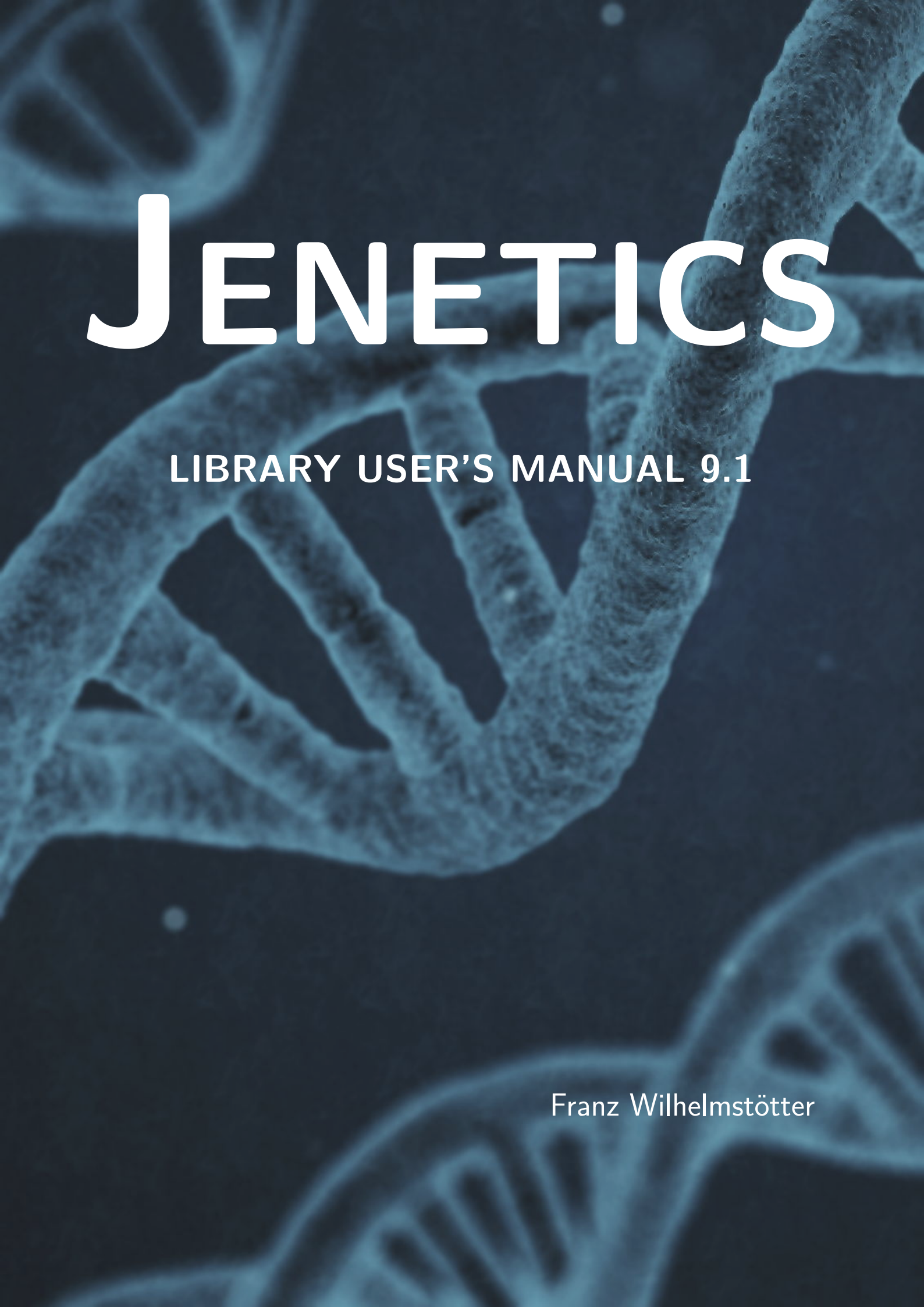


Jenetics

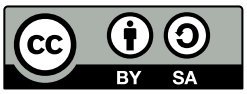
Library User's Manual 9.1

Franz Wilhelmstötter

Franz Wilhelmstötter
franz.wilhelmstoetter@gmail.com

https://jenetics.io

9.1.0-2026/09/02



Contents

Chapter 1

Fundamentals

Jenetics is an advanced **Genetic Algorithm**, **Evolutionary Algorithm** and **Genetic Programming** library, written in modern day 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 a given fitness function without tweaking it. In contrast to other GA implementations, the library uses the concept of an evolution *stream* (`EvolutionStream`) for executing the evolution steps. Since the `EvolutionStream` implements the Java `Stream` interface, it works smoothly with the rest of the Java Stream API. This chapter describes the design concepts and its implementation. It also gives some basic examples and best practice tips.[1]

1.1 Introduction

Jenetics is a library, written in Java[2], which provides a Genetic algorithm (GA), Evolutionary algorithm (EA), Multi-objective optimization (MOO) and Genetic programming (GP) implementation. It has no runtime dependencies to other libraries, except the Java 25 runtime. **Jenetics** is available on the Maven central repository[3] and can be easily integrated into existing projects. The very clear structuring of the different parts of the GA allows an easy adaption for different problem domains.

This manual is not an introduction or a tutorial for genetic and/or evolutionary algorithms in general. It is assumed that the reader has a knowledge about the structure and the functionality of genetic algorithms. Good introductions to GAs can be found in [41], [30], [40], [27], [31] or[45]. For genetic programming you can have a look at [24] or [25].

---

[1] The classes described in this chapter reside in the `io.jenetics.base` module or `io:jenetics:jenetics:9.1.0` artifact, respectively.

[2] The library is build with and depends on Java 25: `https://www.oracle.com/java/technologies/downloads/`

[3] `https://mvnrepository.com/artifact/io.jenetics/jenetics`: If you are using Gradle, you can use the following dependency string: »`io.jenetics:jenetics:9.1.0`«.

To give you a first impression on how to use **Jenetics**, let's start with a simple »Hello World« program. This first example implements the well known *bit counting* problem.

```
1  import io.jenetics.BitChromosome;
2  import io.jenetics.BitGene;
3  import io.jenetics.Genotype;
4  import io.jenetics.engine.Engine;
5  import io.jenetics.engine.EvolutionResult;
6  import io.jenetics.util.Factory;
7
8  public final class HelloWorld {
9      // 2.) Definition of the fitness function.
10     private static int eval(final Genotype<BitGene> gt) {
11         return gt.chromosome()
12             .as(BitChromosome.class)
13             .bitCount();
14     }
15
16     void main() {
17         // 1.) Define the genotype (factory) suitable
18         //     for the problem.
19         final Factory<Genotype<BitGene>> gtf =
20             Genotype.of(BitChromosome.of(10, 0.5));
21
22         // 3.) Create the execution environment.
23         final Engine<BitGene, Integer> engine = Engine
24             .builder(HelloWorld::eval, gtf)
25             .build();
26
27         // 4.) Start the execution (evolution) and
28         //     collect the result.
29         final Genotype<BitGene> result = engine.stream()
30             .limit(100)
31             .collect(EvolutionResult.toBestGenotype());
32
33         IO.println("Hello World:\n\t" + result);
34     }
35 }
```

Listing 1.1: »Hello World« GA

In contrast to other GA implementations, **Jenetics** uses the concept of an evolution *stream* (`EvolutionStream`) for executing the evolution steps. Since the `EvolutionStream` extends the Java `Stream` interface, it works smoothly with the rest of the Java Stream API. Now let's have a closer look at listing 1.1 and discuss this simple program step by step:

1. Probably the most challenging part when setting up a new evolution `Engine`, is to transform the *native* problem domain into an appropriate `Genotype` (factory) representation.[4] In our example we want to count the number of *ones* of a `BitChromosome`. Since we are counting only the ones of one chromosome, we are adding only one `BitChromosome` to our `Genotype`. In general, the `Genotype` can be created with 1 to n chromosomes. For a detailed description of the genotype's structure have a look at section 1.3.1.3.

2. Once this is done, the fitness function, which we are trying to maximize, can be defined. We can simply write a private static method, which

[4] Section2.2 on page 53 describes some common problem encodings.

takes the `Genotype` we defined and calculate its fitness value. If we want to use the optimized bit counting method, `bitCount()`, we have to cast the `Chromosome<BitGene>` class to the actual used `BitChromosome` class. Since we know for sure that we created the `Genotype` with a `BitChromosome`, this is a safe operation. A reference to the `eval` method is then used as a fitness function and passed to the `Engine::build` method.

3. In the third step we are creating the *evolution* `Engine`, which is responsible for evolving the given population. The `Engine` is highly configurable and takes parameters for controlling the evolutionary and the computational environment. For changing the evolutionary behavior, you can set different alterers and selectors (see section 1.3.2). By changing the used `Executor` service, you control the number of threads, the `Engine` is allowed to use. A new `Engine` instance can only be created via its builder, which is created by calling the `Engine.builder` method.

4. In the last step, we will create a new `EvolutionStream` from our `Engine`. The `EvolutionStream` is the model (or view) of the *evolutionary* process. It serves as a »process handle« and allows us, among other things, to control the termination of the evolution. In our example, we simply truncate the stream after 100 generations. If you don't limit the stream, the `EvolutionStream` will never terminate and run forever. The final result, the best `Genotype` in our example, is then collected with one of the predefined collectors of the `EvolutionResult` class.

As the example shows, **Jenetics** makes heavy use of the `Stream` and `Collector` classes. Also lambda expressions and the functional interfaces (SAM types) plays an important roll in the library design.

There are many other GA implementations out there and they may slightly differ in the order of the single execution steps. **Jenetics** uses an classical approach. Listing 1.2 shows the (imperative) pseudocode of the **Jenetics** genetic algorithm steps.

```
1 P_0 ← P_initial
2 F(P_0)
3 while !finished do
4     g ← g + 1
5     S_g ← select_S(P_{g−1})
6     O_g ← select_O(P_{g−1})
7     O_g ← alter(O_g)
8     P_g ← filter[g_i ≥ g_max](S_g) + filter[g_i ≥ g_max](O_g)
9     F(P_g)
```

Listing 1.2: Genetic algorithm

In line (1) the initial population is created and line (2) calculates the fitness value of the individuals. The initial population is created implicitly before the first evolution step is performed. Line (4) increases the generation number and line (5) and (6) selects the survivor and the offspring population. The offspring/survivors fraction is determined by the `offspringFraction` property of the `Engine.Builder`. The selected offspring are altered in line (7). The next line combines the survivor population and the altered offspring population—after removing the *killed* individuals—to the new population. The steps from line (4) to (9) are repeated until a given termination criterion is fulfilled.

1.2 Architecture

The basic metaphor of the **Jenetics** library is the *Evolution Stream*, implemented as Java `Stream`. An evolution stream is powered by—and bound to—an *Evolution Engine*, which performs the needed evolution steps for each generation; the steps are described in the body of the *while* loop of listing 1.2.

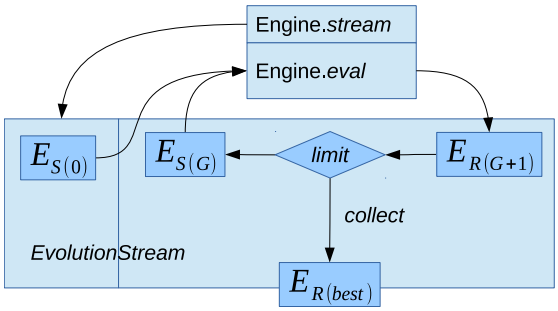


Figure 1.2.1: Evolution workflow

The described evolution workflow is also illustrated in figure 1.2.1, where $E_{S(i)}$ denotes the `EvolutionStart` object at generation$i$ and $E_{R(i)}$ the `EvolutionResult` at the i^{th} generation. Once the evolution `Engine` is created, it can be used by multiple `EvolutionStream`s which can be safely used in different execution threads. This is possible because the evolution `Engine` doesn't have any mutable global state and is therefore thread safe. It is practically a stateless function, $f_E : \mathrm{P} \rightarrow \mathrm{P}$, which maps a start population,P, to an evolved result population. The `Engine` function, f_E, is, of course, *non*deterministic. Calling it twice with the same start population will lead to different result populations.

The evolution process terminates, if the `EvolutionStream` is truncated. The `EvolutionStream` truncation is controlled by the `limit` predicate. As long as the predicate returns true, the evolution is continued.[5] At last, the `EvolutionResult` is collected from the `EvolutionStream` by one of the available `EvolutionResult` collectors.

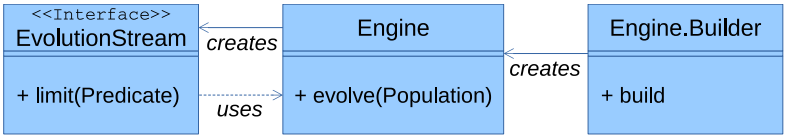


Figure 1.2.2: Evolution engine model

Figure 1.2.2 shows the static view of the main evolution classes, together with its dependencies. Since the `Engine` class itself is immutable, and can't be changed after creation, it is instantiated (configured) via a builder. The `Engine` can be used to create an arbitrary number of `EvolutionStream`s. The `EvolutionStream` is used to control the evolutionary process and collect the final result. This is done in the same way as for the normal `java.util.stream.Stream` classes. With the additional `limit(Predicate)` method, it

[5] See section2.6 on page 73 for a detailed description of the available termination strategies.

is possible to truncate the `EvolutionStream` if some termination criteria is fulfilled. The separation of `Engine` and `EvolutionStream` is the separation of evolution definition and evolution execution.

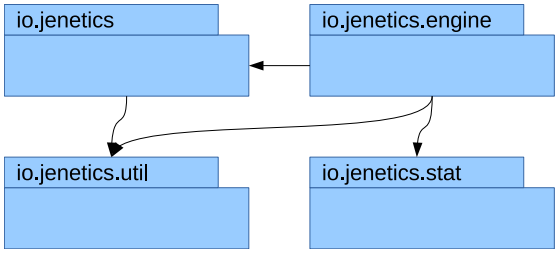


Figure 1.2.3: Package structure

In figure 1.2.3 the package structure of the library is shown and it consists of the following packages:

`io.jenetics` This is the base package of the **Jenetics** library and contains all domain classes like `Gene`, `Chromosome`, `Genotype` or `Phenotype`. All of this types are immutable data classes. It also contains the `Selector` and `Alterer` interfaces and its implementations. The classes in this package are (almost) sufficient to implement an own evolution *engine*.

`io.jenetics.engine` This package contains the actual GA implementation classes, e. g. `Engine`, `EvolutionStream` or `EvolutionResult`. They mainly operate on the domain classes in the `io.jenetics` package.

`io.jenetics.stat` This package contains additional statistics classes which are not available in the Java core library. Java only includes classes for calculating the sum and the average of a given numeric stream (e. g. `DoubleSummaryStatistics`). With this additions it is also possible to calculate the variance, skewness and kurtosis—using the `DoubleMomentStatistics` class. The `EvolutionStatistics` object, which can be calculated for every generation, relies on the classes in this package.

`io.jenetics.util` This package contains the collection classes (`BaseSeq`, `Seq`, `ISeq` and `MSeq`) which are used in the public interfaces of the `Chromosome` and `Genotype`. It also contains the `RandomRegistry`, which implements the *global* PRNG lookup, as well as helper `IO` classes for serializing `Genotype`s and whole populations.

1.3 Base classes

This chapter describes the base classes which are needed to setup and run an genetic algorithm with the **Jenetics**[6] library. They can roughly divided into three types:

[6]The documentation of the whole API is part of the download package or can be viewed online:`https://jenetics.io/javadoc/jenetics/9.1/index.html`.

Domain classes These classes form the domain model of the evolutionary algorithm and contain the structural classes like `Gene` and `Chromosome`. They are directly located in the `io.jenetics` package.

Operation classes These classes operate on the domain classes and includes the `Alterer` and `Selector` interfaces. They are also located in the `io.jenetics` package.

Engine classes These classes implement the actual evolutionary algorithm and can be found in the `io.jenetics.engine` package.

1.3.1 Domain classes

Most of the domain classes are pure data classes and can be treated as *value* objects[7]. All `Gene` and `Chromosome` implementations are immutable as well as the `Genotype` and `Phenotype` class.

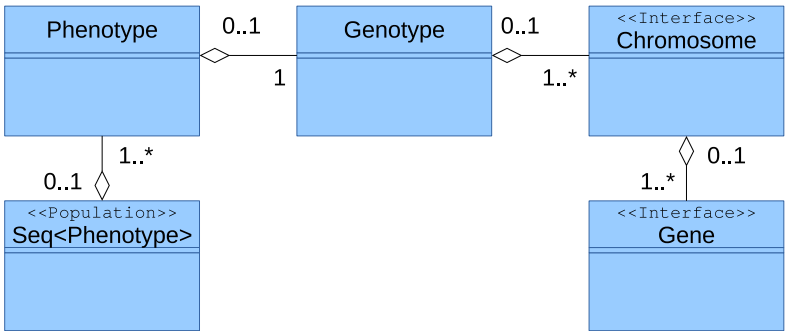


Figure 1.3.1: Domain model

Figure 1.3.1 shows the class diagram of the domain classes. The `Gene` is the base of the class structure. `Gene`s are aggregated in `Chromosome`s, and one to n `Chromosome`s are aggregated in `Genotype`s. A `Genotype` and a fitness `Function` form the `Phenotype`, `which` are collected into a population `Seq`.

1.3.1.1 Gene

The basic building blocks of the **Jenetics** library contain the actual information of the encoded solution, the allele. Some of the implementations also contain domain information of the wrapped allele. This is the case for all `Bounded-Gene`s, which contain the allowed minimum and maximum values. All `Gene` implementations are final and immutable. In fact, they are all value based classes and fulfill the properties which are described in the Java API documentation[34].[8]

Beside the container functionality for the allele, every `Gene` is its own factory and is able to create new, random instances of the same type and with the same constraints[9]. The factory methods are used by the `Alterer`s for creating new

[7] `https://en.wikipedia.org/wiki/Value_object`

[8] It is also worth reading the blog entry from Stephen Colebourne:`http://blog.joda.org/2014/03/valjos-value-java-objects.html`

[9] A constraint can restrict the space of valid values of a given problem domain. An example will be a`DoubleGene`, where the allowed minimal and maximal value of the double allele is part of the gene.

Genes from the existing one and play a crucial role by the exploration of the problem space.

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

Listing 1.3: Gene interface

Listing 1.3 shows the most important methods of the Gene interface. The isValid method, defined in the Verifiable interface, allows the gene to mark itself as invalid, e. g. when its allele is not within the allowed range. All invalid genes are replaced with new ones during the evolution phase. The available Gene implementations in the **Jenetics** library cover a wide range of problem encodings. Refer to chapter 2.1.1 for how to implement your own Gene types.

1.3.1.2 Chromosome

A Chromosome is a collection of Genes which must contain at least one Gene. This allows defining problems which require more than one Gene to encode. Like the Gene interface, the Chromosome is also its own factory and allows creation of a new Chromosome from a given Gene sequence.

```
public interface Chromosome<G extends Gene<?, G>>
    extends Factory<Chromosome<G>>, BaseSeq<G>, Verifiable
{
    G get(int index);
    int length();
    Chromosome<G> newInstance(ISeq<G> genes);
}
```

Listing 1.4: Chromosome interface

Listing 1.4 shows the main methods of the Chromosome interface. These are the methods for accessing single Genes by its index and the factory method for creating a new Chromosome from a given sequence of Genes. The factory method is used by the Alterer classes which were able to create altered Chromosomes from a (changed) Gene sequence. Most of the Chromosome implementations can be created with variable length. E. g. the IntegerChromosome can be created with variable length, where the minimum value of the length range is included and the maximum value of the length range is excluded.

```
IntegerChromosome chromosome = IntegerChromosome.of(
    0, 1_000, new IntRange(5, 9)
);
```

The factory method of the IntegerChromosome will now create chromosome instances with a length between $[range_{min}, range_{max})$, equally distributed. Figure 1.3.2 shows the structure of a Chromosome with variable length.

1.3.1.3 Genotype

The central processing class, the evolution Engine is working with, is the Genotype. It is the *structural* and immutable representative of an individual

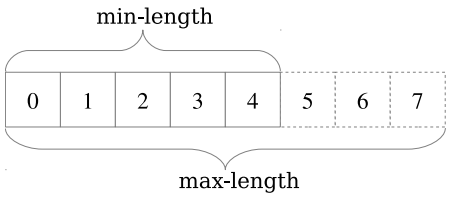


Figure 1.3.2: `Chromosome` structure

and consists of one to n `Chromosome`s. All `Chromosome`s must be parameterized with the same `Gene` type, but each `Chromosome` is allowed to have different lengths and constraints. The allowed minimal and maximal values of a `NumericChromosome` is an example of such a constraint. Within the same `Chromosome`, all alleles must lay within the defined minimal and maximal values.

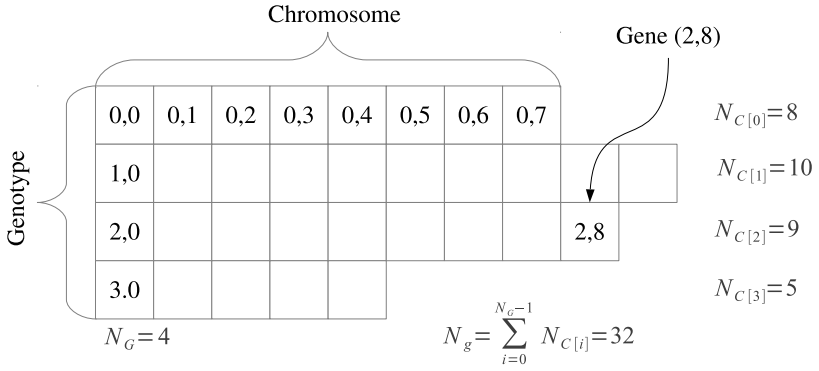


Figure 1.3.3: `Genotype` structure

Figure 1.3.3 shows the `Genotype` structure. A `Genotype` consists of N_G `Chromosome`s and a `Chromosome` consists of $N_{C[i]}$ `Gene`s (depending on the `Chromosome`). The overall number of `Gene`s of a `Genotype` is given by the sum of the `Chromosome`'s `Gene`s, which can be accessed via the `Genotype.geneCount()` method:

$$N_g = \sum_{i=0}^{N_G-1} N_{C[i]} \tag{1.3.1}$$

As already mentioned, the `Chromosome`s of a `Genotype` don't necessarily have to have the same size. It is only required that all genes are from the same type and the `Gene`s within a `Chromosome` have the same constraints; e. g. the same minimal and maximal values for numerical `Gene`s.

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

The code snippet in the listing above creates a `Genotype` with the same structure as shown in figure 1.3.3. In this example the `DoubleGene` has been chosen as the `Gene` type.

`Genotype` vector The `Genotype` is essentially a two-dimensional composition of `Gene`s. This makes it trivial to create `Genotype`s which can be treated as a `Gene` matrices. If its needed to create a vector of `Gene`s, there are two possibilities to do so:

1. creating a *row major* or
2. creating a *column major*

`Genotype` vector. Each of the two possibilities have specific advantages and disadvantages.

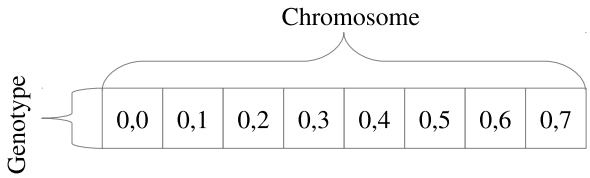


Figure 1.3.4: Row major `Genotype` vector

Figure 1.3.4 shows a `Genotype` vector in row major layout. A `Genotype` vector of length$n$ needs one `Chromosome` of length n. Each `Gene` of such a vector must obey the same constraints. E. g., for `Genotype` vectors containing `NumericGene`s, all `Gene`s must have the same minimum and maximum values. If the problem space doesn't need to have different minimum and maximum values, the row major `Genotype` vector is the preferred choice. Beside the easier `Genotype` creation, the available `Recombinator` alterers are more efficient in exploring the search domain.

If the problem space allows equal `Gene` constraint, the row major `Genotype` vector encoding should be chosen. It is easier to create and the available `Recombinator` classes are more efficient in exploring the search domain.

The following code snippet shows the creation of a row major `Genotype` vector. All `Alterer`s derived from the `Recombinator` do a fairly good job in exploring the problem space for a row major `Genotype` vector.

```
1 Genotype<DoubleGene> genotype = Genotype.of(
2     DoubleChromosome.of(0.0, 1.0, 8)
3 );
```

The column major `Genotype` vector layout must be chosen when the problem space requires `Gene`s with different constraints. This is almost the only reason for choosing the column major layout. The layout of this `Genotype` vector is

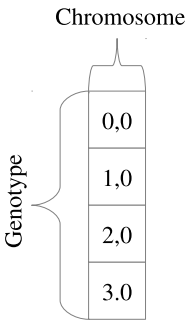


Figure 1.3.5: Column major `Genotype` vector

shown in 1.3.5. For a vector of length n, n `Chromosome`s of length one are needed.

The code snippet below shows how to create a `Genotype` vector in column major layout. It's a little bit more effort to create such a vector, since every `Gene` has to be wrapped into a separate `Chromosome`. The `DoubleChromosome` in the given example has a length of one, when the length parameter is omitted.

```
Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0,  1.0),
    DoubleChromosome.of(1.0,  2.0),
    DoubleChromosome.of(0.0, 10.0),
    DoubleChromosome.of(0.1,  0.9)
);
```

The greater flexibility of a column major `Genotype` vector has to be paid with a lower exploration capability of the `Recombinator` alterers. Using `Crossover` alterers will have the same effect as the `SwapMutator`, when used with row major `Genotype` vectors. Recommended alterers for vectors of `NumericGene`s are:

- `MeanAlterer`[10],
- `LineCrossover`[11] and
- `IntermediateCrossover`[12]

See also 2.3.2 for an advanced description on how to use the predefined vector codecs.

`Genotype` scalar A special case of a `Genotype` contains only one `Chromosome` with length one. The layout of such a `Genotype` *scalar* is shown in 1.3.6. Such `Genotype`s are mostly used for encoding real function problems.

How to create a `Genotype` for a real function optimization problem is shown in the code snippet below. The recommended `Alterer`s are the same as for column major `Genotype` vectors: `MeanAlterer`, `LineCrossover` and `IntermediateCrossover`.

[10] See 1.3.2.2 on page 23.
[11] See 1.3.2.2 on page 23.
[12] See 1.3.2.2 on page 24.

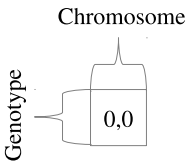


Figure 1.3.6: `Genotype` scalar

```
Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0, 1.0)
);
```

See also 2.3.1 for an advanced description on how to use the predefined scalar codecs.

1.3.1.4 Phenotype

The `Phenotype` is the *actual* representative of an individual and consists of the `Genotype`, the generation where the `Phenotype` has been created and an optional fitness value. Like the `Genotype`, the `Phenotype` is immutable and can't be changed after creation.

```
public final class Phenotype<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Comparable<Phenotype<G, C>>
{
    public Genotype<G> genotype();
    public long generation();
    public C fitness();
    public boolean isEvaluated();
    public Phenotype<G, C> withFitness(C fitness);
}
```

Listing 1.5: `Phenotype` class

Listing 1.5 shows the main methods of the `Phenotype`. The `fitness` property will return the actual fitness value of the `Genotype`, and the `Genotype` can be fetched with the `genotype()` method. If no fitness value is associated with the `Phenotype` yet, the `fitness()` method will throw an `NoSuchElementException`. Whether the fitness value has been set can be checked with the `isEvaluated()` method. Setting a fitness value can be done with the `withFitness(C)` method. Since the `Phenotype` is immutable, this method returns a new `Phenotype` with the set fitness value. Additionally to the fitness value, the `Phenotype` contains the generation when it was created. This allows for the calculation of the current age and allows for the removal of overaged individuals from the population.

1.3.1.5 Population

There is no special class which represents a population. It's *just* a collection of `Phenotype`s. As a collection class, the `ISeq` interface is used. The `ISeq` interface allows for the expression of the immutability of the population at the

type level and makes the code more readable. For a detailed description of this collection classes see section 1.4.4.

1.3.2 Operation classes

Genetic operators are used for creating genetic diversity (`Alterer`) and selecting potentially useful solutions for recombination (`Selector`). This section gives 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.

1.3.2.1 Selector

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 set independently.

The selection process of the **Jenetics** library acts on `Phenotypes` and indirectly, via the fitness function, on `Genotypes`. Direct `Gene` or population selection is not supported by the library.

```
Engine<DoubleGene, Double> engine = Engine.builder(...)
    .offspringFraction(0.7)
    .survivorsSelector(new RouletteWheelSelector<>())
    .offspringSelector(new TournamentSelector<>())
    .build();
```

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

$$N_{O_g} = \|O_g\| = \text{rint}\left(\|P_g\| \cdot f_O\right) \tag{1.3.2}$$

and the number of selected survivors

$$N_{S_g} = \|S_g\| = \|P_g\| - \|O_g\|\,. \tag{1.3.3}$$

The **Jenetics** library contains the following selector implementations:

- `TournamentSelector`
- `TruncationSelector`
- `MonteCarloSelector`
- `ProbabilitySelector`
- `RouletteWheelSelector`
- `LinearRankSelector`
- `ExponentialRankSelector`
- `BoltzmannSelector`
- `StochasticUniversalSelector`
- `EliteSelector`

Beside the well known standard selector implementation, the `Probability-Selector` 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, P_g. 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 in which 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. Compared with fitness proportional selectors, the tournament selector is often used in practice because of its lack of stochastic noise. Tournament selectors are also independent to the scaling of the genetic algorithm fitness function.

Truncation selector In truncation selection individuals are sorted according to their fitness and only the$n$ best individuals are selected. The truncation selection is a very basic selection algorithm. It has its strength in fast selecting individuals in large populations, but is not very often used in practice; whereas the truncation selection is a standard method in animal and plant breeding. Only the best animals, ranked by their phenotypic value, are selected for reproduction.

Monte Carlo selector The Monte Carlo selector selects the individuals from a given population randomly. Instead of a directed search, the Monte Carlo selector performs a random search. This selector can be used to measure the performance of other selectors. In general, the performance of a selector should be better than the selection performance of the Monte Carlo selector. If the Monte Carlo selector is used for selecting the parents for the population, it will be a little bit more disruptive, on average, than roulette wheel selection.[41]

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

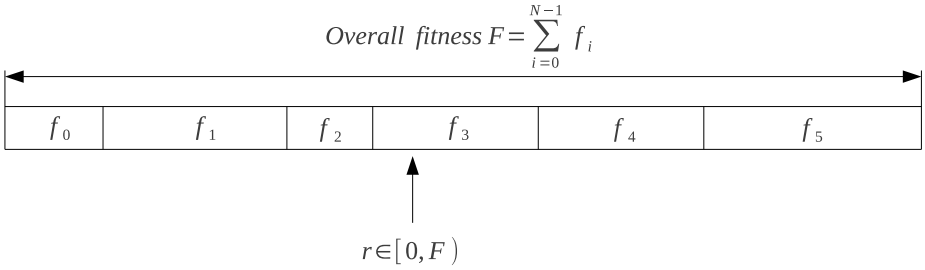


Figure 1.3.7: Fitness proportional selection

figure 1.3.7. An uniform distributed random number $r \in [0, F)$ specifies which individual is selected, by argument minimization:

$$i \leftarrow \underset{n \in [0,N)}{\operatorname{argmin}} \left\{ r < \sum_{i=0}^{n} f_i \right\}, \tag{1.3.4}$$

where N is the number of individuals and f_i the fitness value of the i^{th} 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

$$P(i,k) = \left(\begin{array}{c} n \\ k \end{array} \right) P(i)^{k} (1 - P(i))^{n-k} \tag{1.3.5}$$

where$n$ is the overall number of selected individuals and$k$ the number of individuals$i$ 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 the fitness proportional selector and **Jenetics** implements it as a probability selector. For calculating the selection probability, $P(i)$, the fitness value, f_i, of individual i is used.

$$P(i) = \frac{f_i}{\sum_{j=0}^{N-1} f_j} \tag{1.3.6}$$

Selecting n individuals from a given population is equivalent to play n times on the roulette-wheel. The population doesn't have to be sorted before selecting the individuals. Notice that equation 1.3.6 assumes that all fitness values are positive and the sum of the fitness values is not zero. To cope with negative fitnesses, an adapted formula is used for calculating the selection probabilities.

$$P'(i) = \frac{f_i - f_{\min}}{\sum_{j=0}^{N-1} (f_j - f_{\min})}, \tag{1.3.7}$$

where

$$f_{\min} = \min_{i \in [0,N)} \{f_i, 0\}$$

As you can see, the worst fitness value, $f_{\min}$, if negative, now has a selection probability of zero. In the case that the sum of the corrected fitness values is zero, the selection probability of all fitness values will be set $\frac{1}{N}$.

Linear-rank selector The roulette-wheel selector will have problems when the fitness values differ very much. If the best `Chromosome` fitness is 90%, its circumference occupies 90% of roulette-wheel, and then other `Chromosomes` have too few chances to be selected.[41] 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.

$$P(i) = \frac{1}{N} \left(n^- + \left(n^+ - n^-\right) \frac{i-1}{N-1} \right). \tag{1.3.8}$$

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.[8]

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

$$P(i) = (c-1)\frac{c^{i-1}}{c^N - 1}, \tag{1.3.9}$$

where$c$ must be within the range $[0, 1)$. 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

$$P(i) = \frac{\mathrm{e}^{b \cdot f_i}}{Z}, \tag{1.3.10}$$

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

$$Z = \sum_{i=1}^{n} \mathrm{e}^{f_i}. \tag{1.3.11}$$

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[4] (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 intervals. Weaker members of the population (according to their fitness) have a better chance to be chosen, which reduces the unfair nature of fitness-proportional selection methods. The selection method was introduced by James Baker.[5] Figure 1.3.8 shows the function of the stochastic-universal selection,

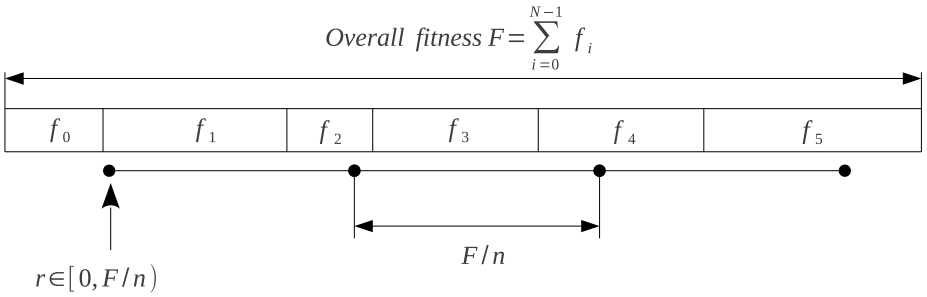


Figure 1.3.8: Stochastic-universal selection

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.[41]

Elite selector The `EliteSelector` copies a small proportion of the fittest candidates, without changes, into the next generation. This may have a dramatic impact on performance by ensuring that the GA doesn't waste time rediscovering previously refused partial solutions. Individuals that are preserved through elitism remain eligible for selection as parents of the next generation. Elitism is also related with memory: *remember the best solution found so far*. A problem with elitism is that it may cause the GA to converge to a *local* optimum, so pure elitism is a race to the nearest local optimum. The elite selector implementation of the **Jenetics** library also lets you specify the selector for the *non* elite individuals.

1.3.2.2 Alterer

The problem encoding/representation determines the bounds of the search space, but the `Alterer`s determine how the space can be traversed: `Alterer`s are responsible for the genetic diversity of the `EvolutionStream`. The two `Alterer` hierarchies used in **Jenetics** are:

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

First we will have a look at the mutation — There are two distinct roles *mutation* plays in the evolution process:

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 converging to a local minimum by stopping the solution to become too close to one another. A genetic algorithm can improve the solution solely by the mutation operator. 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

$$\hat{\mu} = N_P \cdot N_g \cdot P(m) \tag{1.3.12}$$

where N_g is the number of available genes of a genotype and N_P the population size (refer to equation 1.3.1).

Mutator The mutator has to deal with the problem, that the genes are arranged in a *hierarchical* structure with three levels (see chapter 1.3.1.3). 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

$$P_G(m) = P_C(m) = P_g(m) = \sqrt[3]{P(m)}. \tag{1.3.13}$$

Gaussian mutator The Gaussian mutator performs the mutation of number genes. The values created by this mutator follows a Gaussian distribution, and are guaranteed to lie within the range of the number genes. The shape of the distribution can be determined by the `GaussianMutator.Shape` class. It allows to configure a `shift` and a `sigma` parameter. Diagram 1.3.9 shows the distribution's probability density for `shift` parameter, S, between 0 and 1. The gene range is $[0, 20]$. The actual mean value, μ, depends on the gene range and is set to $\mu = (S + 1)\, D$, where $D = \frac{g_{max} - g_{min}}{2}$.

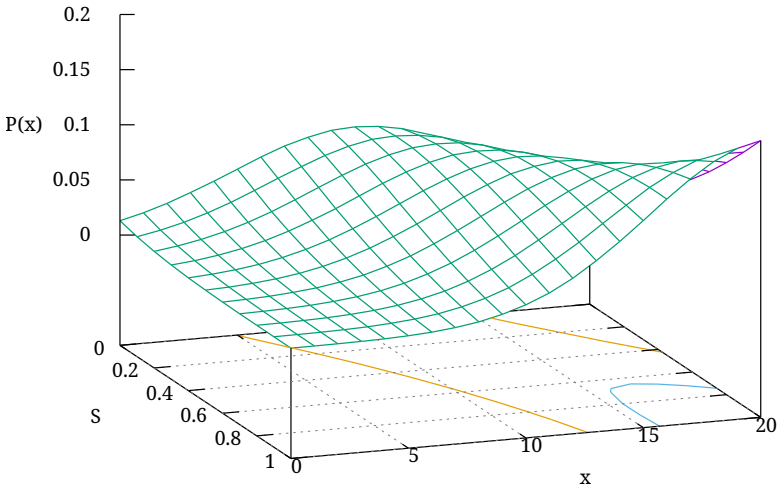


Figure 1.3.9: Gaussian mutator S-shape

Diagram 1.3.10 shows the distribution's probability density for sigma parameter, Σ, between 0.5 and 5. The gene range is $[0, 20]$. The standard deviation, σ, of the new mutation value depends on the gene range and is set to $\sigma = \frac{D}{\Sigma}$.

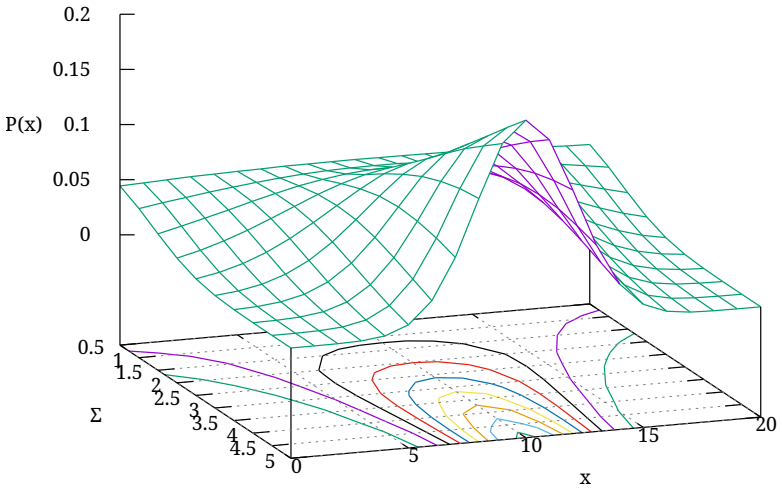


Figure 1.3.10: Gaussian mutator Σ-shape

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.

The second alterer type is the recombination — An enhanced genetic algorithm (EGA) combines elements of existing solutions in order to create a new solution, with some of the properties of each parent. 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 merging those portions of different chromosomes to form new ones.

```
1 void recombine(final MSeq<Phenotype<G, C>> pop) {
2     // Select the Genotypes for crossover.
3     final RandomGenerator random = RandomRegistry.random();
4     final int i1 = random.nextInt(pop.length());
```

```
5       final int i2 = random.nextInt(pop.length());
6       final Phenotype<G, C> pt1 = pop.get(i1);
7       final Phenotype<G, C> pt2 = pop.get(i2);
8       final Genotype<G> gt1 = pt1.genotype();
9       final Genotype<G> gt2 = pt2.genotype();
10
11      //Choosing the Chromosome for crossover.
12      final int chIndex =
13          random.nextInt(min(gt1.length(), gt2.length()));
14      final MSeq<Chromosome<G>> c1 = MSeq.of(gt1);
15      final MSeq<Chromosome<G>> c2 = MSeq.of(gt2);
16      final MSeq<G> genes1 = MSeq.of(c1.get(chIndex));
17      final MSeq<G> genes2 = MSeq.of(c2.get(chIndex));
18
19      // Perform the crossover.
20      crossover(genes1, genes2);
21      c1.set(chIndex, c1.get(chIndex).newInstance(genes1.toISeq()));
22      c2.set(chIndex, c2.get(chIndex).newInstance(genes2.toISeq()));
23
24      //Creating two new Phenotypes and replace the old one.
25      pop.set(i1, Phenotype.of(Genotype.of(c1.toISeq())));
26      pop.set(i2, Phenotype.of(Genotype.of(c2.toISeq())));
27  }
```

Listing 1.6: Chromosome selection for recombination

Listing 1.6 shows how two chromosomes are selected for *recombination*. It is done this way for preserving the given *constraints* and to avoid the creation of invalid individuals.

Because of the possible different `Chromosome` length and/or `Chromosome` constraints within a `Genotype`, only `Chromosomes` with the same `Genotype` position are recombined (see listing1.6).

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 `combine` method.

Shift mutator The shift mutator applies mutation between two randomly chosen points, a and c. A random value between the two points, b, splits the sequences of genes between the positions. The second sequence is then shifted in front of the first one. This mutation operator can be used for combinatorial problems, where no duplicated genes within a chromosome are allowed, e.g., for the TSP.[26] Figure 1.3.11 on the next page shows two shift examples with the chosen shift points and the results of the shift.

Shuffle mutator The shuffle mutator, changes the order of the genes between two randomly chosen positions. The genes between the positions are shuffled. This mutation operator can also be used for combinatorial problems, where no duplicated genes within a chromosome are allowed, e.g., for the TSP.[26] Figure

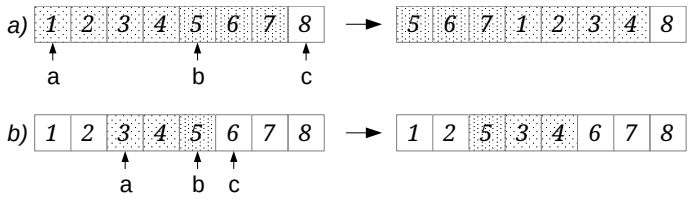


Figure 1.3.11: Shift mutator

1.3.12 shows two shuffle examples with the chosen shuffle points and a possible mutation result.

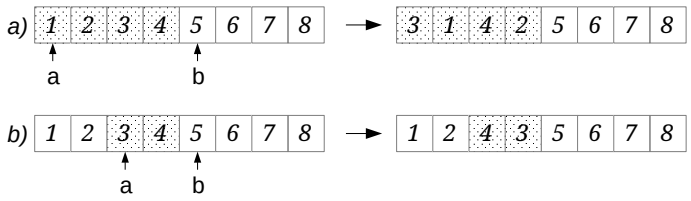


Figure 1.3.12: Shuffle mutator

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 multi-point or uniform crossover.

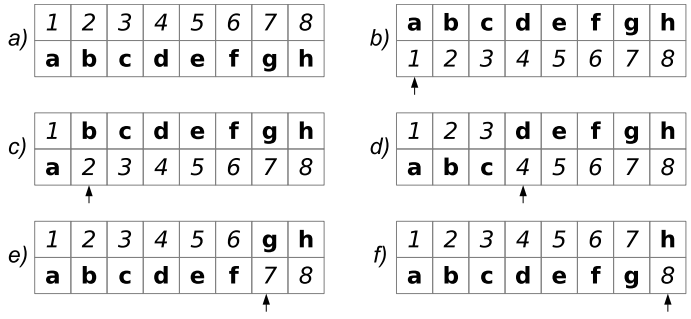


Figure 1.3.13: Single-point crossover

Figure 1.3.13 shows how the `SinglePointCrossover` class is performing the crossover for different crossover points. Sub figure *a)* shows the two chromosomes chosen for crossover. The examples in sub figures *b)* to *f)* illustrate the crossover results for 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 figures in 1.3.14 shows how the multi-point crossover works with two crossover points. Figure *a)* shows the two chromosomes chosen for crossover, *b)* shows the crossover result for the crossover points at index 0 and 4, *c)* uses crossover points at index 3 and 6 and *d*) at index 0 and 7.

