2.2.2 Scalar function .................................................. 46
2.2.3 Vector function .................................................. 47
2.2.4 Affine transformation .......................................... 47
2.2.5 Graph .......................................................... 49
2.3 Codec .................................................................. 51
  2.3.1 Scalar codec ..................................................... 52
  2.3.2 Vector codec ..................................................... 53
  2.3.3 Subset codec ..................................................... 53
  2.3.4 Permutation codec ............................................. 55
  2.3.5 Mapping codec .................................................. 56
  2.3.6 Composite codec .............................................. 56
2.4 Problem .................................................................. 58
2.5 Validation .............................................................. 59
2.6 Termination ............................................................. 60
  2.6.1 Fixed generation .................................................. 61
  2.6.2 Steady fitness ..................................................... 61
  2.6.3 Evolution time .................................................... 63
  2.6.4 Fitness threshold ................................................ 64
  2.6.5 Fitness convergence ......................................... 65
  2.6.6 Population convergence ..................................... 66
  2.6.7 Gene convergence .............................................. 67
2.7 Reproducibility ......................................................... 68
2.8 Evolution performance ........................................... 69
2.9 Evolution strategies ................................................ 70
  2.9.1 ($\mu, \lambda$) evolution strategy ................................ 70
  2.9.2 ($\mu + \lambda$) evolution strategy .............................. 71
2.10 Evolution interception .............................................. 71

3 Internals ..................................................................... 73
  3.1 PRNG testing ........................................................ 73
  3.2 Random seeding ..................................................... 74

4 Modules ..................................................................... 77
  4.1 io.jenetics.ext ......................................................... 78
    4.1.1 Data structures .................................................. 78
      4.1.1.1 Tree .......................................................... 78
      4.1.1.2 Parentheses tree .......................................... 79
      4.1.1.3 Flat tree ..................................................... 80
    4.1.2 Genes ............................................................ 81
      4.1.2.1 BigInteger gene .......................................... 81
      4.1.2.2 Tree gene ................................................... 81
    4.1.3 Operators ........................................................ 81
    4.1.4 Weasel program ............................................... 82
    4.1.5 Modifying Engine .............................................. 84
      4.1.5.1 ConcatEngine ............................................. 85
      4.1.5.2 CyclicEngine ............................................. 86
      4.1.5.3 AdaptiveEngine .......................................... 86
    4.1.6 Multi-objective optimization ................................ 88
      4.1.6.1 Pareto efficiency ........................................ 88
      4.1.6.2 Implementing classes ................................... 89
List of Figures

1.2.1 Evolution workflow ........................................... 4
1.2.2 Evolution engine model ....................................... 4
1.2.3 Package structure ........................................... 5
1.3.1 Domain model ................................................ 6
1.3.2 Chromosome structure ....................................... 8
1.3.3 Genotype structure .......................................... 8
1.3.4 Row-major Genotype vector ................................ 9
1.3.5 Column-major Genotype vector ............................ 10
1.3.6 Genotype scalar ............................................ 11
1.3.7 Fitness proportional selection ............................. 13
1.3.8 Stochastic-universal selection ............................. 15
1.3.9 Single-point crossover ..................................... 19
1.3.10 2-point crossover ......................................... 19
1.3.11 3-point crossover ......................................... 20
1.3.12 Partially-matched crossover .............................. 20
1.3.13 Uniform crossover ......................................... 20
1.3.14 Line crossover hypercube ................................ 21
1.4.1 Evaluation batch ............................................ 30
1.4.2 Block splitting ............................................. 34
1.4.3 Leapfrogging ................................................ 35
1.4.4 Seq class diagram ......................................... 37
2.2.1 Undirected graph and adjacency matrix .................. 49
2.2.2 Directed graph and adjacency matrix ...................... 50
2.2.3 Weighted graph and adjacency matrix .................... 51
2.3.1 Mapping codec types ........................................ 56
2.6.1 Fixed generation termination ............................ 62
2.6.2 Steady fitness termination ................................. 63
2.6.3 Execution time termination ............................... 64
2.6.4 Fitness threshold termination ............................. 65
2.6.5 Fitness convergence termination: \(N_S = 10, N_L = 30\) .... 67
2.6.6 Fitness convergence termination: \(N_S = 50, N_L = 150\) ... 68
2.8.1 Selector-performance (Knapsack) ......................... 69
4.0.1 Module graph .................................................. 77
4.1.1 Example tree ................................................ 78
4.1.2 Parentheses tree syntax diagram .......................... 79
4.1.3 Example FlatTree .......................................... 81
4.1.4 Single-node crossover ..................................... 82
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.5</td>
<td>Engine concatenation</td>
<td>85</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Cyclic Engine</td>
<td>86</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Adaptive Engine</td>
<td>87</td>
</tr>
<tr>
<td>4.1.8</td>
<td>Circle points</td>
<td>89</td>
</tr>
<tr>
<td>4.1.9</td>
<td>Maximizing Pareto front</td>
<td>90</td>
</tr>
<tr>
<td>4.1.10</td>
<td>Minimizing Pareto front</td>
<td>91</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Genotype write performance</td>
<td>99</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Genotype read performance</td>
<td>100</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Real function</td>
<td>106</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Rastrigin function</td>
<td>108</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Evolving images UI</td>
<td>116</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Evolving <em>Mona Lisa</em> images</td>
<td>117</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Symbolic regression polynomial</td>
<td>120</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Pareto front DTLZ1</td>
<td>123</td>
</tr>
</tbody>
</table>
Chapter 1

Fundamentals

Jenetics is an advanced Genetic Algorithm, Evolutionary Algorithm and Genetic Programming library, respectively, 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 Introduction

Jenetics is a library, written in Java\(^1\), which provides an genetic algorithm (GA) and genetic programming (GP) implementation. It has no runtime dependencies to other libraries, except the Java 8 runtime. Although the library is written in Java 8, it is compilable with Java 9, 10 and 11. Jenetics is available on 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 \[^{29}^{,}{21},^{28},^{20},^{22}\text{ or }^{33}\text{. For genetic programming you can have a look at }^{18}\text{ or }^{19}\text{.}

\(^{1}\)The classes described in this chapter reside in the \texttt{io.jenetics.base} module or \texttt{io:jenetics:jenetics:4.3.0} artifact, respectively.

\(^{2}\)The library is build with and depends on Java SE 8: \url{http://www.oracle.com/technetwork/java/javase/downloads/index.html}

\(^{3}\)If you are using Gradle, you can use the following dependency string: \texttt{\textcolor{red}{\textit{\textbackslash \{}io.jenetics:jenetics:4.3.0\textcolor{red}{\textit{\textbackslash \}}}\text{.}}
1.1. INTRODUCTION

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

```java
import io.jenetics.BitChromosome;
import io.jenetics.BitGene;
import io.jenetics.Genotype;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionResult;
import io.jenetics.util.Factory;

public final class HelloWorld {
    // 2.) Definition of the fitness function.
    private static int eval(final Genotype<BitGene> gt) {
        return gt.getChromosome().as(BitChromosome.class).bitCount();
    }

    public static void main(final String[] args) {
        // 1.) Define the genotype (factory) suitable for the problem.
        final Factory<Genotype<BitGene>> gtf = Genotype.of(BitChromosome.of(10, 0.5));

        // 3.) Create the execution environment.
        final Engine<BitGene, Integer> engine = Engine.
            builder(HelloWorld::eval, gtf)
            .build();

        // 4.) Start the execution (evolution) and collect the result.
        final Genotype<BitGene> result = engine.stream()
            .limit(100)
            .collect(EvolutionResult.toBestGenotype());

        System.out.println("Hello World: \n	" + result);
    }
}
```

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 implements 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. The probably most challenging part, when setting up a new evolution Engine, is to transform the problem domain into an appropriate Genotype (factory) representation. 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 on page 8.

2. Once this is done, the fitness function, which should be maximized, can be defined. Utilizing language features introduced in Java 8, we simply

---

*Section 2.2 on page 45 describes some common problem encodings.*
write a private static method, which takes the genotype we defined and calculate it’s 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 can be done safely. A reference to the `eval` method is then used as fitness function and passed to the `Engine.build` method.

3. In the third step we are creating the `evolution Engine`, which is responsible for changing, respectively evolving, a 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 on page 12). By changing the used `Executor` service, you control the number of threads, the `Engine` is allowed to use. An 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 can 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 also allows you, 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 not 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 of Java 8. Also the newly introduced lambda expressions and the functional interfaces (SAM types) play 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) pseudo-code of the Jenetics genetic algorithm steps.

```java
P_0 ← P_{initial}
F(P_0)
while !finished do
  g ← g + 1
  S_g ← selects(P_{g-1})
  O_g ← selects(O_{g-1})
  O_g ← alter(O_g)
  P_g ← filter[y_i ≥ g_{max}](S_g) + filter[y_i ≥ g_{max}](O_g)
  F(P_g)
```

Listing 1.2: Genetic algorithm

Line (1) creates the initial population 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/survivor 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 `died` 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 via the Java 8 Stream API. Therefore it is no longer necessary (and advised) to perform the evolution steps in an imperative way. 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 on the previous page.

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 EvolutionStreams, which can be safely used in different execution threads. This is possible, because the evolution Engine doesn’t have any mutable global state. It is practically a stateless function, \( f_E : P \rightarrow P \), 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 and the EvolutionStream truncation is controlled by the limit predicate. As long as the predicate returns true, the evolution is continued. 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 EvolutionStreams. The EvolutionStream is used to control the evolutionary process and collect the final

---

See section 2.6 on page 60 for a detailed description of the available termination strategies.
result. This is done in the same way as for the normal \texttt{java.util.stream.\-Stream} classes. With the additional \texttt{limit(Predicate)} method, it is possible to truncate the \texttt{EvolutionStream} if some termination criteria is fulfilled. The separation of \texttt{Engine} and \texttt{EvolutionStream} is the separation of the evolution definition and evolution execution.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{package.png}
\caption{Package structure}
\end{figure}

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

\begin{itemize}
\item \texttt{io.jenetics} This is the base package of the \texttt{Jenetics} library and contains all domain classes, like \texttt{Gene}, \texttt{Chromosome} or \texttt{Genotype}. Most of this types are immutable data classes and doesn’t implement any behavior. It also contains the \texttt{Selector} and \texttt{Alterer} interfaces and its implementations. The classes in this package are (almost) sufficient to implement an own GA.

\item \texttt{io.jenetics.engine} This package contains the actual GA implementation classes, e. g. \texttt{Engine}, \texttt{EvolutionStream} or \texttt{EvolutionResult}. They mainly operate on the domain classes of the \texttt{io.jenetics} package.

\item \texttt{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. \texttt{DoubleSummaryStatistics}). With the additions in this package it is also possible to calculate the variance, skewness and kurtosis—using the \texttt{DoubleMomentStatistics} class. The \texttt{EvolutionStatistics} object, which can be calculated for every generation, relies on the classes of this package.

\item \texttt{io.jenetics.util} This package contains the collection classes (\texttt{Seq}, \texttt{ISeq} and \texttt{MSeq}) which are used in the public interfaces of the \texttt{Chromosome} and \texttt{Genotype}. It also contains the \texttt{RandomRegistry} class, which implements the global PRNG lookup, as well as helper \texttt{IO} classes for serializing \texttt{Genotypes} and whole populations.
\end{itemize}

### 1.3 Base classes

This chapter describes the main classes which are needed to setup and run an genetic algorithm with the \texttt{Jenetics} library. They can roughly divided into three types:

\footnote{The documentation of the whole API is part of the download package or can be viewed online: \url{http://jenetics.io/javadoc/jenetics/4.3/index.html}}
Domain classes
This 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
This classes operates on the domain classes and includes the Alterer and Selector classes. They are also located in the io.jenetics package.

Engine classes
This classes implements the actual evolutionary algorithm and reside solely 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. All Gene and Chromosome implementations are immutable as well as the Genotype and Phenotype class.

![Diagram of domain classes]

Figure 1.3.1: Domain model

Figure 1.3.1 shows the class diagram of the domain classes. All domain classes are located in the io.jenetics package. The Gene is the base of the class structure. Genes are aggregated in Chromosomes. One to n Chromosomes are aggregated in Genotypes. A Genotype and a fitness Function form the Phenotype, which are collected into a population Seq.

1.3.1.1 Gene
Genes are the basic building blocks of the Jenetics library. They contain the actual information of the encoded solution, the allele. Some of the implementations also contains domain information of the wrapped allele. This is the case for all BoundedGene, 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.

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. The factory methods are used by the Alterers for creating new
Genes from the existing one and play a crucial role by the exploration of the problem space.

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

Listing 1.3: Gene interface

Listing 1.3 shows the most important methods of the Gene interface. The isValid method, introduced by the Verifiable interface, allows the gene to mark itself as invalid. All invalid genes are replaced with new ones during the evolution phase.

The available Gene implementations in the Jenetics library should cover a wide range of problem encodings. Refer to chapter 2.1.1 on page 39 for how to implement your own Gene types.

### 1.3.1.2 Chromosome

A Chromosome is a collection of Genes which contains at least one Gene. This allows to encode problems which requires more than one Gene. Like the Gene interface, the Chromosome is also it’s own factory and allows to create a new Chromosome from a given Gene sequence.

