000000000             |
|                 Age variance: 1.37434869739             |
|                      Samples: 500                       |
|                 Best fitness: -0.9381718976956661       |
|                Worst fitness: 0.8256770485869761        |
+---------------------------------------------------------+
+---------------------------------------------------------+
|  Fitness Statistics                                     |
+---------------------------------------------------------+
|                 Fitness mean: -0.91895366231            |
|             Fitness variance: 0.01906961656             |
|        Fitness error of mean: -0.04109685714            |
+---------------------------------------------------------+
[[[3.389125781293614]]] --> -0.9381718976956661
```

The GA will generated an console output like above.

7.3 0/1 Knapsack

In the knapsack problem[13] a set of items, together with it's size and value, is given. The task is to select a disjoint subset so that the total size does not exceed the knapsack size. For solving the 0/1 knapsack problem we define a `BitChromosome`, one bit for each item. If the ith bit is set to one the ith item is selected.

```
import static org.jenetics.util.math.random.nextDouble;

import java.util.Random;

import org.jenetics.BitChromosome;
import org.jenetics.BitGene;
import org.jenetics.Chromosome;
import org.jenetics.GeneticAlgorithm;
import org.jenetics.Genotype;
import org.jenetics.Mutator;
import org.jenetics.NumberStatistics;
import org.jenetics.RouletteWheelSelector;
import org.jenetics.SinglePointCrossover;
import org.jenetics.TournamentSelector;
import org.jenetics.util.Factory;
import org.jenetics.util.Function;
import org.jenetics.util.LCG64ShiftRandom;
import org.jenetics.util.RandomRegistry;
import org.jenetics.util.Scoped;

final class Item {
    public final double size;
    public final double value;

```

[13] https://en.wikipedia.org/wiki/Knapsack_problem

```java
    Item(final double size, final double value) {
        this.size = size;
        this.value = value;
    }
}

final class KnapsackFunction
    implements Function<Genotype<BitGene>, Double>
{
    private final Item[] items;
    private final double size;

    public KnapsackFunction(final Item[] items, double size) {
        this.items = items;
        this.size = size;
    }

    @Override
    public Double apply(final Genotype<BitGene> genotype) {
        final Chromosome<BitGene> ch = genotype.getChromosome();
        double size = 0;
        double value = 0;
        for (int i = 0, n = ch.length(); i < n; ++i) {
            if (ch.getGene(i).getBit()) {
                size += items[i].size;
                value += items[i].value;
            }
        }

        return size <= this.size ? value : 0;
    }
}

public class Knapsack {

    private static KnapsackFunction FF(final int n, final double
        size) {
        final Item[] items = new Item[n];
        try (Scoped<? extends Random> random =
            RandomRegistry.scope(new LCG64ShiftRandom(123)))
        {
            for (int i = 0; i < items.length; ++i) {
                items[i] = new Item(
                    nextDouble(random.get(), 0, 100),
                    nextDouble(random.get(), 0, 100)
                );
            }
        }

        return new KnapsackFunction(items, size);
    }

    public static void main(String[] args) throws Exception {
        final int nitems = 15;
        final double kssize = nitems*100.0/3.0;

        final KnapsackFunction ff = FF(nitems, kssize);
        final Factory<Genotype<BitGene>> genotype = Genotype.of(
            BitChromosome.of(nitems, 0.5)
        );

```

```
85          final GeneticAlgorithm<BitGene, Double> ga = new
                GeneticAlgorithm<>(
86              genotype, ff
87          );
88          ga.setPopulationSize(500);
89          ga.setStatisticsCalculator(
90              new NumberStatistics.Calculator<BitGene, Double>()
91          );
92          ga.setSurvivorSelector(
93              new TournamentSelector<BitGene, Double>(5)
94          );
95          ga.setOffspringSelector(
96              new RouletteWheelSelector<BitGene, Double>()
97          );
98          ga.setAlterers(
99               new Mutator<BitGene>(0.115),
100              new SinglePointCrossover<BitGene>(0.16)
101         );
102
103         ga.setup();
104         ga.evolve(100);
105         System.out.println(ga.getBestStatistics());
106         System.out.println(ga.getBestPhenotype());
107     }
108 }
```

The console out put for the Knapsack GA will look like the listing beneath.