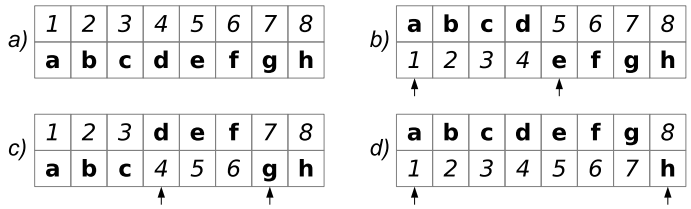


Figure 1.3.14: 2-point crossover

Figure 1.3.15 you can see how the crossover works for an odd number of crossover points.

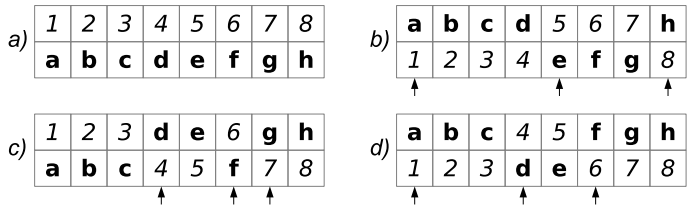


Figure 1.3.15: 3-point crossover

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 1.3.16 *a*)).

After performing the crossover we–normally–got two invalid chromosomes (figure 1.3.16 *b*)). Chromosome 1 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 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

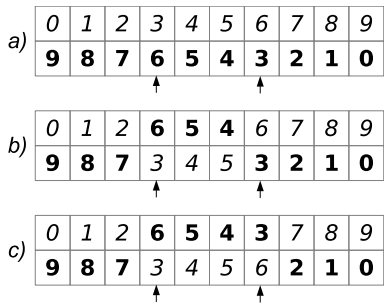


Figure 1.3.16: Partially-matched crossover

this exchange outside the crossing region (figure 1.3.16 *b*)). At the end figure 1.3.16 *c*) shows the repaired chromosome.

Uniform order-based crossover The Uniform order-based crossover guarantees that all genes are found exactly once in each chromosome. No gene is duplicated by this crossover. This crossover can be applied usefully in the TSP or other permutation problem encodings.[26] How the uniform order-based crossover works is shown with the following example chromosomes `C1` and `C2`.

```
C1 = 0123456789
C2 = 9876543210
```

1. Within the uniform order-based crossover, a set of positions are chosen randomly. The genes at the positions are reordered in the order they occur in the other parent.

```
Positions = 2, 4, 5, 7, 8
```

2. The genes at the chosen positions are removed.

```
C1 = 01_3__6__9
C2 = 9__6__3_10
```

3. The order of the removed genes as they occur in the two genes.

```
C1 = 2, 4, 5, 7, 8
C2 = 8, 7, 5, 4, 2
```

4. The removed genes are than added in the order they occur in the other chromosome.

```
C1 = 0183756429
C2 = 9246573810
```

Uniform crossover In uniform crossover, the genes at index$i$ of two chromosomes are swapped with the swap probability, p_S. Empirical studies show that uniform crossover is a more exploitative approach than the traditional exploitative approach that maintains longer schemata. This leads to a better search of the design space with maintaining the exchange of good information.[11]

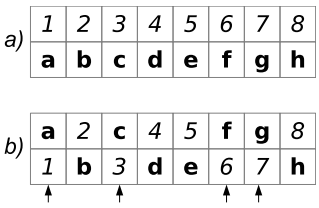


Figure 1.3.17: Uniform crossover

Figure 1.3.17 shows an example of a uniform crossover with four crossover points. A gene is swapped, if a uniformly created random number, $r \in [0, 1]$, is smaller than the swap probability, p_S. The following code snippet shows how these swap indexes are calculated, in a functional way.

```
final RandomGenerator random = RandomRegistry.random();
final int length = 8;
final double ps = 0.5;
final int[] indexes = IntStream.range(0, length)
    .filter(i -> random.nextDouble() < ps)
    .toArray();
```

Combine alterer This alterer changes two genes by combining them. The combine function can be defined when the alterer is created. How this is done, is shown in the code snippet below.

```
final var alterer = new CombineAlterer<DoubleGene, Double>(
    (g1, g2) -> g1.newInstance(g1.doubleValue()/g2.doubleValue())
);
```

Mean alterer The Mean alterer works on genes which implement the `Mean` interface. All numeric genes implement this interface by calculating the arithmetic mean of two genes. This alterer is a specialization of the `CombineAlterer`.

Line crossover The line crossover[13] takes two numeric chromosomes and treats it as a real number vector. Each of these vectors can also be seen as a point in $\mathbb{R}^n$. If we draw a line through these two points (chromosome), we have the possible values of the new chromosomes, which all lie on this line.

Figure 1.3.18 shows how the two chromosomes form the two 3-dimensional vectors (black circles). The dashed line, connecting the two points, form the possible solutions created by the line crossover. An additional variable, p, determines how far out along the line the created children will be. If $p = 0$ then the children will be located along the line within the hypercube. If $p > 0$, the children may be located on an arbitrary place on the line, even outside of the hypercube.This is useful if you want to explore unknown regions, and you need a way to generate chromosomes further out than the parents are.

The internal random parameters, which define the location of the new crossover point, are generated once for the whole vector (chromosome). If

[13]The line crossover, also known as line recombination, was originally described by Heinz Mühlenbein and Dirk Schlierkamp-Voosen.[32]

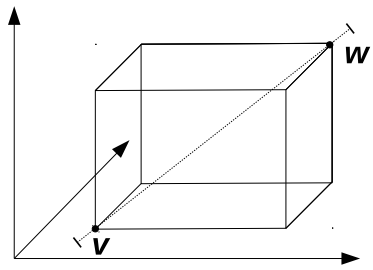


Figure 1.3.18: Line crossover hypercube

the `LineCrossover` generates numeric genes which lie outside the allowed minimum and maximum value, it simply uses the original gene and rejects the generated, invalid one.

Intermediate crossover The intermediate crossover is quite similar to the line crossover. It differs in the way on how the internal random parameters are generated and the handling of the invalid—out of range—genes. The *internal* random parameters of the `IntermediateCrossover` class are generated for *each* gene of the chromosome, instead once for all genes. If the newly generated gene is not within the allowed range, a new one is created. This is repeated, until a valid gene is built.

The crossover parameter, p, has the same properties as for the line crossover. If the chosen value for p is greater than 0, it is likely that some genes must be created more than once, because they are not in the valid range. The probability for gene recreation rises sharply with the value of p. Setting p to a value greater than one, doesn't make sense in most of the cases. A value greater than 10 should be avoided.

Partial alterer Alterers are working on the whole population, which is effectively a sequence of genotypes. If your genotype consists of more than one chromosome, the alterer is applied to all chromosomes. There is no way to bind an alterer to a specific chromosome. The `PartialAlterer` class overcomes this shortcoming and allows you to define the chromosomes the wrapped `Alterer` is using.

```
final Genotype<DoubleGene> gtf = Genotype.of(
    DoubleChromosome.of(0, 1),
    DoubleChromosome.of(1, 2),
    DoubleChromosome.of(2, 3),
    DoubleChromosome.of(3, 4)
);
final Engine<DoubleGene, Double> engine = Engine.builder(ff, gtf)
    .alterers(
        PartialAlterer.of(new Mutator<DoubleGene, Double>(), 0, 3),
        PartialAlterer.of(new MeanAlterer<DoubleGene, Double>(), 1),
        new LineCrossover<>())
    .build();
```

The example above shows how to use the `PartialAlterer`. The wrapped `Mutator` will only operate on the chromosome with the index 0 and 3, the wrapped `MeanAlterer` will alter on the chromosome with index 1 and the `LineCrossover`

will work on all chromosomes. A potential drawback of the `PartialAlterer` is a possible performance penalty. This is because the chromosomes must be *sliced* into different population sequences for each `PartialAlterer`. If this is an issue for the overall performance depends on the concrete application.

1.3.3 Engine classes

The executing classes, which perform the actual evolution, are located in the `io.jenetics.engine` package. The *evolution stream* (`EvolutionStream`) is the base metaphor for performing an GA. On the `EvolutionStream` you can define the termination predicate and *collect* the final `EvolutionResult`. This decouples the static data structure from the executing evolution part. The `EvolutionStream` is also very flexible, when it comes to collecting the final result. The `EvolutionResult` class has several predefined collectors, but you are free to create your own one, which can be seamlessly plugged into the existing stream.

1.3.3.1 Fitness function

The fitness `Function` is also an important part when modeling a genetic algorithm. It takes a `Genotype` as argument and returns a fitness value. The returned fitness value must implement the `Comparable` interface. This allows the evolution `Engine`, respectively the selection operators, to select the offspring- and survivor population. Some selectors have stronger requirements for the fitness value than a `Comparable`, but these constraints are checked by the Java type system at compile time.

The fitness`Function` must be deterministic. Whenever it is applied to the same `Genotype`, it must return the same fitness value. Nondeterministic fitness `functions` can lead to unexpected behavior, since the calculated fitness value is cached by the `Phenotype`.

The following example shows the simplest possible fitness `Function`. This `Function` just returns the allele of a 1x1 *float* `Genotype`.

```
1  public class Main {
2      static Double identity(final Genotype<DoubleGene> gt) {
3          return gt.gene().allele();
4      }
5
6      void main() {
7          // Create fitness function from method reference.
8          Function<Genotype<DoubleGene>, Double>> ff1 =
9              Main::identity;
10
11         // Create fitness function from lambda expression.
12         Function<Genotype<DoubleGene>, Double>> ff2 = gt ->
13             gt.gene().allele();
14     }
15 }
```

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, which must at least implement the `Comparable` interface.

1.3.3.2 Engine

The evolution `Engine` controls how the evolution steps are executed. Once the `Engine` is created, via its `Builder` class, it can't be changed. It doesn't contain any mutable global state and can therefore be safely used/called from different threads. This allows to create more than one `EvolutionStreams` from the same `Engine` and execute them independently.

```
public final class Engine<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Evolution<G, C>,
        EvolutionStreamable<G, C>
{
    // The evolution function, performs one evolution step.
    public EvolutionResult<G,C> evolve(EvolutionStart<G, C> start);

    // Evolution stream for "normal" evolution execution.
    public EvolutionStream<G,C> stream();
}
```

Listing 1.7: `Engine` class

Listing 1.7 shows the main methods of the `Engine` class. The `Engine` is used for performing the actual evolution of a given population. One evolution step is executed by calling the `Engine.evolve` method, which returns an `EvolutionResult` object. This object contains the evolved population plus additional information like the killed and as invalid marked individuals. With the `stream()`method you create a new `EvolutionStream`, which is used for controlling the evolution process. For more information about the `EvolutionStream` see section 1.3.3.4.

As already shown in previous examples, the `Engine` can only be created via its `Builder` class. Only the fitness `Function` and the `Chromosomes`, which represents the problem encoding, must be specified for creating an `Engine` instance. For the rest of the parameters, default values have been specified. This are the `Engine` parameters which can configured:

`alterers` A list of `Alterers` which are applied to the offspring population, in the defined order. The default value of this property is set to `SinglePointCrossover<>(0.2)` followed by `Mutator<>(0.15)`.

`clock` The `java.time.InstantSource` used for calculating the execution durations. A `InstantSource` with nanosecond precision (`System.nanoTime()`) is used as default.

`constraint` This property lets you *override* the default implementation of the `Phenotype::isValid` method, which is useful if the `Phenotype` validity not only depends on valid property of the elements it consists of. A description of the `Constraint` interface is given in section 2.5.

`executor` With this property it is possible to change the `java.util.concurrent.Executor` engine used for evaluating the evolution steps. This property can be used to define an application wide `Executor` or for controlling the number of execution threads. The default value is set to `ForkJoinPool.commonPool()`.

`fitnessFunction` This property defines the fitness `Function` used by the evolution `Engine`. (See section 1.3.3.1.)

`fitnessExecutor` This property lets you define the executor, which is responsible for evaluating the fitness function. It is possible to evaluate each fitness function in its own *virtual* thread. The fitness executor is only used for evaluating the fitness function. For executing the genetic operations, the configured `executor` is used.

`genotypeFactory` Defines the `Genotype Factory` used for creating new individuals. Since the `Genotype` is its own `Factory`, it is sufficient to create a `Genotype`, which serves as a template.

`interceptor` The interceptor lets you define functions, which are able to change the `EvolutionResult` before and after an evolution step. An `EvolutionInterceptor` can be seen as a crosscutting aspect of the evolution process. One implementation of the EvolutionInterceptor is the `FitnessNullifier`, which allows you to enforce the reevaluation of the fitness values of all individuals. This might be handy, if the fitness function is not time invariant and can change during the evolution process.

`maximalPhenotypeAge` Set the maximal allowed age of an individual (`Phenotype`). This prevents super individuals to live forever. The default value is set to 70.

`offspringFraction` Through this property it is possible to define the fraction of offspring (and survivors) for evaluating the next generation. The fraction value must within the interval $[0, 1]$. The default value is set to 0.6. Additionally to this property, it is also possible to set the `survivorsFraction`, `survivorsSize` or `offspringSize`. All these additional properties effectively set the `offspringFraction`.

`offspringSelector` This property defines the `Selector` used for selecting the offspring population. The default values are set to `TournamentSelector<>(3)`.

`optimize` With this property it is possible to define whether the fitness `Function` should be maximized or minimized. By default, the fitness `Function` is maximized.

`populationSize` Defines the number of individuals of a population. The evolution `Engine` keeps the number of individuals constant. That means, the population of the `EvolutionResult` always contains the number of entries defined by this property. The default value is set to 50.

`selector` This method allows to set the `offspringSelector` and `survivorsSelector` in one step with the same selector.

survivorsSelector This property defines the `Selector` used for selecting the survivors population. The default values are set to `TournamentSelector<>(3)`.

The `EvolutionStream`s, created by the `Engine` class, are unlimited. Such streams must be limited by calling the available `EvolutionStream::limit` methods. Alternatively, the `Engine` instance itself can be limited with the `Engine::limit` methods. This limited `Engine`s no longer creates infinite `EvolutionStream`s, they are truncated by the limit predicate defined by the `Engine`. This feature is needed for concatenating evolution `Engine`s (see section 3.1.6.1).

```
final EvolutionStreamable<DoubleGene, Double> engine =
    Engine.builder(problem)
        .minimizing()
        .build()
        .limit(() -> Limits.bySteadyFitness(10));
```

As shown in the example code above, one important difference between the `Engine.limit` and the `EvolutionStream::limit` method is, that the limit method of the `Engine` takes a limiting `Predicate Supplier` instead of the `Predicate` itself. The reason for this is that some `Predicate`s have to maintain internal state to work properly. This means, every time the `Engine` creates a new stream, it must also create a new limiting `Predicate`. The `Engine::limit` function will return an `EvolutionStreamable` instead of an `Engine`. This interface lets you create `EvolutionStream`s, which is what you usually want to do with the `Engine`.

1.3.3.3 Evolution

This functional interface represents the *evolution* function, which is implemented by the `Engine` class. The main purpose of the `Evolution` interface is to decouple the evolution function/strategy from the actual evolution process, represented by the `EvolutionStream`. Listing 1.8 shows the definition of the `Evolution` *functional* interface.

```
@FunctionalInterface
public interface Evolution<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    EvolutionResult<G, C> evolve(EvolutionStart<G, C> start);
}
```

Listing 1.8: `Evolution` interface

1.3.3.4 EvolutionStream

The `EvolutionStream` controls the execution of the evolution process and can be seen as a kind of execution handle. This handle can be used to define the termination criteria and to collect the final evolution result. Since the `EvolutionStream` extends the Java `Stream` interface, it integrates smoothly with the rest of the Java Stream API.[14]

[14]It is recommended to make yourself familiar with the Java Stream API. A good introduction can be found here:`http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/`

```java
public interface EvolutionStream<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    extends Stream<EvolutionResult<G, C>>
{
    EvolutionStream<G, C>
    limit(Predicate<? super EvolutionResult<G, C>> proceed);
}
```

Listing 1.9: `EvolutionStream` interface

Listing 1.9 shows the whole `EvolutionStream` interface. As it can be seen, it only adds one additional method. But this additional `limit` method allows you to truncate the `EvolutionStream` based on a `Predicate` which takes an `EvolutionResult`. Once the `Predicate` returns `false`, the evolution process is stopped. Since the `limit` method returns an `EvolutionStream`, it is possible to define more than one `Predicate`, which both must be fulfilled to continue the evolution process.

```java
final Engine<DobuleGene, Double> engine = ...
final EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .limit(predicate1)
    .limit(predicate2)
    .limit(100);
```

The `EvolutionStream`, created in the example above, will be truncated if one of the two predicates is `false` or if the maximal allowed generations, of 100, is reached. An `EvolutionStream` is usually created via the `Engine::stream` method. The immutable and stateless nature of the evolution `Engine` allows you to create more than one `EvolutionStream` with the same `Engine`.

The generations of the `EvolutionStream` are evolved serially. Calls of the `EvolutionStream` methods (e. g. `limit`, `peek`, ...) are executed in the thread context of the created `Stream`. In a *typical* setup, no additional synchronization and/or locking is needed.

In cases where you appreciate the usage of the `EvolutionStream` but need a different `Engine` implementation, you can use the `EvolutionStream::of` factory method for creating an new `EvolutionStream`.

```java
static <G extends Gene<?, G>, C extends Comparable<? super C>>
EvolutionStream<G, C> of(
    Supplier<EvolutionStart<G, C>> start,
    Function<? super EvolutionStart<G, C>, EvolutionResult<G, C>> f
);
```

This factory method takes a start value, of type `EvolutionStart`, and an evolution `Function`. The evolution `Function` takes the start value and returns an `EvolutionResult` object. To make the runtime behavior more predictable, the start value is fetched/created lazily at the evolution start time.

```java
final Supplier<EvolutionStart<DoubleGene, Double>> start = ...
final EvolutionStream<DoubleGene, Double> stream =
    EvolutionStream.of(start, new MySpecialEngine());
```

1.3.3.5 EvolutionResult

The `EvolutionResult` contains the result data of an evolution step and is the element type of the `EvolutionStream`, as described in section 1.3.3.4.

```
public final class EvolutionResult<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Comparable<EvolutionResult<G, C>>
{
    public ISeq<Phenotype<G, C>> population();
    public long generation();
}
```

Listing 1.10: `EvolutionResult` class

Listing 1.3.3.5 shows the two most important properties, the `population` and the `generation` the result belongs to. These are also the two properties needed for the next evolution step. The `generation` is, of course, incremented by one. To make collecting the `EvolutionResult` object easier, it also implements the `Comparable` interface. Two `EvolutionResult`s are compared by its best `Phenotype`, depending on the optimization direction. The `EvolutionResult` classes has three predefined factory methods, which will return `Collector`s usable for the `EvolutionStream`:

`toBestEvolutionResult()` Collects the best `EvolutionResult` of a `EvolutionStream` according to the defined optimization strategy (minimization or maximization).

`toBestPhenotype()` This collector can be used if you are only interested in the best `Phenotype`.

`toBestGenotype()` Use this collector if you only need the best `Genotype` of the `EvolutionStream`.

The following code snippets show how to use the different `EvolutionStream` collectors.

```
// Collecting the best EvolutionResult of the EvolutionStream.
final EvolutionResult<DoubleGene, Double> result = stream
    .collect(EvolutionResult.toBestEvolutionResult());

// Collecting the best Phenotype of the EvolutionStream.
final Phenotype<DoubleGene, Double> result = stream
    .collect(EvolutionResult.toBestPhenotype());

// Collecting the best Genotype of the EvolutionStream.
final Genotype<DoubleGene> result = stream
    .collect(EvolutionResult.toBestGenotype());
```

Sometimes it is useful not only to collect one *final* result, but to collect the*n* best evolution results instead. This can be achieved by combining the `MinMax::toStrictlyIncreasing` and `ISeq::toISeq(int)` method.

```
final ISeq<EvolutionResult<DoubleGene, Double>> results = engine
    .stream()
    .limit(1000)
    .flatMap(MinMax.toStrictlyIncreasing())
    .collect(ISeq.toISeq(10));
```

The code snippet above collects the best 10 evolution results into the results sequence in increasing order.

1.3.3.6 EvolutionStatistics

The `EvolutionStatistics` class allows you to gather additional statistical information from the `EvolutionStream`. This is especially useful during the development phase of an application, when you have to find the right parametrization of the evolution `Engine`. Besides other information, the `EvolutionStatistics` contains (statistical) information about the fitness, invalid and killed `Phenotypes` and runtime information of the different evolution steps. Since the `EvolutionStatistics` class implements the `Consumer<EvolutionResult<?, C>>` interface, it can be easily plugged into the `EvolutionStream`, adding it with the `peek` method of the stream.

```
final Engine<DoubleGene, Double> engine = ...
final EvolutionStatistics<?, Double> statistics =
    EvolutionStatistics.ofNumber();
engine.stream()
    .limit(100)
    .peek(statistics)
    .collect(toBestGenotype());
```

Listing 1.11: `EvolutionStatistics` usage

Listing 1.11 shows how to add the `EvolutionStatistics` to the `EvolutionStream`. Once the algorithm tuning is finished, it can be removed in the production environment.

There are two different specializations of the `EvolutionStatistics` object available. The first is the general one, which will be working for every kind of `Genes` and fitness types. It can be created via the `EvolutionStatistics::ofComparable` method. The second one collects additional statistical data for numerical fitness values. This can be created with the `EvolutionStatistics::ofNumber` method.

```
+--------------------------------------------------------------------------+
|  Time statistics                                                         |
+--------------------------------------------------------------------------+
|             Selection: sum=0.046538278000 s; mean=0.003878189833 s       |
|              Altering: sum=0.086155457000 s; mean=0.007179621417 s       |
|   Fitness calculation: sum=0.022901606000 s; mean=0.001908467167 s       |
|     Overall execution: sum=0.147298067000 s; mean=0.012274838917 s       |
+--------------------------------------------------------------------------+
|  Evolution statistics                                                    |
+--------------------------------------------------------------------------+
|           Generations: 12                                                |
|               Altered: sum=7,331; mean=610.916666667                     |
|                Killed: sum=0; mean=0.000000000                           |
|              Invalids: sum=0; mean=0.000000000                           |
+--------------------------------------------------------------------------+
|  Population statistics                                                   |
+--------------------------------------------------------------------------+
|                   Age: max=11; mean=1.951000; var=5.545190               |
|               Fitness:                                                   |
|                      min  = 0.000000000000                               |
|                      max  = 481.748227114537                             |
|                      mean = 384.430345078660                             |
|                      var  = 13006.132537301528                           |
+--------------------------------------------------------------------------+
```

A typical output of an number `EvolutionStatistics` object will look like the example above.

The `EvolutionStatistics` object is a simple way for inspecting the `EvolutionStream` after it is finished. It doesn't give you a *live* view of the current evolution process, which can be necessary for long running streams. In such cases you have to maintain/update the statistics yourself.

```
public class TSM {
    // The locations to visit.
    static final ISeq<Point> POINTS = ISeq.of(...);

    // The permutation codec.
    static final Codec<ISeq<Point>, EnumGene<Point>>
    CODEC = Codecs.ofPermutation(POINTS);

    // The fitness function (in the problem domain).
    static double dist(final ISeq<Point> p) {...}

    // The evolution engine.
    static final Engine<EnumGene<Point>, Double> ENGINE = Engine
        .builder(TSM::dist, CODEC)
        .optimize(Optimize.MINIMUM)
        .build();

    // Best phenotype found so far.
    static Phenotype<EnumGene<Point>, Double> best = null;

    // You will be informed on new results. This allows to
    // react on new best phenotypes, e.g. log it.
    private static void update(
        final EvolutionResult<EnumGene<Point>, Double> result
    ) {
        if (best == null ||
            best.compareTo(result.bestPhenotype()) < 0)
        {
            best = result.bestPhenotype();
            IO.print(result.generation() + ": ");
            IO.println("Found best phenotype: " + best);
        }
    }

    // Find the solution.
    void main() {
        final ISeq<Point> result = CODEC.decode(
            ENGINE.stream()
                .peek(TSM::update)
                .limit(10)
                .collect(EvolutionResult.toBestGenotype())
        );
        IO.println(result);
    }
}
```

Listing 1.12: Live evolution statistics

Listing 1.12 shows how to implement a manual statistics gathering. The update method is called whenever a new `EvolutionResult` has been calculated. If a new best `Phenotype` is available, it is stored and logged. With the `TSM::update` method, which is called on every finished generation, you created a *live* view on the evolution progress.

1.3.3.7 Evaluator

The `Evaluator` is responsible for evaluating the fitness values for a given population. It is the most general way for doing the fitness evaluation. Usually, it is not necessary to implement an own evaluation strategy. If you are creating an evolution `Engine` with a fitness function, this is done for you automatically. Each fitness value is then evaluated concurrently, but independently from each other. Using the `Evaluator` interface is helpful if you have performance problems when the fitness function is evaluated serially—or in small concurrent batches, as it is implemented by the default strategy. In this case, the `Evaluator` interface can be used to calculate the fitness function for a population in one batch. Another use case for the `Evaluator` interface is, when the fitness value also depends on the current composition of the population. E. g. it is possible to normalize the population's fitness values.

```
@FunctionalInterface
public interface Evaluator<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    ISeq<Phenotype<G, C>> eval(Seq<Phenotype<G, C>> population);
}
```

Listing 1.13: `Evaluator` interface

The implementer is free to evaluate the whole population, or only evaluate the not yet evaluated `Phenotypes`. There are only two requirements which must be fulfilled:

1. the size of the returned, evaluated, phenotype sequence must be exactly the size of the input phenotype sequence and
2. all phenotypes of the returned population must have a fitness value assigned. That means, the expression `pop.forAll(Phenotype::isEvaluated)` must be true.

The code snippet below creates an evaluator which evaluates the fitness values of the whole population serially in the main thread.

```
final Function<? super Genotype<G>, ? extends C> fitness = ...;
final Evaluator<G, C> evaluator = population -> population
    .map(pt -> pt.eval(fitness))
    .asISeq();
```

To use the fitness `Evaluator`, you have to use the `Engine.Builder` constructor directly, instead of one of the factory methods.

```
final Engine<G, C> engine = new Engine.Builder(evaluator, gtf)
    .build();
```

The `Evaluators` class contains factory methods, which allows you to create `Evaluator` instances from fitness functions which don't return the fitness value directly, but return `Future<T>` or `CompletableFuture<T>` instead. With these methods, there is no need for waiting for the fitness value, if the fitness function is already asynchronous.

```
static Future<Double> fitness(final double x) {
    return ...;
}
```

```
4  void main() {
5      final Codec<Double, DoubleGene> codec = ...;
6      final Evaluator<DoubleGene, Double> evaluator = Evaluators
7          .async(Main::fitness, codec);
8
9      final Engine<DoubleGene, Double> engine =
10         new Engine.Builder<>(evaluator, codec.encoding())
11             .build();
12 }
```

1.4 Nuts and bolts

1.4.1 Concurrency

The **Jenetics** library parallelizes independent tasks whenever possible. Especially the evaluation of the fitness function is done concurrently. That means that the fitness function must be thread-safe and deterministic. The easiest way for achieving thread safety is to make the fitness function immutable and reentrant. Since the number of individuals of one population, which determines the number of fitness functions to be evaluated, is usually much higher then the number of available CPU cores, the fitness evaluation is done in batches. This reduces the evaluation overhead, for *lightweight* fitness functions.

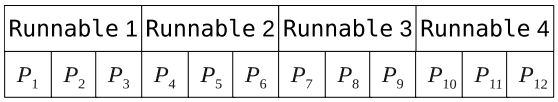


Figure 1.4.1: Evaluation batch

Figure 1.4.1 shows an example population with 12 individuals. The evaluation of the phenotype's fitness functions are evaluated in batches with three elements. For this purpose, the individuals of one batch are wrapped into a `Runnable` object. The batch size is automatically adapted to the available number of CPU cores. It is assumed that the evaluation cost of one fitness function is quite small. If this assumption doesn't hold, you can configure the the maximal number of batch elements with the `io.jenetics.concurrency.maxBatchSize` system property. The usage of this property is described in section 1.4.1.2.

1.4.1.1 Basic configuration

The used `Executor` can be defined when building the evolution `Engine` object. How to do this is shown in the code example below.

```
1  import java.util.concurrent.Executor;
2  import java.util.concurrent.Executors;
3
4  public class Main {
5      private static Double eval(final Genotype<DoubleGene> gt) {
6          // calculate and return fitness
7      }
8
9      void main() {
10         // Creating an fixed size ExecutorService
11         final ExecutorService executor = Executors
```

```
12              .newFixedThreadPool(10)
13          final Factory<Genotype<DoubleGene>> gtf = ...
14          final Engine<DoubleGene, Double> engine = Engine
15              .builder(Main::eval, gtf)
16               // Using 10 threads for evolving.
17              .executor(executor)
18              .build()
19          ...
20      }
21  }
```

If no `Executor` is given, **Jenetics** uses a common `ForkJoinPool`[15] for concurrency. Sometimes it might be useful to run the evaluation `Engine` single-threaded, or even execute all operations in the main thread. This can be easily achieved by setting the appropriate `Executor`.

```
1  final Engine<DoubleGene, Double> engine = Engine.builder(...)
2      // Doing the Engine operations in the main thread
3      .executor((Executor)Runnable::run)
4      .build()
```

The code snippet above shows how to do the `Engine` operations in the main thread. Whereas the snippet below executes the `Engine` operations in a single thread, other than the main thread.

```
1  final Engine<DoubleGene, Double> engine = Engine.builder(...)
2      // Doing the Engine operations in a single thread
3      .executor(Executors.newSingleThreadExecutor())
4      .build()
```

Such a configuration can be useful for performing reproducible (performance) tests, without the uncertainty of a concurrent execution environment.

1.4.1.2 Concurrency tweaks

Jenetics uses different strategies for minimizing the concurrency overhead, depending on the configured `Executor`. For the `ForkJoinPool`, the fitness evaluation of the population is done by recursively dividing it into subpopulations using the abstract `RecursiveAction` class. If a minimal subpopulation size is reached, the fitness values for this subpopulation are directly evaluated. The default value of this threshold is five and can be controlled via the `io.jenetics.concurrency.splitThreshold` system property. Besides the `splitThreshold`, the size of the evaluated subpopulation is dynamically determined by the `ForkJoinTask::getSurplusQueuedTaskCount` method.[16] If this value is greater than three, the fitness values of the current subpopulation are also evaluated immediately. The default value can be overridden by the `io.jenetics.concurrency.maxSurplusQueuedTaskCount` system property.

```
$ java -Dio.jenetics.concurrency.splitThreshold=1 \
```

[15] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ForkJoinPool.html`

[16] Excerpt from the Javadoc: *Returns an estimate of how many more locally queued tasks are held by the current worker thread than there are other worker threads that might steal them. This value may be useful for heuristic decisions about whether to fork other tasks. In many usages of `ForkJoinTasks`, at steady state, each worker should aim to maintain a small constant surplus (for example, 3) of tasks, and to process computations locally if this threshold is exceeded.*

```
-Dio.jenetics.concurrency.maxSurplusQueuedTaskCount=2 \
-cp jenetics-9.1.0.jar:app.jar \
com.foo.bar.MyJeneticsApp
```

You may want to tweak this parameters, if you realize a low CPU utilization during the fitness value evaluation. Long running fitness functions could lead to CPU under utilization while evaluating the last subpopulation. In this case, only one core is busy, while the other cores are idle, because they already finished the fitness evaluation. Since the workload has been already distributed, no *work stealing* is possible. Reducing the `splitThreshold` can help to have a more equal workload distribution between the available CPUs. Reducing the `maxSurplusQueuedTaskCount` property will create a more uniform workload for the fitness function with heavily varying computation cost for different genotype values.

The fitness function shouldn't acquire locks for achieving thread safety. It is also recommended to avoid calls to blocking methods. If such calls are unavoidable, consider using the `ForkJoinPool.managedBlock` method. Especially if you are using a `ForkJoinPool` executor, which is the default.

If the Engine is using an **`ExecutorService`**, a different optimization strategy is used for reducing the concurrency overhead. The original population is divided into a fixed number[17] of subpopulations, and the fitness values of each subpopulation are evaluated by one thread. For long running fitness functions, it is better to have smaller subpopulations for a better CPU utilization. With the `io.jenetics.concurrency.maxBatchSize` system property, it is possible to reduce the subpopulation size. The default value is set to `Integer.MAX_VALUE`. This means, that only the number of CPU cores influences the *batch* size.

```
$ java -Dio.jenetics.concurrency.maxBatchSize=3 \
    -cp jenetics-9.1.0.jar:app.jar \
    com.foo.bar.MyJeneticsApp
```

Another source of underutilized CPUs are lock contentions. It is therefore strongly recommended to avoid locking and blocking calls in your fitness function at all. If blocking calls are unavoidable, consider using the *managed block* functionality of the `ForkJoinPool`.[18]

1.4.1.3 BatchExecutor

The `BatchExecutor` defines the way the fitness functions of a given population are evaluated. It can be changed by setting the `Engine.Builder.fitnessExecutor` property.

```
1 @FunctionalInterface
2 public interface BatchExecutor {
```

[17] The number of sub-populations actually depends on the number of available CPU cores, which are determined with `Runtime.availableProcessors()`.

[18] A good introduction on how to use managed blocks, and the motivation behind it, is given in this talk:`https://www.youtube.com/watch?v=rUDGQQ83ZtI`

```
3      void execute(BaseSeq<? extends Runnable> batch);
4      static BatchExecutor of(Executor executor) {...}
5      static BatchExecutor ofVirtualThreads() {...}
6  }
```

Listing 1.14: `BatchExecutor` interface

A BatchExecutor can be implemented from scratch or created with one of the two factory methods:

1. **`BatchExecutor::of`**: This factory creates a new `BatchExecutor` which uses a given `executor` for evaluating the fitness function. This executor essentially partitions the given batch of `Runnable`s into *equally* sized sub-batches and evaluates these concurrently. By default, this BatchExecutor with the `ForkJoinPool::commonPool`, is used by **Jenetics** for evaluating the fitness functions.

2. **`BatchExecutor::ofVirtualThreads`**: This factory creates a `BatchExecutor` which evaluates every fitness function in its own *virtual* thread.

```
1  final Engine<DoubleGene, Double> engine = Engine.builder(...)
2      // Executes every fitness function in its own virtual thread.
3      .fitnessExecutor(BatchExecutor.ofVirtualThreads())
4      .build()
```

The code snippet above shows how to create an evolution `Engine`, which evaluates every fitness function in a virtual thread.

Using virtual threads for evaluating the fitness functions doesn't make the GA execution automatically faster. But it will help to utilize the available platform threads more efficiently, if your fitness function contains blocking calls. This is likely the case for remote calls, accessing a database or doing file IO.

1.4.2 Randomness

In general, GAs heavily depend on pseudo random number generators (PRNG) for creating new individuals and for the selection and mutation algorithms. **Jenetics** uses the Java `RandomGenerator` interface for generating random numbers. To make the random engine pluggable, the `RandomGenerator` object is always fetched from the `RandomRegistry`. This makes it possible to change the implementation of the random generator without changing the client code. The central `RandomRegistry` also allows for easily changing the `RandomGenerator` even for specific parts of the code.

The following example shows how to change and restore the `RandomGenerator` object. When opening the `with` scope, changes to the `RandomRegistry` are only visible within this scope. Once the `with` scope is left, the original `RandomGenerator` object is restored.

```
1  final var rgf = RandomGeneratorFactory.getDefault();
2  final List<Genotype<DoubleGene>> genotypes =
3      RandomRegistry.with(rgf.create(123), r -> {
```

```
4           Genotype.of(DoubleChromosome.of(0.0, 100.0, 10))
5               .instances()
6               .limit(100)
7               .toList()
8       });
```

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 evaluate the `EvolutionStream` several times with the same initial population.

```
1 final Engine<DoubleGene, Double> engine = ...;
2 // Create a new evolution stream with the given
3 // initial genotypes.
4 final Phenotype<DoubleGene, Double> best = engine.stream(genotypes)
5     .limit(10)
6     .collect(EvolutionResult.toBestPhenotype());
```

The example above uses the generated genotypes for creating the `EvolutionStream`. Each created stream uses the same starting population, but will, most likely, create a different result. This is because the stream evaluation is still nondeterministic.

Setting the PRNG to a `RandomGenerator` object with a defined seed has the effect, that every evolution *stream* will produce the same result—in a single threaded environment.

The parallel nature of the GA implementation requires the creation of streams of random numbers, $t_{i,j}$, which are statistically independent. These streams are numbered with $j = 1, 2, 3, ..., p$, and 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 or threads. 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 genetic algorithm (evolution stream) does not depend on the degree of parallelization.

When the `RandomGenerator` is used in a multi-threaded environment, there must be a way to parallelize the sequential PRNG. Usually this is done by taking the elements of the sequence of pseudo random numbers and distributing them among the threads. There are essentially four different parallelization techniques used in practice: Random seeding, Parameterization, Block splitting and Leapfrogging.

1.4.2.1 Concepts

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. Random

seeding works well for the most problems but without theoretical foundation.[19] The `RandomRegistry` is initialized with the Java `L64X256MixRandom` class.[20]

Parameterization All threads use 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 methods 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 non overlapping contiguous block of random numbers, which should be enough for the whole runtime of the process. If the number of threads is not known in advance, the length of each block should be chosen much larger than the maximal expected number of threads. This strategy is used when using the `LCG64ShiftRandom` 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.

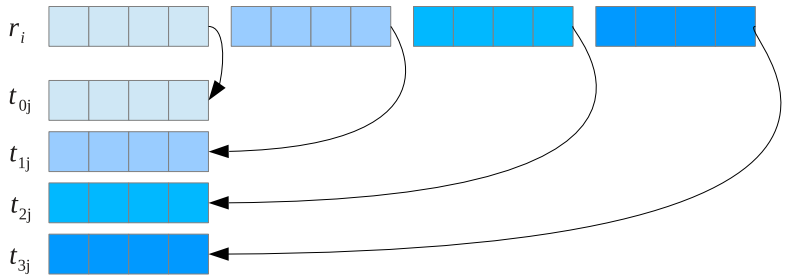


Figure 1.4.2: Block splitting

Leapfrog With the leapfrog method each thread $t \in [0, P)$ only consumes the P^{th} random number and jumps ahead in the random sequence by the number of threads, P. This method requires the ability to jump very quickly ahead in the sequence of random numbers by a given amount. Figure 1.4.3 graphically shows the concept of the leapfrog method.

`LCG64ShiftRandom`[21] The `LCG64ShiftRandom` class is a port of the `trng::lcg64_shift` PRNG class of the TRNG[22] library, implemented in C++.[7] It implements additional methods, which allows to implement the *block splitting*—and also the *leapfrog*—method.

```
1 public class LCG64ShiftRandom
2     implements RandomGenerator.ArbitrarilyJumpableGenerator
3 {
4     public void jump(final double step);
```

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

[20] `RandomRegistry.random(RandomGeneratorFactory.of("L64X256MixRandom"))`

[21] The `LCG64ShiftRandom` PRNG is part of the `io.jenetics.prngine` module (see section3.4 on page 128).

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

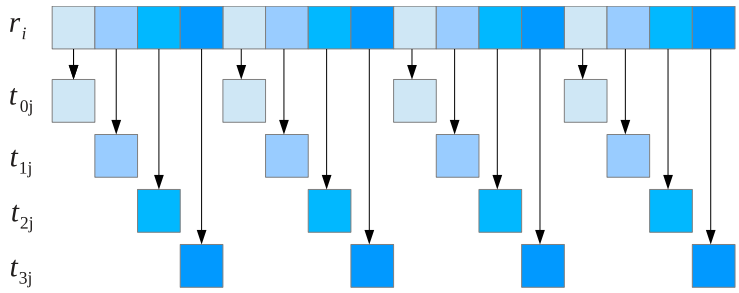


Figure 1.4.3: Leapfrogging

```
5     public void jumpPowerOfTwo(final int s);
6     ...
7 }
```

Listing 1.15: `LCG64ShiftRandom` class

Listing 1.15 shows the interface used for implementing the block splitting and leapfrog parallelization techniques. This methods have the following meaning:

`split` Changes the internal state of the PRNG in a way that future calls to `nextLong` will generate the s^{th} sub-stream of p^{th} 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*.

`jumpPowerOfTwo` Changes the internal state of the PRNG in such a way that the engine jumps 2^s steps ahead. This method is used for parallelization via *block splitting*.

1.4.2.2 Default generator

By default, **Jenetics** uses the `L64X256MixRandom` random generator, which is part of the standard OpenJDK platform. If you want to change this default for your whole application, without changing your code, you can do this via the `io.jenetics.util.defaultRandomGenerator` system property. The code example below shows how to replace the default random generator with `L64X1024MixRandom`[23].

```
$ java -Dio.jenetics.util.defaultRandomGenerator=\
         L64X1024MixRandom \
    -cp jenetics-9.1.0.jar:app.jar \
        com.foo.bar.MyJeneticsApp
```

Some vendors of JDK binaries are also offering »JRE« builds. Oracle doesn't specify such a thing like »JRE« and the vendors decides what to put into it and

[23] A list of all random generators defined in the OpenJDK platform, can be found here: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/random/package-summary.html`

what not. It seems that most of the JREs doesn't fully support the Java random API, as introduced in Java 17. They lack the `L64X256MixRandom` random generator, which prevents the start of applications which uses the **Jenetics** library. Since it is always possible to specify the default random generator at the command line, as described above, it is nevertheless a surprising behavior. In order to make the random generator selection more robust, the following *algorithm* is used when choosing the default random generator:

1. Check if the `io.jenetics.util.defaultRandomGenerator` Java parameter is set. If so, take this generator.
2. Check if the `L64X256MixRandom` generator is available. If so, take this generator.
3. Find the *best* available random generator according to the `RandomGeneratorFactory::stateBits method.`
4. Use the `Random` generator if no *best* generator can be found. This generator is guaranteed to be available on every platform.

The downside of this selection algorithm is, that the used default random generator is no longer the same on all Java platforms.

1.4.2.3 Random sampler

With the `RandomRegistry` it is possible to define the `RandomGenerator` which is used in various parts of the library. For some algorithms, this is not sufficient, they require random numbers which follow a specific distribution. The `Sample` interface in the `io.jenetics.stat` package allows the creation of `double` random numbers within a given range, which follows a specific distribution. Listing 1.16 shows functional `Sampler` interface.

```
@FunctionalInterface
public interface Sampler {
    double sample(RandomGenerator random, DoubleRange range);
}
```

Listing 1.16: `Sampler` interface

The `Sampler` itself is not responsible for the *randomness* of the created numbers. This is delegated to a `RandomGenerator`, which is the first parameter of the `sample` method. The `range` of the created samples is controlled by the second parameter. How to create a simple `Sampler`, which creates uniformly distributed `double` values, is shown in the code snippet below.

```
static final Sampler UNIFORM =
    (random, range) -> random.nextDouble(range.min(), range.max());
```

Two other `Sampler` implementations are currently available; one creates sample points which follows a *linear* distribution and the other one creates number, following a *triangular* distribution. They can be created with the `Samplers::linear` and `Samplers::triangular` factory methods, respectively.

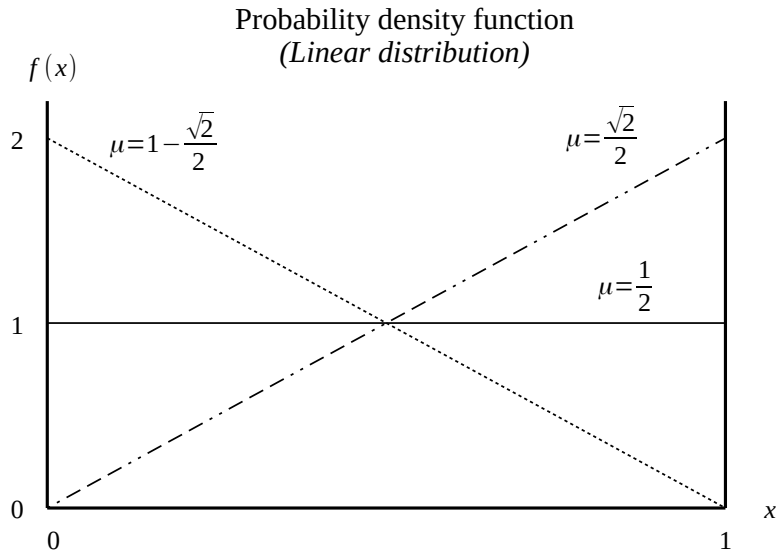


Figure 1.4.4: Linear distribution

Linear sampler Diagram 1.4.4 shows the probability density function (PDF) of the numbers, created by the *linear* sampler.