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

Listing 1.4: Chromosome interface

Listing 1.4 shows the main methods of the Chromosome interface. This are the methods for accessing single Genes by index and as an ISeq respectively, 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 Chromosome 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.

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

The factory method of the IntegerChromosome will now create chromosome instances with a length between \([\text{range}_{\text{min}}, \text{range}_{\text{max}}]\), equally distributed. Figure 1.3.2 on the following page shows the structure of a Chromosome with variable length.
1.3.3 Genotype

The central class, the evolution Engine is working with, is the Genotype. It is the structural and immutable representative of an individual and consists of one to n Chromosomes. All Chromosomes must be parameterized with the same Gene type, but it 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 numeric gene alleles must lay within the defined minimal- and maximal values.

As already mentioned, the Chromosomes of a Genotype doesn’t have to have necessarily the same size. It is only required that all genes are from the same type and the Genes within a Chromosome have the same constraints; e. g. the same min- and max values for numerical Genes.

```java
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(10.0, 5.0, 9),
    DoubleChromosome.of(0.0, 2.0, 10),
    DoubleChromosome.of(10.0, 5.0, 9)
);```
1.3. BASE CLASSES

1.3. BASE CLASSES  CHAPTER 1. FUNDAMENTALS

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

**Genotype vector**  The Genotype is essentially a two-dimensional composition of Genes. This makes it trivial to create Genotypes which can be treated as a Gene matrices. If its needed to create a vector of Genes, 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](image)

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 obeys the same constraints. E. g., for Genotype vectors containing Numeric-Genes, all Genes 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 alters 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 Alterers derived from the Recombinator do a fairly good job in exploring the problem space for row-major Genotype vector.

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

The column-major Genotype vector layout must be chosen when the problem space requires components (Genes) with different constraints. This is almost the
only reason for choosing the column-major layout. The layout of this Genotype vector is shown in Figure 1.3.5. For a vector of length \( n \), \( n \) Chromosomes of length one are needed.

![Figure 1.3.5: Column-major Genotype vector](image)

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

```csharp
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 NumericGenes are:

- MeanAlterer\(^9\)
- LineCrossover\(^10\)
- IntermediateCrossover\(^11\)

See also 2.3.2 on page 53 for an advanced description on how to use the predefined vector codecs.

**Genotype scalar** A very special case of a Genotype contains only one Chromosome with length one. The layout of such a Genotype scalar is shown in Figure 1.3.6 on the next page. Such Genotypes 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 Alterers are the same as for column-major Genotype vectors: MeanAlterer, LineCrossover and IntermediateCrossover.

\(^9\) See 1.3.2.2 on page 20
\(^10\) See 1.3.2.2 on page 21
\(^11\) See 1.3.2.2 on page 21
1.3. BASE CLASSES

CHAPTER 1. FUNDAMENTALS

Figure 1.3.6: Genotype scalar

![Genotype scalar diagram](image)

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

See also [2.3.1 on page 52](#) 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 and the fitness Function, which is used to (lazily) calculate the Genotype’s fitness value. It is only a container which forms the environment of the Genotype and doesn’t change the structure. Like the Genotype, the Phenotype is immutable and can’t be changed after creation.

```java
public final class Phenotype<G extends Gene<?, G>,
    C extends Comparable<? super C>> implements Comparable<Phenotype<G, C>>
{
    public C getFitness()
    public Genotype<G> getGenotype()
    public long getAge(long currentGeneration)
    public void evaluate()
}
```

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, which can be fetched with the getGenotype method. To make the runtime behavior predictable, the fitness value is evaluated lazily. Either by querying the fitness property or through the call of the evaluate method. The evolution Engine is calling the evaluate method in a separate step and makes the fitness evaluation time available through the EvolutionDurations class. Additionally to the fitness value, the Phenotype contains the generation when it was created. This allows to calculate the current age and 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 phenotypes. As collection class, the Seq interface is used. The own Seq

---

1^See since the fitness Function is shared by all Phenotypes, calls to the fitness Function must be idempotent. See section [1.3.3.1 on page 22](#).

---
implementations allows to express the mutability state of the population at the

type level and makes the coding more reliable. For a detailed description of this
collection classes see section 1.4.4 on page 36.

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 chosen
independently.

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

```java
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 off-

spring

\[
N_{O_s} = \|O_g\| = \text{rint}(\|P_g\| \cdot f_O)
\]  

(1.3.2)

and the number of selected survivors

\[
N_{S_s} = \|S_g\| = \|P_g\| - \|O_g\|.
\]  

(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 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.  

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

\[
Overall \text{ fitness } P = \sum_{i=0}^{N-1} f_i
\]

<table>
<thead>
<tr>
<th>( f_0 )</th>
<th>( f_1 )</th>
<th>( f_2 )</th>
<th>( f_3 )</th>
<th>( f_4 )</th>
<th>( f_5 )</th>
</tr>
</thead>
</table>

\[ r \in [0, F) \]

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:

\[
i \leftarrow \arg \min_{n \in [0,N)} \left\{ r < \sum_{i=0}^{n} f_i \right\}, \quad (1.3.4)
\]

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

\[
P(i, k) = \binom{n}{k} P(i)^k (1 - P(i))^{n-k}
\]  

(1.3.5)

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

Roulette-wheel selector  
The roulette-wheel selector is also known as fitness proportional selector and Jenetics implements it as probability selector. For calculating the selection probability \( P(i) \), the fitness value \( f_i \) of individual \( i \) is used.

\[
P(i) = \frac{f_i}{\sum_{j=0}^{N-1} f_j}
\]  

(1.3.6)

Selecting \( n \) individuals from a given population is equivalent to play \( n \) times on the roulette-wheel. The population don’t have to be sorted before selecting the individuals. 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.

\[
P'(i) = \frac{f_i - f_{\min}}{\sum_{j=0}^{N-1} (f_j - f_{\min})}
\]  

(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, has now 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. 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.

\[
P(i) = \frac{1}{N} \left( n^- + (n^+ - n^-) \frac{i - 1}{N - 1} \right)
\]  

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

\[ P(i) = (c - 1) \frac{e^{i-1}}{e^N - 1} \tag{1.3.9} \]

where \( c \) must 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

\[ P(i) = \frac{e^{b f_i}}{Z} \tag{1.3.10} \]

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

\[ Z = \sum_{i=1}^{n} e^{f_i} \tag{1.3.11} \]

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 (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. Figure 1.3.8 shows the function of the stochastic-universal selection,

\[ \text{Overall fitness } F = \sum_{i=1}^{N-1} f_i \]

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.
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 re-discovering 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 causes 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 Alterers determine how the space can be traversed: Alterers 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 to 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

$$\hat{\mu} = N_P \cdot N_g \cdot P(m)$$  \hspace{1cm} (1.3.12)

where $N_g$ is the number of available genes of a genotype and $N_P$ the population size (revere to equation 1.3.1 on page 8).
Mutator  The mutator has to deal with the problem, that the genes are arranged in a 3D structure (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
\[
P_G(m) = P_C(m) = P_g(m) = \sqrt[3]{P(m)}.
\] (1.3.13)

Gaussian mutator  The Gaussian mutator performs the mutation of number genes. This mutator picks a new value based on a Gaussian distribution around the current value of the gene. The variance of the new value (before clipping to the allowed gene range) will be
\[
\hat{\sigma}^2 = \left( \frac{g_{\text{max}} - g_{\text{min}}}{4} \right)^2
\] (1.3.14)
where \( g_{\text{min}} \) and \( g_{\text{max}} \) are the valid minimum and maximum values of the number gene. The new value will be cropped to the gene’s boundaries.

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

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

```c
void recombine(final ISeq<Phenotype<G, C>> pop) {
    // Select the Genotypes for crossover.
    final Random random = RandomRegistry.getRandom();
    final int i1 = random.nextInt(pop.length());
    final int i2 = random.nextInt(pop.length());
    final Phenotype<G, C> pt1 = pop.get(i1);
    final Phenotype<G, C> pt2 = pop.get(i2);
    final Genotype<G> gt1 = pt1.getGenotype();
    final Genotype<G> gt2 = pt2.getGenotype();

    // Choosing the Chromosome for crossover.
    final int chIndex =
```
random.nextInt(min(gt1.length(), gt2.length()));
final MSeq<Chromosome<G>> c1 = gt1.toSeq().copy();
final MSeq<Chromosome<G>> c2 = gt2.toSeq().copy();
final MSeq<G> genes1 = c1.get(chIndex).toSeq().copy();
final MSeq<G> genes2 = c2.get(chIndex).toSeq().copy();

// Perform the crossover.
crossover(genes1, genes2);
c1.set(chIndex, c1.get(chIndex).newInstance(genes1.toISeq()));
c2.set(chIndex, c2.get(chIndex).newInstance(genes2.toISeq()));

// Creating two new Phenotypes and replace the old one.
MSeq<Phenotype<G, C>> result = pop.copy();
result.set(i1, pt1.newInstance(gt1.newInstance(c1.toISeq())));
result.set(i2, pt2.newInstance(gt1.newInstance(c2.toISeq())));
}

Listing 1.6: Chromosome selection for recombination

Listing 1.6 on the preceding page 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 listing 1.6 on the previous page).

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.

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

Figure 1.3.9 shows how the **SinglePointCrossover** class is performing the crossover for different crossover points—in the given example for the chromosome indexes 0,1,3,6 and 7.

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

Figure 1.3.11 you can see how the crossover works for an odd number of crossover points.
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 1.3.12a)).

After performing the crossover we—normally—got two invalid chromosomes (figure 1.3.12b)). 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 $3 \rightarrow 6$, $4 \rightarrow 5$ and $5 \rightarrow 4$. To repair the two chromosomes we have to apply this exchange outside the crossing region (figure 1.3.12b)). At the end figure 1.3.12c) shows the repaired chromosome.

Uniform crossover In uniform crossover, the genes at index $i$ of two chromosomes are swapped with the swap-probability, $p_S$. Empirical studies shows 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. 

Figure 1.3.13: Uniform crossover

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.
Line crossover  The line crossover\footnote{The line crossover, also known as line recombination, was originally described by Heinz Mühlenbein and Dirk Schlierkamp-Voosen.\cite{Muehlenbein93}} takes two numeric chromosomes and treats it as a real number vector. Each of this vectors can also be seen as a point in $\mathbb{R}^n$. If we draw a line through this two points (chromosome), we have the possible values of the new chromosomes, which all lie on this line.

![Figure 1.3.14: Line crossover hypercube](image)

Figure 1.3.14: Line crossover hypercube

Figure[1.3.14] shows how the two chromosomes form the two three-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 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 re-creation 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.

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
1.3. BASE CLASSES

base metaphor for performing an GA. On the EvolutionStream you can define the termination predicate and than 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 an genetic algorithm. It takes a Genotype as argument and returns, at least, a Comparable object as result—the fitness value. This allows the evolution Engine, respectively the selection operators, to select the offspring- and survivor population. Some selectors have stronger requirements to the fitness value than a Comparable, but this constraints is checked by the Java type system at compile time.

Since the fitness Function is shared by all Phenotypes, calls to the fitness Function has to be idempotent. A fitness Function is idempotent if, whenever it is applied twice to any Genotype, it returns the same fitness value as if it were applied once. In the simplest case, this is achieved by Functions which doesn’t contain any global mutable state.

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

```java
public class Main {
    static Double identity (final Genotype<DoubleGene> gt) {
        return gt.getGene().getAllele();
    }

    public static void main(final String[] args) {
        // Create fitness function from method reference.
        Function<Genotype<DoubleGene>, Double>> ff1 =
            Main::identity;

        // Create fitness function from lambda expression.
        Function<Genotype<DoubleGene>, Double>> ff2 =
            gt.getGene().getAllele();
    }
}
```

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 be, at least, a Comparable type.

1.3.3.2 Engine

The evolution Engine controls how the evolution steps are executed. Once the Engine is created, via a Builder class, it can’t be changed. It doesn’t contain any mutable global state and can therefore safely used/called from different threads. This allows to create more than one EvolutionStreams from the Engine and execute them in parallel.
Listing 1.7: Engine class

Listing 1.7 shows the main methods of the Engine class. It 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 execution duration of the several evolution sub-steps and information about the killed and as invalid marked individuals. With the stream method you create a new EvolutionStream, which is used for controlling the evolution process—see section 1.3.3.3 on page 25. Alternatively it is possible to iterate through the evolution process in an imperative way (for whatever reasons this should be necessary). Just create an Iterator of EvolutionResult object by calling the iterator method.

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 are 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.Clock used for calculating the execution durations. A Clock with nanosecond precision (System.nanoTime()) is used as default.

**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().

**evaluator** This property allows you to replace the evaluation strategy of the Phenotype’s fitness function. Normally, each fitness value is evaluated concurrently, but independently from each other. In some configuration
it is necessary, for performance reason, to evaluate the fitness values of a population at once. This is then performed by the Engine.Evaluator or Engine.GenotypeEvaluator interface.

**fitnessFunction** This property defines the fitness function used by the evolution Engine. (See section 1.3.3.1 on page 22)

**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 then serves as template.

**genotypeValidator** This property lets you override the default implementation of the Genotype.isValid method, which is useful if the Genotype validity not only depends on valid property of the elements it consists of.

**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 this additional properties effectively set the offspringFraction.

**offspringSelector** This property defines the Selector used for selecting the offspring population. The default values is 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.

**phenotypeValidator** 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.

**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 is set to TournamentSelector<>(3).

**individualCreationRetries** The evolution Engine tries to create only valid individuals. If a newly created Genotype is not valid, the Engine creates another one, till the created Genotype is valid. This parameter sets the maximal number of retries before the Engine gives up and accept invalid individuals. The default value is set to 10.
mapper This property lets you define a mapper, which transforms the final EvolutionResult object after every generation. One usage of the mapper is to remove duplicate individuals from the population. The EvolutionResult.toUniquePopulation() method provides such a de-duplication mapper.

The EvolutionStreams 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 Engines no long creates infinite EvolutionStreams, they are truncated by the limit predicate defined by the Engine. This feature is needed when you are concatenating evolution Engines (see section 4.1.5.1 on page 85).