```
1  +---------------------------------------------------------+
2  |  Population Statistics                                  |
3  +---------------------------------------------------------+
4  |                     Age mean: 2.57200000000            |
5  |                 Age variance: 7.41564729459            |
6  |                      Samples: 500                      |
7  |                 Best fitness: 643.239770840163         |
8  |                Worst fitness: 0.0                      |
9  +---------------------------------------------------------+
10 +---------------------------------------------------------+
11 |  Fitness Statistics                                     |
12 +---------------------------------------------------------+
13 |                 Fitness mean: 580.26211709025          |
14 |             Fitness variance: 16717.53287769222        |
15 |        Fitness error of mean: 25.95011077163           |
16 +---------------------------------------------------------+
17 [01101111|01011111] --> 643.239770840163
```

7.4 Traveling salesman

The Traveling Salesman problem[14] is one of the classical problems in computational mathematics and it is the most notorious NP-complete problem. The goal is to find the shortest distance, or the path, with the least costs, between N different cities. Testing all possible path for N cities would lead to $N!$ checks to find the shortest one.

The following example uses a path where the cities are lying on a circle. That means, the optimal path will be a polygon. This makes it easier to check the quality of the found solution.

```
1 import static java.lang.Math.PI;
2 import static java.lang.Math.abs;
3 import static java.lang.Math.sin;
4
```

[14] https://en.wikipedia.org/wiki/Travelling_salesman_problem

```
5  import java.io.Serializable;
6
7  import org.jenetics.Chromosome;
8  import org.jenetics.EnumGene;
9  import org.jenetics.GeneticAlgorithm;
10 import org.jenetics.Genotype;
11 import org.jenetics.NumberStatistics.Calculator;
12 import org.jenetics.Optimize;
13 import org.jenetics.PartiallyMatchedCrossover;
14 import org.jenetics.PermutationChromosome;
15 import org.jenetics.SwapMutator;
16 import org.jenetics.util.Factory;
17 import org.jenetics.util.Function;
18
19
20 public class TravelingSalesman {
21
22     private static class FF
23         implements Function<Genotype<EnumGene<Integer>>, Double>,
24                    Serializable
25     {
26         private static final long serialVersionUID = 1L;
27
28         private final double[][] adjacence;
29
30         public FF(final double[][] adjacence) {
31             this.adjacence = adjacence;
32         }
33
34         @Override
35         public Double apply(Genotype<EnumGene<Integer>> gt) {
36             final Chromosome<EnumGene<Integer>>
37                 path = gt.getChromosome();
38
39             double length = 0.0;
40             for (int i = 0, n = path.length(); i < n; ++i) {
41                 final int from = path.getGene(i).getAllele();
42                 final int to = path.getGene((i + 1)%n).getAllele();
43                 length += adjacence[from][to];
44             }
45             return length;
46         }
47
48         @Override
49         public String toString() {
50             return "Point distance";
51         }
52     }
53
54     public static void main(String[] args) {
55         final int stops = 20;
56
57         final Function<Genotype<EnumGene<Integer>>, Double> ff =
58             new FF(adjacencyMatrix(stops));
59         final Factory<Genotype<EnumGene<Integer>>> gtf = Genotype.
           of(
60             PermutationChromosome.ofInteger(stops)
61         );
62         final GeneticAlgorithm<EnumGene<Integer>, Double>
63             ga = new GeneticAlgorithm<>(gtf, ff, Optimize.MINIMUM);
64         ga.setStatisticsCalculator(
65                 new Calculator<EnumGene<Integer>, Double>()
```

```
66            );
67        ga.setPopulationSize(500);
68        ga.setAlterers(
69            new SwapMutator<EnumGene<Integer>>(0.2),
70            new PartiallyMatchedCrossover<Integer>(0.3)
71        );
72
73        ga.setup();
74        ga.evolve(100);
75        System.out.println(ga.getBestStatistics());
76        System.out.println(ga.getBestPhenotype());
77    }
78
79    private static double[][] adjacencyMatrix(int stops) {
80        double[][] matrix = new double[stops][stops];
81        for (int i = 0; i < stops; ++i) {
82            for (int j = 0; j < stops; ++j) {
83                matrix[i][j] = chord(stops, abs(i - j), RADIUS);
84            }
85        }
86        return matrix;
87    }
88    private static double chord(int stops, int i, double r) {
89        return 2.0*r*abs(sin((PI*i)/stops));
90    }
91    private static double RADIUS = 10.0;
92 }
```

The Traveling Salesman problem is a very good example which shows you how to solve combinatorial problems with an GA. **Jenetics** contains several classes which will work very well with this kind of problems. Wrapping the base *type* into an `EnumGene` is the first thing to do. In our example, every city has an unique number, that means we are wrapping an `Integer` into an `EnumGene`. Creating a genotype for integer values is very easy with the factory method of the `PermutationChromosome`. For other data types you have to use one of the constructors of the permutation chromosome. As alterers, we are using a swap-mutator and a partially-matched crossover. These alterers guarantees that no invalid solutions are created—every city exists exactly once in the altered chromosomes.