The static factory method, `Samplers.linear(double mean)`, which creates the `Sampler`, takes the mean value, μ, as parameter. This parameter defines the mean value of the created sample points and must be within the range $[0, 1]$. Only points created within the range $[0, 1)$ will have the defined mean value.

Triangular sampler Diagram 1.4.5 on the following page shows the PDF of the sample points, created by the triangular sampler.

The factory method for this `Sampler`, `Samplers.triangular(double a, double c, double b)`, takes the parameters a, b and c. These parameters, also shown in the diagram, must within the range $[0, 1)$.

1.4.3 Serialization

Jenetics supports serialization for a number of classes, most of them are located in the `io.jenetics` package. Only the concrete implementations of the `Gene` and the `Chromosome` interfaces implements the `Serializable` interface. This gives a greater flexibility when implementing own `Gene`s and `Chromosome`s.

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

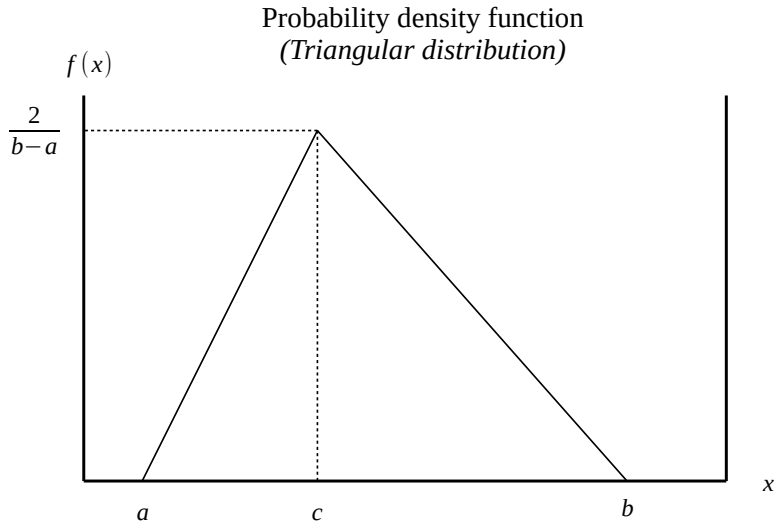


Figure 1.4.5: Triangular distribution

- `Genotype`
- `Phenotype`

With the serialization mechanism you can write a population to disk and load it into a new `EvolutionStream` at a later time. It can also be used to transfer populations to evolution engines, running on different hosts, over a network link. The `IO` class, located in the `io.jenetics.util` package, supports native Java serialization in a convenient way.

```
// Creating result population.
final EvolutionResult<DoubleGene, Double> result = stream
    .limit(100)
    .collect(toBestEvolutionResult());

// Writing the population to disk.
final File file = new File("population.obj");
IO.object.write(result.population(), file);

// Reading the population from disk.
final ISeq<Phenotype<G, C>> population =
    (ISeq<Phenotype<G, C>>)IO.object.read(file);
final EvolutionStream<DoubleGene, Double> stream = Engine
    .build(ff, gtf)
    .stream(population, 1);
```

1.4.4 Utility classes

The `io.jenetics.util` and the `io.jenetics.stat` package of the library contains utility and helper classes which are essential for the implementation of the GA.

`io.jenetics.util.BaseSeq` This interface defines a minimal contract for sequential data, which can be accessed by its index or position. The algorithms, implemented by the **Jenetics** library, assumes that accessing elements of a

`BaseSeq` is done in $O(1)$. `Chromosome` and `Genotype` implement the `BaseSeq` interface. This expresses the intent that the `Chromosome` is a sequence of `Gene`s and the `Genotype` is a sequence of `Chromosome`s.

`io.jenetics.util.Seq` Most notable are the `Seq` interfaces and its implementation. They are used, among others, in the `Chromosome` and `Genotype` classes and hold the `Gene`s and `Chromosome`s, 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.

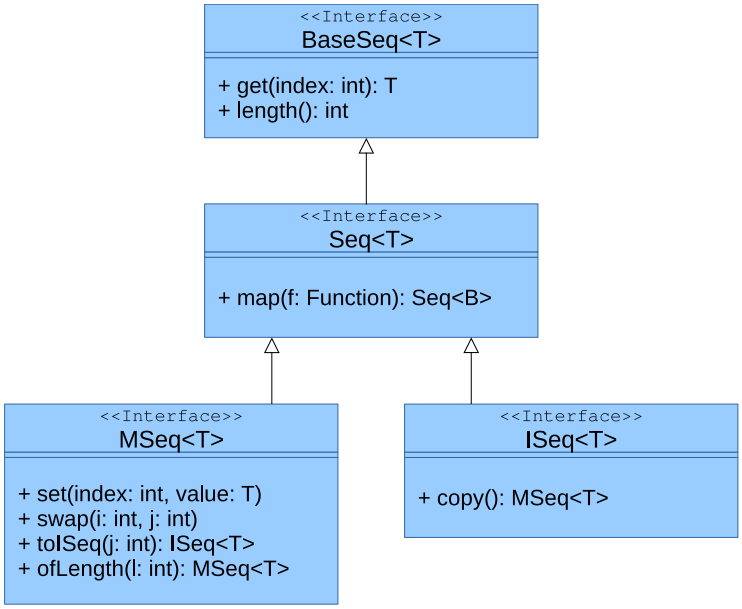


Figure 1.4.6: `Seq` class diagram

Figure 1.4.6 shows the `Seq` class diagram with their most important methods. The interfaces `MSeq` and `ISeq` are mutable, respectively immutable specializations of the basis interface. Creating instances of the `Seq` interfaces is possible via the static factory methods of the interfaces.

```
1  // Create "different" sequences.
2  final Seq<Integer> a1 = Seq.of(1, 2, 3);
3  final MSeq<Integer> a2 = MSeq.of(1, 2, 3);
4  final ISeq<Integer> a3 = MSeq.of(1, 2, 3).toISeq();
5  final MSeq<Integer> a4 = a3.copy();
6
7  // The 'equals' method performs element-wise comparison.
8  assert(a1.equals(a2) && a1 != a2);
9  assert(a2.equals(a3) && a2 != a3);
10 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.

`io.jenetics.util.ProxySorter` This is a special sorter, which allows you to sort even an immutable collection. As the name suggests, it doesn't sort a given sequence directly. Instead it sorts or rearranges a *proxy* `int[]` array, which can then be used for accessing the original sequence in a sorted order. The main usage for this special sorter in **Jenetics** is where you need to access of the population in sorted order, but have to preserve the original order of the population.[24] Many of the algorithms, implemented in the `io.jenetics.ext.moea` package, uses the `ProxySorter`, which leads to simpler code at this places. How the proxy sorter is used can be seen in the following code snippet.

```
final double[] array = RandomGenerator.getDefault()
    .doubles(100)
    .toArray();
final int[] proxy = ProxySorter.sort(array);

// Doing 'Classical' array sort.
final double[] sorted = array.clone();
Arrays.sort(sorted);

// Iterating the array in ascending order.
for (int i = 0; i < array.length; ++i) {
    assert sorted[i] == array[proxy[i]];
}
```

The `ProxySorter` increases the set of sortable objects. It is even possible to sort objects where you only know the access function to the elements and the number of elements.

```
final IntFunction<String> access = ...;
final int length = 100;
final int[] proxy = ProxySorter.sort(
    access, length,
    (a, i, j) -> a.apply(i).compareTo(a.apply(j))
);
```

The code snipped above shows how to sort an `IntFunction`. With the proxy array you are now able to access the `access` function in ascending order. The `ProxySorter` uses the Timsort[25] algorithm for sorting the proxy `int[]` array.

`io.jenetics.stat` This package contains classes for calculating statistical moments. They are designed to work smoothly with the Java Stream API and are divided into mutable (number) consumers and immutable value classes, which holds the statistical moments. The additional classes calculate the

- *minimum*,
- *maximum*,
- *sum*,
- *mean*,
- *variance*,
- *skewness* and
- *kurtosis* value.

Table 1.4.1 contains the available statistical moments for the different numeric types. The following code snippet shows an example on how to collect double statistics from a given `DoubleGene` stream.

[24] For this specific problem you could also do this by copying the population and sorting the copy instead of the original. But using a sorted*proxy* array can lead to simpler code.

[25] `https://en.wikipedia.org/wiki/Timsort`

Numeric type	Consumer class	Value class
`int`	`IntMomentStatistics`	`IntMoments`
`long`	`LongMomentStatistics`	`LongMoments`
`double`	`DoubleMomentStatistics`	`DoubleMoments`

Table 1.4.1: Statistics classes

```
// Collecting into an statistics object.
final DoubleChromosome chromosome = ...
final DoubleMomentStatistics statistics = chromosome.stream()
    .collect(DoubleMomentStatistics
        .toDoubleMomentStatistics(v -> v.doubleValue()));

// Collecting into an moments object.
final DoubleMoments moments = chromosome.stream()
    .collect(DoubleMoments.toDoubleMoments(v -> v.doubleValue()));
```

The `stat` package also contains a class for calculating the quantile[26] of a stream of double values. Its implementing algorithm, which is described in [21], calculates—or *estimates*—the quantile value on the fly, without storing the consumed double values. This allows for using the `Quantile` class even for very large sets of double values. How to calculate the first quartile of a given, random `DoubleStream` is shown in the code snippet below.

```
final Quantile quartile = new Quantile(0.25);
RandomGenerator.getDefault()
    .doubles(10_000)
    .forEach(quartile);
final double value = quartile.value();
```

Be aware, that the calculated quartile is *just* an estimation. For sufficient accuracy, the stream size should be sufficiently large ($size \gg 100$).

[26] `https://en.wikipedia.org/wiki/Quantile`

Chapter 2

Advanced topics

This section describes some advanced topics for setting up an evolution `Engine` or `EvolutionStream`. It contains some problem encoding examples and how to override the default validation strategy of the given `Genotype`s. The last section contains a detailed description of the implemented termination strategies.

2.1 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 a GA. It also comes with different implementations for the base data types (genes and chromosomes) and 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.

2.1.1 Genes

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

For implementing your own gene type you have to implement the `Gene` interface with three methods: (1) the `Gene::allele` method which will return the wrapped data, (2) the `Gene::newInstance` method for creating new, random instances of the gene—must be of the same type and have the same constraint—and (3) the `Gene::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 to extend it from the `NumericGene` interface.

The custom `Genes` and `Chromosomes` implementations must use the `RandomGenerator` available via the `RandomRegistry::random` method when implementing their factory methods. Otherwise it is not possible to seamlessly change the `RandomGenerator` by using the `RandomRegistry::random(RandomGenerator)` method.

If you want to support your own allele type, but want to avoid the effort of implementing the `Gene` interface, you can alternatively use the `AnyGene` class. It can be created with `AnyGene::of(Supplier, Predicate)`. The given `Supplier` is responsible for creating new random alleles, similar to the `newInstance` method in the `Gene` interface. Additional validity checks are performed by the given `Predicate`.

```
1  class LastMonday {
2      // Creates new random 'LocalDate' objects.
3      private static LocalDate nextMonday() {
4          final var random = RandomRegistry.random();
5          LocalDate
6              .of(2015, 1, 5)
7              .plusWeeks(random.nextInt(1000));
8      }
9
10     // Do some additional validity check.
11     private static boolean isValid(final LocalDate date) {...}
12
13     // Create a new gene from the random 'Supplier' and
14     // validation 'Predicate'.
15     private final AnyGene<LocalDate> gene = AnyGene
16         .of(LastMonday::nextMonday, LastMonday::isValid);
17 }
```

Listing 2.1: `AnyGene` example

Example listing 2.1 shows the (almost) minimal setup for creating user defined `Gene` allele types. By convention, the `RandomGenerator`, used for creating the new `LocalDate` objects, must be requested from the `RandomRegistry`. With the optional validation function, `isValid`, it is possible to reject `Genes` whose alleles don't conform to some criteria. The simple usage of the `AnyGene` has also its downsides. Since the `AnyGene` instances are created from function objects, serialization is not supported by the `AnyGene` class. It is also not possible to use some `Alterer` implementations with the `AnyGene`, like:

- `GaussianMutator`,
- `MeanAlterer` and
- `PartiallyMatchedCrossover`

2.1.2 Chromosomes

A new `Gene` type usually comes with a corresponding `Chromosome` implementation. One of the important parts of a `Chromosome` is the factory method `newInstance(ISeq)`, which lets the evolution `Engine` create a new `Chromosome` instance from a sequence of `Gene`s. This method is used by the `Alterer` when

creating a new combined `Chromosome`. The newly created `Chromosome` may have a different length than the original one. The other methods should be self explanatory. The `Chromosome` implementations uses the same serialization mechanism as the `Gene`. In the minimal case it can extends the `Serializable` interface.[1]

Just implementing the `Serializable` interface is sometimes not enough. You might also need to implement the `readObject` and `writeObject` methods for a more concise serialization result. Consider using the serialization proxy pattern, item 90, described in *Effective Java*[9].

Corresponding to the `AnyGene`, it is possible to create chromosomes with arbitrary allele types with the `AnyChromosome`.

```
1  public class LastMonday {
2      // The used problem Codec.
3      private static final Codec<LocalDate, AnyGene<LocalDate>>
4      CODEC = Codec.of(
5          Genotype.of(AnyChromosome.of(LastMonday::nextMonday)),
6          gt -> gt.gene().allele()
7      );
8
9      // Creates new random 'LocalDate' objects.
10     private static LocalDate nextMonday() {
11         final var random = RandomRegistry.random();
12         LocalDate
13             .of(2015, 1, 5)
14             .plusWeeks(random.nextInt(1000));
15     }
16
17     // The fitness function: find a monday at the end of the month.
18     private static int fitness(final LocalDate date) {
19         return date.getDayOfMonth();
20     }
21
22     void main() {
23         final Engine<AnyGene<LocalDate>, Integer> engine = Engine
24             .builder(LastMonday::fitness, CODEC)
25             .offspringSelector(new RouletteWheelSelector<>())
26             .build();
27
28         final Phenotype<AnyGene<LocalDate>, Integer> best =
29             engine.stream()
30                 .limit(50)
31                 .collect(EvolutionResult.toBestPhenotype());
32
33         IO.println(best);
34     }
35 }
```

Listing 2.2: `AnyChromosome` example

Listing 2.2 shows a full usage example of the `AnyGene` and `AnyChromosome`. The example tries to find a Monday with a maximal day of month. An interesting detail is, that an `Codec`[2] definition is used for creating new `Genotypes` and

[1] `http://www.oracle.com/technetwork/articles/java/javaserial-1536170.html`

[2] See section2.3 on page 60 for a more detailed `Codec` description.

for converting them back to `LocalDate` alleles. The convenient usage of the `AnyChromosome` has to be payed by the same restriction as for the `AnyGene`: no serialization support for the chromosome and not usable for all `Alterer` implementations.

2.1.3 Selectors

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

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

Listing 2.3: `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 extending 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 $P(i)$. Another way for reusing existing `Selector` implementation is by composition.

```
public class EliteSelector<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Selector<G, C>
{
    private final TruncationSelector<G, C>
    _elite = new TruncationSelector<>();

    private final TournamentSelector<G, C>
    _rest = new TournamentSelector<>(3);

    public EliteSelector() {
    }

    @Override
    public ISeq<Phenotype<G, C>> select(
        final Seq<Phenotype<G, C>> population,
        final int count,
        final Optimize opt
    ) {
        ISeq<Phenotype<G, C>> result;
```

```
23          if (population.isEmpty() || count <= 0) {
24              result = ISeq.empty();
25          } else {
26              final int ec = min(count, _eliteCount);
27              result = _elite.select(population, ec, opt);
28              result = result.append(
29                  _rest.select(population, max(0, count - ec), opt)
30              );
31          }
32          return result;
33      }
34  }
```

Listing 2.4: Elite selector

Listing 2.4 shows how an *elite* selector could be implemented by using the existing `Truncation`- and `TournamentSelector`. With *elite* selection, the quality of the best solution in each generation monotonically increases over time.[6] It is not necessary to use an elite selector if you want to preserve the best individual in the final result. The evolution `Engine`/`Stream` doesn't throw away the best solution found during the evolution process.

2.1.4 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.

```
1  @FunctionalInterface
2  public interface Alterer<
3      G extends Gene<?, G>,
4      C extends Comparable<? super C>
5  > {
6      AltererResult<G, C> alter(
7          Seq<Phenotype<G, C>> population,
8          long generation
9      );
10 }
```

Listing 2.5: `Alterer` interface

The first parameter of the `alter` method is the `population` which has to be altered. The second parameter is the `generation` of the newly created individuals and the return value is the number of genes that has been altered and the altered population, aggregated in the `AltererResult` class.

To maximize the range of application of an `Alterer`, it is recommended that they can handle `Genotypes` and `Chromosomes` with variable length.

2.1.5 Statistics

During the developing phase of an application which uses the **Jenetics** library, additional statistical data about the evolution process is crucial. Such data can help to optimize the parametrization of the evolution `Engine`. A good

starting point is to use the `EvolutionStatistics` class in the `io.jenetics.engine` package (see listing 1.11). If the data in the `EvolutionStatistics` class doesn't fit your needs, you simply have to write your own statistics class. It is not possible to derive from the existing `EvolutionStatistics` class. This is not a real restriction, since you still can use the class by delegation. Just implement the Java `Consumer<EvolutionResult<G, C>>` interface.

2.1.6 Engine

The evolution `Engine` itself can't be extended, but it is still possible to create an `EvolutionStream` without using the `Engine` class.[3] Because the `EvolutionStream` has no direct dependency to the `Engine`, it is possible to use an different, special evolution `Function`.

```
1  public final class SpecialEngine {
2      // The Genotype factory.
3      private static final Factory<Genotype<DoubleGene>> GTF =
4          Genotype.of(DoubleChromosome.of(0, 1));
5
6      // Create new evolution start object.
7      private static EvolutionStart<DoubleGene, Double>
8      start(final int populationSize, final long generation) {
9          final ISeq<Phenotype<DoubleGene, Double>> population = GTF
10             .instances()
11             .map(gt -> Phenotype.of(gt, generation))
12             .limit(populationSize)
13             .collect(ISeq.toISeq());
14
15         return EvolutionStart.of(population, generation);
16     }
17
18     // The special evolution function.
19     private static EvolutionResult<DoubleGene, Double>
20     evolve(final EvolutionStart<DoubleGene, Double> start) {
21         return ...; // Add implementation!
22     }
23
24     void main() {
25         final Genotype<DoubleGene> best = EvolutionStream
26             .of(() -> start(50, 0), SpecialEngine::evolve)
27             .limit(Limits.bySteadyFitness(10))
28             .limit(100)
29             .collect(EvolutionResult.toBestGenotype());
30
31         IO.println("Best Genotype: " + best);
32     }
33 }
```

Listing 2.6: Special evolution engine

Listing 2.6 shows an implementation stub for using an own special evolution `Function`. It is also possible to change the used evolution function, depending on the actual population. The `EvolutionStream::ofAdjustableEvolution` give you this possibility. In the following example two evolution functions are used, depending on the fitness variance of the previous population.

[3] Also refer to section1.3.3.4 on page 28 on how to create an `EvolutionStream` from an evolution `Function`.

```java
void main() {
    final Problem<double[], DoubleGene, Double> problem = ...;

    // Engine.Builder template.
    final Engine.Builder<DoubleGene, Double> bld = Engine
        .builder(problem)
        .minimizing();

    // Evolution used for low fitness variance.
    final Evolution<DoubleGene, Double> lowVar = builder.copy()
        .alterers(new Mutator<>(0.5))
        .selector(new MonteCarloSelector<>())
        .build();

    // Evolution used for high fitness variance.
    final Evolution<DoubleGene, Double> highVar = builder.copy()
        .alterers(
            new Mutator<>(0.05),
            new MeanAlterer<>())
        .selector(new RouletteWheelSelector<>())
        .build();

    final EvolutionStream<DoubleGene, Double> stream =
        EvolutionStream.ofAdjustableEvolution(
            EvolutionStart::empty,
            er -> var(er) < 0.2 ? lowVar : highVar
        );

    final Genotype<DoubleGene> result = stream
        .limit(Limits.bySteadyFitness(50))
        .collect(EvolutionResult.toBestGenotype());
}

static double var(final EvolutionResult<DoubleGene, Double> er) {
    return er != null
        ? er.population().stream()
            .map(Phenotype::fitness)
            .collect(toDoubleMoments())
            .variance()
        : 0.0;
}
```

Listing 2.7: Adjustable evolution stream

The purpose of such an adjustment is to broaden the search if the population variance tends to be to small. This can reduce the risk of converging to a local minimum. If the population variance is to big, a different engine configuration can help to speed up the optimization.

2.2 Encoding

This section presents some encoding examples for common optimization problems. The encoding should be a complete, and minimal representation of the problem domain. An encoding is complete if it contains enough information to represent every solution to the problem. Whereas a minimal encoding contains only the information needed to represent a solution to the problem. If an encoding contains more information than is needed to uniquely identify solutions to the problem, the search space will be larger than necessary. In the best case, there is a one-to-one mapping from the `Genotype` space to problem domain. Whenever

possible, the encoding should not represent infeasible solutions. If a `Genotype` represents an infeasible solution, care must be taken in the fitness function to give partial credit to the `Genotype` for its »good« genetic material while sufficiently penalizing it for being infeasible. Implementing a specialized `Chromosome`, which won't create invalid encodings can be a solution to this problem. In general, it is much more desirable to design a representation that can only represent valid solutions so that the fitness function measures only fitness, not validity. An encoding that includes invalid individuals enlarges the search space and makes the search more costly. A deeper analysis of how to create encodings can be found in [37] and [36]. Some of the encodings represented in the following sections have been implemented by **Jenetics**, using the `Codec`[4] interface, and are available through static factory methods of the `io.jenetics.engine.Codecs` class.

2.2.1 Real function

Jenetics contains three different numeric `Gene` and `Chromosome` implementations, which can be used to encode a real function, $f : \mathbb{R} \to \mathbb{R}$:

- `IntegerGene`/`Chromosome`,
- `LongGene`/`Chromosome` and
- `DoubleGene`/`Chromosome`.

It is quite easy to encode a real function. Only the minimum and maximum value of the function domain must be defined. The `DoubleChromosome` of length 1 is then wrapped into a `Genotype`.

```
Genotype.of(
    DoubleChromosome.of(min, max, 1)
);
```

Decoding the double value from the `Genotype` is also straight forward. Just get the first `Gene` from the first `Chromosome`, with the `gene` method, and convert it to a `double`.

```
static double toDouble(final Genotype<DoubleGene> gt) {
    return gt.gene().doubleValue();
}
```

When the `Genotype` only contains *scalar* `Chromosomes`[5], it should be clear, that it can't be altered by every `Alterer`. That means, that none of the `Crossover` alterers will be able to create modified `Genotypes`. For *scalars* the appropriate alterers would be the `MeanAlterer`, `GaussianAlterer` and `Mutator`.

Scalar `Chromosomes` and/or `Genotypes` can only be altered by `MeanAlterer`, `GaussianAlterer` and `Mutator` classes. Other alterers are allowed, but will have no effect on the `Chromosomes`.

---

[4] See section2.3 on page 60.
[5] Scalar chromosomes contains only one gene.

2.2.2 Scalar function

Optimizing a function, $f(x_1, ..., x_n)$, of one or more variable whose range is one-dimensional, we have two possibilities for the `Genotype` encoding.[43] For the *first* encoding we expect that all variables, x_i, have the same minimum and maximum value. In this case we can simply create a `Genotype` with a *`Numeric`*`Chromosome` of the desired length n.

```
Genotype.of(
    DoubleChromosome.of(min, max, n)
);
```

The decoding of the `Genotype` requires a cast of the first `Chromosome` to a `DoubleChromosome`. With a call to the `DoubleChromosome.toArray()` method we return the variables $(x_1, ..., x_n)$ as `double[]` array.

```
static double[] toScalars(final Genotype<DoubleGene> gt) {
    return gt.chromosome()
        .as(DoubleChromosome.class)
        .toArray();
}
```

With the *first* encoding you have the possibility to use all available alterers, including all `Crossover` alterer classes.

The *second* encoding *must* be used if the minimum and maximum value of the variables x_i can't be the same for all i. For the different domains, each variable, x_i, is represented by a *`Numeric`*`Chromosome` with length one. The final `Genotype` will consist of n `Chromosome`s with length one.

```
Genotype.of(
    DoubleChromosome.of(min1, max1),
    DoubleChromosome.of(min2, max2),
    ...
    DoubleChromosome.of(minn, maxn)
);
```

With the help of the Java Stream API, the decoding of the `Genotype` can be done in a view lines. The `DoubleChromosome` stream, which is created from the `Chromosome` `Seq`, is first mapped to `double` values and then collected into an array.

```
static double[] toScalars(final Genotype<DoubleGene> gt) {
    return gt.stream()
        .mapToDouble(c -> c.gene().doubleValue())
        .toArray();
}
```

As already mentioned, with the use of scalar `Chromosome`s we can only use the `MeanAlterer`, `GaussianAlterer` or `Mutator` alterer class. If there are performance issues in converting the `Genotype` into a `double[]` array, or any other numeric array, you can access the `Gene`s directly via the `Genotype-.get(i).get(j)` method and than convert it to the desired numeric value, by calling `intValue()`, `longValue()` or `doubleValue()`.

2.2.3 Vector function

A function, $f(X_1, ..., X_n)$, of one to n variables whose range is m-dimensional, is encoded by m `DoubleChromosome`s of length n.[44] The domain–minimum and maximum values–of one variable X_i are the same in this encoding.

```
Genotype.of(
    DoubleChromosome.of(min1, max1, m),
    DoubleChromosome.of(min2, max2, m),
    ...
    DoubleChromosome.of(minn, maxn, m)
);
```

The decoding of the vectors is quite easy with the help of the Java Stream API. In the first `map` we have to cast the `Chromosome<DoubleGene>` object to the actual `DoubleChromosome`. The second `map` then converts each `DoubleChromosome` to a `double[]` array, which is collected to an 2-dimensional `double[n][m]` array afterwards.

```
static double[][] toVectors(final Genotype<DoubleGene> gt) {
    return gt.stream()
        .map(dc -> dc.as(DoubleChromosome.class).toArray())
        .toArray(double[][]::new);
}
```

For the special case of $n = 1$, the decoding of the `Genotype` can be simplified to the decoding we introduced for scalar functions in section 2.2.2.

```
static double[] toVector(final Genotype<DoubleGene> gt) {
    return gt.chromosome().as(DoubleChromosome.class).toArray();
}
```

2.2.4 Affine transformation

An affine transformation[6][7] is usually performed by a matrix multiplication with a transformation matrix—in a homogeneous coordinates system[8]. For a transformation in $\mathbb{R}^2$, we can define the matrix$A$[9]:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.2.1}$$

A simple representation can be done by creating a `Genotype` which contains two `DoubleChromosomes` with a length of 3.

```
Genotype.of(
    DoubleChromosome.of(min, max, 3),
    DoubleChromosome.of(min, max, 3)
);
```

The drawback with this kind of encoding is, that we will create a lot of *invalid* (non-affine transformation matrices) during the evolution process, which must be detected and discarded. It is also difficult to find the right parameters for the *min* and *max* values of the `DoubleChromosomes`. A better approach will be to encode the transformation parameters instead of the transformation matrix. The affine transformation can be expressed by the following parameters:

- s_x – the scale factor in x direction
- s_y – the scale factor in y direction

[6] https://en.wikipedia.org/wiki/Affine_transformation
[7] http://mathworld.wolfram.com/AffineTransformation.html
[8] https://en.wikipedia.org/wiki/Homogeneous_coordinates
[9] https://en.wikipedia.org/wiki/Transformation_matrix

- t_x – the offset in x direction
- t_y – the offset in y direction
- θ – the rotation angle clockwise around origin
- k_x – shearing parallel to x axis
- k_y – shearing parallel to y axis

This parameters can then be represented by the following `Genotype`.

```
Genotype.of(
    // Scale
    DoubleChromosome.of(sxMin, sxMax),
    DoubleChromosome.of(syMin, syMax),
    // Translation
    DoubleChromosome.of(txMin, txMax),
    DoubleChromosome.of(tyMin, tyMax),
    // Rotation
    DoubleChromosome.of(thMin, thMax),
    // Shear
    DoubleChromosome.of(kxMin, kxMax),
    DoubleChromosome.of(kyMin, kxMax)
)
```

This encoding ensures that no invalid `Genotype` will be created during the evolution process, since the crossover will be only performed on the same kind of chromosome (same chromosome index). To convert the `Genotype` back to the transformation matrix A, the following equations can be used [20]:

$$\begin{aligned} a_{11} &= s_x \cos\theta + k_x s_y \sin\theta \\ a_{12} &= s_y k_x \cos\theta - s_x \sin\theta \\ a_{13} &= t_x \\ a_{21} &= k_y s_x \cos\theta + s_y \sin\theta \\ a_{22} &= s_y \cos\theta - s_x k_y \sin\theta \\ a_{23} &= t_y \end{aligned} \tag{2.2.2}$$

This corresponds to an transformation order of $T \cdot S_h \cdot S_c \cdot R$:

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & k_x & 0 \\ k_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

In Java code, the conversion from the `Genotype` to the transformation matrix, will look like this:

```
static double[][] toMatrix(final Genotype<DoubleGene> gt) {
    final double sx = gt.get(0).gene().doubleValue();
    final double sy = gt.get(1).gene().doubleValue();
    final double tx = gt.get(2).gene().doubleValue();
    final double ty = gt.get(3).gene().doubleValue();
    final double th = gt.get(4).gene().doubleValue();
    final double kx = gt.get(5).gene().doubleValue();
    final double ky = gt.get(6).gene().doubleValue();

    final double cos_th = cos(th);
    final double sin_th = sin(th);
```

```
12      final double a11 = cos_th*sx + kx*sy*sin_th;
13      final double a12 = cos_th*kx*sy - sx*sin_th;
14      final double a21 = cos_th*ky*sx + sy*sin_th;
15      final double a22 = cos_th*sy - ky*sx*sin_th;
16
17      return new double[][] {
18          {a11, a12, tx},
19          {a21, a22, ty},
20          {0.0, 0.0, 1.0}
21      };
22  }
```

For the introduced encoding all kind of alterers can be used. Since we have one scalar `DoubleChromosome`, the rotation angle θ, it is recommended also to add a `MeanAlterer` or `GaussianAlterer` to the list of alterers.

2.2.5 Graph

A graph can be represented in many different ways. The most known graph representation is the adjacency matrix. The following encoding examples uses adjacency matrices with different characteristics.

Undirected graph In an undirected graph the edges between the vertices have no direction. If there is a path between nodes i and j, it is assumed that there is also path from j to i.

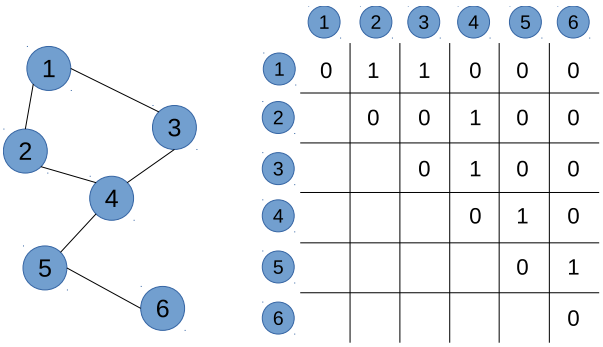


Figure 2.2.1: Undirected graph and adjacency matrix

Figure 2.2.1 shows an undirected graph and its corresponding matrix representation. Since the edges between the nodes have no direction, the values of the lower diagonal matrix are not taken into account. An application which optimizes an undirected graph has to ignore this part of the matrix.[10]

```
1  final int n = 6;
2  final Genotype<BitGene> gt = Genotype.of(BitChromosome.of(n), n);
```

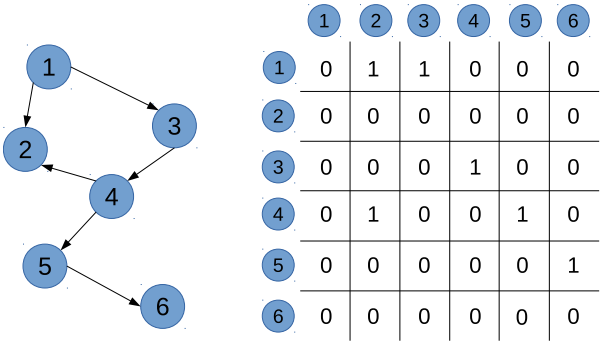
The code snippet above shows how to create an adjacency matrix for a graph with $n = 6$ nodes. It creates a `Genotype` which consists of n `BitChromosomes` of length n each. Whether the node i is connected to node j can be easily checked

[10]This property violates the *minimal* encoding requirement we mentioned at the beginning of section2.2 on page 53. For simplicity reason this will be ignored for the undirected graph encoding.

by calling `gt.get(i-1).get(j-1).booleanValue()`. For extracting the whole matrix as `int[]` array, the following code can be used.

```
final int[][] array = gt.toSeq().stream()
    .map(c -> c.toSeq().stream()
        .mapToInt(gene -> gene.bit() ? 1 : 0)
        .toArray())
    .toArray(int[][]::new);
```

Directed graph A directed graph (digraph) is a graph where the path between the nodes has a direction associated with them. The encoding of a directed graph looks exactly like the encoding of an undirected graph. This time the whole matrix is used and the second diagonal matrix is no longer ignored.

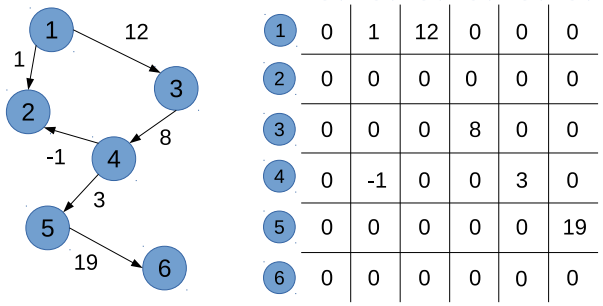


	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	0	0	0
3	0	0	0	1	0	0
4	0	1	0	0	1	0
5	0	0	0	0	0	1
6	0	0	0	0	0	0

Figure 2.2.2: Directed graph and adjacency matrix

Figure 2.2.2 shows the adjacency matrix of a digraph. This time the whole matrix is used for representing the graph.

Weighted directed graph A weighted graph associates a weight (label) with every path in the graph. Weights are usually real numbers. They may be restricted to rational numbers or integers.



	1	2	3	4	5	6
1	0	1	12	0	0	0
2	0	0	0	0	0	0
3	0	0	0	8	0	0
4	0	-1	0	0	3	0
5	0	0	0	0	0	19
6	0	0	0	0	0	0

Figure 2.2.3: Weighted graph and adjacency matrix

The following code snippet shows how the `Genotype` of the matrix is created.

```
final int n = 6;
final double min = -1;
final double max = 20;
final Genotype<DoubleGene> gt = Genotype
    .of(DoubleChromosome.of(min, max, n), n);
```

For accessing the single matrix elements, you can simply call `Genotype.get(i).get(j).doubleValue()`. If the interaction with another library requires a `double[][]` array, the following code can be used.

```
final double[][] array = gt.stream()
    .map(dc -> dc.as(DoubleChromosome.class).toArray())
    .toArray(double[][]::new);
```

2.3 Codec

The `Codec` interface, located in the `io.jenetics.engine` package, narrows the gap between the fitness `Function`, which should be maximized/minimized, and the `Genotype` representation, which can be understood by the evolution `Engine`. With the `Codec` interface it is possible to implement the encodings of section 2.2 in a more formalized way.

Normally, the `Engine` expects a fitness function which takes a `Genotype` as input. This `Genotype` has then to be transformed into an object of the problem domain. The usage `Codec` interface allows a tighter coupling of the `Genotype` definition and the transformation code.[11]

```
public interface Codec<T, G extends Gene<?, G>> {
    Factory<Genotype<G>> encoding();
    Function<Genotype<G>, T> decoder();
    default T decode(final Genotype<G> gt) {...}
}
```

Listing 2.8: `Codec` interface

Listing 2.8 shows the `Codec` interface. The `encoding` method returns the `Genotype` factory, which is used by the `Engine` for creating new `Genotypes`. The decoder `Function`, which is returned by the `decoder` method, transforms the `Genotype` to the argument type of the fitness `Function`. Without the `Codec` interface, the implementation of the fitness Function is *polluted* with code, which transforms the `Genotype` into the argument type of the actual fitness `Function`.

```
static double eval(final Genotype<DoubleGene> gt) {
    final double x = gt.gene().doubleValue();
    // Do some calculation with 'x'.
    return ...
}
```

The `Codec` for the example above is quite simple and is shown below. It is not necessary to implement the `Codec` interface, instead you can use the `Codec::of` factory method for creating new `Codec` instances.

```
final DoubleRange domain = new DoubleRange(0, 2*PI);
final Codec<Double, DoubleGene> codec = Codec.of(
    Genotype.of(DoubleChromosome.of(domain)),
    gt -> gt.chromosome().gene().allele()
);
```

[11] Section2.2 on page 53 describes some possible encodings for common optimization problems.

When using a `Codec` instance, the fitness `Function` solely contains code from your actual problem domain—no dependencies to classes of the **Jenetics** library.

```
static double eval(final double x) {
    // Do some calculation with 'x'.
    return ...
}
```

Jenetics comes with a set of standard encodings, which are created via static factory methods in the `io.jenetics.engine.Codecs` class. The following subsections describe the most important predefined `Codecs`.

2.3.1 Scalar codec

Listing 2.9 shows the implementation of the `Codecs::ofScalar` factory method—for `Integer` scalars.

```
static Codec<Integer, IntegerGene> ofScalar(IntRange domain) {
    return Codec.of(
        Genotype.of(IntegerChromosome.of(domain)),
        gt -> gt.chromosome().gene().allele()
    );
}
```

Listing 2.9: `Codec` factory method: `ofScalar`

The usage of the `Codec`, created by this factory method, simplifies the implementation of the fitness `Function` and the creation of the evolution `Engine`. For scalar types, the saving, in complexity and lines of code, is not that big, but using the factory method is still quite handy. The following listing demonstrates the interaction between `Codec`, fitness `Function` and evolution `Engine`.

```
class Main {
    // Fitness function directly takes an 'int' value.
    static double fitness(int arg) {
        return ...;
    }
    void main() {
        final Engine<IntegerGene, Double> engine = Engine
            .builder(Main::fitness, ofScalar(new IntRange(0, 100)))
            .build();
        ...
    }
}
```

2.3.2 Vector codec

In listing 2.10, the `ofVector` factory method returns a `Codec` for an `int[]` array. The `domain` parameter defines the allowed range of the `int` values and the `length` defines the length of the encoded `int` array.

```
static Codec<int[], IntegerGene>
ofVector(IntRange domain, int length) {
    return Codec.of(
        Genotype.of(IntegerChromosome.of(domain, length)),
        gt -> gt.chromosome()
            .as(IntegerChromosome.class)
            .toArray()
    );
```

```
}
```

Listing 2.10: `Codec` factory method: `ofVector`

The usage example of the *vector* `Codec` is almost the same as for the *scalar* `Codec`. As an additional parameter, we need to define the length of the desired array and we define our fitness function with an `int[]` array.

```
class Main {
    // Fitness function directly takes an 'int[]' array.
    static double fitness(int[] args) {
        return ...;
    }
    void main() {
        final Engine<IntegerGene, Double> engine = Engine
            .builder(
                Main::fitness,
                ofVector(new IntRange(0, 100), 10))
            .build();
        ...
    }
}
```

2.3.3 Matrix codec

In listing 2.11, the `ofMatrix` factory method returns a `Codec` for an `int[][]` matrix. The `domain` parameter defines the allowed range of the `int` values and the `rows` and `cols` defines the dimension of the matrix.

```
static Codec<int[][], IntegerGene> ofMatrix(
    IntRange domain,
    int rows,
    int cols
) {
    return Codec.of(
        Genotype.of(
            IntegerChromosome.of(domain, cols).instances()
                .limit(rows)
                .collect(ISeq.toISeq())
        ),
        gt -> gt.stream()
            .map(ch -> ch.stream()
                .mapToInt(IntegerGene::intValue)
                .toArray())
            .toArray(int[][]::new)
    );
}
```

Listing 2.11: `Codec` factory method: `ofMatrix`

2.3.4 Subset codec

There are currently two kinds of subset codecs you can choose from: finding subsets with variable size and with fixed size.

Variable sized subsets A `Codec` for variable sized subsets can be easily implemented with the use of a `BitChromosome`, as shown in listing 2.12.

```java
static <T> Codec<ISeq<T>, BitGene> ofSubSet(ISeq<T> basicSet) {
    return Codec.of(
        Genotype.of(BitChromosome.of(basicSet.length())),
        gt -> gt.chromosome()
            .as(BitChromosome.class).ones()
            .mapToObj(basicSet)
            .collect(ISeq.toISeq())
    );
}
```

Listing 2.12: `Codec` factory method: `ofSubSet`

The following usage example of subset `Codec` shows a simplified version of the Knapsack problem (see section 6.4). We try to find a subset, from the given basic `SET`, where the sum of the values is as big as possible, but smaller or equal than 20.

```java
class Main {
    // The basic set from where to choose an 'optimal' subset.
    final static ISeq<Integer> SET =
        ISeq.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    // Fitness function directly takes an 'int' value.
    static int fitness(ISeq<Integer> subset) {
        assert(subset.size() <= SET.size());
        final int size = subset.stream().collect(
            Collectors.summingInt(Integer::intValue));
        return size <= 20 ? size : 0;
    }
    void main() {
        final Engine<BitGene, Double> engine = Engine
            .builder(Main::fitness, ofSubSet(SET))
            .build();
        ...
    }
}
```

Fixed sized subsets The second kind of subset `Codec` allows you to find the *best* subset of a given, fixed size. A classical usage for this encoding is the Subset sum problem[12]:
Given a set (or multi-set) of integers, is there a non-empty subset whose sum is zero? For example, given the set $\{-7, -3, -2, 5, 8\}$*, the answer is yes because the subset* $\{-3, -2, 5\}$ *sums to zero. The problem is NP-complete.*[13]

```java
public class SubsetSum
    implements Problem<ISeq<Integer>, EnumGene<Integer>, Integer>
{
    private final ISeq<Integer> _basicSet;
    private final int _size;

    public SubsetSum(ISeq<Integer> basicSet, int size) {
        _basicSet = basicSet;
        _size = size;
    }

    @Override
    public Function<ISeq<Integer>, Integer> fitness() {
```

[12] `https://en.wikipedia.org/wiki/Subset_sum_problem`
[13] `https://en.wikipedia.org/wiki/NP-completeness`

```
14          return subset -> abs(
15              subset.stream().mapToInt(Integer::intValue).sum());
16      }
17
18      @Override
19      public Codec<ISeq<Integer>, EnumGene<Integer>> codec() {
20          return Codecs.ofSubSet(_basicSet, _size);
21      }
22  }
```

2.3.5 Permutation codec

This kind of `Codec` can be used for problems where the optimal solution depends on the order of the input elements. A classical example for such problems is the Knapsack problem (chapter 6.5).

```
1  static <T> Codec<T[], EnumGene<T>> ofPermutation(T... alleles) {
2      return Codec.of(
3          Genotype.of(PermutationChromosome.of(alleles)),
4          gt -> gt.chromosome().stream()
5              .map(EnumGene::allele)
6              .toArray(length -> (T[])Array.newInstance(
7                  alleles[0].getClass(), length))
8      );
9  }
```

Listing 2.13: `Codec` factory method: `ofPermutation`

Listing 2.13 shows the implementation of a permutation `Codec`, where the order of the given alleles influences the value of the fitness function. An alternate formulation of the traveling salesman problem is shown in the following listing. It uses the permutation `Codec` in listing 2.13 and uses `io.jenetics.jpx.WayPoints`, from the *JPX*[14] project, for representing the city locations.

```
1  public class TSM {
2      // The locations to visit.
3      static final ISeq<WayPoint> POINTS = ISeq.of(...);
4
5      // The permutation codec.
6      static final Codec<ISeq<WayPoint>, EnumGene<WayPoint>>
7      CODEC = Codecs.ofPermutation(POINTS);
8
9      // The fitness function (in the problem domain).
10     static double dist(final ISeq<WayPoint> path) {
11         return path.stream()
12             .collect(Geoid.DEFAULT.toTourLength())
13             .to(Length.Unit.METER);
14     }
15
16     // The evolution engine.
17     static final Engine<EnumGene<WayPoint>, Double> ENGINE = Engine
18         .builder(TSM::dist, CODEC)
19         .optimize(Optimize.MINIMUM)
20         .build();
21
22     // Find the solution.
23     void main() {
24         final ISeq<WayPoint> result = CODEC.decode(
25             ENGINE.stream()
```

[14] `https://github.com/jenetics/jpx`

```
26                      .limit(10)
27                      .collect(EvolutionResult.toBestGenotype())
28              );
29
30              IO.println(result);
31          }
32      }
```

2.3.6 Mapping codec

This `Codec` is a variation of the permutation `Codec`. Instead of permuting the elements of a given array, it permutes the mapping of elements of a source set to the elements of a target set. The code snippet below shows the method of the `Codecs` class, which creates a mapping codec from a given source and target set.