```java
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 the Predicate itself. The reason for this is that some Predicates has 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.

1.3.3.3 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.\(^\text{14}\)

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

Listing 1.8: EvolutionStream class

Listing 1.8 shows the whole EvolutionStream interface. As it can be seen, it only adds one additional method. But this additional limit method allows 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.

---
\(^\text{14}\)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/](http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/)
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 to create more than one `EvolutionStream` with the same `Engine` instance.

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 a new `EvolutionStream`.

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.

### 1.3.4.3.4 EvolutionResult

The `EvolutionResult` collects the result data of an evolution step into an immutable value class. This class is the type of the stream elements of the `EvolutionStream`, as described in section 1.3.3.3 on the previous page.

Listing 1.3.3.4 shows the two most important properties, the `population` and the `generation` the result belongs to. This 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 EvolutionResults are compared by its best Phenotype.

The EvolutionResult classes has three predefined factory methods, which will return Collectors usable for the EvolutionStream:

- toBestEvolutionResult() Collects the best EvolutionResult of a EvolutionStream according to the defined optimization strategy.
- 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 shows how to use the different EvolutionStream collectors.

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

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

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

### 1.3.3.5 EvolutionStatistics

The EvolutionStatistics class allows you to gather additional statistical information from the EvolutionStream. This is especially useful during the development phase of the application, when you have to find the right parametrization of the evolution Engine. Besides other informations, 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.

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

Listing 1.10: EvolutionStatistics usage

Listing 1.10 shows how to add the 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 `Gene` and fitness types. It can be created via the `EvolutionStatistics.ofComparable()` method. The second one collects additional statistical data for numeric fitness values. This can be created with the `EvolutionStatistics.ofNumber()` method.

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

The `EvolutionStatistics` object is a simple 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.

```java
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.getBestPhenotype()) < 0)
{
    best = result.getBestPhenotype();
    System.out.print(result.getGeneration() + ": ");
    System.out.println("Found best phenotype: " + best);
}

// Find the solution.
public static void main(final String[] args) {
    final ISeq<Point> result = CODEC.decode(
            ENGINE.stream()
                .peek(TSM::update)
                .limit(10)
                .collect(EvolutionResult.toBestGenotype())
            );
    System.out.println(result);
}

Listing 1.11: Live evolution statistics

Listing 1.11 on the preceding page shows how to implement a manual statistics gathering. The update method is called whenever a new EvolutionResult is 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 have alive view on the evolution progress.

1.3.3.6 Engine.Evaluator

This interface allows to define different strategies for evaluating the fitness functions of a given population. Usually, it is not necessary to override the default evaluation strategy. It 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 Engine.Evaluator interface can be used to calculate the fitness function for a population in one batch.

1 public interface Evaluator<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    ISeq<Phenotype<G, C>> evaluate(Seq<Phenotype<G, C>> pop);
}

Listing 1.12: Engine.Evaluator interface

The implementer is free to do the evaluation in place, or create new Phenotype instance and return the newly created one.

A second 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.
1.4 Nuts and bolts

1.4.1 Concurrency

The Jenetics library parallelizes independent task whenever possible. Especially the evaluation of the fitness function is done concurrently. That means that the fitness function must be thread safe, because it is shared by all phenotypes of a population. The easiest way for achieving thread safety is to make the fitness function immutable and re-entrant.

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.  

1.4.1.1 Basic configuration

The used Executor can be defined when building the evolution Engine object.

```java
package main;

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

public class Main {
    private static Double eval(final Genotype<DoubleGene> gt) {
        // calculate and return fitness
    }

    public static void main(String[] args) {
        // Creating an fixed size ExecutorService
        final ExecutorService executor = Executors.newFixedThreadPool(10)
        final Factory<Genotype<DoubleGene>> gtf = ...
        final Engine<DoubleGene, Double> engine = Engine.builder(Main::eval, gtf)
            .executor(executor)
            .build();
    }
}
```

See section 1.4.1.2 on the following page.
1.4. NUTS AND BOLTS
CHAPTER 1. FUNDAMENTALS

If no Executor is given, Jenetics uses a common ForkJoinPool\textsuperscript{16} 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.

```java
final Engine<DoubleGene, Double> engine = Engine.builder(...)
  .executor((Executor)Runnable::run)
  .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.

```java
final Engine<DoubleGene, Double> engine = Engine.builder(...)
  .executor(Executors.newSingleThreadExecutor())
  .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 sub-populations using the abstract RecursiveAction class. If a minimal sub-population size is reached, the fitness values for this sub-population 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 sub-population is dynamically determined by the ForkJoinTask.getSurplusQueuedTaskCount\textsuperscript{17} method. If this value is greater than three, the fitness values of the current sub-population are also evaluated immediately. The default value can be overridden by the io.jenetics.concurrency.maxSurplusQueuedTaskCount system property.

```bash
$ java -Dio.jenetics.concurrency.splitThreshold=1 -Dio.jenetics.concurrency.maxSurplusQueuedTaskCount=2 -cp jenetics-4.3.0.jar:app.jar\ncom.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 sub-population. In this case, only one core is busy, while the other cores are idle, because they already finished

\textsuperscript{16}https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

\textsuperscript{17}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.
1.4. NUTS AND BOLTS

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 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 of sub-populations, and the fitness values of each sub-population are evaluated by one thread. For long running fitness functions, it is better to have smaller sub-populations for a better CPU utilization. With the io.jenetics.concurrency.maxBatchSize system property, it is possible to reduce the sub-population 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-4.3.0.jar:app.jar \
   com.foo.bar.MyJeneticsApp
```

Another source of under-utilized 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.\textsuperscript{19}

1.4.2 Randomness

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

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

\textsuperscript{18} The number of sub-populations actually depends on the number of available CPU cores, which are determined with Runtime.availableProcessors().

\textsuperscript{19} A good introduction on how to use managed blocks, and the motivation behind it, is given in this talk: \url{https://www.youtube.com/watch?v=rUDQQQBz3tI}
List<Genotype<DoubleGene>> genotypes =
    RandomRegistry.with(new Random(123), r -> {
        Genotype.of(DoubleChromosome.of(0.0, 100.0, 10))
        .instances()
        .limit(100)
        .collect(Collectors.toList())
    });

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.

```java
Engine<DoubleGene, Double> engine = ...;
// Create a new evolution stream with the given initial genotypes.
Phenotype<DoubleGene, Double> best = engine.stream(genotypes)
    .limit(10)
    .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 non-deterministic.

---

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

The parallel nature of the GA implementation requires the creation of streams \( t_{i,j} \) of random numbers which are statistically independent, where the streams are numbered with \( j = 1, 2, 3, ..., p \), \( p \) denotes the number of processes. We expect statistical independence between the streams as well. The used PRNG should enable the GA to *play fair*, which means that the outcome of the GA is strictly independent from the underlying hardware and the number of parallel processes 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 Random engine 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 distribute them among the threads. There are essentially four different parallelizations techniques used in practice: Random seeding, Parameterization, Block splitting and Leapfrogging.

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

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

Block splitting With this method each thread will be assigned a 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 then the maximal expected number of threads. This strategy is used when using the LCG64ShiftRandom.ThreadLocal class. This class assigns every thread a block of \(2^{64} \approx 7 \cdot 10^{16}\) random numbers. After 128 threads, the blocks are recycled, but with changed seed.

Leapfrog With the leapfrog method each thread \(t \in [0, P)\) only consumes the \(P^{th}\) random number and jump 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 on the next page graphically shows the concept of the leapfrog method.

LCG64ShiftRandom The LCG64ShiftRandom class is a port of the trng::lcf64_shift PRNG class of the TRNG library, implemented in C++. It implements additional methods, which allows to implement the block splitting—and also the leapfrog—method.

---

20 This is also expressed by Donald Knuth’s advice: »Random number generators should not be chosen at random.«
21 The LCG64ShiftRandom PRNG is part of the io.jenetics.prgine module (see section 4.4 on page 99).
22 http://numbercrunch.de/trng/
1.4.2 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 Genes and Chromosomes.

- BitGene
- BitChromosome
- CharacterGene
- CharacterChromosome
- IntegerGene
- IntegerChromosome
- LongGene
- LongChromosome
- DoubleGene
- DoubleChromosome
1.4 NUTS AND BOLTS CHAPTER 1. FUNDAMENTALS

- EnumGene • Genotype
- PermutationChromosome • 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.

```
// Creating result population.
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.getPopulation(), file);

// Reading the population from disk.
ISeq<Phenotype<G, C>> population =
    (ISeq<Phenotype<G, C>>)IO.object.read(file);
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.Seq Most notable are the Seq interfaces and its implementation. They are used, among others, in the Chromosome and Genotype classes and holds the Genes and Chromosomes, respectively. The Seq interface itself represents a fixed-sized, ordered sequence of elements. It is an abstraction over the Java build-in array-type, but much safer to use for generic elements, because there are no casts needed when using nested generic types.

Figure 1.4.4 on the next page 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.

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

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

How to create instances of the three Seq types is shown in the listing above. The Seq classes also allows a more functional programming style. For a full method description refer to the [Javadoc](#).
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>
<thead>
<tr>
<th>Numeric type</th>
<th>Consumer class</th>
<th>Value class</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>IntMomentStatistics</td>
<td>IntMoments</td>
</tr>
<tr>
<td>long</td>
<td>LongMomentStatistics</td>
<td>LongMoments</td>
</tr>
<tr>
<td>double</td>
<td>DoubleMomentStatistics</td>
<td>DoubleMoments</td>
</tr>
</tbody>
</table>

Table 1.4.1: Statistics classes

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 an given `DoubleGene` stream.

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

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

The `stat` package also contains a class for calculating the quantile\(^{23}\) of a stream of double values. It’s implementing algorithm, which is described in [15], calculates—or estimates—the quantile value on the fly, without storing the consumed

\(^{23}\)https://en.wikipedia.org/wiki/Quantile
double values. This allows to use the `Quantile` class even for very large number of double values. How to calculate the first quartile of a given, random `DoubleStream` is shown in the code snipped below.

```java
final Quantile quartile = new Quantile(0.25);
new Random().doubles(10_000).forEach(quarter);
final double value = quartile.getValue();
```

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

Genes 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 getAllele() method which will return the wrapped data, (2) the newInstance method for creating new, random instances of the gene—must be of the same type and have the same constraint—and (3) the isValid() method which checks if the gene fulfill the expected constraints. The gene constraint might be violated after mutation and/or recombination. If you want to implement a new number-gene, e. g. a gene which holds complex values, you may want extend it from the abstract NumericGene class. Every Gene extends the Serializable interface. For normal genes there is no more work to do for using the Java serialization mechanism.
The custom Genes and Chromosomes implementations must use the Random engine available via the RandomRegistry.getRandom method when implementing their factory methods. Otherwise it is not possible to seamlessly change the Random engine by using the RandomRegistry.setRandom 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.