```
1  +---------------------------------------------------------+
2  |  Population Statistics                                  |
3  +---------------------------------------------------------+
4  |                    Age mean: 1.95200000000              |
5  |                Age variance: 8.55881362725              |
6  |                     Samples: 500                        |
7  |                Best fitness: 68.677087189481            |
8  |               Worst fitness: 310.38702459691393         |
9  +---------------------------------------------------------+
10 +---------------------------------------------------------+
11 |  Fitness Statistics                                     |
12 +---------------------------------------------------------+
13 |                Fitness mean: 112.17546321371            |
14 |            Fitness variance: 4854.12951297338           |
15 |        Fitness error of mean: 5.01663922307             |
16 +---------------------------------------------------------+
17 [0|1|2|3|4|5|6|7|8|10|9|11|12|13|14|15|16|17|18|19] --> 68.677087189481
```

The listing above shows the output generated by our example. The last line represents the phenotype of the best solution found by the GA, which represents the traveling path. As you can see, the GA has found the shortest path, in reverse order.

8 Build

For building the **Jenetics** library from source, download the most recent, stable package version from `https://sourceforge.net/projects/jenetics/files/latest/download` and extract it to some build directory.

```
$ unzip jenetics-<version>.zip -d <builddir>
```

`<version>` denotes the actual **Jenetics** version and `<builddir>` the actual build directory. Alternatively you can check out the latest–unstable–version from the Mercurial `default` branch.

```
$ hg clone https://fwilhelm@bitbucket.org/fwilhelm/jenetics\
           <builddir>
# or
$ hg clone http://hg.code.sf.net/p/jenetics/main\
           <builddir>
# or
$ git clone https://github.com/jenetics/jenetics.git\
           <builddir>
```

Jenetics uses Gradle[15] as build system and organizes the source into *sub*-projects (*modules*).[16] Each *sub*-project is located in it's own *sub*-directory:

- `org.jenetics`: This project contains the source code and tests for the **Jenetics** *core*-module.
- `org.jenetics.example`: This project contains example code for the *ore*-module.
- `org.jenetics.doc`: Contains the *code* of the web-site and *this* manual.

For building the library change into the `<builddir>` directory (or one of the *module* directory) and call one of the available *tasks*:

- `compileJava`: Compiles the **Jenetics** sources and copies the class files to the `<builddir>/<module-dir>/build/classes/main` directory.
- `test`: Compiles and executes the unit tests. The test results are printed onto the console and a test-report, created by *TestNG*, is written to `<builddir>/<module-dir>` directory.
- `javadoc`: Generates the API documentation. The Javadoc is stored in the `<builddir>/<module-dir>/build/docs` directory
- `jar`: Compiles the sources and creates the JAR files. The artifacts are copied to the `<builddir>/<module-dir>/build/libs` directory.

[15] `http://gradle.org/downloads`

[16] If you are calling the `gradlew` script (instead of `gradle`), which are part of the downloaded package, the proper Gradle version is automatically downloaded and you don't have to install Gradle explicitly.

- `packaging`: Compiles the sources of all modules, creates the JAR files and the Javadoc and creates a complete library package—the very same which you can download from the home page. The build artifacts are copied into the `<builddir>/build/package/jenetics-<version> directory.`
- `clean`: Deletes the `<builddir>/build/*` directories and removes all generated artifacts.

For `packaging` (building) the source, call

```
$ cd <build-dir>
$ gradle packaging
```

or

```
$ ./gradlew packaging
```

if you don't have the the Gradle build system installed—calling the the Gradle wrapper script will download all needed files and trigger the build task afterwards.

IDE integration Gradle has tasks which creates the project file for Eclipse[17] and IntelliJ IDEA[18]. Call