```
1 public static <A, B> Codec<Map<A, B>, EnumGene<Integer>>
2 ofMapping(ISeq<? extends A> source, ISeq<? extends B> target);
```

It is not necessary that the source and target set are of the same size. If |source| > |target|, the returned mapping function is *surjective*, if |source| < |target|, the mapping is *injective* and if |source| = |target|, the created mapping is *bijective*. In every case the size of the encoded `Map` is |target|. Figure 2.3.1 shows the described different mapping types in graphical form.

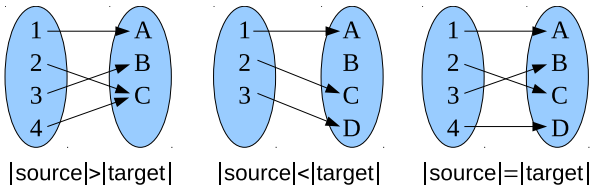


Figure 2.3.1: Mapping codec types

With |source| = |target|, you will create a `Codec` for the *assignment problem*. The problem is defined by a number of workers and a number of jobs. Every worker can be assigned to perform any job. The cost for a worker may vary depending on the worker job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each worker in such a way which optimizes the total assignment costs.[15] The costs for such worker job assignments are usually given by a matrix. Such an example matrix is shown in table 2.3.1.

	Job 1	**Job 2**	**Job 3**	**Job 4**
Worker 1	13	4	7	6
Worker 2	1	11	5	4
Worker 3	6	7	3	8
Worker 4	1	3	5	9

Table 2.3.1: Worker job cost

[15] `https://en.wikipedia.org/wiki/Assignment_problem`

If your worker job cost can be expressed by a matrix, the Hungarian algorithm[16] can find an optimal solution in $O\left(n^3\right)$ time. You should consider this deterministic algorithm before using a GA.

2.3.7 Composite codec

The *composite* `Codec` factory method allows to combine two or more `Codecs` into one. Listing 2.14 shows the method signature of the factory method, which is implemented directly in the `Codec` interface.

```
static <G extends Gene<?, G>, A, B, T> Codec<T, G> of(
    final Codec<A, G> codec1,
    final Codec<B, G> codec2,
    final BiFunction<A, B, T> decoder
);
```

Listing 2.14: Composite `Codec` factory method

As you can see from the method definition, the combining `Codecs` and the combined `Codec` have the same `Gene` type.

Only `Codecs` with the same `Gene` type can be composed by the combining factory methods of the `Codec` class.

The following listing shows a full example which uses a combined `Codec`. It uses the subset `Codec`, introduced in section 2.3.4, and combines it into a `Tuple` of subsets.

```
class Main {
    static final ISeq<Integer> SET =
        ISeq.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

    // Result type of the combined 'Codec'.
    static final class Tuple<A, B> {
        final A first;
        final B second;
        Tuple(final A first, final B second) {
            this.first = first;
            this.second = second;
        }
    }

    static int fitness(Tuple<ISeq<Integer>, ISeq<Integer>> args) {
        return args.first.stream()
                    .mapToInt(Integer::intValue)
                    .sum() -
                args.second.stream()
                    .mapToInt(Integer::intValue)
                    .sum();
    }

    void main() {
        // Combined 'Codec'.
        final Codec<Tuple<ISeq<Integer>, ISeq<Integer>>, BitGene>
            codec = Codec.of(
                Codecs.ofSubSet(SET),
```

[16] https://en.wikipedia.org/wiki/Hungarian_algorithm

```
                    Codecs.ofSubSet(SET),
                    Tuple::new
                );

        final Engine<BitGene, Integer> engine = Engine
            .builder(Main::fitness, codec)
            .build();

        final Phenotype<BitGene, Integer> pt = engine.stream()
            .limit(100)
            .collect(EvolutionResult.toBestPhenotype());

        // Use the codec for converting the result 'Genotype'.
        final Tuple<ISeq<Integer>, ISeq<Integer>> result =
            codec.decoder().apply(pt.genotype());
    }
}
```

If you have to combine more than one `Codec` into one, you have to use the second, more general, *combining* function: `Codec::of(ISeq<Codec<?, G>>, Function<Object[], T>)`. The example above shows how to use the general combining function. It is just a little bit more verbose and requires explicit casts for the *sub-codec* types.

```
final Codec<Triple<Long, Long, Long>, LongGene>
    codec = Codec.of(ISeq.of(
        Codecs.ofScalar(new LongRange(0, 100)),
        Codecs.ofScalar(new LongRange(0, 1000)),
        Codecs.ofScalar(new LongRange(0, 10000))),
        values -> {
            final Long first = (Long)values[0];
            final Long second = (Long)values[1];
            final Long third = (Long)values[2];
            return new Triple<>(first, second, third);
        }
    );
```

2.3.8 Invertible codec

The `InvertibleCodec` extends the `Codec` interface and allows to create a `Genotype` from a given value of the native problem domain.

```
public interface InvertibleCodec<T, G extends Gene<?, G>>
    extends Codec<T, G>
{
    Function<T, Genotype<G>> encoder();
    default Genotype<G> encode(final T value) {...}
}
```

Listing 2.15: `InvertibleCodec` interface

Listing 2.15 shows the additional methods of the `InvertibleCodec` interface. Creating a `Genotype` from a given domain value simplifies the implementation of the `Constraint::repair` method. Most of the factory methods in the `Codecs` class will return `InvertibleCodec` instances. The `encoder` function is not necessarily the inverse of the `decoder` function of the `Codec` interface. This is the case if more then one `Genotype` maps to the same value of the problem domain.

2.4 Problem

The `Problem` interface is a further abstraction level, which allows you to *bind* the problem encoding and the fitness function into one data structure.

```
public interface Problem<
    T,
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    Function<T, C> fitness();
    Codec<T, G> codec();
}
```

Listing 2.16: `Problem` interface

Listing 2.16 shows the `Problem` interface. The generic type `T` represents the type of the *native* problem domain. This is the argument type of the fitness`Function`, and `C` the `Comparable` result of the fitness`Function`. `G` is the `Gene` type, which is used by the evolution `Engine`.

```
// Definition of the Ones counting problem.
final Problem<ISeq<BitGene>, BitGene, Integer> ONES_COUNTING =
    Problem.of(
        // Fitness Function<ISeq<BitGene>, Integer>
        genes -> (int)genes.stream()
            .filter(BitGene::bit).count(),
        Codec.of(
            // Genotype Factory<Genotype<BitGene>>
            Genotype.of(BitChromosome.of(20, 0.15)),
            // Genotype conversion
            // Function<Genotype<BitGene>, <BitGene>>
            gt -> gt.chromosome().toSeq()
        )
    );

// Engine creation for Problem solving.
final Engine<BitGene, Integer> engine = Engine
    .builder(ONES_COUNTING)
    .populationSize(150)
    .survivorsSelector(newTournamentSelector<>(5))
    .offspringSelector(new RouletteWheelSelector<>())
    .alterers(
        new Mutator<>(0.03),
        new SinglePointCrossover<>(0.125))
    .build();
```

The listing above shows how a new `Engine` is created by using a predefined `Problem` instance. This allows the complete decoupling of problem and `Engine` definition.

2.5 Constraint

Constraints delimit the feasible space of solutions of an optimization problem and are considered in evolutionary algorithms [13, 28, 12, 29]. This influence the desirability of each possible solution. If the constraints are satisfied, the solution is accepted and it is called a feasible solution; otherwise the solution is removed or modified. For a fitness function, $f(\mathbf{x})$, the constraints are usually given as a

list of inequalities,

$$g_i(\mathbf{x}) \leq 0, \tag{2.5.1}$$

and a list of equations,

$$h_j(\mathbf{x}) = 0. \tag{2.5.2}$$

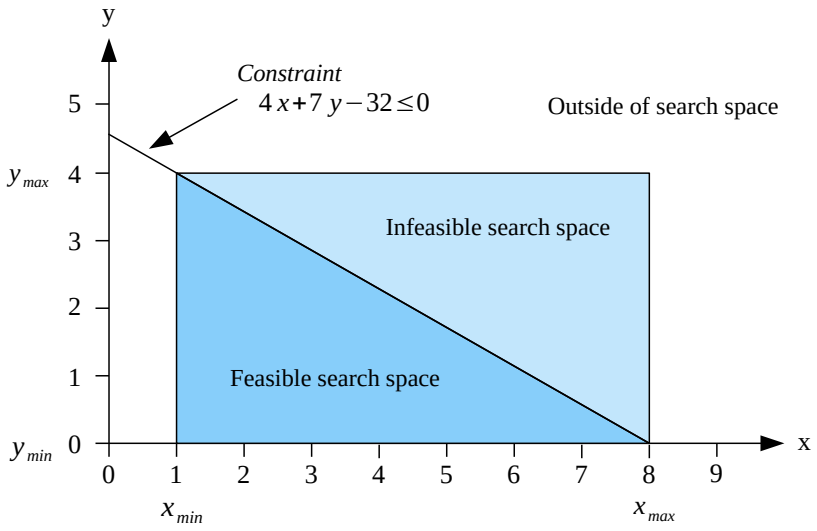


Figure 2.5.1: Constrained 2-dimensional search space

Figure 2.5.1 shows how the inequality, $4x + 7y - 32 \leq 0$, divides the search space into a feasible and an infeasible part. There are different approaches for handling constraints. *Penalty methods* try to convert a constrained optimization problem into an unconstrained one by incorporating its constraints into the fitness function. *Transformation methods* try to map the feasible region into a regular mapped space while preserving the feasibility somehow. The `Constraint` interface of **Jenetics** takes the second approach and tries to preserve feasibility through a *repair* step for invalid candidate solutions.

Usually, a given problem should be encoded in a way, that it is not possible for the evolution `Engine` to create invalid individuals (`Genotypes`). Some possible encodings for common data structures are described in section 2.2. The `Engine` creates new individuals in the altering step, by rearranging (or creating new) `Genes` within a `Chromosome`. Since a `Genotype` is treated as valid if every single `Gene` in every `Chromosome` is valid, the validity property of the `Gene`s determines the validity of the whole `Genotype`. The `Engine` tries to create only valid individuals when creating the initial population and when it replaces `Genotype`s which has been destroyed by the altering step. Individuals which has exceeded its lifetime are also replaced by new ones. Although this behavior will work for most `Genotype`s, it is still possible that invalid individuals will be created during the evolution. If you need a more advanced validation strategy, the `Constraint` interface comes into play.

```java
public interface Constraint<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    boolean test(Phenotype<G, C> ind);
    Phenotype<G, C> repair(Phenotype<G, C> ind, long gen);
}
```

Listing 2.17: `Constraint` interface

Listing 2.5 shows the definition of the `Constraint` interface. The `test` method of the interface checks the validity of the given `Phenotype` and the `repair` method creates a new individual using the invalid individual as template.

The `RetryConstraint` class is the default implementation of the `Constraint` interface. It implements the `repair` method by creating new `Phenotype`s until the created individual is valid. Although this approach seems a little bit simplistic, it has an important and desirable property: the *repaired* individuals follow the same distribution then the original. This means, that no part of the problem domain is left out or is *overcrowded*. The number of necessary retries is also not a problem, for *normal* constraints. For example, the probability that a randomly created point lies outside the unit circle is $1 - \frac{\pi}{4} \approx 0.2146$. This leads to a failure probability after 10 retries, which is the default value of the `RetryConstraint`, of $\left(1 - \frac{\pi}{4}\right)^{10} \approx 0.000000207$. You can parameterize a different `Constraint` definition with the `constraint` method of the `Engine.Builder`.

The behavior of the `Phenotype::isValid` method is overridden by the `Constraint` interface. A `Phenotype` is treated as invalid if the `Constraint::test` method returns false, even if the `Phenotype::isValid` method returns true.

Figure 2.5.2 shows the distribution of the domain points in our *unit circle* example. Rejecting invalid points and recreating new ones leads to an uniform point distribution. Every part of the domain is explored with the same probability. This is a very welcome property of the `RetryConstraint` strategy.

Trying to create only valid domain points can sometimes lead to a nonuniform distribution. This can be seen in figure 2.5.3. The points were created by choosing the angle, α, and the radius, r, randomly, and calculate the point coordinates, $\mathbf{x} = (r\cos\alpha, r\sin\alpha)$,where $r \in [-1, 1]$ and $\alpha \in [0, 2\pi)$. As you can see, the points near the center are much denser than at the domain border. This makes it harder for the `Engine` to explore the whole problem domain.

The `RetryConstraint` is the default implementation of the `Constraint` interface, but it might not be the best one for every given problem. If it is possible, it is better to try to repair an invalid `Phenotype` instead of creating a new one. Suppose you need to optimize the fitness function, $f : \mathbb{R}^3 \to \mathbb{R}$, with the following constraints:

$$\begin{aligned} x_1 + x_2 - 1 &\leq 0, \\ x_2 \cdot x_3 - 0.5 &\leq 0. \end{aligned}$$

A repairing `Constraint` implementation checks the validity of a `Phenotype`

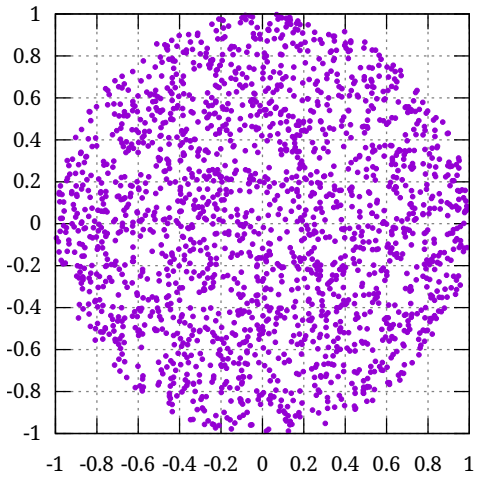


Figure 2.5.2: Domain points with retry-constraint

and repairs it, if it's invalid.

```
public class RepairingConstraint
    implements Constraint<DoubleGene, Double>
{
    @Override
    public boolean test(Phenotype<DoubleGene, Double> pt) {
        return isValid(
            pt.genotype().chromosome()
                .as(DoubleChromosome.class)
                .toArray()
        );
    }
    static boolean isValid(double[] x) {
        return x[0] + x[1] <= 1 && x[1]*x[2] <= 0.5;
    }

    @Override
    public Phenotype<DoubleGene, Double> repair(
        final Phenotype<DoubleGene, Double> pt,
        final long generation
    ) {
        final double[] x = pt.genotype().chromosome()
            .as(DoubleChromosome.class)
            .toArray();

        return newPhenotype(repair(x), generation);
    }
    static double[] repair(final double[] x) {
        if (x[0] + x[1] > 1) x[0] = 1 - x[1];
        if (x[1]*x[2] > 0.5) x[2] = 0.5/x[1];
        return x;
    }
}
```

The implementation of the new depends on your actual encoding and might look like this

```
Phenotype<DoubleGene, Double> newPhenotype(double[] r, long gen) {
```

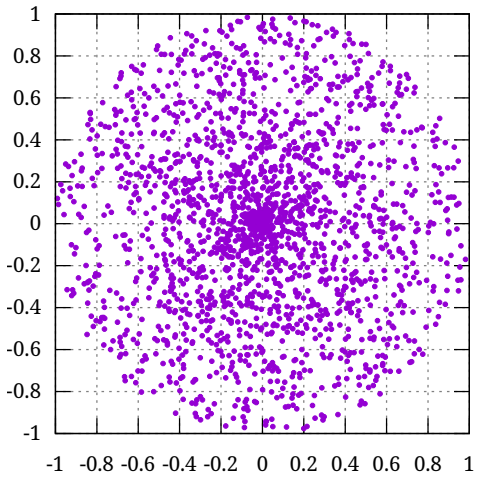


Figure 2.5.3: Only valid domain points

```
    final Genotype<DoubleGene> gt = Genotype.of(
        DoubleChromosome.of(
            DoubleStream.of(r).boxed()
                .map(v -> DoubleGene
                    .of(v, new DoubleRange(0, 1)))
                .collect(ISeq.toISeq())
        )
    );
    return Phenotype.of(gt, gen);
}
```

Writing a repair method this way is quite tedious. The `InvertibleCodec` interface, see section 2.15, allows to implement the repair function in a more natural way. Imagine you want to encode a split range, as shown in figure 2.5.4. Only the values between $[0, 2)$ and $[8, 10)$ are valid.

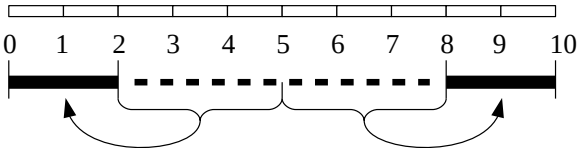


Figure 2.5.4: Split range domain

The following listing shows how to create a constraint, which fulfills the desired codec property.

```
final InvertibleCodec<Double, DoubleGene> codec =
    Codecs.ofScalar(new DoubleRange(0, 10));
final Constraint<DoubleGene, Double> constraint = Constraint.of(
    codec,
    v -> v < 2 || v >= 8,
    v -> {
        if (v >= 2 && v < 8) {
            return v < 5 ? ((v - 2)/3)*2 : ((8 - v)/3)*2 + 8;
```

```
        }
        return v;
    }
);
```

Using an `InvertibleCodec` instead of a `Codec`, the repair function can be expressed in the problem domain. In the given example, the repair function maps the invalid range $[2, 5)$ to $[0, 2)$ and the invalid range $[5, 8)$ to $[8, 10)$. An alternative implementation for this `Codec` can also be created by mapping the scalar range `Codec` directly, as shown in the following listing.

```
final Codec<Double, DoubleGene> codec = Codecs
    .ofScalar(new DoubleRange(0, 10))
    .map(v -> {
            if (v >= 2 && v < 8) {
                return v < 5 ? ((v - 2)/3)*2 : ((8 - v)/3)*2 + 8;
            }
            return v;
        });
```

Creating a new evolution `Engine` with a `Constraint`, only repairs individuals which has been destroyed by the alterer step. It is still possible that the defined `Genotype` factory will create invalid individuals. If your `Genotype` factory can't guarantee that only valid individuals are created, an additional setup step is necessary.

```
final Constraint<DoubleGene, Double> constraint = ...;
final Factory<Genotype<DoubleGene>> gtf = ...;
final Engine<DoubleGene, Double> engine = Engine
    .builder(fitness, constraint.constrain(gtf))
    .constraint(constraint)
    .build();
```

The `Constraint::constrain` method takes an unreliable genotype factory and wraps it into a reliable one. As long as the constraint is implemented *correctly*, only valid individuals are generated by the `Engine`.

The `Constraint`, defined in the Engine, only fixes individuals which has been destroyed during the evolution process. Individuals, created by the `Genotype` factory may still be invalid. Use the `Constraint::constrain` method for creating safe `Genotype` factories.

2.6 Termination

Termination is the criterion by which the evolution stream decides whether to continue or truncate the stream. This section gives a deeper insight into the different ways of terminating or truncating the `EvolutionStream`. The `EvolutionStream` of the **Jenetics** library offers an additional method for limiting the evolution. With the `limit(Predicate<EvolutionResult<G,C>>)` method it is possible to use more advanced termination strategies. If the predicate, given

to the `limit` function, returns false, the `EvolutionStream` is truncated. The `EvolutionStream.limit(r -> true)` will create an infinite evolution stream.

The predicate given to the `EvolutionStream::limit` function must return *false* for truncating the evolution stream. If it returns *true*, the evolution is continued.

All termination strategies described in the following sections are part of the library and can be created by factory methods of the `io.jenetics.engine.Limits` class. The termination strategies were tested by solving the Knapsack problem[17] (see section 6.4) with 250 items. This makes it a real problem with a search space size of $2^{250} \approx 10^{75}$ elements.

Population size:	150
Survivors selector:	`TournamentSelector<>(5)`
Offspring selector:	`RouletteWheelSelector<>()`
Alterers:	`Mutator<>(0.03)` and `SinglePointCrossover<>(0.125)`

Table 2.6.1: Knapsack evolution parameters

Table 2.6.1 shows the evolution parameters used for the termination tests. To make the tests comparable, all test runs use the same evolution parameters and the very same set of knapsack items. Each termination test was repeated 1,000 times, which should give enough data to draw the given candlestick diagrams.

Some of the implemented termination strategies need to maintain an internal state. These strategies can't be reused in different evolution streams. To be on the safe side, it is recommended to always create a `Predicate` instance for each stream. Calling `Stream.limit(Limits::by`*`TerminationStrategy`*`)` will always work as expected.

2.6.1 Fixed generation

The simplest way for terminating the evolution process, is to define a maximal number of generations on the `EvolutionStream`. It just uses the existing `limit` method of the Java `Stream` interface.

```
final long MAX_GENERATIONS = 100;
final EvolutionStream<DoubleGene, Double> stream = engine
    .stream()
    .limit(MAX_GENERATIONS);
```

This kind of termination method can always be applied—usually additional with other evolution terminators—, to guarantee the truncation of the `EvolutionStream` and to define an upper limit of the executed generations. Additionally, the `Limits::byFixedGeneration(long)` predicate can be used instead of the `Stream::limit(long)` method. This predicate is mainly there

[17] The actual implementation used for the termination tests can be found in the Github repository:`https://github.com/jenetics/jenetics/blob/master/jenetics.example/src/main/java/io/jenetics/example/Knapsack.java`

for the completion reason and behaves exactly as the `Stream::limit(long)` function, except for the number of evaluations performed by the resulting stream. The evaluation of the population is *max generations* + 1. This is because the limiting predicate works on the `EvolutionResult` object, which guarantees to contain an *evaluated* population. That means, that the population must be evaluated at least once, even for a generation limit of zero. If this is an unacceptable performance penalty, better use the `Stream::limit(long)` function instead.

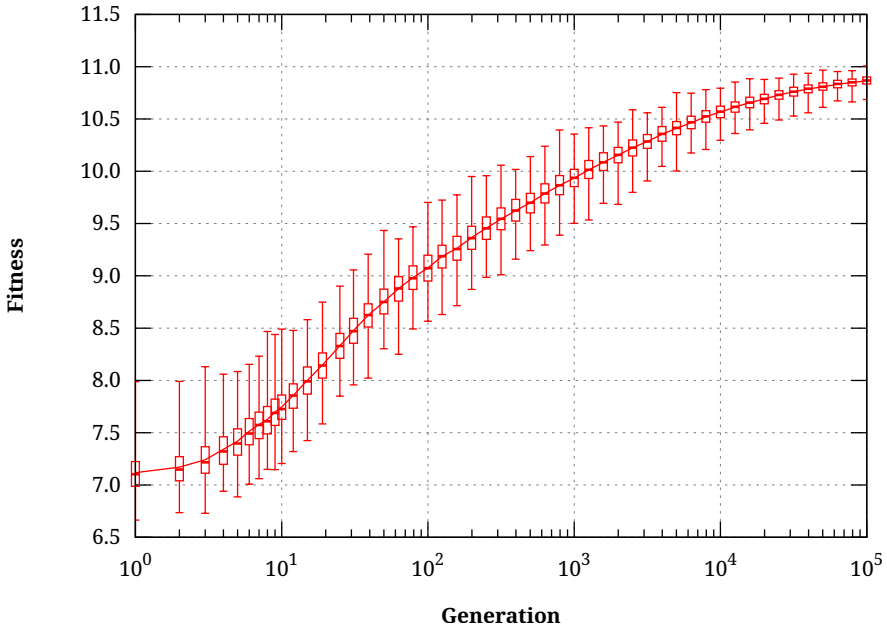


Figure 2.6.1: Fixed generation termination

Figure 2.6.1 shows the best fitness values of the used Knapsack problem after a given number of generations, whereas the candlestick points represents the *min*, 25^{th} *percentile*, *median*, 75^{th} *percentile* and *max* fitness after 250 repetitions per generation. The solid line shows for the *mean* of the best fitness values. For a small increase of the fitness value, the needed generations grows exponentially. This is especially the case when the fitness is approaching its *maximal* value.

2.6.2 Steady fitness

The steady fitness strategy truncates the `EvolutionStream` if its best fitness hasn't changed after a given number of generations. The predicate maintains an internal state—the number of generations with non increasing fitness—, and must be newly created for every `EvolutionStream`.

```
1 final class SteadyFitnessLimit<C extends Comparable<? super C>>
2     implements Predicate<EvolutionResult<?, C>>
3 {
4     private final int _generations;
5     private boolean _proceed = true;
6     private int _stable = 0;
```

```
7       private C _fitness;
8
9       public SteadyFitnessLimit(final int generations) {
10          _generations = generations;
11      }
12
13      @Override
14      public boolean test(final EvolutionResult<?, C> er) {
15          if (!_proceed) return false;
16          if (_fitness == null) {
17              _fitness = er.bestFitness();
18              _stable = 1;
19          } else {
20              final Optimize opt = result.optimize();
21              if (opt.compare(_fitness, er.bestFitness()) >= 0) {
22                  _proceed = ++_stable <= _generations;
23              } else {
24                  _fitness = er.bestFitness();
25                  _stable = 1;
26              }
27          }
28          return _proceed;
29      }
30  }
```

Listing 2.18: Steady fitness

Listing 2.18 shows the implementation of the `Limits::bySteadyFitness(int)` in the `io.jenetics.engine` package. It should give you an impression of how to implement own termination strategies, which possible holds an internal state.

```
1  final Engine<DobuleGene, Double> engine = ...
2  final EvolutionStream<DoubleGene, Double> stream = engine
3      .stream()
4      .limit(Limits.bySteadyFitness(15));
```

The steady fitness terminator can be created by the `bySteadyFitness` factory method of the `io.jenetics.engine.Limits` class. In the example above, the evolution stream is terminated after 15 stable generations.

Figure 2.6.2 shows the actual total executed generation depending on the desired number of steady fitness generations. The variation of the total generation is quite big, as shown by the candlesticks. Though the variation can be quite big—the termination test has been repeated 250 times for each data point—, the tests showed that the steady fitness termination strategy always terminated, at least for the given test setup. The lower diagram give an overview of the fitness progression. Only the mean values of the maximal fitness is shown.

2.6.3 Evolution time

This termination strategy stops the evolution when the elapsed evolution time exceeds an user specified maximal value. The `EvolutionStream` is only truncated at the end of a generation and will not interrupt the current evolution step. A maximal evolution time of zero *ms* will at least evaluate one generation. In a time critical environment, where a solution must be found within a maximal time period, this terminator lets you define the desired guarantees.

```
1  final Engine<DobuleGene, Double> engine = ...
2  final EvolutionStream<DoubleGene, Double> stream = engine
3      .stream()
```

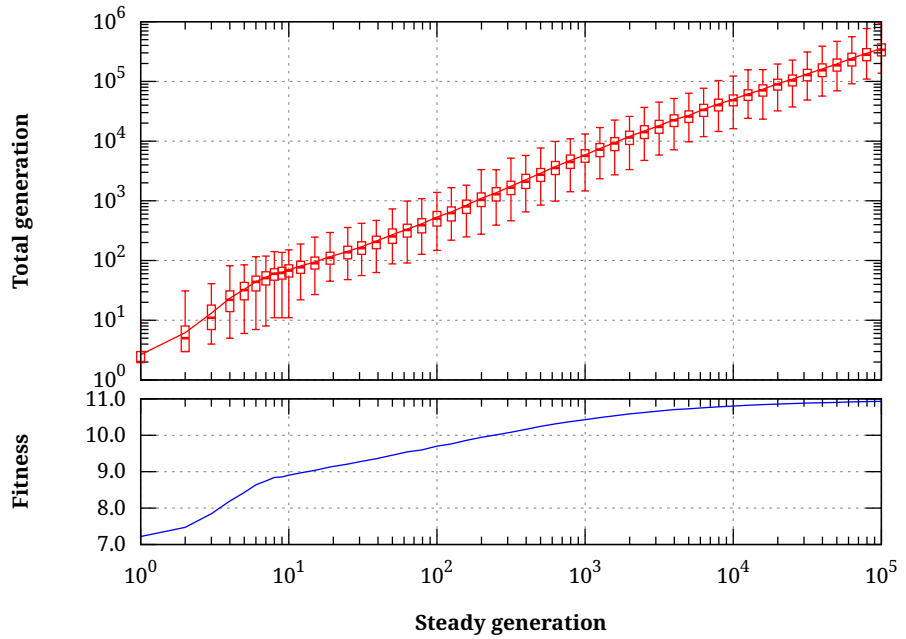


Figure 2.6.2: Steady fitness termination

```
4       .limit(Limits.byExecutionTime(Duration.ofMillis(500)));
```

In the code example above, the `byExecutionTime(Duration)` method is used for creating the termination object. Another method, `byExecutionTime(Duration, InstantSource)`, lets you define the `java.time.InstantSource`, which is used for measure the execution time. **Jenetics** uses the nano precision clock `io.jenetics.util.NanoClock` for measuring the time. To have the possibility to define a different `InstantSource` implementation is especially useful for testing purposes.

Figure 2.6.3 shows the evaluated generations depending on the execution time. Except for very small execution times, the evaluated generations per time unit stays quite stable.[18] That means that a doubling of the execution time will double the number of evolved generations.

2.6.4 Fitness threshold

A termination method that stops the evolution when the best fitness in the current population becomes less than the specified fitness threshold and the objective is set to minimize the fitness. This termination method also stops the evolution when the best fitness in the current population becomes greater than the specified fitness threshold when the objective is to maximize the fitness.

```
1 final Engine<DobuleGene, Double> engine = ...
2 final EvolutionStream<DoubleGene, Double> stream = engine
3     .stream()
```

[18]While running the tests, all other CPU intensive process has been stopped. The measuring started after a warm-up phase.

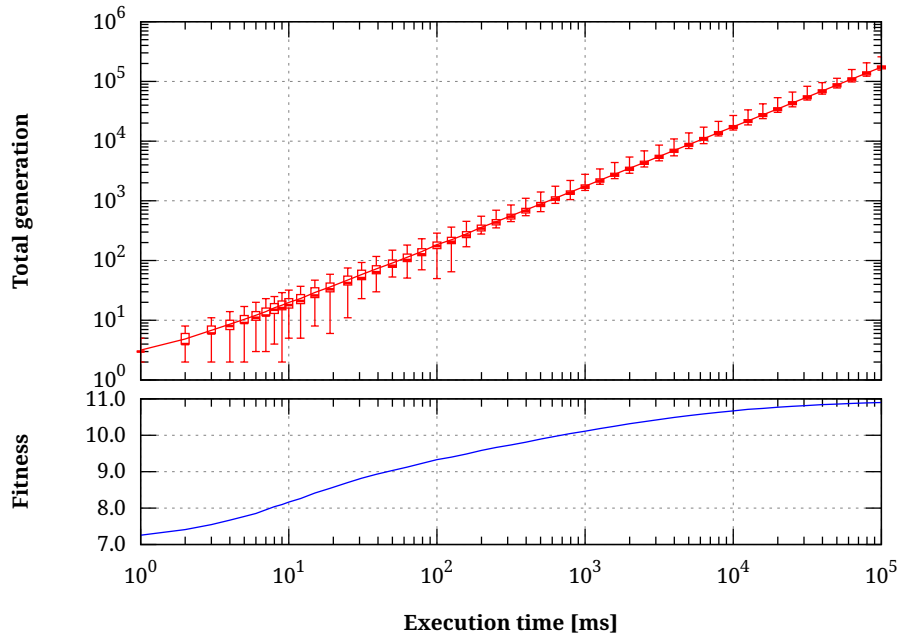


Figure 2.6.3: Execution time termination

```
4       .limit(Limits.byFitnessThreshold(10.5)
5       .limit(5000);
```

When limiting the `EvolutionStream` by a fitness threshold, you have to have knowledge about the expected maximal fitness. This can be the case if you are minimizing an error function with a known optimal value of zero. If there is no such knowledge, it is advisable to add an additional fixed sized generation limit as a safety net.

Figure 2.6.4 shows executed generations depending on the minimal fitness value. The total generations grow exponentially with the desired fitness value. This means, that this termination strategy will (practically) not terminate, if the value for the fitness threshold is chosen too high. And it will definitely not terminate if the fitness threshold is higher than the global maximum of the fitness function. It will be a *perfect* strategy if you can define some *good enough* fitness value, which can be *easily* achieved.

2.6.5 Fitness convergence

In this termination strategy, the evolution stops when the fitness is deemed as converged. Two filters of different lengths are used to smooth the best fitness across the generations. When the best smoothed fitness of the long filter is less than a specified percentage away from the best smoothed fitness from the short filter, the fitness is deemed as converged. **Jenetics** offers a generic version fitness convergence predicate and a version where the smoothed fitness is the moving average of the used filters.

```
1 public static <N extends Number & Comparable<? super N>>
```

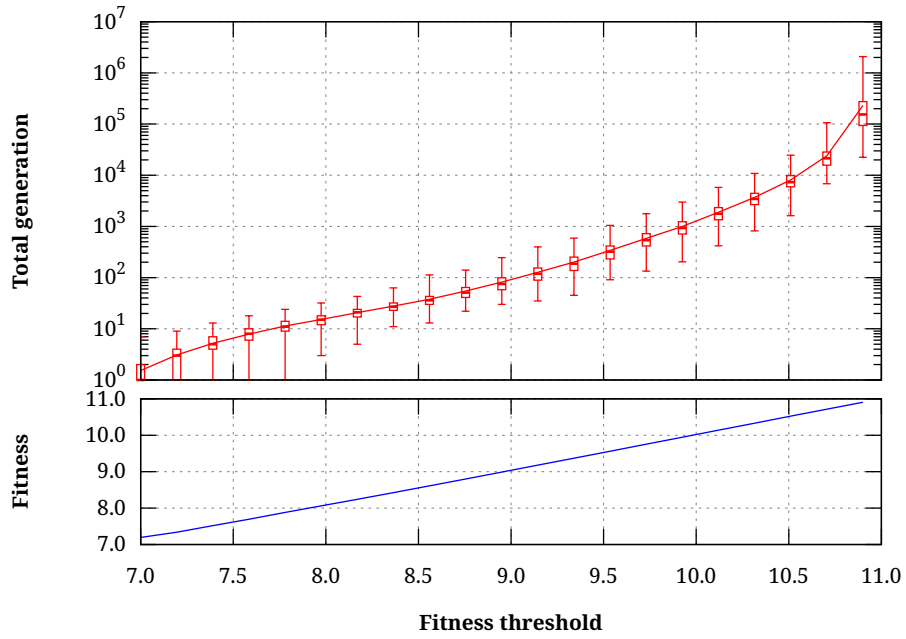


Figure 2.6.4: Fitness threshold termination

```
Predicate<EvolutionResult<?, N>> byFitnessConvergence(
    final int shortFilterSize,
    final int longFilterSize,
    final BiPredicate<DoubleMoments, DoubleMoments> proceed
);
```

Listing 2.19: General fitness convergence

Listing 2.19 shows the factory method which creates the *generic* fitness convergence predicate. This method allows to define the evolution termination according to the statistical moments of the short and long fitness filter.

```
public static <N extends Number & Comparable<? super N>>
Predicate<EvolutionResult<?, N>> byFitnessConvergence(
    final int shortFilterSize,
    final int longFilterSize,
    final double epsilon
);
```

Listing 2.20: Mean fitness convergence

The second factory method (shown in listing 2.20) creates a fitness convergence predicate, which uses the moving average[19] for the two filters. The smoothed fitness value is calculated as follows:

$$\sigma_F(N) = \frac{1}{N}\sum_{i=0}^{N-1} F_{[G-i]} \tag{2.6.1}$$

[19] https://en.wikipedia.org/wiki/Moving_average

where N is the length of the filter, $F_{[i]}$ the fitness value at generation$i$ and G the current generation. If the condition

$$\frac{|\sigma_F(N_S) - \sigma_F(N_L)|}{\delta} < \epsilon \tag{2.6.2}$$

is fulfilled, the `EvolutionStream` is truncated. Where δ is defined as follows:

$$\delta = \begin{cases} \max\left(|\sigma_F(N_S)|, |\sigma_F(N_L)|\right) & \text{if} \quad \neq 0 \\ 1 & \text{otherwise} \end{cases} . \tag{2.6.3}$$

```
final Engine<DobuleGene, Double> engine = ...
final EvolutionStream<DoubleGene, Double> stream = engine
    .stream()
    .limit(Limits.byFitnessConvergence(10, 30, 10E-4);
```

For using the fitness convergence strategy you have to specify three parameters. The length of the short filter, N_S, the length of the long filter, N_L, and the relative difference between the smoothed fitness values, ϵ.

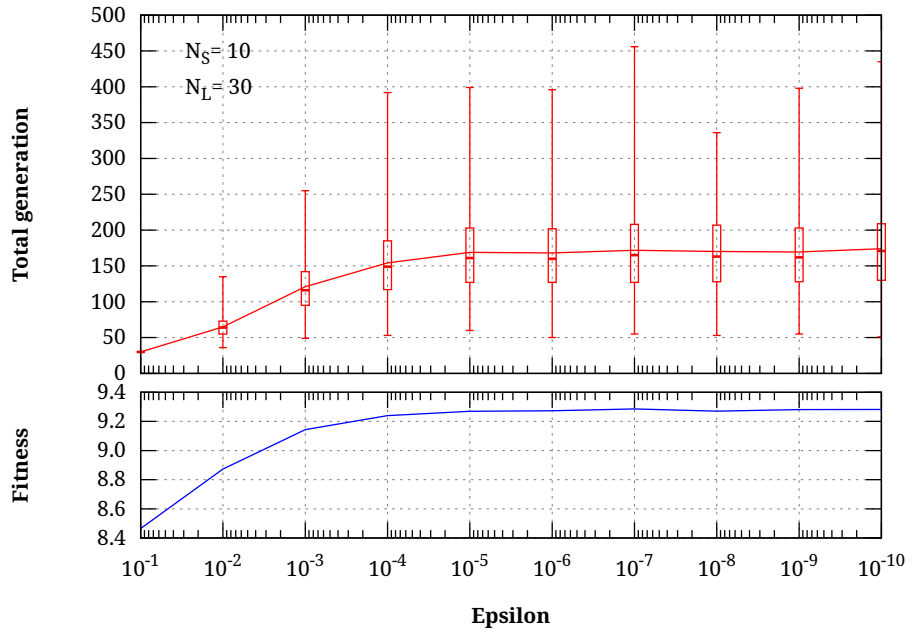


Figure 2.6.5: Fitness convergence termination:$N_S = 10$,$N_L = 30$

Figure 2.6.5 shows the termination behavior of the fitness convergence termination strategy. It can be seen that the minimum number of evolved generations is the length of the long filter, N_L.

Figure 2.6.6 shows the generations needed for terminating the evolution for higher values of the N_S and N_L parameters.

2.6.6 Population convergence

This termination method stops the evolution when the population is deemed as converged. A population is deemed as converged when the average fitness across

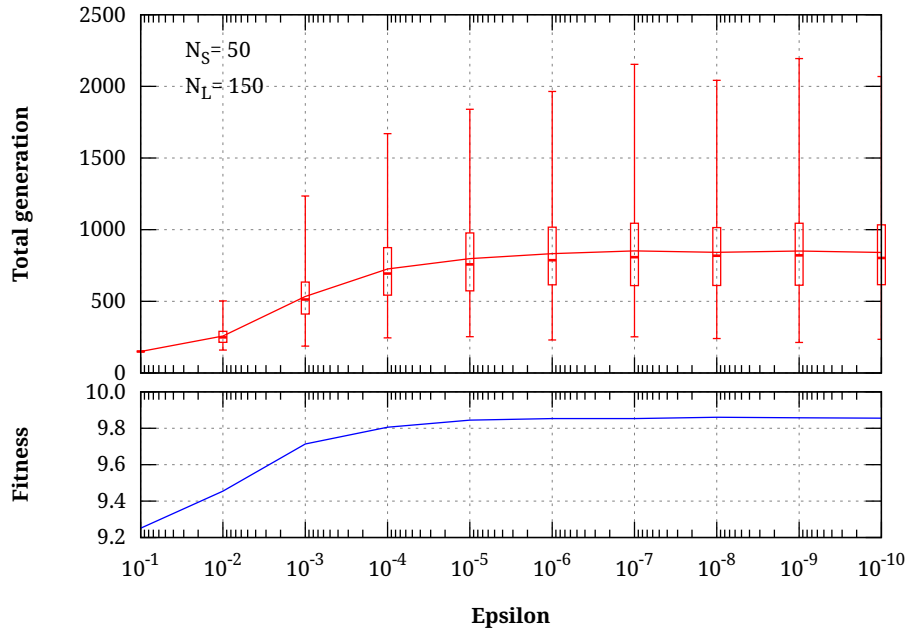


Figure 2.6.6: Fitness convergence termination:$N_S = 50$,$N_L = 150$

the current population is less than a user specified percentage away from the best fitness of the current population. The population is deemed as converged and the `EvolutionStream` is truncated if

$$\frac{\left|f_{max} - \bar{f}\right|}{\delta} < \epsilon, \tag{2.6.4}$$

where

$$\bar{f} = \frac{1}{N}\sum_{i=0}^{N-1} f_i, \tag{2.6.5}$$

$$f_{max} = \max_{i \in [0,N)} \{f_i\} \tag{2.6.6}$$

and

$$\delta = \begin{cases} \max\left(\left|f_{max}\right|, \left|\bar{f}\right|\right) & \text{if} \quad \neq 0 \\ 1 & \text{otherwise} \end{cases}. \tag{2.6.7}$$

N denotes the number of individuals of the population.

```
1 final Engine<DobuleGene, Double> engine = ...
2 final EvolutionStream<DoubleGene, Double> stream = engine
3     .stream()
4     .limit(Limits.byPopulationConvergence(0.1));
```

The `EvolutionStream` in the example above will terminate, if the difference between the population's fitness mean value and the maximal fitness value of the population is less than 10%.

2.6.7 Gene convergence

This termination strategy is different, in the sense that it takes the genes or alleles, respectively, for terminating the `EvolutionStream`. In the gene convergence termination strategy the evolution stops when a specified percentage of the genes of a genotype are deemed as converged. A gene is treated as converged when the average value of that gene across all of the genotypes in the current population is less than a given percentage away from the maximum allele value across the genotypes.

2.7 Reproducibility

Some problems can be defined with different kinds of fitness functions or encodings. Which combination works best can't usually be decided a priori. To choose one, some testing is needed. **Jenetics** allows you to set up an evolution `Engine` in a way that will produce the very same result on every run.