```java
class LastMonday {
    // Creates new random 'LocalDate' objects.
    private static LocalDate nextMonday() {
        final Random random = RandomRegistry.getRandom();
        LocalDate
            .of(2015, 1, 5)
            .plusWeeks(random.nextInt(1000));
    }

    // Do some additional validity check.
    private static boolean isValid(final LocalDate date) {...}

    // Create a new gene from the random 'Supplier' and
    // validation 'Predicate'.
    private final AnyGene<LocalDate> gene = AnyGene.of(LastMonday::nextMonday, LastMonday::isValid);
}
```

Listing 2.1: AnyGene example

Example listing 2.1 shows the (almost) minimal setup for creating user defined Gene allele types. By convention, the Random engine, 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 doesn’t conform 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 normally needs a corresponding chromosome implementation. The most important part of a chromosome is the factory method newInstance, which lets the evolution Engine create a new Chromosome instance from a
### 2.1. EXTENDING JENETICS

#### CHAPTER 2. ADVANCED TOPICS

sequence of **Genes**. This method is used by the **Alterers** when creating new, combined **Chromosomes**. It is allowed, that the newly created chromosome has a different length than the original one. The other methods should be self-explanatory. The chromosome has the same serialization mechanism as the gene. For 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* [6].

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

```java
public class LastMonday {
    // The used problem Codec.
    private static final Codec<LocalDate, AnyGene<LocalDate>> CODEC = Codec.of(
        Genotype.of(AnyChromosome.of(LastMonday::nextMonday)),
        gt -> gt.getGene().getAllele()
    );

    // Creates new random 'LocalDate' objects.
    private static LocalDate nextMonday() {
        final Random random = RandomRegistry.getRandom();
        return LocalDate.of(2015, 1, 5).plusWeeks(random.nextInt(1000));
    }

    // The fitness function: find a monday at the end of the month.
    private static int fitness(final LocalDate date) {
        return date.getDayOfMonth();
    }

    public static void main(final String[] args) {
        final Engine<AnyGene<LocalDate>, Integer> engine = Engine
            .builder(LastMonday::fitness, CODEC)
            .offspringSelector(new RouletteWheelSelector<>())
            .build();

        final Phenotype<AnyGene<LocalDate>, Integer> best =
            engine.stream()
                .limit(50)
                .collect(EvolutionResult.toBestPhenotype());

        System.out.println(best);
    }
}
```

Listing 2.2: **AnyChromosome** example

Listing 2.2 shows a full usage example of the **AnyGene** and **AnyChromosome** class. The example tries to find a Monday with a maximal day of month. An interesting detail is, that an **Codec**¹ definition is used for creating new **Genotypes** and


²See section [2.3 on page 51](#) 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.

```java
@FunctionalInterface
public interface Selector<G extends Gene<?, G>, C extends Comparable<? super C> {
    public ISeq<Phenotype<G, C>> select (fina l Seq<Phenotype<G, C>> population , fina l int count , fina l 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 to extend your selector from the ProbabilitySelector (or any other available Selector implementation). It is worth the effort to try to express your selection strategy in terms of selection property \( P(i) \). Another way for re-using existing Selector implementation is by composition.

```java
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<>();

    public EliteSelector() {
    }

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

42
2.1. EXTENDING JENETICS

CHAPTER 2. ADVANCED TOPICS

Listing 2.4: Elite selector

Listing 2.4 on the preceding page 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. Although this is not necessary, since 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.

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.

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 1.10 on page 27). 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<F, 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. Because the `EvolutionStream` has no direct dependency to the `Engine`, it is possible to use a different special evolution `Function`.

```java
public final class SpecialEngine {
    // The Genotype factory.
    private static final Factory<Genotype<DoubleGene>> GTF =
        Genotype.of(DoubleChromosome.of((0, 1)));

    // The fitness function.
    private static Double fitness(final Genotype<DoubleGene> gt) {
        return gt.getGene().getAllele();
    }

    // Create new evolution start object.
    private static EvolutionStart<DoubleGene, Double> start(final int populationSize, final long generation) {
        final ISeq<Phenotype<DoubleGene, Double>> population = GTF
            .instances()
            .map(gt -> Phenotype.of(gt, generation, SpecialEngine::fitness))
            .limit(populationSize)
            .collect(ISeq.toISeq());

        return EvolutionStart.of(population, population);
    }

    // The special evolution function.
    private static EvolutionResult<DoubleGene, Double> evolve(final EvolutionStart<DoubleGene, Double> start) {
        return ...; // Add implementation!
    }

    public static void main(final String[] args) {
        final Genotype<DoubleGene> best = EvolutionStream
            .of(() -> start(50, 0, SpecialEngine::evolve))
            .limit(Limits.bySteadyFitness(10))
            .limit(100)
            .collect(EvolutionResult.toBestGenotype());

        System.out.println("Best Genotype: " + best);
    }
}
```

Listing 2.6: Special evolution engine

Listing 2.6 shows a complete implementation stub for using an own special evolution `Function`.  

---

3 Also refer to section 1.3.3.3 on page 25 on how to create an `EvolutionStream` from an evolution `Function`. 

2.2 Encoding

This section presents some encoding examples for common problems. The encoding should be a complete and minimal expression of a solution to the problem. An encoding is complete if it contains enough information to represent every solution to the problem. An 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.

Whenever possible, the encoding should not be able to represent infeasible solutions. If a genotype can represent 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 [27] and [28].

Some of the encodings represented in the following sections has been implemented by Jenetics, using the Codec4 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.

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

Decoding the double value from the Genotype is also straightforward. Just get the first gene from the first chromosome, with the getGene() method, and convert it to a double.

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

When the Genotype only contains scalar chromosomes\[4\], it should be clear, that it can’t be altered by every Alterer. That means, that none of the Crossover

\[4\]See section 2.3 on page 51
\[5\]Scalar chromosomes contains only one gene.
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.

### 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. 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 \( n \).

```java
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.

```java
static double[] toScalars(final Genotype<DoubleGene> gt) {
    return gt.getChromosome().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 \( n \) Chromosomes with length one.

```java
Genotype.of(
    DoubleChromosome.of(min1, max1, 1),
    DoubleChromosome.of(min2, max2, 1),
    ...
    DoubleChromosome.of(minn, maxn, 1)
);
```

With the help of the new 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.

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

As already mentioned, with the use of scalar chromosomes 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 Genes directly via the
Genotype.get(i, j) method and than convert it to the desired numeric value, by calling intValue(), longValue() or doubleValue().

2.2.3 Vector function

A function \( f(X_1, \ldots, X_n) \), of one to \( n \) variables whose range is \( m \)-dimensional, is encoded by \( m \) DoubleChromosomes of length \( n \).\[^{32}\] 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 a 2-dimensional double[][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.getChromosome().as(DoubleChromosome.class).toArray();
}
```

2.2.4 Affine transformation

An affine transformation\[^{7}\] 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 \( A \):\[^{9}\]

\[
A = \begin{pmatrix}
    a_{11} & a_{12} & a_{13} \\
    a_{21} & a_{22} & a_{23} \\
    0 & 0 & 1
\end{pmatrix}.
\] (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)
);
```

\[^{32}\]http://mathworld.wolfram.com/AffineTransformation.html
\[^{7}\]https://en.wikipedia.org/wiki/Affine_transformation
\[^{8}\]https://en.wikipedia.org/wiki/Homogeneous_coordinates
\[^{9}\]https://en.wikipedia.org/wiki/Transformation_matrix
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 \( \text{min} \) and \( \text{max} \) 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
- \( t_x \) – the offset in \( x \) direction
- \( t_y \) – the offset in \( y \) direction
- \( \theta \) – 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.

\[
\text{Genotype. of}(
\begin{array}{c}
\text{// Scale} \\
\text{DoubleChromosome. of}(s_{x\text{Min}}, s_{x\text{Max}}), \\
\text{DoubleChromosome. of}(s_{y\text{Min}}, s_{y\text{Max}}), \\
\text{// Translation} \\
\text{DoubleChromosome. of}(t_{x\text{Min}}, t_{x\text{Max}}), \\
\text{DoubleChromosome. of}(t_{y\text{Min}}, t_{y\text{Max}}), \\
\text{// Rotation} \\
\text{DoubleChromosome. of}(\theta_{\text{Min}}, \theta_{\text{Max}}), \\
\text{// Shear} \\
\text{DoubleChromosome. of}(k_{x\text{Min}}, k_{x\text{Max}}), \\
\text{DoubleChromosome. of}(k_{y\text{Min}}, k_{y\text{Max}})
\end{array}
\)
\]

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 [13]:

\[
\begin{align*}
    a_{11} &= s_x \cos \theta + k_x s_y \sin \theta \\
    a_{12} &= s_y k_x \cos \theta - s_x \sin \theta \\
    a_{13} &= t_x \\
    a_{21} &= k_y s_x \cos \theta + s_y \sin \theta \\
    a_{22} &= s_y \cos \theta - s_x k_y \sin \theta \\
    a_{23} &= t_y
\end{align*}
\]

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, 0).doubleValue();
    final double sy = gt.get(1, 0).doubleValue();
    final double tx = gt.get(2, 0).doubleValue();
    final double ty = gt.get(3, 0).doubleValue();
    final double kx = gt.get(5, 0).doubleValue();
    final double ky = gt.get(6, 0).doubleValue();

    final double cos_th = cos(th);
    final double sin_th = sin(th);
    final double a11 = cos_th*sx + kx*sy*sin_th;
    final double a12 = cos_th*kx*sy - sx*sin_th;
    final double a21 = cos_th*ky*sy + sy*sin_th;
    final double a22 = cos_th*sy - ky*sx*sin_th;

    return new double[][] {
        {a11, a12, tx},
        {a21, a22, ty},
        {0.0, 0.0, 1.0}
    };
}

For the introduced encoding all kind of alterers can be used. Since we have one scalar DoubleChromosome, the rotation angle \( \theta \), 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](image)

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
2.2. ENCODING  

CHAPTER 2. ADVANCED TOPICS

optimizes an undirected graph has to ignore this part of the matrix.\footnote{This property violates the minimal encoding requirement we mentioned at the beginning of section 2.2 on page 45. For simplicity reason this will be ignored for the undirected graph encoding.}

```java
final int n = 6;
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 by calling \( \text{gt.get}(i-1, j-1).booleanValue() \). For extracting the whole matrix as \( \text{int}[][] \) array, the following code can be used.

```java
final int[][] array = gt.toSeq().stream()
    .map(c -> c.toSeq().stream())
    .mapToObj(BitGene::ordinal)
    .toArray()
    .toArray(new int[6][6]);
```

**Directed graph**  
A directed graph (digraph) is a graph where the path between the nodes have 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.

![Directed graph and adjacency matrix](image)

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.

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

```java
final int n = 6;
final double min = -1;
final double max = 20;
final Genotype<DoubleGene> gt = Genotype.of(DoubleChromosome.of(min, max, n), n);
```
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 on page 45 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.

Listing 2.7: Codec interface

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

Listing 2.7 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.getGene().doubleValue();
}
```

Section 2.2 on page 45 describes some possible encodings for common optimization problems.
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.

```java
final DoubleRange domain = DoubleRange.of(0, 2*PI);
final Codec<Double, DoubleGene> codec = Codec.of(
    Genotype.of(DoubleChromosome.of(domain)),
    gt -> gt.getChromosome().getGene().getAllele()
);
```

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

```java
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 of the `io.jenetics.engine.Codecs` class. The following subsections shows some of the implementation of this methods.

### 2.3.1 Scalar codec

Listing 2.8 shows the implementation of the `Codecs.ofScalar` factory method—for Integer scalars.

```java
static Codec<Integer, IntegerGene> ofScalar(IntRange domain) {
    return Codec.of(
        Genotype.of(IntegerChromosome.of(domain)),
        gt -> gt.getChromosome().getGene().getAllele()
    );
}
```

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.

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

2.3.2 Vector codec

In the listing 2.9, 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.

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

Listing 2.9: Codec factory method: `ofVector`

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

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

2.3.3 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 2.10.

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

Listing 2.10: Codec factory method: `ofSubSet`
The following usage example of subset Codec shows a simplified version of the Knapsack problem (see section 5.4 on page 110). 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) {
        final int size = subset.stream().collect(
            Collectors.summingInt(Integer::intValue));
        return size <= 20 ? size : 0;
    }

    public static void main(String[] args) {
        final Engine<BitGene, Double> engine = Engine
            .builder(Main::fitness, ofSubSet(SET)).build();
        ...
    }
}
```

**Fixed-size 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. 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.

```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() {
        return subset -> abs(
            subset.stream().mapToInt(Integer::intValue).sum());
    }

    @Override
    public Codec<ISeq<Integer>, EnumGene<Integer>> codec() {
        return Codecs.ofSubSet(_basicSet, _size);
    }
}
```

---

2.3.4 Permutation codec

This kind of codec can be used for problems where optimal solution depends on the order of the input elements. A classical example for such problems is the Knapsack problem (chapter 5.5 on page 112).

```java
static <T> Codec<T[], EnumGene<T>> ofPermutation(T... alleles) {
    return Codec.of(
        Genotype.of(PermutationChromosome.of(alleles)),
        gt -> gt.getChromosome().stream()
            .map(EnumGene::getAllele)
            .toArray(length -> (T[]) Array.newInstance(alleles[0].getClass(), length))
    );
}
```

Listing 2.11: Codec factory method: ofPermutation

Listing 2.11 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 2.11 and uses io.jenetics.jpx.WayPoints, from the JPX project, for representing the city locations.

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

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

    // The fitness function (in the problem domain).
    static double dist(final ISeq<WayPoint> path) {
        return path.stream()
            .collect(Length.Unit.METER);
    }

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

    // Find the solution.
    public static void main(final String[] args) {
        final ISeq<WayPoint> result = CODEC.decode(
            ENGINE.stream()
                .limit(10)
                .collect(EvolutionResult.toBestGenotype())
        );
        System.out.println(result);
    }
}
```

[1](https://github.com/jenetics/jpx)
2.3. CODEC

CHAPTER 2. ADVANCED TOPICS

2.3.5 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.

```
public static <A, B> Codec<Map<A, B>, EnumGene<Integer>> 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 \(|\text{source}| > |\text{target}|\), the returned mapping function is surjective, if \(|\text{source}| < |\text{target}|\), the mapping is injective and if \(|\text{source}| = |\text{target}|\), the created mapping is bijective. In every case the size of the encoded Map is \(|\text{target}|\). Figure 2.3.1 shows the described different mapping types in graphical form.

![Mapping codec types](image)

Figure 2.3.1: Mapping codec types

With \(|\text{source}| = |\text{target}|\), 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.\(^\text{15}\) The costs for such worker-job assignments are usually given by a matrix. Such an example matrix is shown in table 2.3.1.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker 1</td>
<td>13</td>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Worker 2</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Worker 3</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Worker 4</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 2.3.1: Worker-job cost

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

2.3.6 Composite codec

The composite Codec factory method allows to combine two or more Codecs into one. Listing 2.12 on the following page shows the method signature of the

\(^{15}\)https://en.wikipedia.org/wiki/Assignment_problem

\(^{16}\)https://en.wikipedia.org/wiki/Hungarian_algorithm
factory method, which is implemented directly in the Codec interface.