```
$ ./gradlew <eclipse|idea>
```

for creating the project files for Eclipse or IntelliJ, respectively.

External library dependencies The following external projects are used for running and/or building the **Jenetics** library.

- ***TestNG***
  - **Version**: *6.8.8*
  - **Homepage**: *http://testng.org/doc/index.html*
  - **Download**: *http://testng.org/testng-6.8.8.zip*
  - **License**: *Apache License, Version 2.0*
  - **Scope**: *test*
- ***Apache Commons Math***
  - **Version**: *3.3*
  - **Homepage**: *http://commons.apache.org/proper/commons-math/*
  - **Download**: *http://tweedo.com/mirror/apache/commons/math/binaries/commons-math3-3.3-bin.zip*
  - **License**: *Apache License, Version 2.0*
  - **Scope**: *test*

[17] http://www.eclipse.org/
[18] http://www.jetbrains.com/idea/

- ***Java2Html***
  - **Version**: *5.0*
  - **Homepage**: *http://www.java2html.de/*
  - **Download**: *http://www.java2html.de/java2html_50.zip*
  - **License**: *GPL or CPL1.0*
  - **Scope**:*javadoc*
- ***Gradle***
  - **Version**: 1.10 (*or later*)
  - **Homepage**: *http://gradle.org/*
  - **Download**: *http://services.gradle.org/distributions/gradle-1.10-bin.zip*
  - **License**: *Apache License, Version 2.0*
  - **Scope**: *build*

Maven Central The whole Jenetics package can also be downloaded from the *Maven Central* repository:

```
org.bitbucket.fwilhelm:org.jenetics:2.0.2
```

9 License

The library itself is licensed under the Apache License, Version 2.0.


```
Copyright 2007-2014 Franz Wilhelmstötter

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

References

[1] Thomas Back. *Evolutionary Algorithms in Theory and Practice.* Oxford Univiversity Press, 1996.

[2] James E. Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 14–21, 1987.

[3] Heiko Bauke. Tina's random number generator library. *http://numbercrunch.de/trng/trng.pdf*, 2011.

[4] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4:361–394, 1997.

[5] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution.* Springer, 1996.

[6] Daniel Shiffman. *The Nature of Code.* The Nature of Code, 1 edition, 12 2012.

[7] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms.* Springer, 2010.

[8] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

Index

0/1 Knapsack, 30
2-point crossover, 12
3-point crossover, 12

Accumulator, 22
Allele, 5, 23
Alterer, 9, 24
    diagram, 9
Apache Commons Math, 36

Base classes, 3
Block splitting, 17
Boltzmann selector, 8
Build, 35
    Gradle, 35
    gradlew, 35

Chromosome, 4, 23
Class structure, 2
Compile, 35
Concurrency, 13
Crossover
    2-point crossover, 12
    3-point crossover, 12
    Multiple-point crossover, 12
    Partially-matched crossover, 12, 13
    Single-point crossover, 11

Download, 1, 35

Examples, 27
    0/1 Knapsack, 30
    Ones counting, 27
    Real function, 28
    Traveling salesman, 32
Exponential-rank selector, 8

Fitness function, 4
Fitness scaler, 4

Gaussian mutator, 10
Gene, 5, 23
Genetic algorithm, 1
Genetic operator, 5
Genotyp, 3
Genotype, 3
Git repository, 35
Gradle, 1, 35, 37
gradlew, 35

IndexStream, 21
Installation, 35

Java2Html, 37
JDK, 1

`LCG64ShiftRandom`, 18
Leapfrog, 17
License, i, 37
Linear-rank selector, 8

Mercurial repository, 35
Monte Carlo selector, 7
Multiple-point crossover, 12
Mutation, 10
Mutator, 10

Ones counting, 27

Partially-matched crossover, 12, 13
Phenotype, 3
PRNG, 16
    Block splitting, 17
    `LCG64ShiftRandom`, 18
    Leapfrog, 17
    Parameterization, 17
    Performance, 19
    Random seeding, 17
Probability selector, 7

Random, 16
    Engine, 16
    LCG64ShiftRandom, 18
    Registry, 16
Random seeding, 17
Randomness, 16
Real function, 28
Recombination, 11
Roulette-wheel selector, 7

Selector, 5, 24
    diagram, 6
Seq, 20
Serialization, 19
Single-point crossover, 11
Source code, 35

Statistics, 13, 25
Stochastic-universal selector, 8
Swap mutator, 11

Termination, 15
TestNG, 36
Tournament selector, 6
Traveling salesman, 32
Truncation selector, 6