```
final Engine<DoubleGene, Double> engine =
    Engine.builder(fitnessFunction, codec)
        .executor(Runnable::run)
        .build();
final EvolutionResult<DoubleGene, Double> result =
    RandomRegistry.with(new Random(456), r ->
        engine.stream(population)
            .limit(100)
            .collect(EvolutionResult.toBestEvolutionResult())
    );
```

Listing 2.21: Reproducible evolution`Engine`

Listing 2.21 shows the basic setup of such a reproducible evolution `Engine`. Firstly, you have to make sure that all evolution steps are executed serially. This is done by configuring a single threaded executor. In the simplest case the evolution is performed solely on the main thread—`Runnable::run`. If the evolution `Engine` uses more than one worker thread, the reproducibility is no longer guaranteed. The second step configures the random generator, the evolution `Engine` is working with. Just *wrap* the `EvolutionStream` execution in a `RandomRegistry::with` block. Additionally you can start the `EvolutionStream` with a predefined, initial population. Once you have setup the `Engine`, you can vary the fitness function and the `Codec` and compare the results.

If you are using user defined implementations of the `Gene` and `Chromosome` interface, make sure to obtain the `RandomGenerator` object from the `RandomRegistry`. This is also required for every initialization code used in your problem implementation. Also check your code for hidden nondeterministic parts, e. g.`Collections::shuffle` method.

2.8 Evolution performance

This section contains an empirical proof, that evolutionary selectors deliver significantly better fitness results than a random search. The `MonteCarlo-`

`Selector` is used for creating the comparison (random search) fitness values.

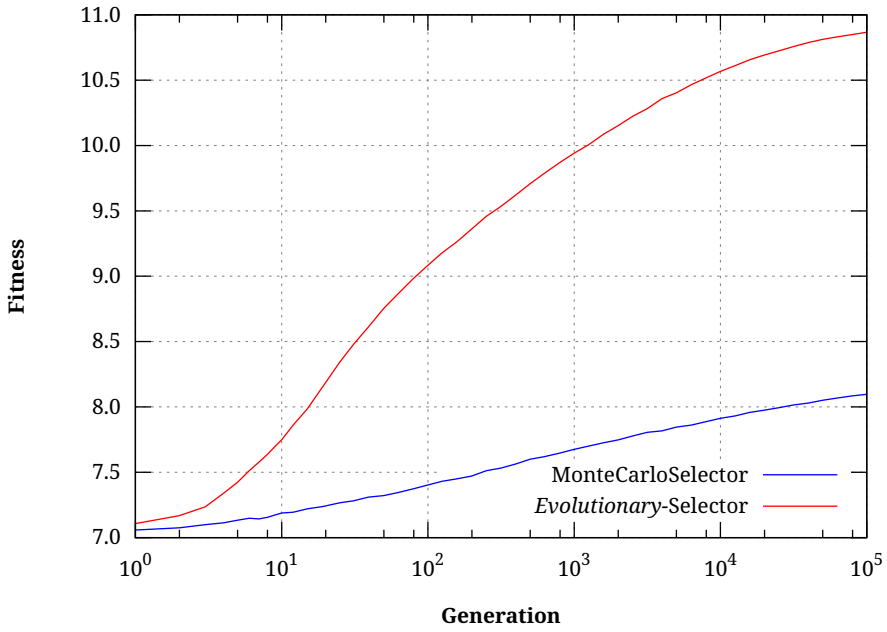


Figure 2.8.1: Selector performance (Knapsack)

Figure 2.8.1 shows the evolution performance of the `Selector`[20] used by the examples in section 2.6. The lower, blue line shows the (mean) fitness values of the *Knapsack* problem when using the `MonteCarloSelector` for selecting the survivors and offspring population. It can be easily seen, that the performance of the real evolutionary `Selectors` is much better than a random search.

2.9 Evolution strategies

Evolution Strategies, ES, were developed by Ingo Rechenberg and Hans-Paul Schwefel at the Technical University of Berlin in the mid 1960s.[42] It is a global optimization algorithm in continuous search spaces and is an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. ES uses truncation selection[21] for selecting the individuals and usually mutation[22] for changing the next generation. This section describes how to configure the evolution `Engine` of the library for the (μ, λ)- and $(\mu + \lambda)$-ES.

2.9.1 (μ, λ) evolution strategy

The (μ, λ) algorithm starts by generating λ individuals randomly. After evaluating the fitness of all the individuals, all but the μ fittest ones are deleted.

[20]The termination tests are using a `TournamentSelector`, with tournament-size 5, for selecting the survivors, and a `RouletteWheelSelector` for selecting the offspring.
[21]See1.3.2.1 on page 13.
[22]See1.3.2.2 on page 17.

Each of the μ fittest individuals gets to produce $\frac{\lambda}{\mu}$ children through an ordinary mutation. The newly created children just replaces the discarded parents.[27]

To summarize it: μ is the number of parents which survive, and λ is the number of offspring, created by the μ parents. The value of λ should be a multiple of μ. ES practitioners usually refer to their algorithm by the choice of μ and λ. If we set $\mu = 5$ and $\lambda = 20$, then we have a $(5, 20)$-ES.

```
final Engine<DoubleGene, Double> engine =
    Engine.builder(fitness, codec)
        .populationSize(lambda)
        .survivorsSize(0)
        .offspringSelector(new TruncationSelector<>(mu))
        .alterers(new Mutator<>(p))
        .build();
```

Listing 2.22: (μ, λ) `Engine` configuration

Listing 2.22 shows how to configure the evolution Engine for (μ, λ)-ES. The population size is set to λ and the survivors size to zero, since the best parents are not part of the final population. Step three is configured by setting the offspring selector to the `TruncationSelector`. Additionally, the `TruncationSelector` is parameterized with μ. This lets the `TruncationSelector` only select the μ best individuals, which corresponds to step two of the ES. There are mainly three levers for the (μ, λ)-ES where we can adjust exploration versus exploitation:[27]

- **Population size** λ: This parameter controls the sample size for each population. For the extreme case, as λ approaches ∞, the algorithm would perform a simple random search.
- **Survivors size of** μ: This parameter controls how selective the ES is. Relatively low μ values push the algorithm towards exploitative search, because only the best individuals are used for reproduction.[23]
- **Mutation probability** p: A high mutation probability pushes the algorithm toward a fairly random search, regardless of the selectivity of μ.

2.9.2 $(\mu + \lambda)$ evolution strategy

In the $(\mu + \lambda)$-*ES*, the next generation consists of the selected best μ parents and the λ new children. This is also the main difference compared to (μ, λ), where the μ parents are not part of the next generation. Thus the next and all successive generations are $\mu + \lambda$ in size.[27] **Jenetics** works with a constant population size and it is therefore not possible to implement an increasing population size. Besides this restriction, the `Engine` configuration for the $(\mu + \lambda)$-ES is shown in listing 2.23.

```
final Engine<DoubleGene, Double> engine =
    Engine.builder(fitness, codec)
        .populationSize(lambda)
        .survivorsSize(mu)
```

[23] As you can see in listing2.22, the survivors size (reproduction pool size) for the(μ, λ)-ES must be set *indirectly* via the `TruncationSelector` parameter. This is necessary, since for the(μ, λ)-ES, the selected bestμ individuals are not part of the population of the next generation.

```
5           .selector(new TruncationSelector<>(mu))
6           .alterers(new Mutator<>(p))
7           .build();
```

Listing 2.23: $(\mu + \lambda)$ `Engine` configuration

Since the selected μ parents are part of the next generation, the `survivorsSize` property must be set to μ. This also requires setting the survivors selector to the `TruncationSelector`. With the `selector(Selector)` method, both selectors and the selector for the survivors and for the offspring, can be set. Because the best parents are also part of the next generation, the $(\mu + \lambda)$-ES may be more exploitative than the (μ, λ)-ES. This has the risk, that very fit parents can defeat other individuals over and over again, which leads to a premature convergence to a local optimum.

2.10 Evolution interception

Once the `EvolutionStream` is created, it will continuously create `EvolutionResult` objects, one for every generation. It is not possible to alter the results, although it is tempting to use the `Streams::map` method for this purpose. The problem with the `map` method is, that the altered `EvolutionResult` will not be fed back to the `Engine` when evolving the next generation.

```
1 private EvolutionResult<DoubleGene, Double>
2 mapping(EvolutionResult<DoubleGene, Double> result) {...}
3
4 final Genotype<DobuleGene> result = engine.stream()
5     .map(this::mapping)
6     .limit(100)
7     .collect(toBestGenotype());
```

Performing the `EvolutionResult` mapping as shown in the code snippet above, will only change the results for the operations after the mapper definition. The evolution processing of the `Engine` is *not* affected. If we want to intercept the evolution process, the interceptor must be defined when the `Engine` is created.

```
1 final Engine<DobuleGene, Double> engine = Engine.builder(problem)
2     .interceptor(EvolutionInterceptor.ofAfter(this::mapping))
3     .build();
```

The code snippet above shows the correct way for intercepting the evolution stream. The mapper given to the `Engine` will change the stream of `EvolutionResults` and the will also feed the altered result back to the evolution `Engine`. Changing the evolved `EvolutionResult` is a powerful tool and should used cautiously.

Distinct population This kind of intercepting the evolution process is very flexible. **Jenetics** comes with one predefined stream interception method, which allows for removing duplicate individuals from the resulting population.

```
1 final Engine<DobuleGene, Double> engine = Engine.builder(problem)
2     .interceptor(EvolutionResult.toUniquePopulation())
3     .build();
```

Despite the de-duplication, it is still possible to have duplicate individuals. This will be the case when domain of the possible `Genotype`s is not big enough, and the same individual is created by chance. You can control the number of `Genotype` creation retries using the `EvolutionResult::toUniquePopulation(-int)` method, which allows you to define the maximal number of retries if an individual already exists.

Chapter 3

Modules

The **Jenetics** library has been split into several modules, which allows keeping the base EA module as small as possible. It currently consists of the modules shown in table 3.0.1, including the **Jenetics** base module.[1]

Module	Artifact
`io.jenetics.base`	`io.jenetics:jenetics:9.1.0`
`io.jenetics.ext`	`io.jenetics:jenetics.ext:9.1.0`
`io.jenetics.prog`	`io.jenetics:jenetics.prog:9.1.0`
`io.jenetics.xml`	`io.jenetics:jenetics.xml:9.1.0`
`io.jenetics.prngine`	`io.jenetics:prngine:2.0.0`

Table 3.0.1: **Jenetics** modules

With this module split, the code is easier to maintain and doesn't force the user to use parts of the library he or she isn't using. This keeps the `io.jenetics.base` module as small as possible. The additional **Jenetics** modules will be described in this chapter. Figure 3.0.1 shows the dependency graph of the **Jenetics** modules.

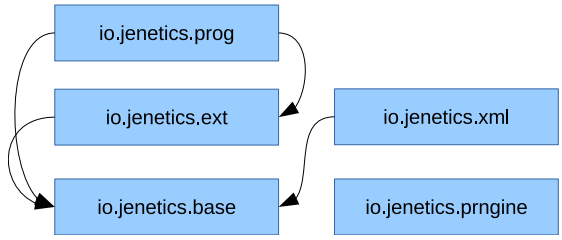


Figure 3.0.1: Module graph

[1]The used module names follow the recommended naming scheme for the JPMS automatic modules:`http://blog.joda.org/2017/05/java-se-9-jpms-automatic-modules.html`.

3.1 io.jenetics.ext

The `io.jenetics.ext` module implements additional *non* standard `Genes` and evolutionary operations. It also contains data structures which are used by these additional `Genes` and operations.

3.1.1 Data structures

3.1.1.1 CSV

Some optimization problems operate on data, stored in CSV format[2]. Symbolic regression, as described in section 3.2.6 on page 121, is one example of such a problem.

```
try (Stream<String> lines =
     CsvSupport.lines(new FileReader("data.csv")))
{
    final List<String[]> rows = lines
        .map(CsvSupport::split)
        .toList();
}
```

Listing 3.1: General usage example of `CsvSupport` class.

Listing 3.1 shows the most general way reading CSV data from a file. Additional helper functions allows to parse CSV data directly from a CSV string and parse it either as `String` values:

```
final List<String[]> rows = CsvSupport.parse("""
    ad,aixas,Aixàs,06,,42.4833333,1.4666667
    ad,aixirivali,Aixirivali,06,,42.4666667,1.5
    """
);
```

Or `double` values:

```
final List<double[]> rows = CsvSupport.parseDoubles("""
    0.7,0.6020
    0.8,0.9280
    0.9,1.3860
    1.0,2.0000
    """
);
```

The functional design of the CsvSupport makes it highly configurable and allows to read and write CSV data with different *quote* and/or *separator* characters.

```
final var reader = new LineReader(new Quote('\"'));
final var splitter = new LineSplitter(new Separator(','));
try (Stream<String> lines = reader.read(new FileReader(csv))) {
    final List<String[]> rows = lines
        .map(splitter::split)
        .toList();
}
```

[2]The `CsvSupport` class fully implements RFC-4180: *Common Format and MIME Type for Comma-Separated Values (CSV) Files* (`https://tools.ietf.org/html/rfc4180`).

3.1.1.2 Tree

The `Tree` interface defines a general tree data type, where each tree node can have an arbitrary number of children.

```
1 public interface Tree<V, T extends Tree<V, T>> {
2     V value();
3     Optional<T> parent();
4     T childAt(int index);
5     int childCount();
6 }
```

Listing 3.2: `Tree` interface

Listing 3.2 shows the `Tree` interface with its basic abstract tree methods. All other needed tree methods, e. g. for node traversal and search, are implemented by default methods, which are derived from these four abstract tree methods. A mutable default implementation of the `Tree` interface is given by the `TreeNode` class.

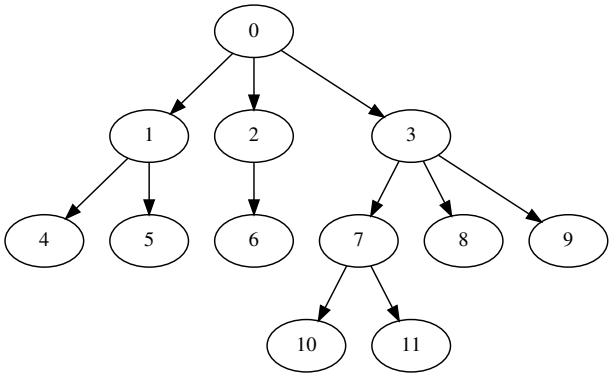


Figure 3.1.1: Example tree

To illustrate the usage of the `TreeNode` class, we will create a `TreeNode` instance from the tree shown in figure 3.1.1. The example tree consists of 12 nodes with a maximal depth of three and a varying child count from one to three.

```
1  final TreeNode<Integer> tree = TreeNode.of(0)
2      .attach(TreeNode.of(1)
3          .attach(4, 5))
4      .attach(TreeNode.of(2)
5          .attach(6))
6      .attach(TreeNode.of(3)
7          .attach(TreeNode.of(7)
8              .attach(10, 11))
9          .attach(8)
10         .attach(9));
```

Listing 3.3: Example `TreeNode`

Listing 3.3 shows the `TreeNode` representation of the given example tree. New children are added by using the `attach` method. For full `Tree` method list have a look at the Javadoc documentation.

3.1.1.3 Parentheses tree

A parentheses tree[3] is a serialized representation of a tree and is a simplified form of the *Newick* tree format[4]. The parentheses tree representation of the tree in figure 3.1.1 will look like the following string:

```
0(1(4,5),2(6),3(7(10,11),8,9))
```

As you can see, nodes on the same tree level are separated by a comma, ',`. New tree levels are created with an opening parentheses '`(`' and closed with a closing parentheses '`)`'. No additional spaces are inserted between the separator character and the node value. Any spaces in the parentheses tree string will be part of the node value. Figure 3.1.2 shows the syntax diagram of the parentheses tree. The `NodeValue` in the diagram is the string representation of the `Tree::value` object.

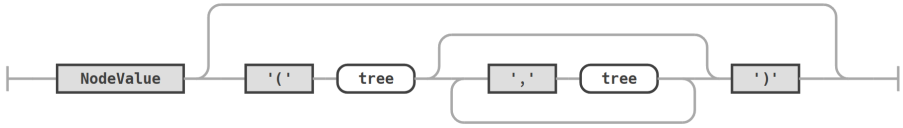


Figure 3.1.2: Parentheses tree syntax diagram

To get the parentheses tree representation, you just have to call `Tree::toParenthesesTree`. This method uses the `Object::toString` method for serializing the tree node value. If you need a different string representation you can use the `Tree::toParenthesesTree(Function<? super V, String>)` method. A simple example, on how to use this method, is shown in the code snippet below.

```
final Tree<Path, ?> tree = ...;
final String string = tree.toParenthesesString(Path::getFileName);
```

If the string representation of the tree node value contains one of the protected characters, '`,`', '`(`' or '`)`', they will be escaped with a '`\`' character.

```
final Tree<String, ?> tree = TreeNode.of("(root)")
    .attach(",", "(", ")")
```

The tree in the code snippet above will be represented as the following parentheses string:

```
\(root\)(\„\(,\))
```

Serializing a tree into parentheses form is just one part of the story. It is also possible to read back the parentheses string as tree object. The `TreeNode::parse(String)` method allows you to parse a tree string back to a `TreeNode<String>` object. If you need to create a tree with the original node type, you can call the `parse` method with an additional string mapper function. How you can parse a given parentheses tree string is shown in the code below.

[3] https://www.i-programmer.info/programming/theory/3458-parentheses-are-trees.html

[4] http://evolution.genetics.washington.edu/phylip/newicktree.html

```
final Tree<Integer, ?> tree = TreeNode.parse(
    "0(1(4,5),2(6),3(7(10,11),8,9))",
    Integer::parseInt
);
```

The `TreeNode::parse` method will throw an `IllegalArgumentException` if it is called with an invalid tree string.

3.1.1.4 Flat tree

The main purpose for the `Tree` data type in the `io.jenetics.ext` module is to support hierarchical `TreeGene`s, which are needed for genetic programming (see section 3.2). Since the `Chromosome` type is essentially an array, a mapping from the hierarchical tree structure to a 1-dimensional array is needed.[5] For general trees with arbitrary child count, additional information needs to be stored for a bijective mapping between tree and array. The `FlatTree` interface extends the `Tree` node with a `childOffset` method, which returns the absolute start index of the tree's children.

```
public interface FlatTree<V, T extends FlatTree<V, T>>
    extends Tree<V, T>
{
    int childOffset();
    default ISeq<T> flattenedNodes() {...};
}
```

Listing 3.4: `FlatTree` interface

Listing 3.4 shows the additional child offset needed for reconstructing the tree from the flattened array version. When flattening an existing tree, the nodes are traversed in breadth first order.[6] For each node the absolute array offset of the first child is stored, together with the child count of the node. If the node has no children, the child offset is set to -1.

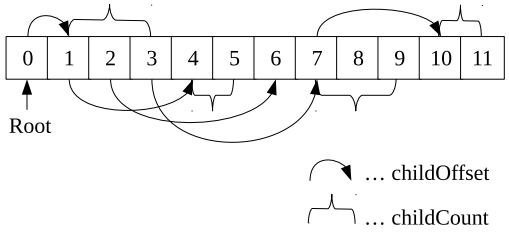


Figure 3.1.3: Example `FlatTree`

Figure 3.1.3 illustrates the flattened example tree shown in figure 3.1.1. The curved arrows denotes the child offset of a given parent node and the curly braces denotes the child count of a given parent node.

```
final TreeNode<Integer> tree = ...;
final ISeq<FlatTreeNode<Integer>> nodes = FlatTreeNode.of(tree)
    .flattenedNodes();
```

[5] There exists mapping schemes for *perfect* binary trees, which allows a bijective mapping from tree to array without additional storage need: `https://en.wikipedia.org/wiki/Binary_tree#Arrays`. For general trees with arbitrary child count, such simple mapping doesn't exist.

[6] `https://en.wikipedia.org/wiki/Breadth-first_search`

```
4 assert Tree.equals(tree, nodes.get(0));
5
6 final TreeNode<Integer> unflattened = TreeNode.of(nodes.get(0));
7 assert tree.equals(unflattened);
8 assert unflattened.equals(tree);
```

The code snippet above shows how to flatten a given integer tree and convert it back to a regular tree. The first element of the flattened tree node sequence is always the root node.

Since the `TreeGene` and the `ProgramGene` are implementing the `FlatTree` interface, it is helpful to know and understand the used tree to array mapping.

Since there is no possibility to change the nodes of a `FlatTree`, it can be used as an immutable version of the `Tree` interface. The tree nodes are also stored more memory efficient than in the `TreeNode` class.

3.1.1.5 Tree formatting

Using the parentheses tree is one possibility for creating a string representation of a given tree. Although it is the default format returned by the `toString()` method, it is sometime desirable to use different formats. The `TreeFormatter` class lets you to implement your own formats and also defines additional tree formats.

`TreeFormatter.PARENTHESES` Converts a tree to its default parentheses format. This is the default format and is also used by the `Tree::toString` method.

`TreeFormatter.TREE` Creates a verbose tree string, which spans multiple lines, e. g.

```
div
├── cos
│   └── 1.3
└── cos
    └── 3.14
```

`TreeFormatter.DOT` Creates a tree string in the dot format, which can be used to create nice graphs with Graphviz[7].

`TreeFormatter.LISP` Creates a *Lisp* tree from a given `Tree` instance. E. g.

```
(mul (div (cos 1.0) (cos 3.14)) (sin (mul 1.0 z)))
```

3.1.1.6 Tree reduction

The `Tree` is a very mighty data structure. It implements methods, which allows you to traverse and manipulate it. One of these methods is the `Tree::reduce` function, which traverses the tree in pre-order.

[7] `https://www.graphviz.org/`

```
public interface Tree<V, T> {
    default <U> U reduce(
        U[] neutral,
        BiFunction<? super V, ? super U[], ? extends U> reducer
    );
}
```

Listing 3.5: `Tree::reduce` method

Listing 3.5 shows the signature of the `reduce` method. The `neutral` array is used in the `reducer` function for leaf elements. After the reduction process, one element is returned, which might be `null` for empty trees. The following code snippet shows how to use the `Tree::reduce` method for evaluating a simple arithmetic expression tree.

```
final Tree<String, ?> formula = TreeNode.parse(
    "add(sub(6,div(230,10)),mul(5,6))"
);
final double result = formula.reduce(new Double[0], (op, args) ->
    switch (op) {
        case "add" -> args[0] + args[1];
        case "sub" -> args[0] - args[1];
        case "mul" -> args[0] * args[1];
        case "div" -> args[0] / args[1];
        default -> Double.parseDouble(op);
    }
);
```

The result of the reduced arithmetic tree will be 13.0, as expected.

3.1.2 Rewriting

Tree rewriting is a synonym for term rewriting, i.e., the process of transforming trees (tree structured data) into other trees by applying rewriting rules. Rewriting trees is not necessarily deterministic. One rewrite rule can be applied in many different ways to that term, or more than one rule will be applicable to a tree node. The rewriting system implementation in **Jenetics** is currently used for simplifying program trees, which are evolved in genetic programming problems (see section 3.2 and 3.2.4). A good introduction in tree/term rewriting systems can be found in [3].

. (Tree rewrite rule): A tree rewrite rule is a pair of terms (sub trees), $l \to r$. The notation indicates that the *left*-hand side, l, can be replaced by the *right*-hand side, r.

A rule, $l \to r$, can be applied to a tree, t, if the left tree (pattern) matches a sub tree of$t$. The matching sub tree is then replaced by the right tree (pattern)r.

. (Tree rewrite system): A tree rewrite system is a set, $\mathcal{R}$, of rewrite rules, $l \to r$.

In contrast to string rewriting systems, whose objects are flat sequences of symbols, the objects a term rewriting system works on, i.e. the terms, form a term algebra. A term can be visualized as a tree of symbols, the set of admitted symbols being fixed by a given signature.

3.1.2.1 Tree pattern

The `TreePattern` class is used for the left-hand and the right-hand side of a rewrite rule. It is typed and consists of variable (`Var`) and value (`Val`) nodes which form a sum type[8] of the *sealed* `Decl` class.

```
public final class TreePattern<V> {
    public TreePattern(Tree<Decl<V>, ?> pattern) {...}
    public TreeMatcher<V> filters(Tree<V, ?> tree) {...}
    public TreeNode<V> expand(Map<Var<V>, Tree<V, ?>> vars) {...}
}
```

Listing 3.6: `TreePattern` class

Listing 3.6 shows the constructor and main methods of the `TreePattern`. The `filters` method is used when used for the left-hand side and the `expand` method for the right-hand side. How to create a simple tree pattern is shown in the code snippet below.

```
final Tree<Decl<String>, ?> t = TreeNode
    .<Decl<String>>of(new Val<>("add"))
    .attach(new Var<>("x"), new Val<>("1"));
final TreePattern<String> p = new TreePattern<>(t);
assert p.filters(TreeNode.parse("add(sub(x,y),1)")).matches();
```

You can see that the variable `x` will match for arbitrary sub trees. For more complicated patterns it is quite cumbersome to create it via a `Decl` tree. Usually you will create a `TreePattern` object by compiling a proper pattern string. For creating the same pattern as in the example above you can write `TreePattern.compile("add($x,1)")`. The base syntax for the tree pattern follows the parentheses tree DSL described in 3.1.1.3. It only differs in the declaration of tree variables, which start with a '$' and must be a valid Java identifier. If you want to match *non* string trees you must specify an additional mapper function with the `compile` method.

```
final TreePattern<Integer> pattern = TreePattern
    .compile("0($x,1)", Integer::parseInt);
```

The right-hand side functionality of the rewrite rule is used to expand a given pattern. For expanding a given pattern you have to deliver a `Var` to sub tree mapping.

```
final TreePattern<String> pattern = TreePattern
    .compile("add($x,$y,1)");
final Map<Var<String>, Tree<String, ?>> vars = new HashMap<>();
vars.put(new TreePattern.Var<>("x"), TreeNode.parse("sin(x)"));
vars.put(new TreePattern.Var<>("y"), TreeNode.parse("sin(y)"));

final Tree<String, ?> tree = pattern.expand(vars);
assert tree.toParenthesesString().equals("add(sin(x),sin(y),1)");
```

3.1.2.2 Tree rewriter

The `TreeRewriter` interface is an abstraction of the tree *rewriting* functionality. Its `rewrite` method takes a `TreeNode`, which will be rewritten, and the maximal number the rule should be applied to the input tree.

[8] `https://en.wikipedia.org/wiki/Algebraic_data_type`

```
public interface TreeRewriter<V> {
    int rewrite(TreeNode<V> tree, int limit);
}
```

Listing 3.7: `TreeRewriter` interface

With the `TreeRewriter` interface you are able to combine two or more tree rewriter to one. This can be done with the `concat(final TreeRewriter<V>... rewriters)` factory method. There are two implementations of the `TreeRewriter` interface: the `TreeRewriteRule` class and the `TRS` class.

3.1.2.3 Tree rewrite rule

A `TreeRewriteRule` consists of left *matching* pattern and a right *replacing* pattern. To simplify the creation of a rewrite rule, it is possible to create one via a simple DSL: `add(0,$x) -> $x`. The left and the right tree pattern is separated by an arrow, `->`, and the pattern DSL is described in section 3.1.2.1.

```
final TreeRewriteRule<String> rule = TreeRewriteRule
    .compile("add($x,0) -> $x");
final TreeNode<String> t = parse("add(5,0)");
rule.rewrite(t);
```

Since the `TreeRewriteRule` implements the `TreeRewriter` interface, it can directly be used for rewriting input trees.

3.1.2.4 Tree rewrite system (TRS)

The `TRS` class puts all things together and allows for defining a complete tree (term) rewriting system. The primary constructor will take a sequence of `TreeRewriteRule`s (`ISeq<TreeRewriteRule<V>>`), but the `TRS` creation can be simplified by using a simple DSL.

```
final TRS<String> trs = TRS.of(
    "add(0,$x) -> $x",
    "add(S($x),$y) -> S(add($x,$y))",
    "mul(0,$x) -> 0",
    "mul(S($x),$y) -> add(mul($x,$y),$y)"
);
```

The example above defines a tree rewrite system with four rewrite rules, which are applied in the given order. Each rule is applied until the given tree stays unchanged. This also means, that the termination of the `TRS` can't be guaranteed. It's mainly your responsibility to create a rewrite system which will *always* terminate. If you are not sure whether the system is terminating or not, you better call the `TreeRewriter.rewrite(TreeNode, int)` method, which also takes the maximal number, the rule should be applied to the input tree.

```
final TreeNode<String> t = parse("add(S(0),S(mul(S(0),S(S(0)))))");
trs.rewrite(t);
assert t.equals(parse("S(S(S(S(0))))"));
```

Since the given tree rewrite system is terminating, we can safely apply the `TRS` to add(S(0),S(mul(S(0),S(S(0))))), which will then be rewritten to S(S(S(S(0)))).

3.1.2.5 Constant expression rewriter

The `ConstExprRewriter` class allows for the evaluation of constant tree expressions. In the code snippet below, it is shown how to evaluate a constant double expression.

```
final TreeNode<Op<Double>> tree = MathExpr
    .parse("1+2+3+4")
    .toTree();
ConstRewriter.ofType(Double.class).rewrite(tree);
assert tree.value().equals(Const.of(10.0))
```

Since the `ConstExprRewriter` can rewrite constant expressions of arbitrary types, a rewrite instance of the appropriate type, `Double`, must be created first.

3.1.3 Genes

3.1.3.1 BigInteger gene

The `BigIntegerGene` implements the `NumericGene` interface and can be used when the range of the existing `LongGene` or `DoubleGene` is not enough. Its allele type is a `BigInteger`, which can store arbitrary precision integers. There also exists a corresponding `BigIntegerChromosome`.

3.1.3.2 Tree gene

The `TreeGene` interface extends the `FlatTree` interface and serves as basis for the `ProgramGene`, used for genetic programming. Its tree nodes are stored in the corresponding `TreeChromosome`. How the tree hierarchy is flattened and mapped to an array is described in section 3.1.1.4.

3.1.4 Operators

Simulated binary crossover The `SimulatedBinaryCrossover` performs the simulated binary crossover (SBX) on `NumericChromosomes` such that each position is either crossed contracted or expanded with a certain probability. The probability distribution is designed such that the children will lie closer to their parents as is the case with the single point binary crossover. It is implemented as described in [16].

Single-node crossover The `SingleNodeCrossover` class works on `TreeChromosomes`. It swaps two, randomly chosen, nodes from two tree chromosomes. Figure 3.1.4 shows how the single-node crossover works. In this example node 3 of the first tree is swapped with node h of the second tree.

Reverse sequence mutator (RSM) The `RSMutator` chooses two positions i and j randomly. The gene order in a chromosome will then be reversed between these two points. This mutation operator can also be used for combinatorial problems, where no duplicated genes within a chromosome are allowed, e.g. for the TSP.[1]

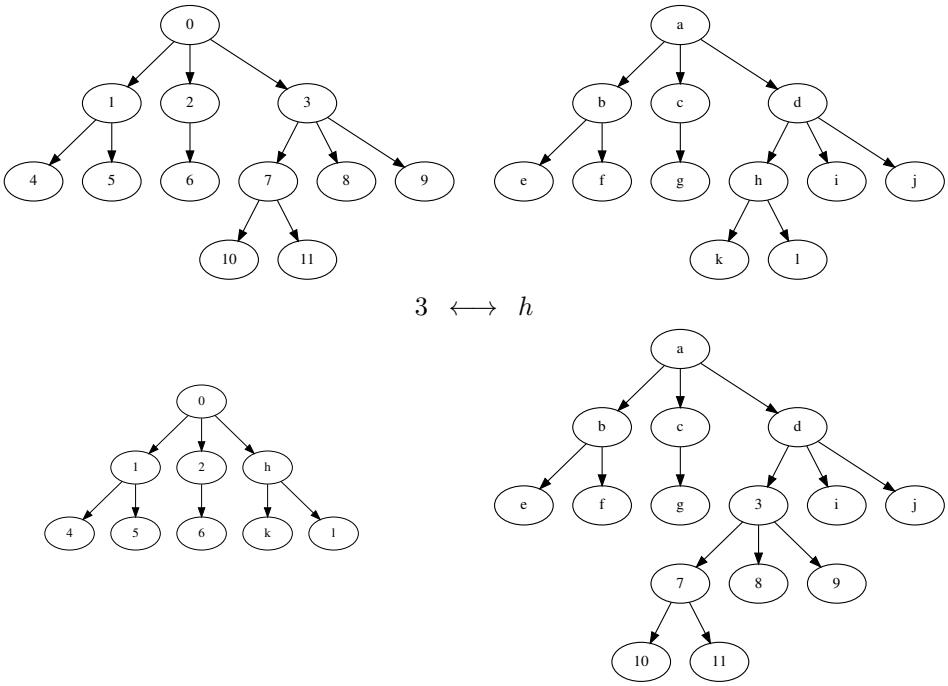


Figure 3.1.4: Single-node crossover

Hybridizing PSM and RSM (HPRM) The `HPRMutator` constructs an offspring from a pair of parents by hybridizing two mutation operators, PSM (`SwapMutator`) and RSM. Its main application is for combinatorial problems, like the TSP.[2]

3.1.5 Weasel program

The Weasel program[9] is thought experiment from Richard Dawkins, in which he tries to illustrate the function of genetic *mutation* and *selection*.[10] For this reason he chooses the well known example of typewriting monkeys.

> I don't know who it was first pointed out that, given enough time, a monkey bashing away at random on a typewriter could produce all the works of Shakespeare. The operative phrase is, of course, given enough time. Let us limit the task facing our monkey somewhat. Suppose that he has to produce, not the complete works of Shakespeare but just the short sentence »Methinks it is like a weasel«, and we shall make it relatively easy by giving him a typewriter with a restricted keyboard, one with just the 26 (uppercase) letters, and a space bar. How long will he take to write this one little sentence?[14]

The search space of the 28 character long target string is $27^{28} \approx 10^{40}$. If the monkey writes $1,000,000$ different *sentences* per second, it would take about 10^{26}

[9] `https://en.wikipedia.org/wiki/Weasel_program`
[10] The classes are located in the `io.jenetics.ext` module.

years (in average) writing the correct one. Although Dawkins did not provide the source code for his program, a »Weasel« style algorithm could run as follows:

1. Start with a random string of 28 characters.
2. Make*n* copies of the string (reproduce).
3. Mutate the characters with an mutation probability of 5%.
4. Compare each new string with the target string »METHINKS IT IS LIKE A WEASEL«, and give each a score (the number of letters in the string that are correct and in the correct position).
5. If any of the new strings has a perfect score (28), halt. Otherwise, take the highest scoring string, and go to step 2.

Richard Dawkins was also very careful to point out the limitations of this simulation:

> Although the monkey/Shakespeare model is useful for explaining the distinction between single-step selection and cumulative selection, it is misleading in important ways. One of these is that, in each generation of selective »breeding«, the mutant »progeny« phrases were judged according to the criterion of resemblance to a distant ideal target, the phrase METHINKS IT IS LIKE A WEASEL. Life isn't like that. Evolution has no long-term goal. There is no long-distance target, no final perfection to serve as a criterion for selection, although human vanity cherishes the absurd notion that our species is the final goal of evolution. In real life, the criterion for selection is always short-term, either simple survival or, more generally, reproductive success.[14]

If you want to write a Weasel program with the **Jenetics** library, you need to use the special `WeaselSelector` and `WeaselMutator`.

```
public class WeaselProgram {
    private static final String TARGET =
        "METHINKS IT IS LIKE A WEASEL";

    private static int score(final Genotype<CharacterGene> gt) {
        final var src = gt.chromosome().as(CharSequence.class);
        return IntStream.range(0, TARGET.length())
            .map(i -> src.charAt(i) == TARGET.charAt(i) ? 1 : 0)
            .sum();
    }

    void main() {
        final CharSeq chars = CharSeq.of("A–Z ");
        final Factory<Genotype<CharacterGene>> gtf = Genotype.of(
            new CharacterChromosome(chars, TARGET.length())
        );
        final Engine<CharacterGene, Integer> engine = Engine
            .builder(WeaselProgram::score, gtf)
            .populationSize(150)
            .selector(new WeaselSelector<>())
            .offspringFraction(1)
            .alterers(new WeaselMutator<>(0.05))
            .build();
        final Phenotype<CharacterGene, Integer> result = engine
```

```
25                .stream()
26                .limit(byFitnessThreshold(TARGET.length() - 1))
27                .peek(r -> IO.println(
28                    r.totalGenerations() + ": " +
29                    r.bestPhenotype()))
30                .collect(toBestPhenotype());
31            IO.println(result);
32        }
33    }
```

Listing 3.8: `Weasel program`

Listing 3.8 shows how to implement the `WeaselProgram` with **Jenetics**. Step (1) and (2) of the algorithm is done implicitly when the initial population is created. The third step is done by the `WeaselMutator`, with mutation probability of 0.05. Step (4) is done by the `WeaselSelector` together with the configured offspring fraction of one. The `EvolutionStream` is limited by the `Limits.byFitnessThreshold`, which is set to $score_{max} - 1$. In the current example this value is set to `TARGET.length()` - 1 = 27.

```
1:  [UBNHLJUS RCOXR LFIYLAWRDCCNY] --> 6
2:  [UBNHLJUS RCOXR LFIYLAWWDCCNY] --> 7
3:  [UBQHLJUS RCOXR LFIYLAWWECCNY] --> 8
5:  [UBQHLJUS RCOXR LFICLAWWECCNL] --> 9
6:  [W QHLJUS RCOXR LFICLA WEGCNL] --> 10
7:  [W QHLJKS RCOXR LFIHLA WEGCNL] --> 11
8:  [W QHLJKS RCOXR LFIHLA WEGSNL] --> 12
9:  [W QHLJKS RCOXR LFIS A WEGSNL] --> 13
10: [M QHLJKS RCOXR LFIS A WEGSNL] --> 14
11: [MEQHLJKS RCOXR LFIS A WEGSNL] --> 15
12: [MEQHIJKS ICOXR LFIN A WEGSNL] --> 17
14: [MEQHINKS ICOXR LFIN A WEGSNL] --> 18
16: [METHINKS ICOXR LFIN A WEGSNL] --> 19
18: [METHINKS IMOXR LFKN A WEGSNL] --> 20
19: [METHINKS IMOXR LIKN A WEGSNL] --> 21
20: [METHINKS IMOIR LIKN A WEGSNL] --> 22
23: [METHINKS IMOIR LIKN A WEGSEL] --> 23
26: [METHINKS IMOIS LIKN A WEGSEL] --> 24
27: [METHINKS IM IS LIKN A WEHSEL] --> 25
32: [METHINKS IT IS LIKN A WEHSEL] --> 26
42: [METHINKS IT IS LIKN A WEASEL] --> 27
46: [METHINKS IT IS LIKE A WEASEL] --> 28
```

The (shortened) output of the Weasel program (listing 3.8) shows, that the optimal solution is reached in generation 46.

3.1.6 Modifying `Engine`

The current design of `Engine` allows for creating multiple independent `EvolutionStreams` from a single `Engine` instance. One drawback of this approach is, that the `EvolutionStream` runs with the same evolution parameters until the stream is truncated. It is not possible to change the stream's `Engine` configuration during the evolution process. This is the purpose of the `EvolutionStreamable` interface. It is similar to the Java `Iterable` interface and abstracts the `EvolutionStream` creation.

```
1  public interface EvolutionStreamable<
2      G extends Gene<?, G>,
3      C extends Comparable<? super C>
4  > {
5      EvolutionStream<G, C>
6      stream(Supplier<EvolutionStart<G, C>> start);
7
```

```
8       EvolutionStream<G, C> stream(EvolutionInit<G> init);
9
10      EvolutionStreamable<G, C>
11      limit(Supplier<Predicate<? super EvolutionResult<G, C>>> p)
12  }
```

Listing 3.9: `EvolutionStreamable` interface

Listing 3.9 shows the main methods of the `EvolutionStreamable` interface. The existing `stream` methods take an initial value, which allows to concatenate different engines. With the `limit` method it is possible to limit the size of the created `EvolutionStream` instances. The `io.jenetics.ext` module contains additional classes which allows for concatenating evolution `Engines` with different configurations, which will then create one varying `EvolutionStream`. This additional `Engine` classes are:

1. `ConcatEngine` and
2. `CyclicEngine`.

3.1.6.1 ConcatEngine

The `ConcatEngine` class allows for creating more than one `Engine` with different configurations, and combine it into one `EvolutionStreamable` (`Engine`).

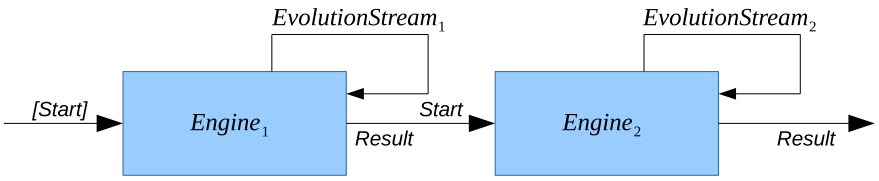


Figure 3.1.5: `Engine` concatenation

Figure 3.1.5 shows how the `EvolutionStream` of two concatenated `Engines` works. You can create the first partial `EvolutionStream` with an optional start value. If the first `EvolutionStream` stops, its final `EvolutionResult` is used as start value of the second `EvolutionStream`, created by the second evolution `Engine`. It is important that the evolution `Engines` used for concatenation are limited. Otherwise the created `EvolutionStream` will only use the first `Engine`, since it is not limited.

The concatenated evolution Engines must be limited (by calling `Engine::limit`), otherwise only the first `Engine` is used executing the resulting EvolutionStream.

The following code sample shows how to create an `EvolutionStream` from two concatenate `Engines`. As you can see, the two `Engines` are limited.

```
1  final Engine<DoubleGene, Double> engine1 = ...;
2  final Engine<DoubleGene, Double> engine2 = ...;
3
4  final Genotype<DoubleGene> result =
```

```
5       ConcatEngine.of(
6           engine1.limit(50),
7           engine2.limit(() -> Limits.bySteadyFitness(30)))
8       .stream()
9           .collect(EvolutionResult.toBestGenotype());
```

A practical use case for the `Engine` concatenation is, when you want to do a broader exploration of the search space at the beginning and narrow it with the following `Engine`. In such a setup, the first `Engine` would be configured with a `Mutator` with a relatively big mutation probability. The mutation probabilities of the following `Engine` would then be gradually reduced.

3.1.6.2 CyclicEngine

The `CyclicEngine` is similar to the `ConcatEngine`. Where the `ConcatEngine` stops the evolution, when the `EvolutionStream` of the last engine terminates, the `CyclicEngine` continues with a new `EvolutionStream` from the first `Engine`. The evolution flow of the `CyclicEngine` is shown in figure 3.1.6.

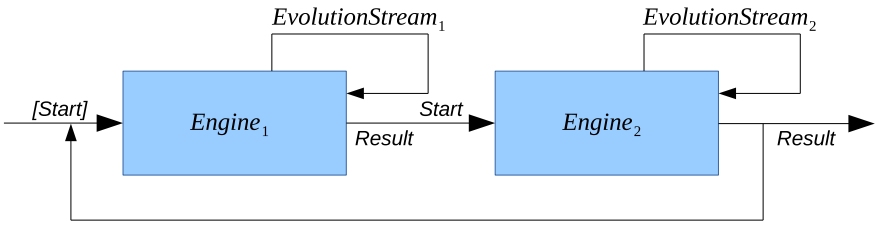


Figure 3.1.6: Cyclic`Engine`

Since the `CyclicEngine` creates unlimited streams, although the participating `Engine`s are all creating limited streams, the resulting `EvolutionStream` *must* be limited as well. The code snippet below shows the creation and execution of a*cyclic* `EvolutionStream`.

```
1   final Genotype<DoubleGene> result =
2       CyclicEngine.of(
3           engine1.limit(50),
4           engine2.limit(() -> Limits.bySteadyFitness(15)))
5       .stream()
6           .limit(Limits.bySteadyFitness(50))
7           .collect(EvolutionResult.toBestGenotype());
```

The reason for using a cyclic `EvolutionStream` is similar to the reason for using a concatenated `EvolutionStream`. It allows you to do a broad search, followed by a narrowed exploration. This cycle is then repeated until the limiting predicate of the *outer* stream terminates the evolution process.

3.1.7 Multi-objective optimization

A Multi-objective optimization Problem (MOP) can be defined as the problem of finding

> a vector of decision variables which satisfies constraints and optimizes
> a vector function whose elements represent the objective functions.
> These functions form a mathematical description of performance

> criteria which are usually in conflict with each other. Hence, the term »optimize« means finding such a solution which would give the values of all the objective functions acceptable to the decision maker.[35]

There are several ways for solving multiobjective problems. An excellent theoretical foundation is given in [10]. The algorithms implemented by **Jenetics** are based in therms of Pareto optimality as described in [18], [15] and [23].

3.1.7.1 Pareto efficiency

Pareto efficiency is named after the Italian economist and political scientist Vilfredo Pareto[11]. He used the concept in his studies of economic efficiency and income distribution. The concept has been applied in different academic fields such as economics, engineering, and the life sciences. Pareto efficiency says that an allocation is efficient if an action makes some individual better off and no individual worse off. In contrast to single-objective optimization, where usually only one optimal solution exits, the multi-objective optimization creates a set of optimal solutions. The optimal solutions are also known as the Pareto front or Pareto set.

. (Pareto efficiency[10]**):** A solution, $\mathbf{x}$, is said to be Pareto optimal iff there is no $\mathbf{x}'$ for which $\mathbf{v} = (f_1(\mathbf{x}'), ..., f_k(\mathbf{x}'))$ dominates $\mathbf{u} = (f_1(\mathbf{x}), ..., f_k(\mathbf{x}))$.

The definition says that $\mathbf{x}^*$ is Pareto optimal if there exists no feasible vector,$\mathbf{x}$, which would decrease some criterion without causing a simultaneous increase in at least one other criterion.

. (Pareto dominance[10]**):** A vector $\mathbf{u} = (u_1, ..., u_k)$ is said to **dominate** another vector $\mathbf{v} = (v_1, ..., v_k)$ (denoted by $\mathbf{u} \succeq \mathbf{v}$) iff $\mathbf{u}$ is partially greater than $\boldsymbol{v}$, i.e., $\forall i \in \{1, ..., k\}$,$u_i \geq v_i \wedge \exists i \in \{1, ..., k\} : \ u_i > v_i$.

After this two basic definitions, lets have a look at a simple example. Figure 3.1.7 shows some points of a two-dimensional solution space. For simplicity, the points will all lie within a circle with radius1 and center point of $(1, 1)$.

Figure 3.1.8 shows the Pareto front of a maximization problem. This means we are searching for solutions that try to maximize the x and y coordinate at the same time.

Figure 3.1.9 shows the Pareto front if we try to minimize the$x$ and y coordinate at the same time.

3.1.7.2 Implementing classes

The classes, used for solving multi-objective problems, reside in the `io.jenetics.ext.moea` package. Originally, the **Jenetics** library focuses on solving single-objective problems. This drives the design decision to force the return value of the fitness function to be `Comparable`. If the result type of the fitness function is a vector, it is no longer clear how to make the results comparable. **Jenetics** chooses to use the Pareto dominance relation (see section 3.1.7.1). The Pareto dominance relation, $\succ$, defines a strict partial order, which means $\succ$ is

[11] `https://en.wikipedia.org/wiki/Vilfredo_Pareto`

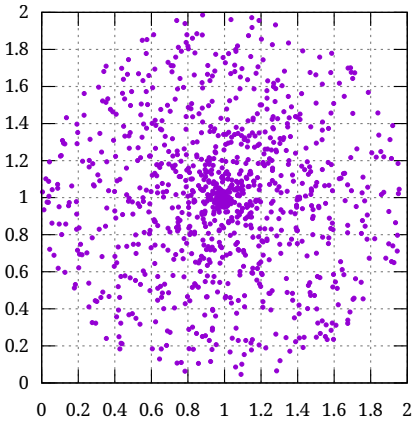


Figure 3.1.7: Circle points

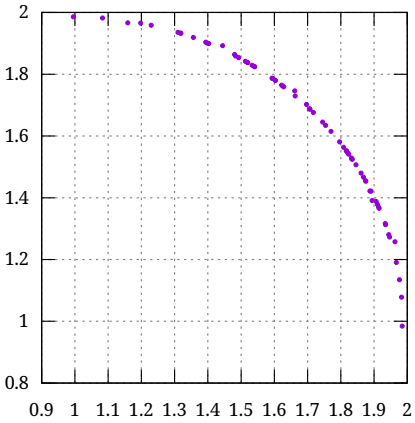


Figure 3.1.8: Maximizing Pareto front

1. irreflexive: $\mathbf{u} \not\succ \mathbf{u}$,
2. transitive: $\mathbf{u} \succ \mathbf{v} \wedge \mathbf{v} \succ \mathbf{w} \Rightarrow \mathbf{u} \succ \mathbf{w}$ and
3. asymmetric: $\mathbf{u} \succ \mathbf{v} \Rightarrow \mathbf{v} \not\succ \mathbf{u}$.

The `io.jenetics.ext.moea` package contains the classes needed for doing multi-objective optimization. One of the central types is the `Vec` interface, which allows you to wrap a vector of any element type into a `Comparable`.