```java
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 2.12: 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 which 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 2.3.3 on page 53 and combines it into a Tuple of subsets.

```java
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();
    }

    public static void main(String[] args) {
        // Combined 'Codec'.
        final Codec<Tuple<ISeq<Integer>, ISeq<Integer>>, BitGene> codec = Codec.of(
            Codecs.ofSubSet(SET),
            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.getGenotype());
}

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(LongRange.of(0, 100)),
    Codecs.ofScalar(LongRange.of(0, 1000)),
    Codecs.ofScalar(LongRange.of(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);
});

2.4 Problem

The Problem interface is a further abstraction level, which allows 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>> {
    public Function<T, C> fitness();
    public Codec<T, G> codec();
}

Listing 2.13: Problem interface

Listing 2.13 shows the Problem interface. The generic type T represents the native 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.
2.5. VALIDATION

CH. 2. ADVANCED TOPICS

2.5 Validation

A given problem should usually 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 on page 45.

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 Genes determines the validity of the whole Genotype.

The Engine tries to create only valid individuals when creating the initial population and when it replaces Genotypes which has been destroyed by the altering step. Individuals which has exceeded its lifetime are also replaced by new valid ones. To guarantee the termination of the Genotype creation, the Engine is parameterized with the maximal number of retries (individualCreationRetries). If the described validation mechanism doesn’t fulfill your needs, you can override the validation mechanism by creating the Engine with an external Genotype validator.

```
final Predicate<? super Genotype<DoubleGene>> validator = gt -> {
  // Implement advanced Genotype check.
  boolean valid = ...;
  return valid;
};
final Engine<DoubleGene, Double> engine = Engine.builder(gtf, ff)
  .limit(100)
  .genotypeValidator(validator)
  .individualCreationRetries(15)
  .build();
```

Having the possibility to replace the default validation check is a nice thing, but it is better to not create invalid individuals in the first place. For achieving this goal, you have two possibilities:

1. Creating an explicit Genotype factory and
2. Implementing new Gene/Chromosome/Alterer classes.

\[\text{See section 1.3.3.2 on page 22}\]
2.6. TERMINATION  CHAPTER 2. ADVANCED TOPICS

Genotype factory  The usual mechanism for defining an encoding is to create a Genotype prototype\(^{18}\). Since the Genotype implements the Factory interface, an prototype instance can easily passed to the Engine.builder method. For a more advanced Genotype creation, you only have to create an explicit Genotype factory.

```java
final Factory<Genotype<DoubleGene>> gtf = () -> {
    // Implement your advanced Genotype factory.
    Genotype<DoubleGene> genotype = ...;
    return genotype;
};
final Engine<DoubleGene, Double> engine = Engine.builder(gtf, ff)
    .limit(100)
    .individualCreationRetries(15)
    .build();
```

With this method you can avoid that the Engine creates invalid individuals in the first place, but it is still possible that the alterer step will destroy your Genotypes.

Gene/Chromosome/Alterer  Creating your own Gene, Chromosome and Alterer classes is the most heavy-weighted possibility for solving the validity problem. Refer to section 2.1 on page 39 for a more detailed description on how to implement this classes.

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 evolution stream, respectively. 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 evolution stream is truncated. `EvolutionStream.limit(r -> true)` will create an infinite evolution stream.

All termination strategies described in the following sub-sections are part of the library and can be created by factory methods of the `io.jenetics.engine.Limits` class. The termination strategies where tested by solving the Knapsack problem\(^{19}\) (see section 5.4 on page 110) with 250 items. This makes it a real problem with a search-space size of $2^{250} \approx 10^{75}$ elements.

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

Table 2.6.1 on the next page shows the evolution parameters used for the termination tests. To make the tests comparable, all test runs uses the same

---

\(^{18}\)https://en.wikipedia.org/wiki/Prototype_pattern  
\(^{19}\)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
termination

2.6. TERMINATION CHAPTER 2. ADVANCED TOPICS

| Population size: | 150 |
| Survivors selector: | TournamentSelector<>(5) |
| Offspring selector: | RouletteWheelSelector<>( ) |
| Alterers: | Mutator<>(0.03) and SinglePointCrossover<>(0.125) |
| Fitness scaler: | Identity function |

Table 2.6.1: Knapsack evolution parameters

evolution parameters and the very same set of knapsack items. Each termination test was repeated 1,000 times, which gives us enough data to draw the given candlestick diagrams.

Some of the implemented termination strategy needs to maintain an internal state. This strategies can’t be re-used 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.byTerminationStrategy)` 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.

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

This kind of termination method should always be applied—usually additional with other evolution terminators—, to guarantee the truncation of the evolution stream and to define an upper limit of the executed generations.

Figure 2.6.1 on the following page shows the best fitness values of the used Knapsack problem after a given number of generations, whereas the candle-stick points represents the `min`, `25th percentile`, `median`, `75th 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 to its `maximal` value.

2.6.2 Steady fitness

The steady fitness strategy truncates the evolution stream 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 evolution stream.

```java
final class SteadyFitnessLimit<C extends Comparable<? super C>>
 implements Predicate<EvolutionResult<?, C>>
 {
 private final int _generations;
 private boolean _proceed = true;
 private int _stable = 0;
 private C _fitness;
```
2.6. TERMINATION 

CHAPTER 2. ADVANCED TOPICS

Figure 2.6.1: Fixed generation termination

Listing 2.14: Steady fitness

```java
public SteadyFitnessLimit(final int generations) {
    _generations = generations;
}

@Override
public boolean test(final EvolutionResult<?, C> er) {
    if (!_proceed) return false;
    if (_fitness == null) {
        _fitness = er.getBestFitness();
        _stable = 1;
    } else {
        final Optimize opt = result.getOptimize();
        if (opt.compare(_fitness, er.getBestFitness()) >= 0) {
            _proceed = ++_stable <= _generations;
        } else {
            _fitness = er.getBestFitness();
            _stable = 1;
        }
    }
    return _proceed;
}
```

Listing 2.14 on the previous page 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 and internal state.
EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .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: Steady fitness termination

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 candle-sticks. 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 evolution stream is only truncated at the end of an generation and will not interrupt the current evolution step. An maximal evolution time of zero ms will at least evaluate one generation. In an time-critical environment, where a solution must be found within a maximal time period, this terminator let you define the desired guarantees.
In the code example above, the `byExecutionTime(Duration)` method is used for creating the termination object. Another method, `byExecutionTime(Duration, Clock)`, lets you define the `java.time.Clock`, which is used for measure the execution time. `Jenetics` uses the nano precision clock `io.jenetics.utilNanoClock` for measuring the time. To have the possibility to define a different Clock implementation is especially useful for testing purposes.

![Figure 2.6.3: Execution time termination](image)

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. 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 user-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 user-specified fitness threshold when the objective is to maximize the fitness.

```java
Engine<DoubleGene, Double> engine = ... EvolutionStream<DoubleGene, Double> stream = engine.stream() .limit(Limits.byFitnessThreshold(10.5)) .limit(5000);
```

While running the tests, all other CPU intensive process has been stopped. The measuring started after a warm-up phase.
When limiting the evolution stream by a fitness threshold, you have to have a knowledge about the expected maximal fitness. If there is no such knowledge, it is advisable to add an additional fixed sized generation limit as safety net.

Figure 2.6.4 shows executed generations depending on the minimal fitness value. The total generations grows 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 to 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.

```java
public static <N extends Number & Comparable<? super N>>
    Predicate<EvolutionResult<?, N>> byFitnessConvergence(
        final int shortFilterSize,
        final int longFilterSize,
        final BiPredicate<DoubleMoments, DoubleMoments> proceed
    )
```

...
2.6. TERMINATION

CHAPTER 2. ADVANCED TOPICS

Listing 2.15: General fitness convergence

Listing 2.15 on the previous page 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.

1. \[
\text{public static } \langle N \text{ extends Number & Comparable<}% \\text{super } N\rangle > \text{ byFitnessConvergence(}
2. \text{ final int shortFilterSize,}
3. \text{ final int longFilterSize,}
4. \text{ final double epsilon)
5. \text{)};
6. \]

Listing 2.16: Mean fitness convergence

The second factory method (shown in listing 2.16) creates a fitness convergence 
predicate, which uses the moving average 21 for the two filters. The smoothed 
fitness value is calculated as follows:

\[
\sigma_F(N) = \frac{1}{N} \sum_{i=0}^{N-1} F[G-i] \quad (2.6.1)
\]

where \(N\) is the length of the filter, \(F[i]\) the fitness value at generation \(i\) and \(G\) the current generation. If the condition

\[
\frac{|\sigma_F(N_S) - \sigma_F(N_L)|}{\delta} < \epsilon \quad (2.6.2)
\]

is fulfilled, the evolution stream is truncated. Where \(\delta\) is defined as follows:

\[
\delta = \begin{cases} 
\max(|\sigma_F(N_S)|, |\sigma_F(N_L)|) & \text{if } \neq 0 \\
1 & \text{otherwise}
\end{cases} \quad (2.6.3)
\]

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, \(\epsilon\).

Figure 2.6.5 on the next page shows the termination behavior of the fitness 
convergence termination strategy. It can be seen that the minimum number of 
evoluted generations is the length of the long filter, \(N_L\).

Figure 2.6.6 on page 68 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 that stops the evolution when the population is deemed 
as converged. A population is deemed as converged when the average fitness 
across the current population is less than a user-specified percentage away from

21https://en.wikipedia.org/wiki/Moving_average
2.6. TERMINATION

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

the best fitness of the current population. The population is deemed as converged and the evolution stream is truncated if

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

where

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

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

and

$$\delta = \begin{cases} \max(\lvert f_{\text{max}} \rvert, \lvert \bar{f} \rvert) & \text{if } \neq 0 \\ 1 & \text{otherwise} \end{cases}.$$

$N$ denotes the number of individuals of the population.

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

The evolution stream 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 then 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 evolution stream. 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 define 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 setup an evolution **Engine** in a way that will produce the very same result on every run.

```java
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 2.17: Reproducible evolution Engine

Listing 2.17 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. The second step configures the random engine, the evolution Engine is working with. Just wrap the evolution stream execution in a RandomRegistry::with block. Additionally you can start the evolution stream with a predefined, initial population. Once you have setup the Engine, you can vary the fitness function and the codec and compare the results.

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 used by the examples in section 2.6 on page 60. 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.

The termination tests are using a TournamentSelector, with tournament-size 5, for selecting the survivors, and a RouletteWheelSelector for selecting the offspring.
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. 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 for selecting the individuals and usually mutation for changing the next generation. This section describes how to configure the evolution Engine of the library for the \((\mu, \lambda)\)- and \((\mu + \lambda)\)-ES.

2.9.1 \((\mu, \lambda)\) evolution strategy

The \((\mu, \lambda)\) algorithm starts by generating \(\lambda\) individuals randomly. After evaluating the fitness of all the individuals, all but the \(\mu\) fittest ones are deleted. Each of the \(\mu\) fittest individuals gets to produce \(\lambda/\mu\) children through an ordinary mutation. The newly created children just replaces the discarded parents.

To summarize it: \(\mu\) is the number of parents which survive, and \(\lambda\) is the number of offspring, created by the \(\mu\) parents. The value of \(\lambda\) should be a multiple of \(\mu\). ES practitioners usually refer to their algorithm by the choice of \(\mu\) and \(\lambda\).

If we set \(\mu = 5\) and \(\lambda = 5\), then we have a \((5, 20)\)-ES.

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

Listing 2.18: \((\mu, \lambda)\) Engine configuration

Listing 2.18 shows how to configure the evolution Engine for \((\mu, \lambda)\)-ES. The population size is set to \(\lambda\) 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 \(\mu\). This lets the `TruncationSelector` only select the \(\mu\) best individuals, which corresponds to step two of the ES.

There are mainly three levers for the \((\mu, \lambda)\)-ES where we can adjust exploration versus exploitation:

- **Population size** \(\lambda\): This parameter controls the sample size for each population. For the extreme case, as \(\lambda\) approaches \(\infty\), the algorithm would perform a simple random search.

- **Survivors size of** \(\mu\): This parameter controls how selective the ES is. Relatively low \(\mu\) values pushes the algorithm towards exploitative search, because only the best individuals are used for reproduction.