```
1 public interface Vec<T> extends Comparable<Vec<T>> {
2     T data();
3     int length();
4     ElementComparator<T> comparator();
5     ElementDistance<T> distance();
6     Comparator<T> dominance();
7 }
```

Listing 3.10: `Vec` interface

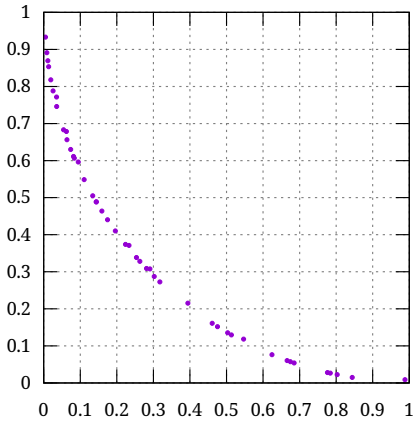


Figure 3.1.9: Minimizing Pareto front

Listing 3.10 shows the necessary methods of the `Vec` interface. These methods are sufficient to do all the optimization calculations. The `data()` method returns the underlying vector type, like `double[]` or `int[]`. With the `ElementComparator`, which is returned by the `comparator()` method, it is possible to compare single elements of the vector type `T`. This is similar to the `ElementDistance` function, returned by the `distance()` method, which calculates the distance of two vector elements. The last method, `dominance()`, returns the Pareto dominance comparator, $\succ$. Since it is quite a bothersome to implement all these needed methods, the `Vec` interface comes with a set of factory methods, which allows for creating `Vec` instance for some primitive array types.

```
final Vec<int[]> ivec = Vec.of(1, 2, 3);
final Vec<long[]> lvec = Vec.of(1L, 2L, 3L);
final Vec<double[]> dvec = Vec.of(1.0, 2.0, 3.0);
```

For efficiency reason, the primitive arrays are not copied, when the `Vec` instance is created. This lets you, theoretically, change the value of a created `Vec` instance, which will lead to unexpected results.

Although the `Vec` interface extends the `Comparable` interface, it violates its general contract. It only implements the Pareto dominance relation, which defines a partial order. Trying to sort a list of `Vec` objects, might lead to an exception (thrown by the sorting method) at runtime.

The second difference to the single-objective setup is the `EvolutionResult` collector. In the single-objective case, we will only get one *best* result, which is different in the multi-object optimization. As we have seen in section 3.1.7.1, we no longer have only one result, we have a set of Pareto optimal solutions. There is a predefined collector in the `io.jenetics.ext.moea` package, `MOEA::toParetoSet(IntRange)`, which collects the Pareto optimal `Phenotypes` into an `ISeq`.

```
final ISeq<Phenotype<DoubleGene, Vec<double[]>>> paretoSet =
```

```
    engine.stream()
        .limit(100)
        .collect(MOEA.toParetoSet(new IntRange(30, 50)));
```

Since there exists a potential infinite number of Pareto optimal solutions, you have to define desired number of set elements. This is done with an `IntRange` object, where you can specify the minimal and maximal set size. The example above will return a Pareto size with size in the range of $[30, 50)$. For reducing the Pareto set size, the distance between two vector elements is taken into account. Points which lie very close to each other are removed. This leads to a result, where the Pareto optimal solutions are, more or less, evenly distributed over the whole Pareto front. The *crowding distance*[12] measure is used for calculating the proximity of two points and it is described in [10] and [18].

Till now we have described the multi-objective result type (`Vec`) and the final collecting of the Pareto optimal solution. So lets create a simple multi-objective problem and an appropriate `Engine`.

```
final Problem<double[], DoubleGene, Vec<double[]>> problem =
    Problem.of(
        v -> Vec.of(v[0]*cos(v[1]) + 1, v[0]*sin(v[1]) + 1),
        Codecs.ofVector(
            new DoubleRange(0, 1),
            new DoubleRange(0, 2*PI)
        )
    );

final Engine<DoubleGene, Vec<double[]>> engine =
    Engine.builder(problem)
        .offspringSelector(new TournamentSelector<>(4))
        .survivorsSelector(UFTournamentSelector.ofVec())
        .build();
```

The fitness function in the example problem above will create 2D-points which will all lies within a circle with a center of $(1, 1)$. In figure 3.1.8 you can see how the resulting solution will look like. There is almost no difference in creating an evolution `Engine` for single- or multi-objective optimization. You only have to take care to choose the right `Selector`. Not all `Selectors` will work for multi-objective optimization. This include all `Selectors` which needs a `Number` fitness type and where the population needs to be sorted[13]. The `Selector` which works fine in a multi-objective setup is the `TournamentSelector`. Additionally you can use one of the special MO selectors: `NSGA2Selector` and `UFTournamentSelector`.

NSGA2 selector This selector selects the first elements of the population, which has been sorted by the Crowded-comparison operator (equation 3.1.1), $\underset{n}{\succ}$, as described in [15]

$$i \underset{n}{\succ} j \quad \text{if} \quad (i_{rank} < j_{rank}) \vee ((i_{rank} = j_{rank}) \wedge i_{dist} > j_{dist}), \qquad (3.1.1)$$

[12]The crowding distance value of a solution provides an estimate of the density of solutions surrounding that solution. The crowding distance value of a particular solution is the average distance of its two neighboring solutions.`https://www.igi-global.com/dictionary/crowding-distance/42740`.

[13]Since the$\succ$ relation doesn't define a total order, sorting the population will lead to an`IllegalArgumentException` at runtime.

where i_{rank} denotes the non-domination rank of individual i and i_{dist} the crowding distance of individual$i$.

Unique fitness tournament selector The selection of unique fitnesses lifts the selection bias towards over represented fitnesses by reducing multiple solutions sharing the same fitness to a single point in the objective space. It is therefore no longer required to assign a crowding distance of zero to individual of equal fitness as the selection operator correctly enforces diversity preservation by picking unique points in the objective space.[18]

Since the multi-objective optimization (MOO) classes are an extensions to the existing evolution `Engine`, the implementation doesn't exactly follow an established algorithm, like NSGA2 or SPEA2. The results and performance, described in the relevant papers, are therefore not directly comparable. See listing1.2 for comparing the **Jenetics** evolution flavor.

3.1.7.3 Termination

Most of the existing termination strategies, implemented in the `Limits` class, presumes a total order of the fitness values. This assumption holds for single-objective optimization problems, but not for multi-objective problems. Only termination strategies which don't rely on the total order of the fitness value, can be safely used. The following termination strategies can be used for multi-objective problems:

- `Limits::byFixedGeneration`,
- `Limits::byExecutionTime` and
- `Limits::byGeneConvergence`.

All other strategies doesn't have a well defined termination behavior.

3.1.7.4 Mixed optimization

Till now, we have only considered MOO problems, where all objectives where either minimized or maximized. This property might be to restrictive for some problem classes. If you have MOO problem with three objectives, for example, where objective one and three must be minimized and objective two has to be maximized, you need some additional mechanisms for doing this. Defining the optimization direction of the `Engine` is not sufficient. The fitness result `Vec` has to be configured accordingly. This can be done by using the most generic factory method of the `Vec` interface. Since this is quite bothersome, the `VecFactory` can be used for this task. Listing 3.11 shows the main method of the interface. The additional static factory methods has been omitted.

```
1 @FunctionalInterface
2 public interface VecFactory<T> {
3     Vec<T> newVec(final T array);
4 }
```

Listing 3.11: `VecFactory` interface

Instead of creating the solution `Vec` instances directly, the fitness function must create it with a properly configured `VecFactory` instance.

```
final VecFactory<double[]> factory = VecFactory.ofDoubleVec(
    Optimize.MINIMUM,
    Optimize.MAXIMUM,
    Optimize.MINIMUM
);

Vec<double[]> fitness(final double[] point) {
    final double x = point[0];
    final double y = point[1];
    return factory.newVec(new double[] {
        sin(x)*y,
        cos(y)*x,
        x + y
    });
}
```

The example code above shows how the `VecFactory` must be configured to create `Vec<double[]>` objects with the desired optimization properties. In the fitness function you will then use the `VecFactory` instance for creating the fitness values instead of the `Vec::of(double...)` factory method. The optimization direction of the evolution `Engine` will remain at its default value, `Optimize.MAXIMUM`. If you configure the `Engine` for minimization, the configured optimization directions in the `VecFactory` will be reversed. That means, the first objective will be maximized instead of minimized, and so on.

3.1.8 Grammatical evolution

One of the difficulties when defining a codec[14] for a given problem, is the creation of invalid solution candidates or individuals. Using constraints is one way of handling this problem, which is described in section 2.5 on page 68. Another possibility is to assign bad fitness values to invalid individuals. This might work to some degree, but at the cost of additional CPU and evolution time. In the best case, your codec is creating no invalid solutions at all. Grammatical evolution (GE) can help you to achieve this. GE was introduced by Michael O'Neill and Conor Ryan[38, 33, 39] for solving genetic programming (GP)[15] problems. Although the initial realm for GE was creating/evolving programs in arbitrary languages, it is not restricted to this area. GE can be used for every problems, where the problem domain can be expressed as *sentences* of a context-free language (CFL)[16].

GE requires an additional *mapping* step for creating a sentence (program) from a given CFG. Diagram 3.1.10 on the next page shows the principal decoding process of GE. A `BitChromosome` or `IntegerChromosome` determines the rules which are used for creating the sentences. The grammar is usually given in Backus-Naur form (BNF). The result is the construction of a syntactically correct program from a binary string which can then be evaluated by a fitness function. In the following sections, the building blocks of GE are described in a greater detail.

[14] See section2.3 on page 60.
[15] See section3.2 on page 116.
[16] `https://en.wikipedia.org/wiki/Context-free_language`

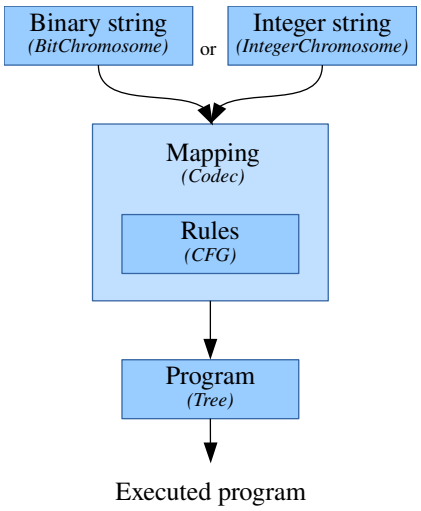


Figure 3.1.10: Grammatical evolution

3.1.8.1 Context-free grammar

A context-free grammar (CFG)[17][19] basically consists of a finite set of grammar rules. The grammar rules consists of two kind of symbols: 1) the terminals, which are the symbols of the alphabet underlying the languages, and 2) the *non*-terminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \to \alpha$, where A is a non-terminal, and α is a terminal or non-terminal symbol. CFGs are used to *generate* strings (sentences), rather than *recognize* strings.

. (CFG): A context-free grammar, G, is defined by the tuple $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$, where

1. $\mathcal{N}$ is a finite set of *non-terminal* symbols called the *variables*. Every symbol represents a different type of phrase in the sentence.
2. $\mathcal{T}$ is a finite set of *terminal* symbols, not element of $\mathcal{N}$. Terminals make up the actual content of the sentences defined by G. The terminals are the alphabet of the language.
3. $\mathcal{R}$ is a finite relation in$\mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$. Members of $\mathcal{R}$ are the rewrite rules of the grammar.
4. $\mathcal{S} \in \mathcal{N}$ is the start symbol and represents the whole sentence.

3.1.8.2 Backus-Naur form

The Backus-Naur form (BNF)[18] is a traditional form to represent a CFG. It is used to formally define the grammar of a language, so that there is no ambiguity as to what is allowed and what is not. BNF uses a range of symbols and

[17] `https://en.wikipedia.org/wiki/Context-free_grammar`
[18] `https://en.wikipedia.org/wiki/Backus-Naur_form`

expressions to create production rules. The names of the production rules are put within angle brackets, `<...>`, and the alternatives are separated by vertical bars, `|`. A simple BNF production rule might look like this:

```
<num> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

In this example `num` is the name of the production rule and the values `1..9` represent the terminal symbols, associated with the rule. If a non-terminal symbol appears on the right hand side, it means that there will be another production rule (or a set of rules) to define its replacement.

```
<expr> ::= <num> | <var> | <expr> <op> <expr>
```

This shows that `exp` is either a `num` or `var` or two `expr`, concatenated by an `op`. All components are non-terminal, therefore further production rules are required.

```
<op>  ::= + | - | * | /
<var> ::= x | y
```

This two rules defines four `op` symbols, $\{\mathtt{x}, -, *, /\}$, and two `var` terminals, $\{\mathtt{x}, \mathtt{y}\}$. Production rules for the syntax of a language might come as a large set of BNF statements that specify how every aspect of the language is defined. Every non-terminal symbol on the right hand side of a production rule, must have a rule that has the symbol on the left side. This continues until everything can be specified in relation to terminal symbols. Here is the whole grammar, which will define a language for simple arithmetic expressions:

```
<expr> ::= <num> | <var> | <expr> <op> <expr>
<op>   ::= + | - | * | /
<var>  ::= x | y
<num>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The symbol of the first rule, `expr`, will automatically serve as start symbol of the grammar, G, defined by the given BNF.

3.1.8.3 Sentence generation

After we have defined the CFG, we want to create a valid (random) sentences from our grammar. To do so, we will introduce a numbering schema for the alternative expressions for every rule. Such a schema, for our simple arithmetic grammar is shown below:

```
              [0]     [1]              [2]
<expr> ::= <num> | <var> | <expr> <op> <expr>
            [0]  [1]  [2]  [3]
<op>   ::= + | - | * | /
           [0]  [1]
<var>  ::= x | y
           [0]  [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]
<num>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Algorithm 3.1 Leftmost derivation

1. $L \leftarrow [\mathcal{S}]$
   Initialize the symbol list, L, with the start symbol, $\mathcal{S}$.
2. $n \leftarrow L[\min\{i\}] \in \mathcal{N}$
   Pick the leftmost non-terminal symbol, $L[\min\{i\}]$, from the current sentence list, L.
3. $E \leftarrow R(n)\,[\text{rand}]$
   Get the rule, $R(n)$, for the chosen non-terminal, n, and select a random rule alternative, E.E will contain one of more terminal and/or non-terminal symbols.
4. $L[\min\{i\}] \leftarrow E$
   Replace the chosen variable, n, with the selected symbols, E.
5. *Repeat steps* 2..4 *until the symbol list, L, contains only terminals.*

The following example shows snapshots of the symbol list, L, for a possible sentence creation run, according to the described leftmost derivation algorithm 3.1.

1. `L = [<expr>]`. Initialize the sentence list with the start symbol.
2. `L = [<expr>, <op>, <expr>]`. Select the first non-terminal symbol in the list,`<expr>`, and replace it with an randomly chosen rule alternative. Alternative `[2]`, `<expr> <op> <expr>`, is chosen and replaces the variable `<expr>`.
3. `L = [<num>, <op>, <expr>]`. Replacing the first `<expr>` with alternative `[0]`, `<num>`.
4. `L = [5, <op>, <expr>]`. Replacing `<num>` with 5.
5. `L = [5, -, <expr>]`
6. `L = [5, -, <var>]`
7. `L = [5, -, x]`

After the 6^{th} iteration the symbol list contains only terminal symbols and the generation process stops. If we convert the symbol list into a string, we get an algebraic expression, $5 - x$, which is an element of the CFL, defined by our simple CFG. Algorithm 3.2 on the next page shows the rightmost derivation method, which is a variation of the leftmost derivation algorithm. The only difference is, that instead of the processing the symbol list from left to right, it is processed from right to left.

3.1.8.4 Mapping

Now we have everything in place to do the last step. How to create a sentence from a given grammar? The original paper [38] uses a bit string for creating

Algorithm 3.2 Rightmost derivation

1. $L \leftarrow [\mathcal{S}]$
   Initialize the symbol list,L, with the start symbol, $\mathcal{S}$.
2. $n \leftarrow L[\max\{i\}] \in \mathcal{N}$
   Pick the rightmost non-terminal symbol, $L[\max\{i\}]$, from the current sentence list, L.
3. $E \leftarrow R(n)$ [rand]
   Get the rule, $R(n)$, for the chosen non-terminal, n, and select a random rule alternative, E. E will contain one of more terminal and/or non-terminal symbols.
4. $L[\min\{i\}] \leftarrow E$
   Replace the chosen variable, n, with the selected symbols, E.
5. *Repeat steps 2..4 until the symbol list, L, contains only terminals.*

a sentence from a given chromosome and called this process *mapping*. Since we have already described how to create a sentence from a grammar, we must describe the last missing part of the process, and this missing part is the selection of alternative symbols from a given rule. This is step three in the described algorithms 3.1 on the preceding page and 3.2. Instead of selecting a *random* alternative, the index of the selected alternative must be determined by the given chromosome.

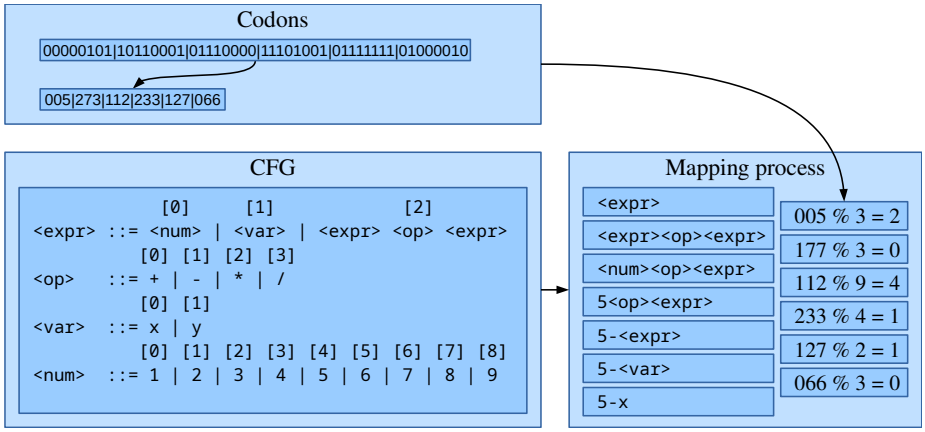


Figure 3.1.11: GE mapping

Figure 3.1.11 shows he GE mapping process. The binary string is split into 8-bit junks and interpreted as unsigned integers, with vales in the range of $[0, 256)$. This 8-bit junks are called *codons*. Whenever a new rule alternative must be selected, a new codon is read from the chromosome. A simple modulo operation is used to get an index within the desired range. If more codons are needed during the mapping process than available in the input chromosome, the reading of the codons is wrapped over and starts at the beginning of the

chromosome again.

3.1.8.5 Implementing classes

This section describes the main classes which are part of the implementation of the GE. All the described classes are part of the `io.jenetics.ext` module and reside in the `io.jenetics.ext.grammar` package.

`Cfg` This class represents a context-free grammar as described in section 3.1.8.1 on page 108.

`Bnf` Although it is possible to create instances of the `Cfg` class manually, it is in most cases easier to define it as BNF string. This class contains methods for parsing and formatting CFGs from and to BNF strings.

`SymbolIndex` The `SymbolIndex` interface defines the strategy, which alternative symbols of a given rule are selected during the sentence generation process. The classical algorithm uses codons, encoded in an binary string.

`Codons` This class is an implementation of the classical symbol selection algorithms as described in the original paper. It implements the `SymbolIndex` interface.

`Generator` The `Generator` interface is an abstraction of the sentence generation algorithm as such as algorithm 3.1 on page 110 or 3.2 on the previous page.

`Mappings` The `Mappings` class contains factory methods for creating different mapping strategies. Since the mapping process, as shown in figure 3.1.11 on the preceding page, is implemented as `Codec`[19], the factory methods will return `Codec` instances for the `BitChromosome` and `IntegerChromosome`.

Let's start with the `Cfg` class and how to create grammars. The `Cfg` class comes with a set of factory methods, which lets you create `Cfg` objects quite easily. Our already known arithmetic expression grammar can be created with the following code. The used factory methods, `N`, `T`, `E`, `R`, were statically imported.

```
final var cfg = Cfg.<String>builder()
    .R("expr", rule -> rule
        .N("num", "some annotation")
        .N("var", "some other annotation")
        .E(exp -> exp
            .T("(")
            .N("expr").N("op", 4).N("expr")
            .T(")")))
    .R("op", rule -> rule.T("+").T("-").T("*").T("/"))
    .R("var", rule -> rule.T("x").T("y"))
    .R("num", rule -> rule
        .T("0").T("1").T("2").T("3").T("4")
        .T("5").T("6").T("7").T("8").T("9")
    )
    .build();
```

[19] See section2.3 on page 60.

As alternative, the above CFG can also been creating via a string in BNF form, as described in section 3.1.8.2 on page 108.[20] The code snippet below shows how to do this.

```
final Cfg<String> cfg = Bnf.parse("""
    <expr> ::= <num> | <var> | <expr> <op> <expr>
    <op>   ::= + | - | * | /
    <var>  ::= x | y
    <num>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    """
);
```

The `SymbolIndex` interface is responsible for selecting the index of the rule alternative to choose. The reason for this interface is to decouple the selection algorithm from the data structure, which determines the selection; it should decouple the algorithm from the underlying `BitChromosome` or `IntegerChromosome`.

```
@FunctionalInterface
public interface SymbolIndex {
    int next(Cfg.Rule<?> rule, int bound);
}
```

Listing 3.12: `SymbolIndex` interface

Listing 3.12 shows the single method of the `SymbolIndex` interface. It takes the rule, for which to select the alternatives and the desired index bounds and returns the selected symbol index. The `rule` parameter of the next function allows to use different index selection strategies, or different index sources, for the different rules. This interface makes it possible to let the sentence creation to be controlled by an `IntegerChromosome` instead the classical bit string.

The `Codons` class implements the `SymbolIndex` interface and can be created from a `BitChromosome` and `IntegerChromosome` instances. `Codons` created from a `BitChromosome` gives you an implementation of the classic symbol selection algorithm.

```
// Create codons backed up by binary string.
var bch = BitChromosome.of(100*8);
var bcodons = Codons.ofBitGenes(bch);
// Creadte codons backed up by integer string.
var ich = IntegerChromosome.of(new IntRange(0, 256), 100);
var icodons = Codons.ofIntegerGenes(ich);
```

The code snipped above shows two ways for creating essentially the same codons. In the first variant, the codons are read from a `BitChromosome` with the length of 800, which results in 100 different indexes readable from the created `Codons` object. In the second variant an `IntegerChromosome`, with a value range of $[0, 256)$ and a length of 100, is used for the `Codons` object. Using an `IntegerChromosome` instead of a `BitChromosome` gives you a greater flexibility, as it allows you to specify an explicit range for the codon values.

Implementations of the functional `Generator` interface are responsible for creating sentences from a given grammar. It takes a `Cfg` object as input and creates a generic result object of type `T`. This genericity lets you use the same interface for different result type, like `List<Symbol<String>>` or `Tree<Symbol<String>, ?>`.

[20] The CFG, created by the BNF representation doesn't allow to add annotations to the CFG elements. This is only possible when using the `Cfg.Builder`. Such annotations can be used by the sentence generator.

```
@FunctionalInterface
public interface Generator<T, R> {
    R generate(Cfg<? extends T> cfg);
}
```

Listing 3.13: `Generator` interface

Listing 3.13 shows the definition of the `Generator` interface. It takes a `Cfg` object as input and returns the created result sentence of type `R`. The kind of the created result is not determined by interfaces. It can be a random instance of the input grammar, a reproducible, deterministic result, or even always the same result. It is the responsibility of the implementations to determine which kind of result to return. The reason for this is, not make assumptions about the sentence creation and bake this assumptions into the API; this avoid a leaky abstraction.

There are currently two implementation of the `Generator` interface: 1) the `SentenceGenerator`, which generates a list of terminal symbols (sentences) from a given grammar, and 2) the `DerivationTreeGenerator`, generating a derivation tree, or parse tree.

```
// Define sentence generator for creating random sentences.
var generator = new SentenceGenerator<String>(
    SymbolIndex.of(RandomGenerator.getDefault()),
    1_000
);
// Create random sentence for given CFG.
List<Cfg.Terminal<String>> sentence = generator.generate(cfg);
String string = sentence.stream()
    .map(Cfg.Symbol::name)
    .collect(Collectors.joining());
```

In the code snippet above you can see how to create a `SentenceGenerator` which will create random sentences with a maximal length of 1000. The generated sentence will be a list of terminal symbols. But it is quite easy to convert it to a string. The `DerivationTreeGenerator` work can be used similarly. Instead of a list of terminals, it creates a `Tree<Symbol<String>, ?>` instead.

The factory methods of the `Mappings` class puts it all together and let you easily specify the needed parts for the mapping process. **Jenetics** already has an interface for expressing such a mapping process, the `Codec`[21] interface. So no additional interface or class was introduce and the `Mappings` factories will return `Codec` instances instead. In the code snippet below you can see how to create the classical mapping, with a single `BitChromosome` used for the rule symbol selection. It is created via the `Mappers::singleBitChromosomeMapper` method.

```
final Codec<List<Terminal<String>>, BitGene> codec =
    Mappers.singleBitChromosomeMapper(
        cfg,
        // Length of the used BitChromosome.
        100*8,
        // The used generator, created from SymbolIndex.
        index -> new SentenceGenerator<>(index, 1_000)
    );
```

For creating this codec you must specify the CFG, the length of the chromosome and the sentence `Generator`. The `Generator` is given as factory function, which

[21] See section2.3 on page 60.

takes an `SymbolIndex` at input. The codec, created in the code snippet below, is essentially the same as created with the `Mappers::singleBitChromosomeMapper` method. It only differs in the way the codons are created. Using an `IntegerChromosome` as codons source gives you a greater flexibility and allows to change the value range of the created codons.

```
final Codec<List<Terminal<String>>, IntegerGene> codec =
    Mappers.singleIntegerChromosomeMapper(
        cfg,
        // Value range of created codons.
        new IntRange(0, 256),
        // Length of the used IntegerChromosome.
        100,
        index -> new SentenceGenerator<>(index, 1_000)
    );
```

Beside the classical mapping algorithms, **Jenetics** contains also a novel method for genotype to sentence mapping. This new approach uses a separate `IntegerChromosome` for every `Cfg.Rule`. This allows to align the range of the chromosome with the number of alternatives of the rule. With this property, no modulo operation is needed for creating codons with the correct bounds. No additional modulo operation means, that all alternatives have the same probability for being selected, for randomly created chromosomes. All rule alternatives have the same changes of being selected. There is no accidental bias towards a given rule alternative or symbol.

```
final Codec<List<Terminal<String>>, IntegerGene> codec =
    Mappers.multiIntegerChromosomeMapper(
        cfg,
        // Chromosome length depends of number of alternatives.
        rule -> new IntRange(rule.alternatives().size()*25),
        index -> new SentenceGenerator<>(index, 1_000)
    );
```

The codes snippet above shows how to create a mapping, which uses a genotype with one chromosome for every rule in the given CFG. Our example CFG

```
                [0]     [1]           [2]
<expr> ::= <num> | <var> | <expr> <op> <expr>
              [0] [1] [2] [3]
<op>   ::= + | - | * | /
              [0] [1]
<var>  ::= x | y
              [0] [1] [2] [3] [4] [5] [6] [7] [8]
<num>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

will use a `Genotype` with the following structure for the encoding of the codons.

```
Genotype.of(
    IntegerChromosome.of(new IntRange(0, 3), 3*25), // <expr>
    IntegerChromosome.of(new IntRange(0, 4), 4*25), // <op>
    IntegerChromosome.of(new IntRange(0, 2), 2*25), // var
    IntegerChromosome.of(new IntRange(0, 9), 9*25)  // <num>
);
```

As you can see, the number range for each rule chromosome reflects the number of available rule alternatives and the chromosome length can be expressed as a multiple of the number of alternatives of the corresponding rule. If you don't

need this flexibility, it is of course also possible to use constant chromosome lengths. Just use a length function like this: `rule -> new IntRange(1_000)`.

3.2 io.jenetics.prog

In artificial intelligence, *genetic programming* (GP) is a technique whereby computer programs are encoded as a set of genes that are then modified (evolved) using an evolutionary algorithm (often a genetic algorithm).[22] The `io.jenetics.prog` module contains classes which enables the **Jenetics** library doing GP. It introduces a `ProgramGene` and `ProgramChromosome` pair, which serves as the main data structure for genetic programs. A `ProgramGene` is essentially a tree (AST[23]) of operations (`Op`) stored in a `ProgramChromosome`.[24]

3.2.1 Operations

When creating own genetic programs, it is not necessary to derive classes from the `ProgramGene` or `ProgramChromosome`. The intended extension point is the `Op` interface.

The extension point for own GP implementations is the `Op` interface. There is in general no need for extending the `ProgramChromosome` class.

```
1 public interface Op<T> {
2     String name();
3     int arity();
4     T apply(T[] args);
5 }
```

Listing 3.14: GP `Op` interface

The generic type of the `Op` interface (see listing 3.14) enforces the data type constraints for the created program tree and makes the implementation a *strongly typed* GP. Using the `Op.of` factory method, a new operation is created by defining the desired operation function.

```
1 final Op<Double> add = Op.of("+", 2, v -> v[0] + v[1]);
2 final Op<String> concat = Op.of("+", 2, v -> v[0] + v[1]);
```

A new `ProgramChromosome` is created with the operations suitable for our problem. When creating a new `ProgramChromosome`, we must distinguish two different kind of operations:

1. *Non-terminal* operations have an arity greater than zero, which means they take at least one argument. These operations need to have child nodes, where the number of children must be equal to the arity of the

[22] `https://en.wikipedia.org/wiki/Genetic_programming`
[23] `https://en.wikipedia.org/wiki/Abstract_syntax_tree`
[24] When implementing the GP module, the emphasis was to not create a parallel world of genes and chromosomes. It was a requirement, that the existing Alterer and Selector classes could also be used for the new GP classes. This has been achieved by flattening the AST of a genetic program to fit into the 1-dimensional (flat) structure of a chromosome.

operation of the parent node. Non-terminal operations will be abbreviated to *operations*.

2. *Terminal* operations have an arity of zero and from the leaves of the program tree. Terminal operations will be abbreviated to *terminals*.

The `io.jenetics.prog` module comes with three predefined terminal operations: `Var`, `Const` and `EphemeralConst`.

`Var` The `Var` operation defines a variable of a program, which is set from outside when it is evaluated.

```
final Var<Double> x = Var.of("x", 0);
final Var<Double> y = Var.of("y", 1);
final Var<Double> z = Var.of("z", 2);
final ISeq<Op<Double>> terminals = ISeq.of(x, y, z);
```

The terminal operations defined in the listing above can be used for defining a program which takes a 3-dimensional vector as input parameters, x, y, and z, with the argument indices 0, 1, and 2. If you have again a look at the `apply` method of the operation interface, you can see that this method takes an object array of type `T`. The variable x will return the first element of the input arguments, because it has been created with index 0.

`Const` The `Const` operation will always return the same, constant value when evaluated.

```
final Const<Double> one = Const.of(1.0);
final Const<Double> pi = Const.of("PI", Math.PI);
```

You can create a constant operation in to flavors: with a value only, and with a dedicated name. If a constant has a name, the symbolic name is used, instead of the value, when the program tree is printed.

`EphemeralConst` An ephemeral constant is a terminal operation, which encapsulates a value that is generated at run time from the `Supplier` it is created from. Ephemeral constants allows you to have terminals that don't have all the same values. To create an ephemeral constant that takes its random value in $[0, 1)$ you will write the following code.

```
final Op<Double> rand1 = EphemeralConst
    .of(RandomRegistry.random()::nextDouble);
final Op<Double> rand2 = EphemeralConst
    .of("R", RandomRegistry.random()::nextDouble);
```

The ephemeral constant value is determined when it is inserted in the tree and never changes until it is replaced by another ephemeral constant.

3.2.2 Program creation

The `ProgramChromosome` comes with some factory methods, which lets you easily create program trees with a given depth and a given set of operations and terminals.

```
final int depth = 5;
final ISeq<Op<Double>> operations = ISeq.of(...);
final ISeq<Op<Double>> terminals = ISeq.of(...);
final ProgramChromosome<Double> program = ProgramChromosome
    .of(depth, operations, terminals);
```

The code snippet above will create a *perfect* program tree[25] of depth 5. All non-leaf nodes will contain operations, randomly selected from the given operations, whereas all leaf nodes are filled with operations from the terminals.

The created program tree is *perfect*, which means that all leaf nodes have the same *depth*. If new trees needs to be created during evolution, they will be created with the *depth*, *operations* and *terminals* defined by the *template* program tree.

During the evolution phase, the size of the `ProgramChromosome` can grow and shrink. The `SingleNodeCrossover`, which is part of the `jenetics.ext` module is responsible for this change in the program size. When a smaller sub tree is exchanged with a bigger sub tree, the size of the first tree will grow and the size of the second tree will shrink. This can lead to undesirable large programs. Because of this reason, it is possible to create a `ProgramChromosome` with an additional *validation* predicate.

```
final ProgramChromosome<Double> program = ProgramChromosome.of(
    depth, ch -> ch.root().size() <= 50,
    operations, terminals
);
```

The predicate, `ch -> ch.root().size() <= 50`, marks all programs with more then 50 nodes as invalid. Invalid chromosomes will then be replaced by newly created one. When defining a validation predicate, you have to take care, that the desired depth and the validation predicate *matches*. If the given program tree depth is too big, e. g. 51, every newly created program will be immediately marked as invalid. This is because a tree with depth 51 will have for sure more than 50 nodes.

The evolution `Engine` used for solving GP problems is created the same way as for normal GA problems. Also the execution of the `EvolutionStream` stays the same. The first `Gene` of the collected final `Genotype` represents the evolved program, which can be used to calculate function values from arbitrary arguments.

```
final Engine<ProgramGene<Double>, Double> engine = Engine
    .builder(Main::error, program)
    .minimizing()
    .alterers(
        new SingleNodeCrossover<>(),
        new Mutator<>())
    .build();

final ProgramGene<Double> program = engine.stream()
    .limit(300)
    .collect(EvolutionResult.toBestGenotype())
```

[25] All leafs of a perfect tree have the same depth and all internal nodes have degree `Op.arity`.

```
12      .gene();
13  final double result = program.eval(3.4);
```

For a complete GP example have a look at the examples in chapter 6.7. The code example above also shows, that the program is represented by the first gene (aka root gene) of the `ProgramChromosome`. Since the `ProgramGene` implements the `Tree<Op<A>,ProgramGene<A>>` interface, it smoothly integrates with existing tree algorithms. Some possible program gene assignments are shown in the code snippet below, which will compile without warnings or additional casts.

```
1 final ProgramChromosome<Double> chromosome = ...;
2 assert chromosome.gene() == chromosome.root();
3 final ProgramGene<Double> program = chromosome.root();
4 final Tree<Op<Double>, ?> opTree = chromosome.root();
5 final Tree<?, ?> tree = chromosome.root();
```

3.2.3 Program repair

The specialized crossover class, `SingleNodeCrossover`, for a `TreeGene` guarantees that the program tree after the alter operation is still valid. It obeys the tree structure of the `Gene`. General alterers, not written for `ProgramGene` of `TreeGene` classes, will most likely destroy the tree property of the altered chromosome. There are essentially two possibility for handling invalid tree chromosomes:

1. Marking the `Chromosome` as invalid. This possibility is easier to achieve, but would also to lead to a large number of invalid `Chromosomes`, which must be recreated. When recreating invalid `Chromosomes` we will also lose possible solutions.

2. Trying to repair the invalid `Chromosome`. This is the approach the **Jenetics** library has chosen. The repair process reuses the operations in a `ProgramChromosome` and rebuilds the tree property by using the operation arity.

Jenetics allows for the usage of arbitrary `Alterer` implementations. Even alterers not implemented for ProgramGenes. Genes *destroyed* by such alterers are repaired.

3.2.4 Program pruning

When you are solving symbolic regression problems, the mathematical expression trees, created during the evolution process, can become quite big. From the diversity point of view, this might be not that bad, but it comes with additional computation cost. With the `MathRewriteAlterer` you are able to simplify some portion of the population in each generation. The rewrite alterer uses the *default* `TreeRewiter`[26] defined by the `MathExpr.REWRITER` field. It is also possible to create a `MathRewriteAlterer` instance with your own `TreeRewiter`.

[26] See section3.1.2 on page 93 for a detailed description of the implemented tree rewrite system.

```
final Engine<ProgramGene<Double>, Double> engine = Engine
    .builder(Main::error, program)
    .minimizing()
    .alterers(
        new SingleNodeCrossover<>(),
        new Mutator<>(),
        new MathRewriteAlterer<>(0.5))
    .build();
```

In the example above, half of the expression trees are simplified in each generation. If you want to prune the final result, you can do this with the `MathExpr::rewrite` method, which uses the `MathExpr.REWRITER` tree rewriter for the rewrite task.

```
final ProgramGene<Double> program = engine.stream()
    .limit(3000)
    .collect(EvolutionResult.toBestGenotype())
    .gene();

final TreeNode<Op<Double>> tree = TreeNode.ofTree(program);
MathExpr.rewrite(tree);
```

The algorithm used for pruning the expression tree, currently only uses some basic mathematical identities, like $x+0=x$, $x\cdot 1=x$ or $x\cdot 0=0$. More advanced simplification algorithms may be implemented in the future. The `MathExpr` helper class can also be used for creating mathematical expression trees from the *`usual`* textual representation.

```
final MathExpr expr = MathExpr
    .parse("5*z + 6*x + sin(y)^3 + (1 + sin(z*5)/4)/6");
final double value = expr.eval(5.5, 4, 2.3);
```

The variables in an expression string are sorted alphabetically. This means, that the expression is evaluated with $x = 5.5$, $y = 4$ and $z = 2.3$, which leads to a result value of 44.19673085074048.

3.2.5 Multi-root programs

The given examples, so far, where using a single `ProgramChromosome` for modeling the program. Since the `Genotype` is able to hold more than one `Chromosome`, it is possible to create more than one program root. These programs are evaluated concurrently.

```
final Codec<ISeq<Function<Double[], Double>>, ProgramGene<Double>>
codec = Codec.of(
    Genotype.of(
        // First 'program'.
        ProgramChromosome.of(
            4, ch -> ch.root().size() <= 30,
            operations, terminals
        ),
        // Second 'program'.
        ProgramChromosome.of(
            5, ch -> ch.root().size() <= 50,
            operations, terminals
        )
    ),
    gt -> gt.stream()
        .map(Chromosome::gene)
        .collect(ISeq.toISeq())
);
```

The code snippet above shows how to create a codec with two independent program roots. These programs are then mapped, in the fitness function, to the combined fitness value. It is also possible to use different operations and terminals for each `ProgramChromosome`.

3.2.6 Symbolic regression

Symbolic regression is a specific type of regression analyses, where the search space consists of mathematical expressions. The task is to find a model, which fits a given data set in terms of accuracy and simplicity. In a classical approach, you will try to optimize the parameters of a predefined function type, e. g. a polynomial of grade n:

$$f(x) = \sum_{k=0}^{n} a_k x^k.$$

The encoding would only be a `DoubleChromosome` of length $n + 1$, where the `Gene` at position $k \in [0, ..., n]$ represents the factor a_k of the polynomial. If the type of mathematical function is not known in advance, GP can be used finding a function which is composed out of a given set of *primitives*.

> Symbolic regression involves finding a mathematical expression, in symbolic form, that provides a good, best, or perfect fit between a given finite sampling of values of the independent variables and the associated values of the dependent variables.[24]

Since symbolic regression is quite a common task in GP, **Jenetics** comes with classes and interfaces, supporting the implementation of such problems. These classes are defined in the `io.jenetics.prog.regression` package. The following sections describes these classes and interfaces and its usage. A complete symbolic regression example is given in section 6.7.

3.2.6.1 Loss function

The loss function measures how good the evolved program (tree) predicts the expected outcome or data set. If the prediction deviates too much from the expected data, the loss function will cough up a larger number. Loss functions are classified into two major categories, depending on the type of the learning task—*regression* losses and *classification* losses. In the following paragraphs, only loss functions suitable for *regression* problems will be described.

Mean squared error The mean squared error is the default loss function used for regression analysis. It is also known as *quadratic* loss or *L2* loss and is calculated as the average of the squared differences between the predicted and actual values.

$$MSE = \frac{1}{n} \sum_{i=0}^{n-1} \left(y_i - \tilde{y_i}\right)^2, \tag{3.2.1}$$

where y_i denotes the expected function value and $\tilde{y_i}$ the calculated (estimated) value for data point, i. The result is always positive and the perfect value is 0. The squaring means that larger mistakes result in more errors than smaller mistakes, meaning that the model penalizes larger mistakes. The mean squared error is the preferred loss function for regression problems.

Mean absolute error The mean absolute error, also known as *L1* loss, is calculated as the average of the absolute difference between the expected and calculated values.

$$MAE = \frac{1}{n}\sum_{i=0}^{n-1}|y_i - \tilde{y}_i| \tag{3.2.2}$$

This loss function is suitable for regression problems where the distribution of the target variable may be mostly Gaussian, but may have outliers, e. g. large or small values far from the mean value. This means that the *MAE* is more robust than the *MSE*, which is useful if the sample data is corrupted with outliers.

The MAE is more robust to outliers, but its derivatives are not continuous, making it less efficient to find the correct solution. The MSE is sensitive to corrupt data, but finds more stable and closed form solutions.

The interface used for calculating the loss between *calculated* and *expected* values is shown in listing 3.15

```
1 @FunctionalInterface
2 public interface LossFunction<T> {
3     double apply(T[] calculated, T[] expected);
4 }
```

Listing 3.15: `LossFunction` interface

3.2.6.2 Complexity function

The complexity function measures the complexity of the evolved tree. If you have two programs with the *same* loss value, you usually want the *simpler* program to survive. A simple complexity measure is the number of nodes a program tree consists of. You can obtain such a measure by the `ofNodeCount(int)` factory method of the `Complexity` interface. The complexity measure, $C(P)$, is defined as

$$C(P) = 1 - \sqrt{1 - \frac{\min(N(P), N_{max})^2}{N_{max}^2}}, \tag{3.2.3}$$

where $N(P)$ is the number of nodes the program, P, consists of and N_{max} the maximal allowed program nodes. If the number of program nodes is equal or greater then the maximal node number, $C(P)$, will return 1.

The graph in figure 3.2.1 shows how the program complexity increases with the number of nodes. For the example graph the maximal node count was set to 28.

```
1 @FunctionalInterface
2 public interface Complexity<T> {
3     double apply(Tree<? extends Op<T>, ?> program);
4 }
```

Listing 3.16: `Complexity` interface

Listing 3.16 shows the interface which calculates the complexity measure of a given program tree.

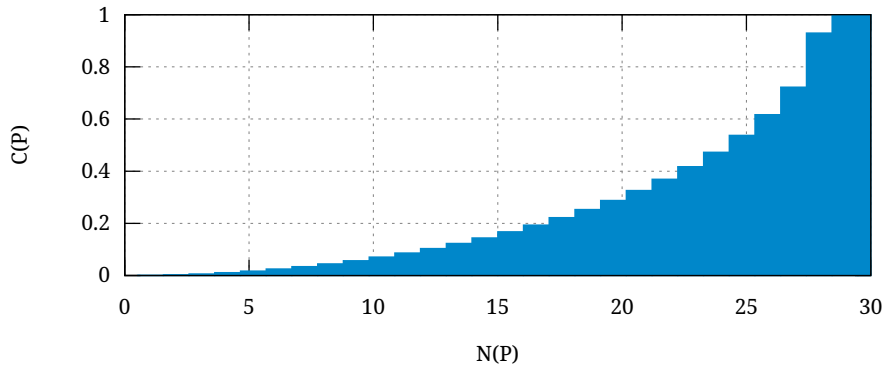


Figure 3.2.1: Node count complexity

3.2.6.3 Error function

The error function combines the loss function and the complexity function into one error measure. It is used as fitness function which the GP is minimizing.

```
@FunctionalInterface
public interface Error<T> {
    double apply(
        Tree<? extends Op<T>, ?> program,
        T[] calculated,
        T[] expected
    );
}
```

Listing 3.17: `Error` interface

Listing 3.17 shows the error (fitness) function used for evolving symbolic regression problems. Instead of implementing the error function from scratch, you will probably want to use one of the factory methods for creating it from one of the predefined `LossFunction` and `Complexity` measure.

```
final Error<Double> error1 = Error.of(LossFunction::mse);
final Error<Double> error2 = Error.of(
    LossFunction::mae,
    Complexity.ofNodeCount(28)
);
final Error<Double> error3 = Error.of(
    LossFunction::mse,
    Complexity.ofNodeCount(28),
    (loss, complexity) -> loss + loss*complexity
);
```

The code snippet above shows the three possibilities to create an error function by using the predefined loss functions and complexity measure. `error1` is created by using the mean squared error, MES.`error2` and `error3` defines the same error function. The only difference is that `error3` defines the loss-complexity composition function explicitly.

3.2.6.4 Sample points

Solving regression problems requires to compare the current solution (program tree) with a set of sample points, which represents the *original* function to be approximated. The `Sample` interface represents such sample point. It actually maps a n-dimensional point of domain, $\mathbb{D}$, to an one-dimensional point of the same domain: $\mathbb{D}^n \to \mathbb{D}$.

```
public interface Sample<T> {
    int arity();
    T argAt(int index);
    T result();
}
```

Listing 3.18: `Sample` interface

The arity of the sample point returns the dimension, n. To make it easier to create `double` sample points, some factory methods are also given in the `Sample` interface.

```
final Sample<Double> sample1 = Sample.ofDouble(0.0, 0.0);
final Sample<Double> sample2 = Sample.ofDouble(1.0, 1.0);
final Sample<Double> sample3 = Sample.ofDouble(2.0, 2.0);
```

The code snippet above shows how to create three sample points for a function $f : \mathbb{R} \to \mathbb{R}$.

3.2.6.5 Regression problem

The `Regression` class is the only concrete type of the public API of the `regression` package. It integrates the interfaces, described in the last sections, into one problem definition.

```
public final class Regression<T>
    implements Problem<Tree<Op<T>, ?>, ProgramGene<T>, Double>
{
    ...
}
```

As you can see in the code snippet above, the `Regression` class implements the `Problem` interface and can be therefore easily used in setting up an appropriate evolution `Engine`. A full such regression example can be found in section 6.7.

3.2.7 Boolean programs

The default data type for doing symbolic regression is the `Double` class. This is supported by a standard set of mathematical operations, defined in the `MathOp` class. Since the GP operations are not restricted to any particular type, the boolean operations, defined in the `BoolOp` class, can be used for defining boolean programs.

3.3 io.jenetics.xml

The `io.jenetics.xml` module allows for writing and reading `Chromosome`s and `Genotype`s to and from XML. Since the existing JAXB marshaling is part of the deprecated `javax.xml.bind` module the `io.jenetics.xml` module is

now the recommended for XML marshalling of the **Jenetics** classes. The XML marshalling, implemented in this module, is based on the Java `XMLStreamWriter` and `XMLStreamReader` classes of the `java.xml` module.

3.3.1 XML writer

The main entry point for writing XML files is the typed XML`Writer` interface. Listing 3.19 shows the interface of the XML`Writer`.

```
@FunctionalInterface
public interface Writer<T> {
    void write(XMLStreamWriter xml, T data)
        throws XMLStreamException;

    static <T> Writer<T> attr(String name);
    static <T> Writer<T> attr(String name, Object value);
    static <T> Writer<T> text();

    static <T> Writer<T>
    elem(String name, Writer<? super T>... children);

    static <T> Writer<Iterable<T>>
    elems(Writer<? super T> writer);
}
```

Listing 3.19: XML`Writer` interface

Together with the static `Writer` factory method, it is possible to define arbitrary writers through composition. There is no need for implementing the `Writer` interface. A simple example will show you how to create (compose) a `Writer` class for the `IntegerChromosome`. The created XML should look like the given example above.

```
<int-chromosome length="3">
    <min>-2147483648</min>
    <max>2147483647</max>
    <alleles>
        <allele>-1878762439</allele>
        <allele>-957346595</allele>
        <allele>-88668137</allele>
    </alleles>
</int-chromosome>
```

The following writer will create the desired XML from an integer `Chromosome`. As the example shows, the structure of the XML can easily be grasped from the XML writer definition and vice versa.

```
final Writer<IntegerChromosome> writer =
    elem("int-chromosome",
        attr("length").map(ch -> ch.length()),
        elem("min", Writer.<Integer>text().map(ch -> ch.min())),
        elem("max", Writer.<Integer>text().map(ch -> ch.max())),
        elem("alleles",
             elems("allele", Writer.<Integer>text())
                 .map(ch -> ch.toSeq().map(g -> g.allele()))
        )
     );
```

3.3.2 XML reader

Reading and writing XML files uses the same concepts. For reading XML there is an abstract `Reader` class, which can be easily composed. The main method

of the Reader class can be seen in listing 3.20.

```
public abstract class Reader<T> {
    public abstract T read(final XMLStreamReader xml)
        throws XMLStreamException;
}
```

Listing 3.20: XML`Reader` class

When creating a XML`Reader`, the structure of the XML must be defined in a similar way as for the XML`Writer`. Additionally, a factory function, which will create the desired object from the extracted XML data, is needed. A `Reader`, which will read the XML representation of an `IntegerChromosome` can be seen in the following code snippet below.

```
 final Reader<IntegerChromosome> reader =
     elem(
         (Object[] v) -> {
             final int length = (int)v[0];
             final int min = (int)v[1];
             final int max = (int)v[2];
             final List<Integer> alleles = (List<Integer>)v[3];
             assert alleles.size() == length;
             return IntegerChromosome.of(
                 alleles.stream()
                     .map(value -> IntegerGene.of(value, min, max)
                     .toArray(IntegerGene[]::new)
             );
         },
         "int-chromosome",
         attr("length").map(Integer::parseInt),
         elem("min", text().map(Integer::parseInt)),
         elem("max", text().map(Integer::parseInt)),
         elem("alleles",
             elems(elem("allele",text().map(Integer::parseInt)))
         )
     );
```

3.3.3 Marshalling performance

Another important aspect when doing marshalling, is the space needed for the marshaled objects and the time needed for doing the marshalling. For the performance tests a genotype with a varying chromosome count is used. The used genotype template can be seen in the code snippet below.

```
final Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0, 1.0, 100),
    chromosomeCount
);
```

Table 3.3.1 shows the required space of the marshaled genotypes for different marshalling methods: (a) Java serialization, (b) JAXB[27] serialization and (c) XML`Writer`.

Using the Java serialization will create the smallest files and the XML`Writer` of the `io.jenetics.xml` module will create files roughly 75% the size of the JAXB serialized genotypes. The size of the marshaled objects also influences

[27]The JAXB marshalling has been removed in version 4.0. It is still part of the table for comparison with the new XML marshalling.

Chromosome count	Java serialization	JAXB	XML writer
1	0.0017 MiB	0.0045 MiB	0.0035 MiB
10	0.0090 MiB	0.0439 MiB	0.0346 MiB
100	0.0812 MiB	0.4379 MiB	0.3459 MiB
1000	0.8039 MiB	4.3772 MiB	3.4578 MiB
10000	8.0309 MiB	43.7730 MiB	34.5795 MiB
100000	80.3003 MiB	437.7283 MiB	345.7940 MiB

Table 3.3.1: Marshaled object size

the write performance. As you can see in diagram 3.3.1 the Java serialization is the fastest marshalling method, followed by the JAXB marshalling. The XML`Writer` is the slowest one, but still comparable to the JAXB method.

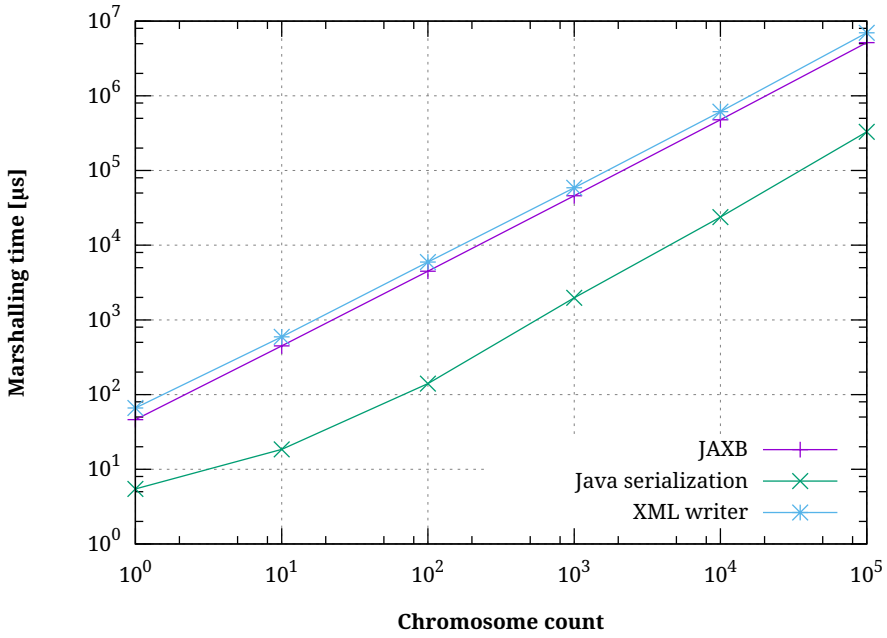


Figure 3.3.1: Genotype write performance

For reading the serialized genotypes, we will see similar results (see diagram 3.3.2). Reading Java serialized genotypes has the best read performance, followed by JAXB and the XML Reader. This time the difference between JAXB and the XML Reader is hardly visible.

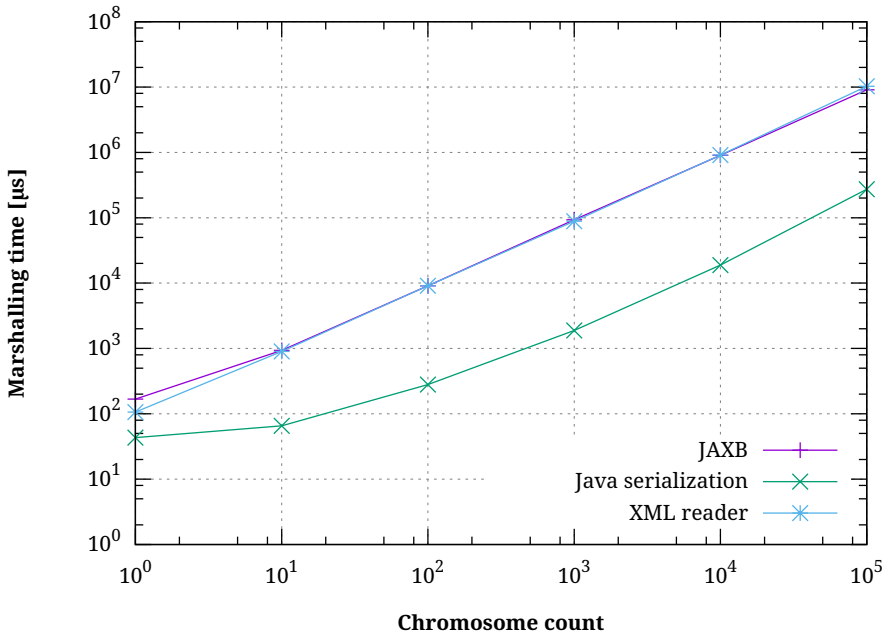


Figure 3.3.2: Genotype read performance

3.4 io.jenetics.prngine

The `prngine`[28] module contains pseudo-random number generators for sequential and parallel Monte Carlo simulations[29]. It has been designed to work smoothly with the **Jenetics** GA library, but it has no dependency to it. All PRNG implementations of this library implements the Java `RandomGenerator` interface, which makes it easily usable in other projects.

The pseudo random number generators of the `io.jenetics.prngine` module are **not** cryptographically strong PRNGs.

The `io.jenetics.prngine` module consists of the following PRNG implementations:

`KISS32Random` Implementation of an simple PRNG as proposed in *Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications* (JKISS32, page 3) David Jones, UCL Bioinformatics Group.[22] The period of this PRNG is $\approx 2.6 \cdot 10^{36}$.

`KISS64Random` Implementation of an simple PRNG as proposed in *Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications*

[28] This module is not part of the **Jenetics** project directly. Since it has no dependency on any of the **Jenetics** modules, it has been extracted to a separate GitHub repository (`https://github.com/jenetics/prngine`) with an independent versioning.

[29] `https://de.wikipedia.org/wiki/Monte-Carlo-Simulation`

(JKISS64, page 10) David Jones, UCL Bioinformatics Group.[22] The PRNG has a period of $\approx 1.8 \cdot 10^{75}$.

`LCG64ShiftRandom` This class implements a linear congruential PRNG with additional bit-shift transition. It is a port of the trng::lcg64_shift PRNG class of the TRNG library created by Heiko Bauke.[30]

`MT19937_32Random` This is a 32-bit version of Mersenne Twister pseudo random number generator.[31]

`MT19937_64Random` This is a 64-bit version of Mersenne Twister pseudo random number generator.

`XOR32ShiftRandom` This generator was discovered and characterized by George Marsaglia [Xorshift RNGs]. In just three XORs and three shifts (generally fast operations) it produces a full period of $2^{32} - 1$ on 32 bits. (The missing value is zero, which perpetuates itself and must be avoided.)[32]

`XOR64ShiftRandom` This generator was discovered and characterized by George Marsaglia [Xorshift RNGs]. In just three XORs and three shifts (generally fast operations) it produces a full period of $2^{64} - 1$ on 64 bits. (The missing value is zero, which perpetuates itself and must be avoided.)

All implemented PRNGs have been tested with the `dieharder` test suite. Table 3.4.1 shows the statistical performance of the implemented PRNGs, including the Java `RandomGenerator, L64X256MixRandom`, which is the default generator used by the library.

PRNG	Passed	Weak	Failed
`KISS32Random`	113	1	0
`KISS64Random`	109	5	0
`LCG64ShiftRandom`	111	3	0
`MT19937_32Random`	111	3	0
`MT19937_64Random`	108	6	0
`XOR32ShiftRandom`	103	6	5
`XOR64ShiftRandom`	113	1	0
`L64X256MixRandom`	111	3	0

Table 3.4.1: Dieharder results

The second important performance measure for PRNGs is the number of random number it is able to create per second.[33] Table 3.4.2 shows the PRN creation speed for all implemented generators.

[30] `https://github.com/jenetics/trng4`

[31] `https://en.wikipedia.org/wiki/Mersenne_Twister`

[32] `http://digitalcommons.wayne.edu/jmasm/vol2/iss1/2/`

[33] Measured on a Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz with Java(TM) SE Runtime Environment (build 1.8.0_102-b14)—Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)—, using the JHM micro-benchmark library.

PRNG	10^6 int/s	10^6 float/s	10^6 long/s	10^6 double/s
`KISS32Random`	189	143	129	108
`KISS64Random`	128	124	115	124
`LCG64ShiftRandom`	258	185	261	191
`MT19937_32Random`	140	115	92	82
`MT19937_64Random`	148	120	148	120
`XOR32ShiftRandom`	227	161	140	120
`XOR64ShiftRandom`	225	166	235	166
`L64X256MixRandom`	162	136	121	166

Table 3.4.2: PRNG speed

Chapter 4

Practical **Jenetics**

In the previous sections, we've described the technical capabilities of the **Jenetics** library in a great detail. What's missing are some general guidelines which will help to solve real world optimization problems using metaheuristic methods, supported by **Jenetics**.

> In computer science and mathematical optimization, a metaheuristic is a higher-level procedure or heuristic designed to find, generate, tune, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem or a machine learning problem, especially with incomplete or imperfect information or limited computation capacity.[47]

The optimization examples shown so far where deliberately kept simple, since the purpose was to illustrate the library usage. Trying to solve more advanced and nontrivial optimization problems, one might be overwhelmed by the problem itself *and* the correct usage of the **Jenetics** library. This might be the case when one is not familiar with Metaheuristics in general, or the field of Evolutionary algorithms in particular. Before starting with the description of some best practices, we will recap what an optimization problem actually is.

> In mathematics, engineering, computer science and economics, an optimization problem is the problem of finding the best solution from all feasible solutions.[46]

We will talk about general optimization problems, which minimization and an maximization problem. A graphical representation of an optimization problem is shown in diagram 4.0.1 on the next page.

From a given solution space, S, we want to find the solution with the optimal fitness value, which fulfills a given constraint, C. With this definition, we can start and suggest some implementation strategies using **Jenetics**.

4.1 General methodology

Jenetics uses the `Genotype` class as unified view onto the solution space, S, of the problem. This means, that the fitness function takes a `Genotype` as input. For trivial *Hello World* problems, it is totally fine to describe the solution space

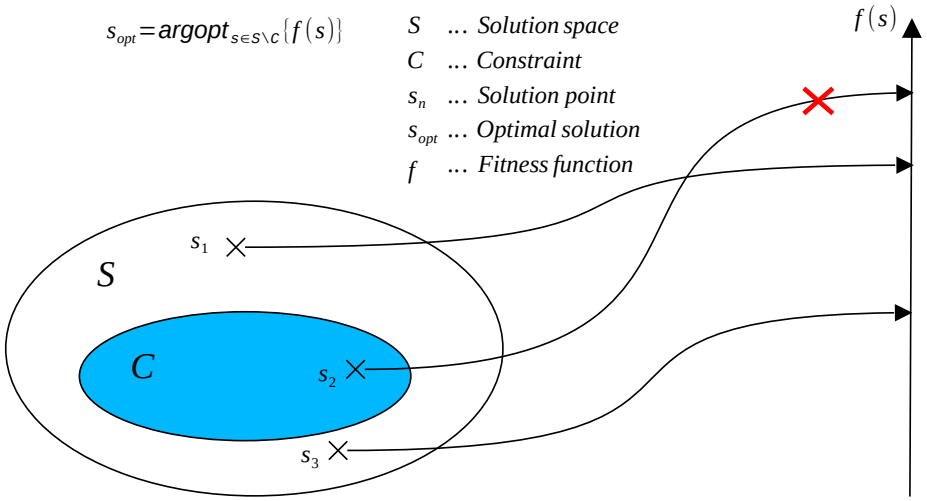


Figure 4.0.1: Optimization problem

in terms of a `Genotype`. E.g., for the *bit counting* problem, where the fitness function directly uses the `Genotype` as input parameter. The code snippet below shows the fitness function for the *bit counting* problem.

```
1 static int fitness(final Genotype<BitGene> gt) {
2     return gt.chromosome()
3         .as(BitChromosome.class)
4         .bitCount();
5 }
```

For nontrivial optimization problems, using the `Genotype` directly is not the best way for implementing the fitness function. The code will become hard to read very quickly, since the implementation of the actual fitness function is mixed with the code needed for decoding the `Genotype` to the *native* solution space, S. It is better to implement the fitness function in terms of S. To do that, it is first needed to find a domain model, which best represents the solution space. Ideally, the domain model doesn't allow the modeling of invalid solutions.

Find a domain model, which is able to represent only valid solution candidates, or as view as possible invalid solution candidates.

To connect the domain model and the fitness function, which uses this domain model, with the **Jenetics** `Engine`[1], an appropriate `Codec`[2] must be implemented. The `Codec` interface is the connector between the domain model and the `Engine` class. With this approach, one can separate the problem of implementing the fitness function from the problem mapping the `Genotype` to the domain model.

[1] See section 1.3.3.2 on page 26.
[2] See section 2.3 on page 60.

The `Codec` should map every `Genotype` to only valid solution candidates in S, or as view as possible invalid solution candidates. Every `Genotype` should also be mapped to only one solution candidate in S.

Methodological approach

1. Find a domain model for the solution space, S.
2. Implement the fitness function in terms of the domain model.
3. Find a `Codec` which maps a `Genotype` to the domain model.
4. Setup `Engine` with `Codec` and fitness function, $f(S)$.

The following code snippet shows how to use a `Codec` for the *bit counting* problem. It shows the steps 2 to 4 from the methodological approach. Step 1, finding the domain model representing the solution space, is trivial in this example: S is set to `int`.

```
// (2) Fitness function working on native solution space.
int fitness(final int count) {
    return count;
}

// (3) Codec for Genotype to int mapping.
final Codec<Integer, BitGene> codec = Codec.of(
    Genotype.of(BitChromosome.of(100)),
    gt -> gt.chromosome()
        .as(BitChromosome.class)
        .bitCount()
);

// (4) Setup Engine.
final static Engine<BitGene, Integer> engine = Engine
    .builder(this::fitness, codec)
    .build();
```

4.2 Encoding strategies

Finding a proper encoding for the domain model of the solution space can be quite challenging. The `Codec`[3] interface has been introduced as a mapping abstraction between the `Genotype` and the domain model, used by the fitness function. While it is possible to create a `Codec` directly by specifying the `Factory<Genotype>` and the `Genotype` to domain model mapping, this is not always the best way for creating a `Codec`. With a basic set of predefined `Codecs` and the composition capability of the `Codec` interface, complex `Codecs` can be created in a declarative style. The following subsections describes solutions for common encoding problems.

[3] See section 2.3 on page 60.

4.2.1 Mapping codecs

Before developing your own `Codec`, have a look at the `Codecs` class, which contains a base set of commonly usable codecs. They support scalars, vectors, matrices and permutations.

- **`Codec<Integer, IntegerGene>`**: Scalar support for `int/long/double` values.[4]
- **`Codec<int[], IntegerGene>`**: Vector support for `int/long/double` values.[5]
- **`Codec<int[][], IntegerGene>`**: Matrix support for `int/long/double` values.[6]
- **`Codec<int[], EnumGene>`**: Permutation support.[7]

Creating a `Codec`, as shown in the code snippet below, is more expressive and less error prone.

```
record Solution(int[] values) {}
final Codec<Solution, IntegerGene> codec = Codecs
    .ofVector(new IntRange(0, 100), 10)
    .map(Solution::new);
```

The `map` method can also be used to *homogenize* the `Gene` types for a `Genotype`, which only allows chromosomes with the same `Gene` types. The following snippet shows how to create an `int[]` arrays from `DoubleGene`s, where this `int[]` array is than used for creating the `Solution` domain object.

```
final Codec<Solution, DoubleGene> codec = Codecs
    .ofVector(new DoubleRange(0, 100), 10)
    .map(Conversions::doubleToIntArray)
    .map(Solution::new);
```

The `Conversions` class implements helper methods for converting `int[]`, `long[]` and `double[]` arrays.

4.2.2 Combining codecs

If the solution model itself consists of nested models, it is possible to define `Codecs` for each of the nested models and combine these sub `Codec` classes to the final `Codec`. The code snippet below shows how to combine two standard scalar `Codecs` to a `Codec` for a given user type.

```
// Codec for GPS point (latitude, longitude).
final Codec<WayPoint, DoubleGene> wpc = Codec.combine(
    Codecs.ofScalar(new DoubleRange(30, 50)), // latitude
    Codecs.ofScalar(new DoubleRange(69, 72)), // longitude
    WayPoint::of
);
```

A more advanced example is shown in the code snippet below. It uses the already created `Codec` and creates a `Codec` for an array of subdomain elements.

[4] See section 2.3.1 on page 61.
[5] See section 2.3.2 on page 61.
[6] See section 2.3.3 on page 62.
[7] See section 2.3.5 on page 64.

```
// Domain model of solution space.
record Locations(WayPoint[] coordinates) {}

// Codec for the path object, using the
// 'wpc' codec for a single vector element.
final Codec<Locations, DoubleGene> pc = Codecs.ofVector(wpc, 10)
    .map(coordinates -> coordinates.toArray(WayPoint[]::new))
    .map(Locations::new);
```

As it can be seen, the composability of the `Codec` interface is a very useful and mighty tool, for creating complex model encodings.

The `Codec` also allows the creation of random `Genotype`s and the decoding of these `Genotype`s into instances of the domain model.

```
// Create a new, random 'Genotype'.
final Genotype<DoubleGene> genotype = pc.encoding().newInstance();
// Decode the 'Genotype' to an instance of the domain model.
final Locations locations = pc.decode(genotype);
```

4.2.3 Partial permutation codecs

Permutation problems are usually encoded with the `EnumGene` and the `PermutationChromosome`, which guarantees that only permutations of a given set of elements are possible. This works fine if your problem is a *pure* permutation problem, like the Traveling salesman problem. For mixed problems, the `PermutationChromosome` can no longer be used, since all `Gene`s in a `Genotype` must be of the same type.

```
// Domain model for a variation of the TSP.
record Path(LocalTime start, WayPoint[] stops) {
}
```

If the optimal route of the TSP additionally depends on the start time of the tour, we no longer have a pure permutation problem. Additionally to the path[8], we have to encode a `LocalTime` object. Lets break down the problem, and start with the `Codec` of for the start time.

```
// Codec for a Java 'LocalTime' object.
final Codec<LocalTime, DoubleGene> ltc = Codecs
    .ofScalar(new DoubleRange(0, Duration.ofHours(24).toSeconds()))
    .map(Double::longValue)
    .map(LocalTime::ofSecondOfDay);
```

We used a `DoubleGene` for the start time encoding and now we have to find a permutation encoding which uses `DoubleGene`s as well.

```
// All waypoints of the tour.
final WayPoint[] points = new WayPoint[] {
    WayPoint.of(23, 42)
    // ...
};

// Permutation codec, using DoubleGens.
final Codec<WayPoint[], DoubleGene> wpc = Codecs
    .ofVector(new DoubleRange(0, 1), points.length) // (1)
    .map(ProxySorter::sort)                         // (2)
    .map(permutation -> {                           // (3)
```

[8]The path is defined by an array of `WayPoints`, which are a data structure of the JPX library: `https://github.com/jenetics/jpx`.

```
12          assert permutation.length == points.length;
13          var result = new WayPoint[points.length];
14          for (int i = 0; i < points.length; ++i) {
15              result[i] = points[permutation[i]];
16          }
17          return result;
18      });
```

There is a lot going on here, so lets explain the steps 1 to 3.

1. In the first step, a `double[]` array with the length, equals to the number of the way points of the tour, is created. The actual value range of the array doesn't matter.

2. In this step, the `double[]` array will be sorted. The `ProxySorter`[9] doesn't sort the array directly, which means that the input array will be not changed. Instead the `ProxySorter::sort` method returns an index array, which represents the sort order of the input.

3. The index array from the previous step is a permutation of the integers in the range [0...*points.length*). By »sorting« the tour points with the index array, we will create a valid TSP tour.

We are now able to combine the two sub `Codecs` to the final `Codec`, and we already know how to do this.

```
1 final Codec<Path, DoubleGene> codec = Codec.combine(
2     ltc, wpc,
3     Path::new
4 );
```

[9] See section 1.4.4 on page 45.

Appendix

Chapter 5

Internals

This section contains internal implementation details which doesn't fit in one of the previous sections. They are not essential for using the library, but would give the user a deeper insight in some design decisions made when implementing the library. It also introduces tools and classes which were developed for testing purpose. These classes are not exported and **not** part of the official API.

5.1 PRNG testing

Jenetics uses the `dieharder`[1] (command line) tool for testing the *randomness* of the used PRNGs. `dieharder` is a random number generator (RNG) testing suite. It is intended to test generators, not files of possibly random numbers. Since `dieharder` needs a huge amount of random data for testing the quality of a RNG, it is usually advisable to pipe the random numbers to the `dieharder` process:

```
$ cat /dev/urandom | dieharder -g 200 -a
```

The example above demonstrates how to stream a raw binary stream of bits to the `stdin` (raw) interface of `dieharder`. With the `DieHarder` class, which is part of the `io.jenetics-.prngine.internal` package, it is easily possible to test PRNGs extending the `java.util.random.RandomGenerator` interface. The only requirement is, that the PRNG must be *default*-constructible and part of the classpath.

```
$ java -cp prngine-2.0.0.jar \
      io.jenetics.prngine.internal.DieHarder \
      <random-engine-name> -a
```

Calling the command above will create an instance of the given random engine and stream the random data (bytes) to the raw interface of `dieharder` process.

```
#==========================================================================#
# Testing: L64X256MixRandom (2022-02-18 19:11)                             #
#==========================================================================#
#==========================================================================#
# Mac OS X 10.15.7 (x86_64)                                                #
```

[1] From Robert G. Brown:`http://www.phy.duke.edu/~rgb/General/dieharder.php`

```
#  java version "17"                                                          #
#  Java(TM) SE Runtime Environment (build 17+35-LTS-2724)                     #
#  Java HotSpot(TM) 64-Bit Server VM (build 17+35-LTS-2724)                   #
#=============================================================================#
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  2.39e+07  | 340119430|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.91974692|  PASSED
      diehard_operm5|   0|   1000000|     100|0.59856940|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.99990675|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.46844654|  PASSED
...
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.76563780|  PASSED
#=============================================================================#
#  Summary: PASSED=111, WEAK=3, FAILED=0                                      #
#           235.031,383 MB of random data created with 79,891 MB/sec          #
#=============================================================================#
#=============================================================================#
#  Runtime: 0:49:01                                                           #
#=============================================================================#
```

In the listing above, a part of the created `dieharder` report is shown. For testing the `LCG64ShiftRandom` class, which is part of the `io.jenetics.prngine` module, the following command can be called:

```
$ java -cp prngine-2.0.0.jar \
       io.jenetics.prngine.internal.DieHarder \
       LCG64ShiftRandom -a
```

Table 5.1.1 shows the summary of the `dieharder` tests. The full report is part of the source file of the `LCG64ShiftRandom` class.[2]

Passed tests	Weak tests	Failed tests
111	3	0

Table 5.1.1: `LCG64ShiftRandom` quality

5.2 Random seeding

The PRNGs[3], used by the **Jenetics** library, needs to be initialized with a proper seed value before they can be used. The usual way for doing this, is to take the current time stamp.

```
public static long nanoSeed() {
    return System.nanoTime();
}
```

Before applying this method throughout the whole library, I decided to perform some statistical tests. For this purpose I treated the `seed` method itself as PRNG and analyzed the created long values with the `DieHarder class`. The

[2] `https://github.com/jenetics/prngine/blob/master/prngine/src/main/java/io/jenetics/prngine/LCG64ShiftRandom.java`

[3] See section1.4.2 on page 37.

`nanoSeed` method has been wrapped into the `io.jenetics.prngine.internal.NanoTimeRandom` class. Assuming that the `dieharder` tool is in the search path, calling

```
$ java -cp prngine-2.0.0.jar \
      io.jenetics.prngine.internal.DieHarder \
      NanoTimeRandom -a
```

will perform the statistical tests for the nano time random engine. The statistical quality is rather bad: every single test failed. Table 5.2.1 shows the summary of the `dieharder` report.[4]

Passed tests	Weak tests	Failed tests
0	0	114

Table 5.2.1: Nano time seeding quality

An alternative source of entropy, for generating seed values, would be the `/dev/random` or `/dev/urandom` file. But this approach is not portable, which was a prerequisite for the **Jenetics** library.

The next attempt tries to fetch the seeds from the JVM, via the `Object::hashCode` method. Since the hash code of an Object is available for every operating system and most likely »randomly« distributed.

```
1  public static long objectSeed() {
2      final long a = new Object().hashCode();
3      final long b = new Object().hashCode();
4      return mixStafford13(a << 32 | b);
5  }
6
7  private static long mixStafford13(final long z) {
8      long v = (z^(z >>> 30))*0xbf58476d1ce4e5b9L;
9      v = (v^(v >>> 27))*0x94d049bb133111ebL;
10     return v^(v >>> 31);
11 }
```

This seed method has been wrapped into the `ObjectHashRandom` class and tested as well with

```
$ java -cp prngine-2.0.0.jar \
      io.jenetics.prngine.internal.DieHarder \
      ObjectHashRandom -a
```

Table 5.2.2 shows the summary of the `dieharder` report[5], which is already excellent.

After additional experimentation, a combination of the nano time seed and the object hash seeding seems to be the *right* solution. The rational behind this was, that the PRNG seed shouldn't rely on a single *source* of entropy.

[4] The detailed test report can be found in the source of the NanoTimeRandom class.`https://github.com/jenetics/prngine/blob/master/prngine/src/main/java/io/jenetics/prngine/internal/NanoTimeRandom.java`

[5] Full report:`https://github.com/jenetics/prngine/blob/master/prngine/src/main/java/io/jenetics/prngine/internal/ObjectHashRandom.java`

Passed tests	Weak tests	Failed tests
113	1	0

Table 5.2.2: Object hash seeding quality

```java
public static long seed() {
    final long a = mixStafford13(System.currentTimeMillis());
    final long b = mixStafford13(System.nanoTime());
    return seed(mix(a, b));
}

private static long seed(final long base) {
    return mix(base, objectSeed());
}

private static long mix(final long a, final long b) {
    long c = a^b;
    c ^= c << 17;
    c ^= c >>> 31;
    c ^= c << 8;
    return c;
}
```

Listing 5.1: Random seeding

The code in listing 5.1 shows how the nano time seed is mixed with the object seed. The `mix` method was inspired by the mixing step of the`lcg64_shift`[6] random engine, which has been reimplemented in the `LCG64ShiftRandom` class. Running the tests with

```
$ java -cp prngine-2.0.0.jar \
      io.jenetics.prngine.internal.DieHarder \
      SeedRandom -a
```

leads to the statistics summary[7], which is shown in table 5.2.3.

Passed tests	Weak tests	Failed tests
110	4	0

Table 5.2.3: Combined random seeding quality

The statistical performance of this seeding is better, according to the `dieharder` test suite, than some of the real random engines, including the default Java `Random` engine. Using the proposed `seed` method is in any case preferable to the simple `System.nanoTime()` call.

Open questions

- How does this method perform on operating systems other than Linux?
- How does this method perform on other JVM implementations?

[6]This class is part of theTRNG library:`https://github.com/rabauke/trng4/blob/master/src/lcg64_shift.hpp`

[7]Full report:`https://github.com/jenetics/prngine/blob/master/prngine/src/main/java/io/jenetics/prngine/internal/SeedRandom.java`

Chapter 6

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.

```
$ ./jenetics.example/src/main/scripts/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 7.

6.1 Ones counting

Ones counting is one of the simplest model problem. It uses a binary chromosome and forms a classic genetic algorithm[1]. The fitness of a `Genotype` is proportional to the number of ones.

```
1  import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
2  import static io.jenetics.engine.Limits.bySteadyFitness;
3
4  import io.jenetics.BitChromosome;
5  import io.jenetics.BitGene;
6  import io.jenetics.Genotype;
7  import io.jenetics.Mutator;
8  import io.jenetics.Phenotype;
9  import io.jenetics.RouletteWheelSelector;
10 import io.jenetics.SinglePointCrossover;
11 import io.jenetics.engine.Engine;
12 import io.jenetics.engine.EvolutionStatistics;
13
14 public class OnesCounting {
15
16     // This method calculates the fitness for a given genotype.
17     private static Integer count(final Genotype<BitGene> gt) {
18         return gt.chromosome()
19                  .as(BitChromosome.class)
20                  .bitCount();
```

[1] 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.

```
    }

    void main() {
        // Configure and build the evolution engine.
        final Engine<BitGene, Integer> engine = Engine
            .builder(
                OnesCounting::count,
                BitChromosome.of(20, 0.15))
            .populationSize(500)
            .selector(new RouletteWheelSelector<>())
            .alterers(
                new Mutator<>(0.55),
                new SinglePointCrossover<>(0.06))
            .build();

        // Create evolution statistics consumer.
        final EvolutionStatistics<Integer, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<BitGene, Integer> best = engine.stream()
            // Truncate the evolution stream after 7 "steady"
            // generations.
            .limit(bySteadyFitness(7))
            // The evolution will stop after maximal 100
            // generations.
            .limit(100)
            // Update the evaluation statistics after
            // each generation
            .peek(statistics)
            // Collect (reduce) the evolution stream to
            // its best phenotype.
            .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

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.[2]

```
+-------------------------------------------------------------------------+
|  Time statistics                                                        |
+-------------------------------------------------------------------------+
|             Selection: sum=0.016580144000 s; mean=0.001381678667 s      |
|              Altering: sum=0.096904159000 s; mean=0.008075346583 s      |
|   Fitness calculation: sum=0.022894318000 s; mean=0.001907859833 s      |
|     Overall execution: sum=0.136575323000 s; mean=0.011381276917 s      |
+-------------------------------------------------------------------------+
|  Evolution statistics                                                   |
+-------------------------------------------------------------------------+
|           Generations: 12                                               |
|               Altered: sum=40,487; mean=3373.916666667                  |
|                Killed: sum=0; mean=0.000000000                          |
|              Invalids: sum=0; mean=0.000000000                          |
+-------------------------------------------------------------------------+
|  Population statistics                                                  |
+-------------------------------------------------------------------------+
```

[2]For the other default values (population size, maximal age, ...) have a look at the Javadoc:`https://jenetics.io/javadoc/jenetics/9.1/index.html`

```
|                    Age: max=9; mean=0.808667; var=1.446299                    |
|                Fitness:                                                       |
|                        min  = 1.000000000000                                  |
|                        max  = 18.000000000000                                 |
|                        mean = 10.050833333333                                 |
|                        var  = 7.839555898205                                  |
|                        std  = 2.799920694985                                  |
+-------------------------------------------------------------------------------+
[00001101|11110111|11111111] --> 18
```

The given example will print the overall timing statistics onto the console. In the *Evolution statistics* section you can see that it actually takes 15 generations to fulfill the termination criteria—finding no better result after 7 consecutive generations.

6.2 Real function

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

$$f(x) = \cos\left(\frac{1}{2} + \sin(x)\right) \cdot \cos(x). \tag{6.2.1}$$

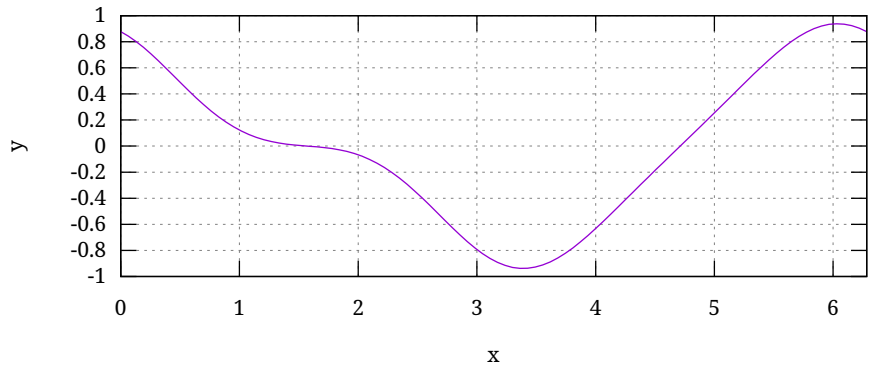


Figure 6.2.1: Real function

The graph of function 6.2.1, in the range of $[0, 2\pi]$, is shown in figure 6.2.1 and the listing beneath shows the GA implementation which will minimize the function.

```
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static java.lang.Math.sin;
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;

import io.jenetics.DoubleGene;
import io.jenetics.MeanAlterer;
import io.jenetics.Mutator;
import io.jenetics.Optimize;
import io.jenetics.Phenotype;
import io.jenetics.engine.Codecs;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionStatistics;
```

```java
import io.jenetics.util.DoubleRange;

public class RealFunction {

    // The fitness function.
    private static double fitness(final double x) {
        return cos(0.5 + sin(x))*cos(x);
    }

    void main() {
        final Engine<DoubleGene, Double> engine = Engine
            // Create a new builder with the given fitness
            // function and chromosome.
            .builder(
                RealFunction::fitness,
                Codecs.ofScalar(new DoubleRange(0.0, 2.0*PI)))
            .populationSize(500)
            .optimize(Optimize.MINIMUM)
            .alterers(
                new Mutator<>(0.03),
                new MeanAlterer<>(0.6))
            // Build an evolution engine with the
            // defined parameters.
            .build();

        // Create evolution statistics consumer.
        final EvolutionStatistics<Double, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<DoubleGene, Double> best = engine.stream()
            // Truncate the evolution stream after 7 "steady"
            // generations.
            .limit(bySteadyFitness(7))
            // The evolution will stop after maximal 100
            // generations.
            .limit(100)
            // Update the evaluation statistics after
            // each generation
            .peek(statistics)
            // Collect (reduce) the evolution stream to
            // its best phenotype.
            .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

The GA works with 1×1 `DoubleChromosomes` whose values are restricted to the range $[0, 2\pi]$.

```
+-------------------------------------------------------------------------+
|  Time statistics                                                        |
+-------------------------------------------------------------------------+
|             Selection: sum=0.064406456000 s; mean=0.003066974095 s      |
|              Altering: sum=0.070158382000 s; mean=0.003340875333 s      |
|   Fitness calculation: sum=0.050452647000 s; mean=0.002402507000 s      |
|     Overall execution: sum=0.169835154000 s; mean=0.008087388286 s      |
+-------------------------------------------------------------------------+
|  Evolution statistics                                                   |
+-------------------------------------------------------------------------+
|           Generations: 21                                               |
|               Altered: sum=3,897; mean=185.571428571                    |
|                Killed: sum=0; mean=0.000000000                          |
|              Invalids: sum=0; mean=0.000000000                          |
```

```
+-------------------------------------------------------------------------+
|  Population statistics                                                  |
+-------------------------------------------------------------------------+
|                    Age: max=9; mean=1.104381; var=1.962625              |
|                Fitness:                                                 |
|                       min  = -0.938171897696                            |
|                       max  = 0.936310125279                             |
|                       mean = -0.897856583665                            |
|                       var  = 0.027246274838                             |
|                       std  = 0.165064456617                             |
+-------------------------------------------------------------------------+
[[[3.389125782657314]]] --> -0.9381718976956661
```

The GA will generated an console output like above. The *exact* result of the function–for the given range–will be $3.389,125,782,8907,939...$ You can also see, that we reached the final result after 19 generations.

6.3 Rastrigin function

The Rastrigin function[3] is often used to test the optimization performance of genetic algorithm.

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left(x_i^2 - A\cos(2\pi x_i)\right). \tag{6.3.1}$$

As the plot in figure 6.3.1 shows, the Rastrigin function has many local minima, which makes it difficult for standard, gradient based methods to find the global minimum. If $A = 10$ and $x_i \in [-5.12, 5.12]$, the function has only one global minimum at $\mathbf{x} = \mathbf{0}$ with $f(\mathbf{x}) = 0$.