---

23 See 1.3.2.1 on page 13.
24 See 1.3.2.2 on page 17.
25 As you can see in listing 2.18, the survivors size (reproduction pool size) for the \((\mu, \lambda)\)-ES must be set *indirectly* via the `TruncationSelector` parameter. This is necessary, since for the \((\mu, \lambda)\)-ES, the selected best \(\mu\) individuals are not part of the population of the next generation.
Mutation probability $p$: A high mutation probability pushes the algorithm toward a fairly random search, regardless of the selectivity of $\mu$.

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

In the $(\mu + \lambda)$-ES, the next generation consists of the selected best $\mu$ parents and the $\lambda$ new children. This is also the main difference to $(\mu, \lambda)$, where the $\mu$ parents are not part of the next generation. Thus the next and all successive generations are $\mu + \lambda$ in size. 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 2.19.

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

Listing 2.19: $(\mu + \lambda)$ Engine configuration

Since the selected $\mu$ parents are part of the next generation, the survivorsSize property must be set to $\mu$. This also requires to set the survivors selector to the TruncationSelector. With the selector(Selector) method, both selectors, 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 $(\mu, \lambda)$-ES. This has the risk, that very fit parents can defeat other individuals over and over again, which leads to a prematurely 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 Stream.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.

```java
private EvolutionResult<DoubleGene, Double> mapping(EvolutionResult<DoubleGene, Double> result) {...}

final Genotype<DoubleGene> result = engine.stream()
    .map(this::mapping)
    .limit(100)
    .collect(toBestGenotype());
```

Doing 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 mapping must be defined when the Engine is created.

```java
final Engine<DoubleGene, Double> engine = Engine.build(problem)
    .mapping(this::mapping)
    .build();
```
The code snippet above shows the correct way for intercepting the evolution stream. The mapper given to the \texttt{Engine} will change the stream of \texttt{EvolutionResults} and the will also feed the altered result back to the evolution \texttt{Engine}.

\textbf{Distinct population} This kind of intercepting the evolution process is very flexible. \texttt{Jenetics} comes with one predefined stream interception method, which allows to remove duplicate individuals from the resulting population.

\begin{verbatim}
final Engine<DoubleGene, Double> engine = Engine.build(problem)
    .mapping(EvolutionResult.toUniquePopulation())
    .build();
\end{verbatim}

Despite the de-duplication, it is still possible to have duplicate individuals. This will be the case when domain of the possible \texttt{Genotypes} is not big enough and the same individual is created by chance. You can control the number of \texttt{Genotype} creation retries using the \texttt{EvolutionResult.toUniquePopulation(int)} method, which allows you to define the maximal number of retries if an individual already exists.
Chapter 3

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 where developed for testing purpose. This classes are not exported and not part of the official API.

3.1 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 class. The only requirement is, that the PRNG must be default-constructible and part of the classpath.

```
$ java -cp io.jenetics.prngine-1.0.1.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.

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 io.jenetics.prngine-1.0.1.jar 
  io.jenetics.prngine.internal.DieHarder 
  io.jenetics.prngine.LCG64ShiftRandom -a
```

Table 3.1.1 shows the summary of the dieharder tests. The full report is part of the source file of the LCG64ShiftRandom class.

```
<table>
<thead>
<tr>
<th>Passed tests</th>
<th>Weak tests</th>
<th>Failed tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Table 3.1.1: LCG64ShiftRandom quality

### 3.2 Random seeding

The PRNGs 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.

```java
public static long seed() {
  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


\[\text{See section 1.4.2 on page 32}\]
3.2. RANDOM SEEDING  

The `seed()` 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 io.jenetics.prngine-1.0.1.jar \
   io.jenetics.prngine.internal.DieHarder \
   io.jenetics.prngine.internal.NanoTimeRandom -a
```

will perform the statistical tests for the nano time random engine. The statistical quality is rather bad: every single test failed. Table 3.2.1 shows the summary of the `dieharder` report.

<table>
<thead>
<tr>
<th>Passed tests</th>
<th>Weak tests</th>
<th>Failed tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>114</td>
</tr>
</tbody>
</table>

Table 3.2.1: 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.

```java
public static long seed() {
    return ((long)new Object().hashCode() << 32) |
    new Object().hashCode();
}
```

This seed method has been wrapped into the `ObjectHashRandom` class and tested as well with

```
$ java -cp io.jenetics.prngine-1.0.1.jar \
   io.jenetics.prngine.internal.DieHarder \
   io.jenetics.prngine.internal.ObjectHashRandom -a
```

Table 3.2.2 shows the summary of the `dieharder` report, which looks better than the nano time seeding, but 86 failing tests was still not very satisfying.

<table>
<thead>
<tr>
<th>Passed tests</th>
<th>Weak tests</th>
<th>Failed tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>0</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 3.2.2: Object hash seeding quality

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.

---


The code in listing 3.1 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` random engine, which has been reimplemented in the `LCG64ShiftRandom` class. Running the tests with

```
$ java -cp io.jenetics.prngine-1.0.1.jar \n  io.jenetics.prngine.internal.DieHarder \n  io.jenetics.prngine.internal.SeedRandom -a
```

leads to the statistics summary, which is shown in table 3.2.3.

<table>
<thead>
<tr>
<th>Passed tests</th>
<th>Weak tests</th>
<th>Failed tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2.3: 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?

---

6This class is part of the TRNG library: [https://github.com/rabauke/trng4/blob/master/src/lcg64_shift.hpp](https://github.com/rabauke/trng4/blob/master/src/lcg64_shift.hpp)

Chapter 4

Modules

The Jenetics library has been split up into several modules, which allows to keep the base EA module as small as possible. It currently consists of the modules shown in table 4.0.1 including the Jenetics base module.

<table>
<thead>
<tr>
<th>Module</th>
<th>Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.jenetics.base</td>
<td>io.jenetics:jenetics:4.3.0</td>
</tr>
<tr>
<td>io.jenetics.ext</td>
<td>io.jenetics:jenetics.ext:4.3.0</td>
</tr>
<tr>
<td>io.jenetics.prog</td>
<td>io.jenetics:jenetics.prog:4.3.0</td>
</tr>
<tr>
<td>io.jenetics.xml</td>
<td>io.jenetics:jenetics.xml:4.3.0</td>
</tr>
<tr>
<td>io.jenetics.prngine</td>
<td>io.jenetics:prngine:1.0.1</td>
</tr>
</tbody>
</table>

Table 4.0.1: 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, which keep the io.jenetics-.base module as small as possible. The additional Jenetics modules will be described in this chapter. Figure 4.0.1 shows the dependency graph of the Jenetics modules.

![Figure 4.0.1: Module graph](http://blog.joda.org/2017/05/java-se-9-jpms-automatic-modules.html)
4.1 **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 this additional genes and operations.

### 4.1.1 Data structures

#### 4.1.1.1 Tree

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

```java
public interface Tree<V, T extends Tree<V, T>> {
  public V getValue();
  public Optional<T> getParent();
  public T getChild(int index);
  public int childCount();
}
```

Listing 4.1: `Tree` interface

Listing 4.1 shows the `Tree` interface with its basic abstract tree methods. All other needed tree methods, e.g. for node traversal and search, are implemented with default methods, which are derived from this four abstract tree methods. A mutable default implementation of the `Tree` interface is given by the `TreeNode` class.

![Example tree](image)

Figure 4.1.1: Example tree

To illustrate the usage of the `TreeNode` class, we will create a `TreeNode` instance from the tree shown in figure 4.1.1. The example tree consists of 12 nodes with a maximal depth of three and a varying child count from one to three.

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

78
Listing 4.2: Example TreeNode

Listing 4.2 on the previous page 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.

4.1.1.2 Parentheses tree

A parentheses tree is a serialized representation of a tree and is a simplified form of the Newick tree format. The parentheses tree representation of the tree in figure 4.1.1 on the preceding page 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 4.1.2 shows the syntax diagram of the parentheses tree. The NodeValue in the diagram is the string representation of the Tree.getValue() object.

![Parentheses tree syntax diagram](https://www.i-programmer.info/programming/theory/3458-parentheses-are-trees.html)

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.

```java
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.

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

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

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

Serializing a tree into parentheses form is just one part of the story. It is also possible to read back the parentheses string as a 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.

```java
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.

### 4.1.3 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 4.2 on page 92). Since the chromosome type is essentially an array, a mapping from the hierarchical tree structure to a 1-dimensional array is needed. 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.

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

Listing 4.3: `FlatTree` interface

Listing 4.3 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. 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 4.1.3 on the following page illustrates the flattened example tree shown in figure 4.1.1 on page 78. 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.

```java
final TreeNode<Integer> tree = ...;
final ISeq<FlatTreeNode<Integer>> nodes = FlatTreeNode.of(tree).
    flattenedNodes();
assert Tree.equals(tree, nodes.get(0));
final TreeNode<Integer> unflattened = TreeNode.of(nodes.get(0));
assert tree.equals(unflattened);
assert unflattened.equals(tree);
```

4There 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](https://en.wikipedia.org/wiki/Binary_tree#Arrays). For general trees with arbitrary child count, such simple mapping doesn’t exist.

4.1. Genes

4.1.2.1 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.

4.1.2.2 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 4.1.1.3 on the preceding page.

4.1.3 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 [11].

Single-node crossover  TheSingleNodeCrossover class works on TreeChromosomes. It swaps two, randomly chosen, nodes from two tree chromosomes. Figure 4.1.4 on the next page 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.
4.1.4 Weasel program

The Weasel program[^6] is thought experiment from Richard Dawkins, in which he tries to illustrate the function of genetic mutation and selection[^7]. 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?[^8]

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} 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).

[^6]: https://en.wikipedia.org/wiki/Weasel_program
[^7]: The classes are located in the io.jenetics.ext module.
[^8]:
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.[9]

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

```java
import java.util.stream.IntStream;

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

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

    public static void main(final String[] args) {
        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.stream()
            .limit(byFitnessThreshold(TARGET.length() - 1))
            .peek(r -> System.out.println("r getTotalGenerations() + " + r.getTotalGenerations() + " r getBestPhenotype() + " + r.getBestPhenotype()))
            .collect(toBestPhenotype());
        System.out.println(result);
    }
}
```
Listing 4.4: Weasel program

Listing 4.4 on the preceding page shows how to implement the WeaselProgram with Jenetics. Step (1) and (2) of the algorithm are 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 evolution stream is limited by the Limits.byFitnessThreshold, which is set to \( \text{score}_{\text{max}} - 1 \). In the current example this value is set to \( \text{TARGET}.\text{length()} - 1 = 27 \). The (shortened) output of the Weasel program (listing 4.4 on the previous page) shows, that the optimal solution is reached in generation 46.

4.1.5 Modifying Engine

The current design of Engine allows to create multiple independent evolution streams from a single Engine instance. One drawback of this approach is, that the evolution stream 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. For this purpose, the EvolutionStreamable interface has been introduced. It is similar to the Java Iterable interface and abstracts the EvolutionStream creation.

```
public interface EvolutionStreamable<
    G extends Gene<?, G>,
    C extends Comparable<? super C>> {
    EvolutionStream<G, C> stream(Supplier<EvolutionStart<G, C>> start);
    EvolutionStream<G, C> stream(EvolutionInit<G> init);
    EvolutionStreamable<G, C> limit(Supplier<Predicate<? super EvolutionResult<G, C>>> p)
}
```

Listing 4.5: EvolutionStreamable interface
Listing 4.5 on the preceding page 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 to concatenate evolution `Engines` with different configurations, which will then create one varying `EvolutionStream`. This additional `Engine` classes are:

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

### 4.1.5.1 ConcatEngine

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

```java
    final Engine<DoubleGene, Double> engine1 = ...;
    final Engine<DoubleGene, Double> engine2 = ...;

    final Genotype<DoubleGene> result =
        ConcatEngine.of(
            engine1.limit(50),
            engine2.limit(() -> Limits.bySteadyFitness(30)))
        .stream()
        .collect(EvolutionResult.toBestGenotype());
```

Figure 4.1.5: Engine concatenation

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.
4.1. IO.JENETICS.EXT  
CHAPTER 4. MODULES

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 Engines. 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 Engines would then be gradually reduced.

4.1.5.2 CyclicEngine

The CyclicEngine is similar to the ConcatEngine. Where the ConcatEngine stops the evolution, when the stream of the last engine terminates, the CyclicEngine continues with a new stream from the first Engine. The evolution flow of the CyclicEngine is shown in figure 4.1.6.

Figure 4.1.6: Cyclic Engine

![Diagram of Cyclic Engine]

Since the CyclicEngine creates unlimited streams, although the participating Engines 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.

```java
final Genotype<DoubleGene> result =
    CyclicEngine.of(
        engine1.limit(50),
        engine2.limit((() -> Limits.bySteadyFitness(15)))
          .stream()
          .limit(Limits.bySteadyFitness(50))
          .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.

4.1.5.3 AdaptiveEngine

The AdaptiveEngine is the most flexible method for creating EvolutionStreams. Instead of defining a fixed set Engines, which then creates the EvolutionStream, you define a function which creates an Engine, depending on last EvolutionResult of the previous EvolutionStream. You can see the evolution flow of the AdaptiveEngine in figure 4.1.7 on the next page.