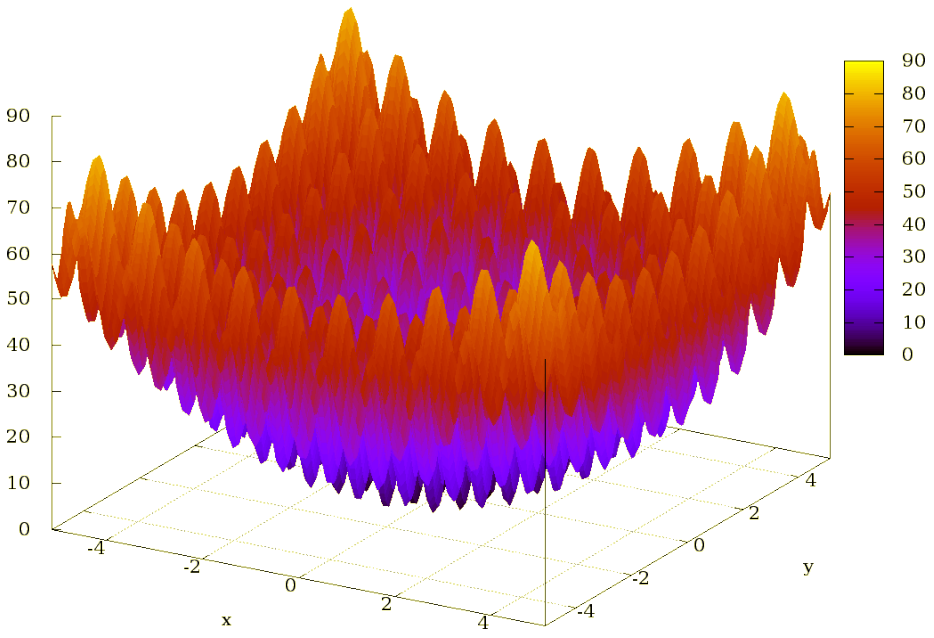


Figure 6.3.1: Rastrigin function

[3] https://en.wikipedia.org/wiki/Rastrigin_function

The following listing shows the `Engine` setup for solving the Rastrigin function, which is very similar to the setup for the real function in section 6.2. Beside the different fitness function, the `Codec` for `double` vectors is used, instead of the `double` scalar `Codec`.

```
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;

import io.jenetics.DoubleGene;
import io.jenetics.MeanAlterer;
import io.jenetics.Mutator;
import io.jenetics.Optimize;
import io.jenetics.Phenotype;
import io.jenetics.engine.Codecs;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionStatistics;
import io.jenetics.util.DoubleRange;

public class RastriginFunction {
    private static final double A = 10;
    private static final double R = 5.12;
    private static final int N = 2;

    private static double fitness(final double[] x) {
        double value = A*N;
        for (int i = 0; i < N; ++i) {
            value += x[i]*x[i] - A*cos(2.0*PI*x[i]);
        }

        return value;
    }

    void main() {
        final Engine<DoubleGene, Double> engine = Engine
            .builder(
                RastriginFunction::fitness,
                // Codec for 'x' vector.
                Codecs.ofVector(new DoubleRange(-R, R), N))
            .populationSize(500)
            .optimize(Optimize.MINIMUM)
            .alterers(
                new Mutator<>(0.03),
                new MeanAlterer<>(0.6))
            .build();

        final EvolutionStatistics<Double, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<DoubleGene, Double> best = engine.stream()
            .limit(bySteadyFitness(7))
            .peek(statistics)
            .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

The console output of the program shows, that **Jenetics** finds the *optimal* solution after 38 generations.

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=0.209185134000 s; mean=0.005504871947 s        |
|              Altering: sum=0.295102044000 s; mean=0.007765843263 s        |
|   Fitness calculation: sum=0.176879937000 s; mean=0.004654735184 s        |
|     Overall execution: sum=0.664517256000 s; mean=0.017487296211 s        |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 38                                                 |
|               Altered: sum=7,549; mean=198.657894737                      |
|                Killed: sum=0; mean=0.000000000                            |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                   Age: max=8; mean=1.100211; var=1.814053                 |
|               Fitness:                                                    |
|                      min  = 0.000000000000                                |
|                      max  = 63.672604047475                               |
|                      mean = 3.484157452128                                |
|                      var  = 71.047475139018                               |
|                      std  = 8.428966433616                                |
+---------------------------------------------------------------------------+
[[[-1.3226168588424143E-9],[-1.096964971404292E-9]]] --> 0.0
```

6.4 0/1 Knapsack

In the Knapsack problem[4] a set of items, together with its 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 i^{th} bit is set to one the i^{th} item is selected.

```
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;

import java.util.function.Function;
import java.util.stream.Collector;
import java.util.stream.Stream;

import io.jenetics.BitGene;
import io.jenetics.Mutator;
import io.jenetics.Phenotype;
import io.jenetics.RouletteWheelSelector;
import io.jenetics.SinglePointCrossover;
import io.jenetics.TournamentSelector;
import io.jenetics.engine.Codec;
import io.jenetics.engine.Codecs;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionStatistics;
import io.jenetics.util.ISeq;
import io.jenetics.util.RandomRegistry;

// The main class.
public class Knapsack {

    // This class represents a knapsack item, with a specific
    // "size" and "value".
    final static class Item {
        public final double size;
```

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

```
28          public final double value;
29
30          Item(final double size, final double value) {
31              this.size = size;
32              this.value = value;
33          }
34
35          // Create a new random knapsack item.
36          static Item random() {
37              final var r = RandomRegistry.random();
38              return new Item(
39                  r.nextDouble()*100,
40                  r.nextDouble()*100
41              );
42          }
43
44          // Collector for summing up the knapsack items.
45          static Collector<Item, ?, Item> toSum() {
46              return Collector.of(
47                  () -> new double[2],
48                  (a, b) -> {a[0] += b.size; a[1] += b.value;},
49                  (a, b) -> {a[0] += b[0]; a[1] += b[1]; return a;},
50                  r -> new Item(r[0], r[1])
51              );
52          }
53      }
54
55      // Creating the fitness function.
56      static Function<ISeq<Item>, Double>
57      fitness(final double size) {
58          return items -> {
59              final Item sum = items.stream().collect(Item.toSum());
60              return sum.size <= size ? sum.value : 0;
61          };
62      }
63
64      void main() {
65          final int nitems = 15;
66          final double kssize = nitems*100.0/3.0;
67
68          final ISeq<Item> items =
69              Stream.generate(Item::random)
70                  .limit(nitems)
71                  .collect(ISeq.toISeq());
72
73          // Defining the codec.
74          final Codec<ISeq<Item>, BitGene> codec =
75              Codecs.ofSubSet(items);
76
77          // Configure and build the evolution engine.
78          final Engine<BitGene, Double> engine = Engine
79              .builder(fitness(kssize), codec)
80              .populationSize(500)
81              .survivorsSelector(new TournamentSelector<>(5))
82              .offspringSelector(new RouletteWheelSelector<>())
83              .alterers(
84                  new Mutator<>(0.115),
85                  new SinglePointCrossover<>(0.16))
86              .build();
87
88          // Create evolution statistics consumer.
89          final EvolutionStatistics<Double, ?>
```

```
90              statistics = EvolutionStatistics.ofNumber();
91
92          final Phenotype<BitGene, Double> best = engine.stream()
93              // Truncate the evolution stream after 7 "steady"
94              // generations.
95              .limit(bySteadyFitness(7))
96              // The evolution will stop after maximal 100
97              // generations.
98              .limit(100)
99              // Update the evaluation statistics after
100             // each generation
101             .peek(statistics)
102             // Collect (reduce) the evolution stream to
103             // its best phenotype.
104             .collect(toBestPhenotype());
105
106         final ISeq<Item> knapsack = codec.decode(best.genotype());
107
108         System.out.println(statistics);
109         System.out.println(best);
110         System.out.println("\n\n");
111         System.out.printf(
112             "Genotype of best item: %s%n",
113             best.genotype()
114         );
115
116         final double fillSize = knapsack.stream()
117             .mapToDouble(it -> it.size)
118             .sum();
119
120         System.out.printf("%.2f%% filled.%n", 100*fillSize/kssize);
121     }
122 }
```

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

```
+-------------------------------------------------------------------------+
|  Time statistics                                                        |
+-------------------------------------------------------------------------+
|             Selection: sum=0.044465978000 s; mean=0.005558247250 s      |
|              Altering: sum=0.067385211000 s; mean=0.008423151375 s      |
|   Fitness calculation: sum=0.037208189000 s; mean=0.004651023625 s      |
|     Overall execution: sum=0.126468539000 s; mean=0.015808567375 s      |
+-------------------------------------------------------------------------+
|  Evolution statistics                                                   |
+-------------------------------------------------------------------------+
|           Generations: 8                                                |
|               Altered: sum=4,842; mean=605.250000000                    |
|                Killed: sum=0; mean=0.000000000                          |
|              Invalids: sum=0; mean=0.000000000                          |
+-------------------------------------------------------------------------+
|  Population statistics                                                  |
+-------------------------------------------------------------------------+
|                   Age: max=7; mean=1.387500; var=2.780039               |
|               Fitness:                                                  |
|                      min  = 0.000000000000                              |
|                      max  = 542.363235999342                            |
|                      mean = 436.098248628661                            |
|                      var  = 11431.801291812390                          |
|                      std  = 106.919601999878                            |
+-------------------------------------------------------------------------+
[01111011|10111101] --> 542.3632359993417
```

6.5 Traveling salesman

The Traveling Salesman problem[5] 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 paths 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.

```
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static java.lang.Math.hypot;
import static java.lang.Math.sin;
import static java.lang.System.out;
import static java.util.Objects.requireNonNull;
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;

import java.util.function.Function;
import java.util.stream.IntStream;

import io.jenetics.EnumGene;
import io.jenetics.Optimize;
import io.jenetics.PartiallyMatchedCrossover;
import io.jenetics.Phenotype;
import io.jenetics.SwapMutator;
import io.jenetics.engine.Codec;
import io.jenetics.engine.Codecs;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionStatistics;
import io.jenetics.engine.Problem;
import io.jenetics.util.ISeq;
import io.jenetics.util.MSeq;
import io.jenetics.util.RandomRegistry;

public class TravelingSalesman
    implements Problem<ISeq<double[]>, EnumGene<double[]>, Double>
{

    private final ISeq<double[]> _points;

     // Create new TSP problem instance with given way points.
    public TravelingSalesman(ISeq<double[]> points) {
        _points = requireNonNull(points);
    }

    @Override
    public Function<ISeq<double[]>, Double> fitness() {
        return p -> IntStream.range(0, p.length())
            .mapToDouble(i -> {
                final double[] p1 = p.get(i);
                final double[] p2 = p.get((i + 1)%p.size());
                return hypot(p1[0] - p2[0], p1[1] - p2[1]); })
            .sum();
    }

    @Override
```

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

```
49      public Codec<ISeq<double[]>, EnumGene<double[]>> codec() {
50          return Codecs.ofPermutation(_points);
51      }
52
53       // Create a new TSM example problem with the given number
54       // of stops. All stops lie on a circle with the given radius.
55      public static TravelingSalesman of(int stops, double radius) {
56          final MSeq<double[]> points = MSeq.ofLength(stops);
57          final double delta = 2.0*PI/stops;
58
59          for (int i = 0; i < stops; ++i) {
60              final double alpha = delta*i;
61              final double x = cos(alpha)*radius + radius;
62              final double y = sin(alpha)*radius + radius;
63              points.set(i, new double[]{x, y});
64          }
65
66          // Shuffling of the created points.
67          final var random = RandomRegistry.random();
68          for (int j = points.length() - 1; j > 0; --j) {
69              final int i = random.nextInt(j + 1);
70              final double[] tmp = points.get(i);
71              points.set(i, points.get(j));
72              points.set(j, tmp);
73          }
74
75          return new TravelingSalesman(points.toISeq());
76      }
77
78      void main() {
79          int stops = 20; double R = 10;
80          double minPathLength = 2.0*stops*R*sin(PI/stops);
81
82          TravelingSalesman tsm = TravelingSalesman.of(stops, R);
83          Engine<EnumGene<double[]>, Double> engine = Engine
84              .builder(tsm)
85              .optimize(Optimize.MINIMUM)
86              .maximalPhenotypeAge(11)
87              .populationSize(500)
88              .alterers(
89                  new SwapMutator<>(0.2),
90                  new PartiallyMatchedCrossover<>(0.35))
91              .build();
92
93          // Create evolution statistics consumer.
94          EvolutionStatistics<Double, ?>
95              statistics = EvolutionStatistics.ofNumber();
96
97          Phenotype<EnumGene<double[]>, Double> best =
98              engine.stream()
99              // Truncate the evolution stream after 25 "steady"
100             // generations.
101             .limit(bySteadyFitness(25))
102             // The evolution will stop after maximal 250
103             // generations.
104             .limit(250)
105             // Update the evaluation statistics after
106             // each generation
107             .peek(statistics)
108             // Collect (reduce) the evolution stream to
109             // its best phenotype.
110             .collect(toBestPhenotype());
```

```
111
112             out.println(statistics);
113             out.println("Known min path length: " + minPathLength);
114             out.println("Found min path length: " + best.fitness());
115         }
116
117 }
```

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 guarantee that no invalid solutions are created—every city exists exactly once in the altered chromosomes.

```
1  +--------------------------------------------------------------------------+
2  |  Time statistics                                                         |
3  +--------------------------------------------------------------------------+
4  |             Selection: sum=0.077451297000 s; mean=0.000619610376 s       |
5  |              Altering: sum=0.205351688000 s; mean=0.001642813504 s       |
6  |   Fitness calculation: sum=0.097127225000 s; mean=0.000777017800 s       |
7  |     Overall execution: sum=0.371304464000 s; mean=0.002970435712 s       |
8  +--------------------------------------------------------------------------+
9  |  Evolution statistics                                                    |
10 +--------------------------------------------------------------------------+
11 |           Generations: 125                                               |
12 |               Altered: sum=177,200; mean=1417.600000000                  |
13 |                Killed: sum=173; mean=1.384000000                         |
14 |              Invalids: sum=0; mean=0.000000000                           |
15 +--------------------------------------------------------------------------+
16 |  Population statistics                                                   |
17 +--------------------------------------------------------------------------+
18 |                   Age: max=11; mean=1.677872; var=5.617299               |
19 |               Fitness:                                                   |
20 |                      min  = 62.573786016092                              |
21 |                      max  = 344.248763720487                             |
22 |                      mean = 144.636749974591                             |
23 |                      var  = 5082.947247878953                            |
24 |                      std  = 71.294791169334                              |
25 +--------------------------------------------------------------------------+
26 Known min path length: 62.57378601609235
27 Found min path length: 62.57378601609235
```

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.

6.6 Evolving images

The following example tries to approximate a given image by semitransparent polygons.[6] It comes with an Swing UI, where you can immediately start your own experiments. After compiling the sources with

[6] Original idea by Roger Johanssonhttp://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa.

```
$ ./gradlew jar
```

you can start the example by calling

```
$ ./jrun io.jenetics.example.image.EvolvingImages
```

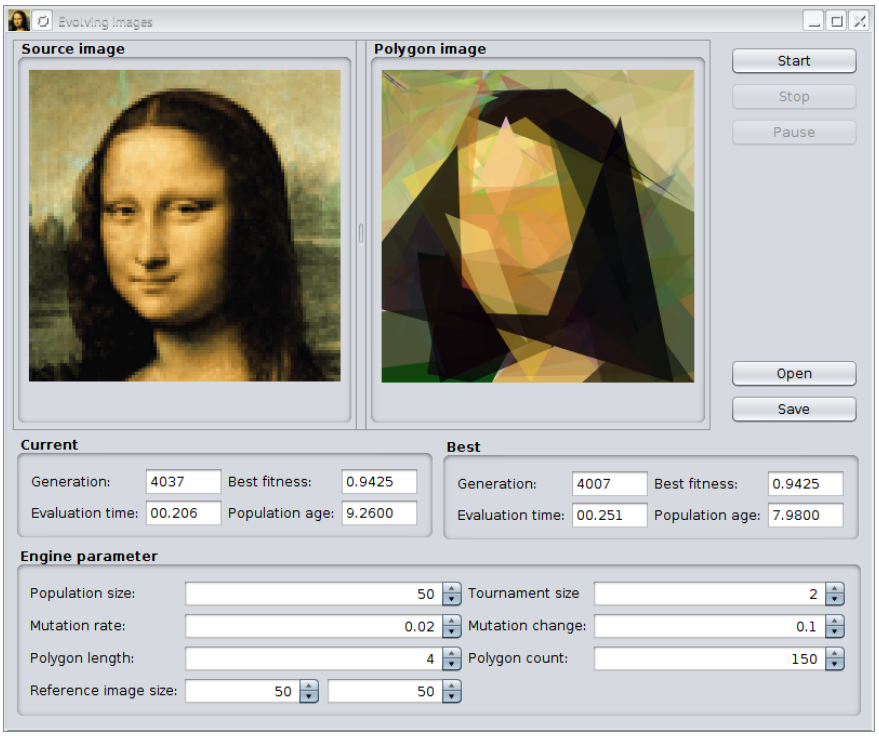


Figure 6.6.1: Evolving images UI

Figure 6.6.1 show the GUI after evolving the default image for about 4,000 generations. With the »Open« button it is possible to load other images for *polygonization*. The »Save« button allows to store *polygonized* images in PNG format to disk. At the button of the UI, you can change some of the GA parameters of the example:

Population size The number of individual in the population.

Tournament size The example uses a `TournamentSelector` for selecting the offspring population. This parameter lets you set the number of individuals used for the tournament step.

Mutation rate The probability that a polygon *component* (color or vertex position) is altered.

Mutation magnitude In case a polygon *component* is going to be mutated, its value will be randomly modified in the uniform range of $[-m, +m]$.

Polygon length The number of edges (or vertices) of the created polygons.

Polygon count The number of polygons of one individual (`Genotype`).

Reference image size To improve the processing speed, the fitness of a given polygon set (individual) is not calculated with the full sized image. Instead a scaled reference image with the given size is used. A smaller reference image will speed up the calculation, but will also reduce the accuracy.

It is also possible to run and configure the *Evolving Images* example from the command line. This allows for performing long running evolution *experiments* and save polygon images every n generations—specified with the `--image--generation` parameter.

```
$ ./jrun io.jenetics.example.image.EvolvingImages evolve \
        --engine-properties engine.properties \
        --input-image monalisa.png \
        --output-dir evolving-images \
        --generations 10000 \
        --image-generation 100
```

Every command line argument has proper default values, so that it is possible to start it without parameters. Listing 6.1 shows the default values for the GA engine if the `--engine-properties` parameter is not specified.

```
1 population_size=50
2 tournament_size=3
3 mutation_rate=0.025
4 mutation_multitude=0.15
5 polygon_length=4
6 polygon_count=250
7 reference_image_width=60
8 reference_image_height=60
```

Listing 6.1: Default `engine.properties`

For a quick start, you can simply call

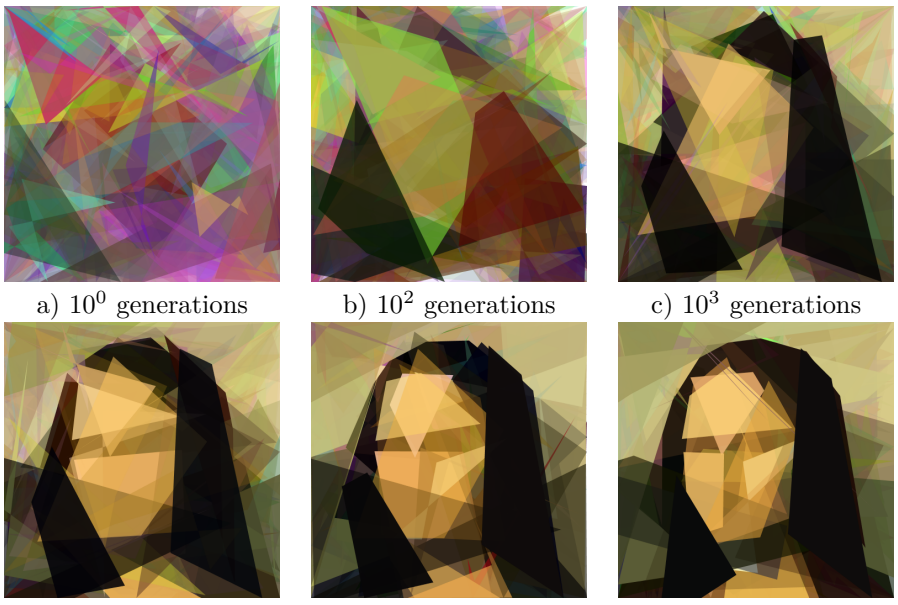
```
$ ./jrun io.jenetics.example.image.EvolvingImages evolve
```

The images in figure 6.6.2 shows the resulting polygon images after the given number of generations. They where created with the command line version of the program using the default `engine.properties` file (listing 6.1):

```
$ ./jrun io.jenetics.example.image.EvolvingImages evolve \
        --generations 1000000 \
        --image-generation 100
```

6.7 Symbolic regression

The following example shows how to set up and solve a symbolic regression problem with the help of GP and **Jenetics**. The data set used for the example was created with the polynomial, $4x^3 - 3x^2 + x$. This allows us to check the quality of the function found by the GP. Setting up a GP requires a little bit



a) 10^0 generations b) 10^2 generations c) 10^3 generations

d) 10^4 generations e) 10^5 generations f) 10^6 generations

Figure 6.6.2: Evolving *Mona Lisa* images

more effort then the setup of a GA. First, you have to define the set of atomic mathematical operations, the GP is working with. These operations influence the search space and is a kind of a *priori* knowledge put into the GP. As a second step you have to define the terminal operations. Terminals are either constants or variables. The number of variables defines the domain dimension of the fitness function.

```
1  import static io.jenetics.util.RandomRegistry.random;
2
3  import java.util.List;
4
5  import io.jenetics.Mutator;
6  import io.jenetics.engine.Engine;
7  import io.jenetics.engine.EvolutionResult;
8  import io.jenetics.engine.Limits;
9  import io.jenetics.util.ISeq;
10
11 import io.jenetics.ext.SingleNodeCrossover;
12 import io.jenetics.ext.util.TreeNode;
13
14 import io.jenetics.prog.ProgramGene;
15 import io.jenetics.prog.op.EphemeralConst;
16 import io.jenetics.prog.op.MathExpr;
17 import io.jenetics.prog.op.MathOp;
18 import io.jenetics.prog.op.Op;
19 import io.jenetics.prog.op.Var;
20 import io.jenetics.prog.regression.Error;
21 import io.jenetics.prog.regression.LossFunction;
22 import io.jenetics.prog.regression.Regression;
23 import io.jenetics.prog.regression.Sample;
24
25 public class SymbolicRegression {
```

```
26
27      // Definition of the allowed operations.
28      private static final ISeq<Op<Double>> OPS =
29          ISeq.of(MathOp.ADD, MathOp.SUB, MathOp.MUL);
30
31      // Definition of the terminals.
32      private static final ISeq<Op<Double>> TMS = ISeq.of(
33          Var.of("x", 0),
34          EphemeralConst.of(() -> (double)random().nextInt(10))
35      );
36
37      // Lookup table for {@code 4*x^3 - 3*x^2 + x}
38      public static final List<Sample<Double>> SAMPLES =
39          Sample.parseDoubles("""
40                  -1.0, -8.0000
41                  -0.9, -6.2460
42                  -0.8, -4.7680
43                  -0.7, -3.5420
44                  -0.6, -2.5440
45                  -0.5, -1.7500
46                  -0.4, -1.1360
47                  -0.3, -0.6780
48                  -0.2, -0.3520
49                  -0.1, -0.1340
50                   0.0,  0.0000
51                   0.1,  0.0740
52                   0.2,  0.1120
53                   0.3,  0.1380
54                   0.4,  0.1760
55                   0.5,  0.2500
56                   0.6,  0.3840
57                   0.7,  0.6020
58                   0.8,  0.9280
59                   0.9,  1.3860
60                   1.0,  2.0000
61              """
62          );
63
64      static final Regression<Double> REGRESSION =
65          Regression.of(
66              Regression.codecOf(
67                  OPS, TMS, 5,
68                  t -> t.gene().size() < 30
69              ),
70              Error.of(LossFunction::mse),
71              SAMPLES
72          );
73
74      void main() {
75          final Engine<ProgramGene<Double>, Double> engine = Engine
76              .builder(REGRESSION)
77              .minimizing()
78              .alterers(
79                  new SingleNodeCrossover<>(0.1),
80                  new Mutator<>())
81              .build();
82
83          final EvolutionResult<ProgramGene<Double>, Double> er =
84              engine.stream()
85                  .limit(Limits.byFitnessThreshold(0.01))
86                  .collect(EvolutionResult.toBestEvolutionResult());
87
```

```
88          final ProgramGene<Double> program = er.bestPhenotype()
89              .genotype()
90              .gene();
91
92          final TreeNode<Op<Double>> tree = program.toTreeNode();
93          MathExpr.rewrite(tree);
94          System.out.println("G: " + er.totalGenerations());
95          System.out.println("F: " + new MathExpr(tree));
96          System.out.println("E: " + REGRESSION.error(tree));
97      }
98  }
```

The error function uses the *mean squared error*[7] as loss function and no additional tree complexity metric. One output of a GP run is shown in figure 6.7.1. If we simplify this program tree, we will get exactly the polynomial which created the sample data.

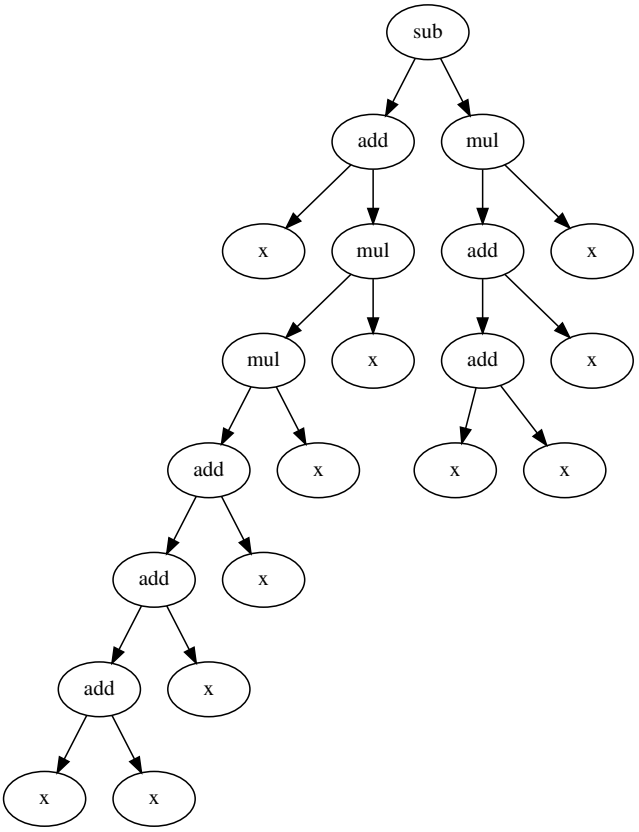


Figure 6.7.1: Symbolic regression polynomial

[7] https://en.wikipedia.org/wiki/Mean_squared_error

6.8 Grammar based regression

The same example as in given in section 6.7 on page 155, can be done with grammatical evolution[8]. How to do this is shown in the following example.

```
import static java.util.Objects.requireNonNull;
import static java.util.stream.Collectors.joining;
import static io.jenetics.prog.op.MathExpr.parseTree;

import java.util.List;
import java.util.function.Function;

import io.jenetics.IntegerGene;
import io.jenetics.Phenotype;
import io.jenetics.SinglePointCrossover;
import io.jenetics.SwapMutator;
import io.jenetics.engine.Codec;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionResult;
import io.jenetics.engine.Limits;
import io.jenetics.engine.Problem;
import io.jenetics.util.IntRange;

import io.jenetics.ext.grammar.Bnf;
import io.jenetics.ext.grammar.Cfg;
import io.jenetics.ext.grammar.Cfg.Symbol;
import io.jenetics.ext.grammar.Mappers;
import io.jenetics.ext.grammar.SentenceGenerator;
import io.jenetics.ext.util.Tree;
import io.jenetics.ext.util.TreeNode;

import io.jenetics.prog.op.Const;
import io.jenetics.prog.op.MathExpr;
import io.jenetics.prog.op.Op;
import io.jenetics.prog.regression.Error;
import io.jenetics.prog.regression.LossFunction;
import io.jenetics.prog.regression.Sample;
import io.jenetics.prog.regression.Sampling;
import io.jenetics.prog.regression.Sampling.Result;

public class GrammarBasedRegression
    implements Problem<Tree<Op<Double>, ?>, IntegerGene, Double>
{

    private static final Cfg<String> GRAMMAR = Bnf.parse("""
        <expr> ::= x | <num> | <expr> <op> <expr>
        <op>   ::= + | - | * | /
        <num>  ::= 2 | 3 | 4
        """
    );

    private static final Codec<Tree<Op<Double>, ?>, IntegerGene>
        CODEC = Mappers.multiIntegerChromosomeMapper(
            GRAMMAR,
            // The length of the chromosome is 25 times the length
            // of the alternatives of a given rule. Every rule
            // gets its own chromosome. It would also be possible
            // to define variable chromosome length with the
            // returned integer range.
            rule -> new IntRange(rule.alternatives().size()*25),
            // The used generator defines the generated data type,
```

[8] See section3.1.8 on page 107.

```
57              // which is `List<Terminal<String>>`.
58              index -> new SentenceGenerator<>(index, 50)
59          )
60          // Map the type of the codec from `List<Terminal<String>>`
61          // to `String`
62          .map(s -> s.stream().map(Symbol::name).collect(joining()))
63          // Map the type of the codec from `String` to
64          // `Tree<Op<Double>, ?>`
65          .map(e -> e.isEmpty()
66              ? TreeNode.of(Const.of(0.0))
67              : parseTree(e));
68
69      private static final Error<Double> ERROR =
70          Error.of(LossFunction::mse);
71
72      private final Sampling<Double> _sampling;
73
74      public GrammarBasedRegression(Sampling<Double> sampling) {
75          _sampling = requireNonNull(sampling);
76      }
77
78      public GrammarBasedRegression(List<Sample<Double>> samples) {
79          this(Sampling.of(samples));
80      }
81
82      @Override
83      public Codec<Tree<Op<Double>, ?>, IntegerGene> codec() {
84          return CODEC;
85      }
86
87      @Override
88      public Function<Tree<Op<Double>, ?>, Double> fitness() {
89          return program -> {
90              final Result<Double> result = _sampling.eval(program);
91              return ERROR.apply(
92                  program, result.calculated(), result.expected()
93              );
94          };
95      }
96
97      void main() {
98          final var regression = new GrammarBasedRegression(
99              SymbolicRegression.SAMPLES
100         );
101
102         final Engine<IntegerGene, Double> engine = Engine
103             .builder(regression)
104             .alterers(
105                 new SwapMutator<>(),
106                 new SinglePointCrossover<>())
107             .minimizing()
108             .build();
109
110         final EvolutionResult<IntegerGene, Double> result = engine
111             .stream()
112             .limit(Limits.byFitnessThreshold(0.05))
113             .collect(EvolutionResult.toBestEvolutionResult());
114
115         final Phenotype<IntegerGene, Double> best =
116             result.bestPhenotype();
117
118         final Tree<Op<Double>, ?> program =
```

```
119                 regression.decode(best.genotype());
120
121             System.out.println(
122                 "Generations: " + result.totalGenerations());
123             System.out.println(
124                 "Function:    " + new MathExpr(program).simplify());
125             System.out.println(
126                 "Error:       " + regression.fitness().apply(program));
127         }
128
129     }
```

6.9 DTLZ1

Deb, Thiele, Laumanns and Zitzler have proposed a set of generational MOPs for testing and comparing MOEAs. This suite of benchmarks attempts to define generic MOEA test problems that are scalable to a user defined number of objectives. Because of the last names of its creators, this test suite is known as DTLZ (Deb-Thiele-Laumanns-Zitzler).[10]

DTLZ1 is an M-objective problem with linear Pareto-optimal front:[17]

$$\begin{aligned} f_1(\mathbf{x}) &= \frac{1}{2} x_1 x_2 \cdots x_{M-1} (1 + g(\mathbf{x}_M)), \\ f_2(\mathbf{x}) &= \frac{1}{2} x_1 x_2 \cdots (1 - x_{M-1}) (1 + g(\boldsymbol{x}_M)), \\ &\vdots \\ f_{M-1}(\mathbf{x}) &= \frac{1}{2} x_1 (1 - x_2) (1 + g(\mathbf{x}_M)), \\ f_M(\mathbf{x}) &= \frac{1}{2} (1 - x_1) (1 + g(\mathbf{x}_M)), \\ & \forall i \in [1, ..n] : \ 0 \leq x_i \leq 1 \end{aligned}$$

The functional $g(\mathbf{x}_M)$ requires $|\mathbf{x}_M| = k$ variables and must take any function with $g \geq 0$. Typically g is defined as:

$$g(\mathbf{x}_M) = 100 \left[|\mathbf{x}_M| + \left(x - \frac{1}{2} \right)^2 - \cos\left(20\pi \left(x - \frac{1}{2} \right) \right) \right].$$

In the above problem, the total number of variables is $n = M + k - 1$. The search space contains $11^k - 1$ local Pareto-optimal fronts, each of which can attract an MOEA.

```
1  import static java.lang.Math.PI;
2  import static java.lang.Math.cos;
3  import static java.lang.Math.pow;
4
5  import io.jenetics.DoubleGene;
6  import io.jenetics.Mutator;
7  import io.jenetics.Phenotype;
8  import io.jenetics.TournamentSelector;
9  import io.jenetics.engine.Codecs;
10 import io.jenetics.engine.Engine;
11 import io.jenetics.engine.Problem;
```

```
12 import io.jenetics.util.DoubleRange;
13 import io.jenetics.util.ISeq;
14 import io.jenetics.util.IntRange;
15
16 import io.jenetics.ext.SimulatedBinaryCrossover;
17 import io.jenetics.ext.moea.MOEA;
18 import io.jenetics.ext.moea.NSGA2Selector;
19 import io.jenetics.ext.moea.Vec;
20
21 public class DTLZ1 {
22     private static final int VARIABLES = 4;
23     private static final int OBJECTIVES = 3;
24     private static final int K = VARIABLES - OBJECTIVES + 1;
25
26     static final Problem<double[], DoubleGene, Vec<double[]>>
27     PROBLEM = Problem.of(
28         DTLZ1::f,
29         Codecs.ofVector(new DoubleRange(0, 1.0), VARIABLES)
30     );
31
32     static Vec<double[]> f(final double[] x) {
33         double g = 0.0;
34         for (int i = VARIABLES - K; i < VARIABLES; i++) {
35             g += pow(x[i] - 0.5, 2.0) - cos(20.0*PI*(x[i] - 0.5));
36         }
37         g = 100.0*(K + g);
38
39         final double[] f = new double[OBJECTIVES];
40         for (int i = 0; i < OBJECTIVES; ++i) {
41             f[i] = 0.5 * (1.0 + g);
42             for (int j = 0; j < OBJECTIVES - i - 1; ++j) {
43                 f[i] *= x[j];
44             }
45             if (i != 0) {
46                 f[i] *= 1 - x[OBJECTIVES - i - 1];
47             }
48         }
49
50         return Vec.of(f);
51     }
52
53     static final Engine<DoubleGene, Vec<double[]>> ENGINE =
54         Engine.builder(PROBLEM)
55             .populationSize(100)
56             .alterers(
57                 new SimulatedBinaryCrossover<>(1),
58                 new Mutator<>(1.0/VARIABLES))
59             .offspringSelector(new TournamentSelector<>(5))
60             .survivorsSelector(NSGA2Selector.ofVec())
61             .minimizing()
62             .build();
63
64     void main() {
65         final ISeq<Vec<double[]>> front = ENGINE.stream()
66             .limit(2500)
67             .collect(MOEA.toParetoSet(new IntRange(1000, 1100)))
68             .map(Phenotype::fitness);
69     }
70
71 }
```

The listing above shows the encoding of the DTLZ1 problem with the **Jenetics**

library. Figure 6.9.1 shows the Pareto-optimal front of the DTLZ1 optimization.

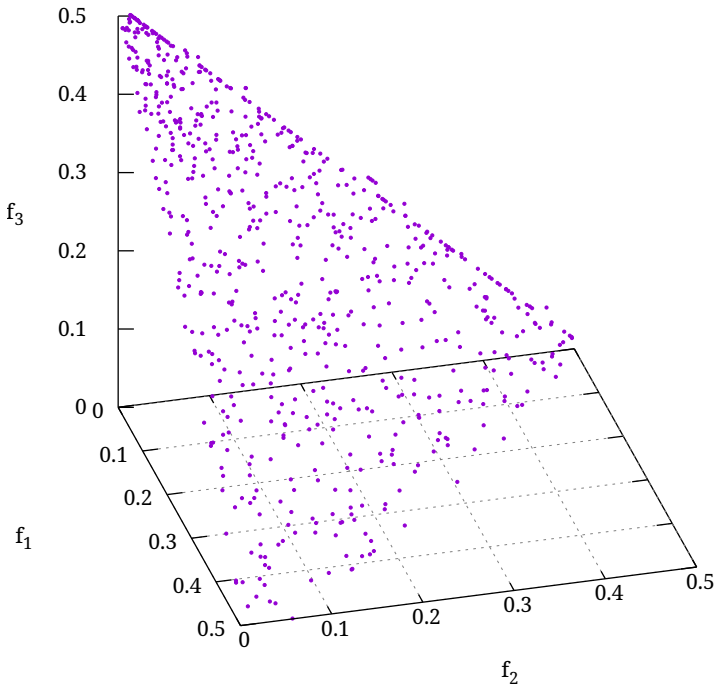


Figure 6.9.1: Pareto front DTLZ1

Chapter 7

Build

For building the **Jenetics** library from source, download the most recent, stable package version from `https://github.com/jenetics/jenetics/releases` 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 version from the Git `master` branch.

```
$ git clone https://github.com/jenetics/jenetics.git\
          <builddir>
```

Jenetics uses Gradle[1] as build system and organizes the source into *sub*-projects (*modules*).[2] Each *sub*-project is located in its own *sub*-directory.

Published projects

- `jenetics`: This project contains the source code and tests for the **Jenetics** *base*-module.
- `jenetics.ext`: This module contains additional *non*-standard GA operations and data types. It also contains classes for solving multi-objective problems (MOEA). Additional classes for defining tree rewrite systems are also part of this module.
- `jenetics.prog`: The modules contains classes which allows to do genetic programming (GP). It seamlessly works with the existing `EvolutionStream` and evolution `Engine`.
- `jenetics.xml`: XML marshalling module for the **Jenetics** base data structures.

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

[2] 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.

- `prngine`: PRNGine is a pseudo-random number generator library for sequential and parallel Monte Carlo simulations. Since this library has no dependencies to one of the other projects, it has its own repository[3] with independent versioning.

Non-published projects

- `jenetics.example`: This project contains example code for the *base*-module.
- `jenetics.incubator`: This project contains experimental code which might find its way into one of the *official* modules.
- `jenetics.doc`: Contains the code of the website and this manual.
- `jenetics.tool`: This module contains classes used for doing integration testing and algorithmic performance testing. It is also used for creating GA performance measures and creating diagrams from the performance measures.

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.
- `jar`: Compiles the sources and creates the JAR files. The artifacts are copied to the `<builddir>/<module-dir>/build/libs` 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.
- `clean`: Deletes the `<builddir>/build/*` directories and removes all generated artifacts.

For `building the library from` the source, call

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

or

```
$ ./gradlew jar
```

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.

Maven Central The whole **Jenetics** package can also be downloaded from the *Maven Central* repository `http://repo.maven.apache.org/maven2`:

[3] `https://github.com/jenetics/prngine`

pom.xml snippet for Maven

```
<dependency>
    <groupId>io.jenetics</groupId>
    <artifactId>module-name</artifactId>
    <version>9.1.0</version>
</dependency>
```

Gradle

```
'io.jenetics:module-name:9.1.0'
```

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

Bibliography

[1] Otman Abdoun, Jaafar Abouchabaka, and Chakir Tajani. Analyzing the performance of mutation operators to solve the travelling salesman problem. *CoRR*, abs/1203.3099, 2012.

[2] Otman Abdoun, Chakir Tajani, and Jaafar Abouchabaka. Hybridizing PSM and RSM operator for solving np-complete problems: Application to travelling salesman problem. *CoRR*, abs/1203.5028, 2012.

[3] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

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

[5] 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.

[6] Shumeet Baluja and Rich Caruana. Removing the genetics from the standard genetic algorithm. pages 38–46. Morgan Kaufmann Publishers, 1995.

[7] Heiko Bauke. Tina's random number generator library. *https://github.com/rabauke/trng4/blob/master/doc/trng.pdf*, 2011.

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

[9] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 3rd edition, 2018.

[10] David A. Van Veldhuizen Carlos A. Coello Coello, Gary B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Genetic and Evolutionary Computation. Springer, Berlin, Heidelberg, 2nd edition, 2007.

[11] P.K. Chawdhry, R. Roy, and R.K. Pant. *Soft Computing in Engineering Design and Manufacturing*. Springer London, 1998.

[12] Carlos Coello. Coello, a.c.: Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art. comput. methods appl. mech. engrg. 191(11-12), 1245-1287. *Computer Methods in Applied Mechanics and Engineering*, 191:1245–1287, January 2002.

[13] Rituparna Datta and Kalyanmoy Deb. *Evolutionary Constrained Optimization*. December 2014.

[14] Richard Dawkins. *The Blind Watchmaker*. New York: W. W. Norton & Company, 1986.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, April 2002.

[16] Kalyanmoy Deb and Hans-Georg Beyer. Self-adaptive genetic algorithms with simulated binary crossover. *COMPLEX SYSTEMS*, 9:431–454, 1999.

[17] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. *Scalable test problems for evolutionary multi-objective optimization*. Number 112 in TIK-Technical Report. ETH-Zentrum, ETH-Zentrum Switzerland, July 2001.

[18] Félix-Antoine Fortin and Marc Parizeau. Revisiting the nsga-ii crowding-distance computation. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 623–630, New York, NY, USA, 2013. ACM.

[19] Raymond Greenlaw. Grammars. In Raymond Greenlaw, editor, *Fundamentals of the Theory of Computation: Principles and Practice*, pages 195–220. Morgan Kaufmann, Oxford, 1998.

[20] J.F. Hughes and J.D. Foley. *Computer Graphics: Principles and Practice*. The systems programming series. Addison-Wesley, 2014.

[21] Raj Jain and Imrich Chlamtac. The p2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Commun. ACM*, 28(10):1076–1085, October 1985.

[22] David Jones. Good practice in (pseudo) random number generation for bioinformatics applications, May 2010.

[23] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Rel. Eng. & Sys. Safety*, 91(9):992–1007, 2006.

[24] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[25] John R. Koza. Introduction to genetic programming: Tutorial. In *Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '08, pages 2299–2338, New York, NY, USA, 2008. ACM.

[26] Rudolf Kruse, Sanaz Mostaghim, Christian Borgelt, Christian Braune, and Matthias Steinbrecher. *Elements of Evolutionary Algorithms*, pages 255–285. Springer International Publishing, Cham, 2022.

[27] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[28] Efrén Mezura-Montes. *Constraint-Handling in Evolutionary Optimization*, volume 198. January 2009.

[29] Zbigniew Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. In *Evolutionary Programming*, 1995.

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

[31] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[32] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. 1(1):25–49.

[33] Michael O'Neil and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer US, Boston, MA, 2003.

[34] Oracle. Value-based classes. *https://docs.oracle.com/javase/8/docs/api/-java/lang/doc-files/ValueBased.html*, 2014.

[35] A. Osyczka. Multicriteria optimization for engineering design. *Design Optimization*, pages 193–227, 1985.

[36] Charles C. Palmer and Aaron Kershenbaum. An approach to a problem in network design using genetic algorithms. *Networks*, 26(3):151–163, 1995.

[37] Franz Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Springer, 2 edition, 2006.

[38] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–96, Paris, 14-15 April 1998. Springer-Verlag.

[39] Conor Ryan, Michael O'Neill, and JJ Collins. *Introduction to 20 Years of Grammatical Evolution*, pages 1–21. Springer International Publishing, Cham, 2018.

[40] Daniel Shiffman. *The Nature of Code*. The Nature of Code, 1 edition, December 2012.

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

[42] W. Vent. Rechenberg, ingo, evolutionsstrategie — optimierung technischer systeme nach prinzipien der biologischen evolution. 170 s. mit 36 abb. frommann-holzboog-verlag. stuttgart 1973. broschiert. *Feddes Repertorium*, 86(5):337–337, 1975.

[43] Eric W. Weisstein. Scalar function. *http://mathworld.wolfram.com/-ScalarFunction.html*, 2015.

[44] Eric W. Weisstein. Vector function. *http://mathworld.wolfram.com/-VectorFunction.html*, 2015.

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

[46] Wikipedia contributors. Optimization problem — Wikipedia, the free encyclopedia, 2023. [Online; accessed 11-September-2024].

[47] Wikipedia contributors. Metaheuristic — Wikipedia, the free encyclopedia, 2024. [Online; accessed 11-September-2024].

Index