From the implementation perspective, the AdaptiveEngine requires a little bit more code. The additional flexibility isn’t for free, as you can see in the code example below.

```java
public static void main(final String[] args) {
    final Problem<double[], DoubleGene, Double> problem = ...;
```
The example tries to broaden the search, once the variance of the population’s fitness values are below a given threshold. When implementing the `Engine` creation function, you have to be aware, that the `EvolutionResult` for the first `Engine` is `null`. 
4.1.6 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.

There are several ways for solving multiobjective problems. An excellent theoretical foundation is given in [7]. The algorithms implemented by Jenetics are based in terms of Pareto optimality as described in [13], [10] and [17].

4.1.6.1 Pareto efficiency

Pareto efficiency is named after the Italian economist and political scientist Vilfredo Pareto. 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.

**Definition. (Pareto efficiency [7]):** A solution $x$ is said to be Pareto optimal if there is no $x'$ for which $v = (f_1(x'), ..., f_k(x'))$ dominates $u = (f_1(x), ..., f_k(x))$.

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

**Definition. (Pareto dominance [7]):** A vector $u = (u_1, ..., u_k)$ is said to dominate another vector $v = (v_1, ..., v_k)$ (denoted by $u \succeq v$) if $u$ is partially less than $v$, i.e., $\forall i \in \{1, ..., k\}, u_i \geq v_i \land \exists i \in \{1, ..., k\} : u_i > v_i$.

After this two basic definitions, let’s have a look at a simple example. Figure 4.1.8 on the following page shows some points of a two-dimensional solution space. For simplicity, the points will all lie within a circle with radius 1 and center point of $(1, 1)$.

Figure 4.1.9 on page 90 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 4.1.10 on page 91 shows the Pareto front if we try to minimize the $x$ and $y$ coordinate at the same time.

---

4.1.6.2 Implementing classes

The classes, used for solving multi-objective problems, reside in the \texttt{io.jenetics.ext.moea} package. Originally, the \texttt{Jenetics} library focuses on solving single-objective problems. This drives the design decision to force the return value of the fitness function to be \texttt{Comparable}. If the result type of the fitness function is a vector, it is no longer clear how to make the results comparable. \texttt{Jenetics} chooses to use the Pareto dominance relation (see section 4.1.6.1 on the previous page). The Pareto dominance relation, \(\succ\), defines a strict partial order, which means \(\succ\) is

1. irreflexive: \(u \not\succ u\),
2. transitive: \(u \succ v \land v \succ w \Rightarrow u \succ w\) and
3. asymmetric: \(u \succ v \Rightarrow v \not\succ u\).

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

```java
public interface Vec<T> extends Comparable<Vec<T>> {
  public T data();
  public int length();
  public ElementComparator<T> comparator();
  public ElementDistance<T> distance();
  public Comparator<T> dominance();
}
```

Listing 4.6: Vec interface

Listing 4.6 shows the necessary methods of the \texttt{Vec} interface. This methods are sufficient to do all the optimization calculations. The \texttt{data()} method returns the underlying vector type, like \texttt{double[]} or \texttt{int[]}. With the \texttt{ElementComparator}, which is returned by the \texttt{comparator()} method, it is possible to compare single elements of the vector type \(T\). This is similar to the \texttt{ElementDistance} function,
4.1. IO.JENETICS.EXT

CHAPTER 4. MODULES

Figure 4.1.9: Maximizing Pareto front

returned by the `distance()` method, which calculates the distance of two vector elements. The last method, `dominance()`, returns the Pareto dominance comparator, `≻`. Since it is quite a bothersome to implement all this needed methods, the `Vec` interface comes with a set of factory methods, which allows to create `Vec` instance for some primitive array types.

```java
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.

The second difference to the single-object 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 4.1.6.1 on page 88, 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`.

```java
final ISeq<Phenotype<DoubleGene, Vec<double[]>>> paretoSet =
    engine.stream()
    .limit(100)
    .collect(MOEA.toParetoSet(IntRange.of(30, 50)));
```

Since there exists a potential infinite number of Pareto optimal solutions, you have to define desired number 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 which 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 measure is used for calculating the proximity of two points and it is described in [7] and [13].

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

```java
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(
            DoubleRange.of(0, 1),
            DoubleRange.of(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 4.1.9 on the preceding page 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.

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](https://www.igi-global.com/dictionary/crowding-distance/42740).

Since the \(\succ\) relation doesn’t define a total order, sorting the population will lead to an IllegalArgumentException at runtime.
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 4.1.1), as described in \[10\]

\[
i \succ j \quad \text{if} \quad (i_{\text{rank}} < j_{\text{rank}}) \lor ((i_{\text{rank}} = j_{\text{rank}}) \land i_{\text{dist}} > j_{\text{dist}}), \tag{4.1.1}
\]

where \(i_{\text{rank}}\) denotes the non-domination rank of individual \(i\) and \(i_{\text{dist}}\) 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. \[13\]

Since the 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 listing 1.2 on page 3 for comparing the Jenetics evolution flavor.

### 4.1.6.3 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 doesn’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.

### 4.2 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)\[14\]. The io.jenetics-prog module contains classes which enables the Jenetics library doing GP. It

\[\text{https://en.wikipedia.org/wiki/Genetic_programming}\]
introduces a `ProgramGene` and `ProgramChromosome` pair, which serves as the main data-structure for genetic programs. A `ProgramGene` is essentially a tree (AST\(^{12}\)) of operations (Op) stored in a `ProgramChromosome`\(^{13}\).

### 4.2 Operations

When creating own genetic programs, it is not necessary to derive own 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.

```java
public interface Op<T> {
    public String name();
    public int arity();
    public T apply(T[] args);
}
```

**Listing 4.7: GP Op interface**

The generic type of the Op interface (see listing 4.7) 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.

```
final Op<Double> add = Op.of("+", 2, v -> v[0] + v[1]);
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. This operations need to have child nodes, where the number of children must be equal to the arity of the 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.
4.2. IO.JENETICS.PROG

CHAPTER 4. MODULES

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 \texttt{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.

\textbf{Const}  
The \texttt{Const} operation will always return the same, constant, value when evaluated.

\begin{verbatim}
final Const<Double> one = Const.of(1.0);
final Const<Double> pi = Const.of("PI", Math.PI);
\end{verbatim}

We 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.

\textbf{EphemeralConst}  
An ephemeral constant is a terminal operation, which encapsulates a value that is generated at run time from the \texttt{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.

\begin{verbatim}
final Op<Double> rand1 = EphemeralConst.of(RandomRegistry.getRandom()::nextDouble);
final Op<Double> rand2 = EphemeralConst.of("R", RandomRegistry.getRandom()::nextDouble);
\end{verbatim}

The ephemeral constant value is determined when it is inserted in the tree and never changes until it is replaced by another ephemeral constant.

4.2.2 Program creation

The \texttt{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.

\begin{verbatim}
final int depth = 5;
final Op<Double> operations = ISeq.of(...);
final Op<Double> terminals = ISeq.of(...);
final ProgramChromosome<Double> program = ProgramChromosome.of(depth, operations, terminals);
\end{verbatim}

The code snippet above will create a \texttt{perfect} program tree\footnote{All leafs of a perfect tree have the same depth and all internal nodes have degree \texttt{Op.arity}.} 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**.

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.

```java
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())
  .getGene();
final double result = program.eval(3.4);
```

For a complete GP example have a look at the examples in chapter 5.7 on page 118.

### 4.2.3 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 loose 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 the usage of arbitrary **Alterer** implementations. Even alterers not implemented for **ProgramGenes**. Genes destroyed by such alterer are repaired.
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 MathTreePruneAlterer you are able to simplify some portion of the population in each generation.

```java
final Engine<ProgramGene<Double>, Double> engine = Engine
    .builder(Main::error, program)
    .minimizing()
    .alterers(
        new SingleNodeCrossover<>(),
        new Mutator<>(),
        new MathTreePruneAlterer<>(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.simplify method.

```java
final ProgramGene<Double> program = engine.stream()
    .limit(3000)
    .collect(EvolutionResult.toBestGenotype())
    .getGene();
final TreeNode<Op<Double>> expr = MathExpr.simplify(program);
```

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.

```java
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.

io.jenetics.xml

The io.jenetics.xml module allows to write and read chromosomes and genotypes 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.

XML writer

The main entry point for writing XML files is the typed XMLWriter interface. Listing 4.8 on the next page shows the interface of the XMLWriter.
Listing 4.8: XMLWriter 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.

```
final Writer<IntegerChromosome> writer =
    elem("int-chromosome",
        attr("length", map(ch -> ch.length())),
        elem("min", Writer.<Integer>text().map(ch -> ch.getMin())),
        elem("max", Writer.<Integer>text().map(ch -> ch.getMax())),
        elem("alleles",
            elems("allele", Writer.<Integer>text().map(ch -> ch.toSeq().map(g -> g.getAllele())))
        )
    );
```

4.3.2 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 4.9.

```
public abstract class Reader<T> {
    public abstract T read(final XMLStreamReader xml)
        throws XMLStreamException;
}
```

Listing 4.9: XMLReader class
When creating a `XMLReader`, the structure of the XML must be defined in a similar way as for the `XMLWriter`. 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.

```java
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)))
    );
```

### 4.3.3 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.

```java
final Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0, 1.0, 100),
    chromosomeCount);
```

Table 4.3.1 shows the required space of the marshaled genotypes for different marshalling methods: (a) Java serialization, (b) JAXB\(^{15}\) serialization and (c) `XMLWriter`.

<table>
<thead>
<tr>
<th>Chromosome count</th>
<th>Java serialization</th>
<th>JAXB</th>
<th>XML writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0017 MiB</td>
<td>0.0045 MiB</td>
<td>0.0035 MiB</td>
</tr>
<tr>
<td>10</td>
<td>0.0090 MiB</td>
<td>0.0439 MiB</td>
<td>0.0346 MiB</td>
</tr>
<tr>
<td>100</td>
<td>0.0812 MiB</td>
<td>0.4379 MiB</td>
<td>0.3459 MiB</td>
</tr>
<tr>
<td>1000</td>
<td>0.8039 MiB</td>
<td>4.3772 MiB</td>
<td>3.4578 MiB</td>
</tr>
<tr>
<td>10000</td>
<td>8.0309 MiB</td>
<td>43.7730 MiB</td>
<td>34.5795 MiB</td>
</tr>
<tr>
<td>100000</td>
<td>80.3003 MiB</td>
<td>437.7283 MiB</td>
<td>345.7940 MiB</td>
</tr>
</tbody>
</table>

Table 4.3.1: Marshaled object size

\(^{15}\)The JAXB marshalling has been removed in version 4.0. It is still part of the table for comparison with the new XML marshalling.
Using the Java serialization will create the smallest files and the XMLWriter of the io.jenetics.xml module will create files roughly 75% the size of the JAXB serialized genotypes. The size of the marshaled also influences the write performance. As you can see in diagram 4.3.1 the Java serialization is the fastest marshalling method, followed by the JAXB marshalling. The XMLWriter is the slowest one, but still comparable to the JAXB method.

![Diagram 4.3.1: Genotype write performance](image)

For reading the serialized genotypes, we will see similar results (see diagram 4.3.2 on the next page). 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.

### 4.4 io.jenetics.prngine

The prngine module contains pseudo-random number generators for sequential and parallel Monte Carlo simulations. 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 extends the Java Random class, which makes it easily usable in other projects.

---

16 This module is not part of the Jenetics project directly. Since it has no dependency to any of the Jenetics modules, it has been extracted to a separate GitHub repository (https://github.com/jenetics/prngine) with an independent versioning.

17 https://de.wikipedia.org/wiki/Monte-Carlo-Simulation
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. 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* (JKISS64, page 10) David Jones, UCL Bioinformatics Group. 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.

**MT19937_32Random** This is a 32-bit version of Mersenne Twister pseudo random number generator.\(^{19}\)

---

\(^{16}\) https://github.com/jenetics/trng4

\(^{18}\) https://en.wikipedia.org/wiki/Mersenne_Twister
4.4. IDENETICS.PRNGINE

CHAPTER 4. MODULES

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.)

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 has been tested with the dieharder test suite. Table 4.4.1 shows the statistical performance of the implemented PRNGs, including the Java Random implementation. Beside the XOR32ShiftRandom class, the j.u.Random implementation has the poorest performance, concerning its statistical performance.

<table>
<thead>
<tr>
<th>PRNG</th>
<th>Passed</th>
<th>Weak</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>KISS32Random</td>
<td>108</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>KISS64Random</td>
<td>109</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>LCG64ShiftRandom</td>
<td>110</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>MT19937_32Random</td>
<td>113</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MT19937_64Random</td>
<td>111</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>XOR32ShiftRandom</td>
<td>101</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>XOR64ShiftRandom</td>
<td>107</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>j.u.Random</td>
<td>106</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.4.1: Dieharder results

The second important performance measure for PRNGs is the number of random number it is able to create per second. Table 4.4.2 on the next page shows the PRN creation speed for all implemented generators. The slowest random engine is the j.u.Random class, which is caused by the synchronized implementations. When the only the creation speed counts, the j.u.c.ThreadLocalRandom is the random engine to use.

---

20. [http://digitalcommons.wayne.edu/jmasm/vol2/iss1/2/](http://digitalcommons.wayne.edu/jmasm/vol2/iss1/2/)

21. 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.
### Table 4.4.2: PRNG speed

<table>
<thead>
<tr>
<th>PRNG</th>
<th>$10^6$ int/s</th>
<th>$10^6$ float/s</th>
<th>$10^6$ long/s</th>
<th>$10^6$ double/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>KISS32Random</td>
<td>189</td>
<td>143</td>
<td>129</td>
<td>108</td>
</tr>
<tr>
<td>KISS64Random</td>
<td>128</td>
<td>124</td>
<td>115</td>
<td>124</td>
</tr>
<tr>
<td>LCG64ShiftRandom</td>
<td>258</td>
<td>185</td>
<td>261</td>
<td>191</td>
</tr>
<tr>
<td>MT19937_32Random</td>
<td>140</td>
<td>115</td>
<td>92</td>
<td>82</td>
</tr>
<tr>
<td>MT19937_64Random</td>
<td>148</td>
<td>120</td>
<td>148</td>
<td>120</td>
</tr>
<tr>
<td>XOR32ShiftRandom</td>
<td>227</td>
<td>161</td>
<td>140</td>
<td>120</td>
</tr>
<tr>
<td>XOR64ShiftRandom</td>
<td>225</td>
<td>166</td>
<td>235</td>
<td>166</td>
</tr>
<tr>
<td>j.u.Random</td>
<td>91</td>
<td>89</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>j.u.c.JLRandom</td>
<td>264</td>
<td>224</td>
<td>268</td>
<td>216</td>
</tr>
</tbody>
</table>
Appendix
Chapter 5

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.

```bash
$ ./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 6 on page 124.

5.1 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.

```java
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;
import io.jenetics.BitChromosome;
import io.jenetics.BitGene;
import io.jenetics.Genotype;
import io.jenetics.Mutator;
import io.jenetics.Phenotype;
import io.jenetics.RouletteWheelSelector;
import io.jenetics.SinglePointCrossover;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionStatistics;

public class OnesCounting {
    
    // This method calculates the fitness for a given genotype.
    private static Integer count(final Genotype<BitGene> gt) {
        return gt.getChromosome()
            .as(BitChromosome.class)
            .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.
5.1. ONES COUNTING  
CHAPTER 5. EXAMPLES

```java
public static void main(String[] args) {
    // 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 For the other default values (population size, maximal age, ...) have a look at the Javadoc: [http://jenetics.io/javadoc/jenetics/4.3/index.html](http://jenetics.io/javadoc/jenetics/4.3/index.html)
5.2. REAL FUNCTION

In this example we try to find the minimum value of the function
\[ f(x) = \cos \left( \frac{1}{2} + \sin (x) \right) \cdot \cos (x). \]  

The graph of function 5.2.1, in the range of \([0, 2\pi]\), is shown in figure 5.2.1 and the listing beneath shows the GA implementation which will minimize the function.

```java
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static java.lang.Math.sin;
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;
```
public class RealFunction {

    // The fitness function.
    private static double fitness(final double x) {
        return cos(0.5 + sin(x)) * cos(x);
    }

    public static void main(final String[] args) {
        final Engine<DoubleGene, Double> engine = Engine
            // Create a new builder with the given fitness
            // function and chromosome.
            .builder(
                RealFunction::fitness,
                Codecs.ofScalar(DoubleRange.of(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 $1 \times 1$ DoubleChromosomes whose values are restricted to the range $[0, 2\pi]$.
5.3. RASTRIGIN FUNCTION

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.

5.3 Rastrigin function

The Rastrigin function\footnote{https://en.wikipedia.org/wiki/Rastrigin_function} is often used to test the optimization performance of genetic algorithm.

\[ f(x) = A n + \sum_{i=1}^{n} \left( x_i^2 - A \cos (2 \pi x_i) \right). \] (5.3.1)

As the plot in figure 5.3.1 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 \( x = 0 \) with \( f(x) = 0 \).
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 5.2 on page 106. Beside the different fitness function, the `Codec` for `double` vectors is used, instead of the `double` scalar `Codec`.

```java
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;
    }

    public static void main(final String[] args) {
        final Engine<DoubleGene, Double> engine = Engine.builder()
            .fitness(RastriginFunction::fitness)
            .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.
5.4 0/1 Knapsack

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

```java
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;
import java.util.Random;
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.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;
    }
```

public final double value;

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

// Create a new random knapsack item.
static Item random() {
    final Random r = RandomRegistry.getRandom();
    return new Item(
        r.nextDouble() * 100,
        r.nextDouble() * 100
    );
}

// Collector for summing up the knapsack items.
static Collector<Item, ?, Item> toSum() {
    return Collector.of(
        () -> new double[2],
        (a, b) -> {a[0] += b.size; a[1] += b.value;},
        (a, b) -> {a[0] += b[0]; a[1] += b[1]; return a;},
        r -> new Item(r[0], r[1])
    );
}

// Creating the fitness function.
static Function<ISeq<Item>, Double> fitness(final double size) {
    return items -> {
        final Item sum = items.stream().collect(Item.toSum());
        return sum.size <= size ? sum.value : 0;
    };
}

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

    final ISeq<Item> items =
        Stream.generate(Item::random)
            .limit(nitems)
            .collect(ISeq.toISeq());

    // Configure and build the evolution engine.
    final Engine<BitGene, Double> engine = Engine.
        builder(fitness(kssize), Codecs.ofSubSet(items))
        .populationSize(500)
        .survivorsSelector(new TournamentSelector<>(5))
        .offspringSelector(new RouletteWheelSelector<>())
        .alterers(new Mutator<>(0.115),
                    new SinglePointCrossover<>(0.16))
        .build();

    // Create evolution statistics consumer.
    final EvolutionStatistics<Double, ?> statistics = EvolutionStatistics.ofNumber();

    final Phenotype<BitGene, Double> best = engine.stream()
        // Truncate the evolution stream after 7 'steady'

    // Configure and build the evolution engine.
    final Engine<BitGene, Double> engine = Engine.
        builder(fitness(kssize), Codecs.ofSubSet(items))
        .populationSize(500)
        .survivorsSelector(new TournamentSelector<>(5))
        .offspringSelector(new RouletteWheelSelector<>())
        .alterers(
            new Mutator<>(0.115),
            new SinglePointCrossover<>(0.16))
        .build();

    // Create evolution statistics consumer.
    final EvolutionStatistics<Double, ?> statistics = EvolutionStatistics.ofNumber();

    final Phenotype<BitGene, Double> best = engine.stream()
        // Truncate the evolution stream after 7 'steady'
5.5. TRAVELING SALESMAN  
CHAPTER 5. EXAMPLES

```java
// 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 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 = 436.0982486289
  - var = 11431.801291812390
  - std = 106.919601999878

---

5.5 Traveling salesman

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

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

```java
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.Random;
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
    public Codec<ISeq<double[]>, EnumGene<double[]>> codec() {
        return Codecs.ofPermutation(_points);
    }

    // Create a new TSM example problem with the given number of stops.
    // All stops lie on a circle with the given radius.
    public static TravelingSalesman of(int stops, double radius) {
        final MSeq<double[]> points = MSeq.ofLength(stops);
        final double delta = 2.0 * PI / stops;
        for (int i = 0; i < stops; ++i) {
            final double alpha = delta * i;
            final double x = cos(alpha) * radius + radius;
            final double y = sin(alpha) * radius + radius;
            points.set(i, new double[]{x, y});
        }
    }
}
5.5. TRAVELING SALESMAN CHAPTER 5. EXAMPLES

```java
// Shuffling of the created points.
final Random random = RandomRegistry.getRandom();
for (int j = points.length() - 1; j > 0; --j) {
    final int i = random.nextInt(j + 1);
    final double[] tmp = points.get(i);
    points.set(i, points.get(j));
    points.set(j, tmp);
}
return new TravelingSalesman(points.toISeq());
}
public static void main(String[] args) {
    int stops = 20; double R = 10;
    double minPathLength = 2.0 * stops * R * sin(PI/stops);
    TravelingSalesman tsm = TravelingSalesman.of(stops, R);
    Engine<EnumGene<double[]>, Double> engine = Engine
        .builder(tsm)
        .optimize(Optimize.MINIMUM)
        .maximalPhenotypeAge(11)
        .populationSize(500)
        .alterers(
            new SwapMutator<>(0.2),
            new PartiallyMatchedCrossover<>(0.35))
        .build();
    // Create evolution statistics consumer.
    EvolutionStatistics<Double, ?> statistics = EvolutionStatistics.ofNumber();
    Phenotype<EnumGene<double[]>, Double> best =
        engine.stream()
        // Truncate the evolution stream after 25 "steady"
        // generations.
        .limit(bySteadyFitness(25))
        // The evolution will stop after maximal 250
        // generations.
        .limit(250)
        // Update the evaluation statistics after
        // each generation
        .peek(statistics)
        // Collect (reduce) the evolution stream to
        // its best phenotype.
        .collect(toBestPhenotype());
    out.println(statistics);
    out.println("Known min path length: " + minPathLength);
    out.println("Found min path length: " + best.getFitness());
}
```

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

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.

### 5.6 Evolving images

The following example tries to approximate a given image by semitransparent polygons. It comes with an Swing UI, where you can immediately start your own experiments. After compiling the sources with

```
$ ./gradlew jar
```

you can start the example by calling

```
$ ./jrun io.jenetics.example.image.EvolvingImages
```

Figure 5.6.1 on the next page 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 of the population.
- **Tournament size** The example uses a TournamentSelector for selecting the offspring population. This parameter lets you set the number of individual 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 an 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 to do 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 \
```
5.6. **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 5.1 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 5.1: Default `engine.properties`

For a quick start, you can simply call

```
$ ./jrun io.jenetics.example.image.EvolvingImages evolve
```

![Image a) $10^0$ generations](image_a.png)  
![Image b) $10^2$ generations](image_b.png)  
![Image c) $10^4$ generations](image_c.png)

![Image d) $10^4$ generations](image_d.png)  
![Image e) $10^5$ generations](image_e.png)  
![Image f) $10^6$ generations](image_f.png)

**Figure 5.6.2: Evolving Mona Lisa images**

The images in figure 5.6.2 shows the resulting polygon images after the given number of generations. They were created with the command line version of the program using the default `engine.properties` file (listing 5.1):

```
$ ./jrun io.jenetics.example.image.EvolvingImages evolve \  
--generations 1000000 \  
--image-generation 100
```

117
5.7 Symbolic regression

Symbolic regression is a classical example in genetic programming and tries to find a mathematical expression for a given set of values.

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. [18]

The following example shows how to solve the GP problem with Jenetics. We are trying to find the polynomial, \(4x^3 - 3x^2 + x\), which fits a given data set. The sample data where created with the polynomial we are searching for. This makes it easy to check the quality of the approximation found by the GP.

```java
import static java.lang.Math.pow;
import java.util.Arrays;
import io.jenetics.Genotype;
import io.jenetics.Mutator;
import io.jenetics.engine.Codec;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionResult;
import io.jenetics.util.ISeq;
import io.jenetics.util.RandomRegistry;
import io.jenetics.extSingleNodeCrossover;
import io.jenetics.prog.ProgramChromosome;
import io.jenetics.prog.ProgramGene;
import io.jenetics.prog.op.EphemeralConst;
import io.jenetics.prog.op.MathOp;
import io.jenetics.prog.op.Op;
import io.jenetics.prog.op.Var;

public class SymbolicRegression {
    // Sample data created with 4x^3 - 3x^2 + x
    static final double [][] SAMPLES = new double [][] {
        {-1.0, -8.0000},
        {-0.9, -6.2460},
        {-0.8, -4.7680},
        {-0.7, -3.5420},
        {-0.6, -2.5440},
        {-0.5, -1.7500},
        {-0.4, -1.1360},
        {-0.3, -0.6780},
        {-0.2, -0.3520},
        {-0.1, -0.1340},
        {0.0, 0.0000},
        {0.1, 0.0740},
        {0.2, 0.1120},
        {0.3, 0.1380},
        {0.4, 0.1760},
        {0.5, 0.2500},
        {0.6, 0.3840},
        {0.7, 0.6020},
        {0.8, 0.9280},
        {0.9, 1.3860},
        {1.0, 2.0000}
    }
```
5.7. SYMBOLIC REGRESSION  CHAPTER 5. EXAMPLES

One output of a GP run is shown in figure 5.7.1 on the next page. If we simplify this program tree, we will get exactly the polynomial which created the sample data.
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). [7]

DTLZ1 is an $M$-objective problem with linear Pareto-optimal front. [12]
5.8. DTLZ1

\[
f_1(x) = \frac{1}{2} x_1 x_2 \cdots x_{M-1} (1 + g(x_M)),
\]

\[
f_2(x) = \frac{1}{2} x_1 x_2 \cdots (1 - x_{M-1}) (1 + g(x_M)),
\]

\[\vdots\]

\[
f_{M-1}(x) = \frac{1}{2} x_1 (1 - x_2) (1 + g(x_M)),
\]

\[
f_M(x) = \frac{1}{2} (1 - x_1) (1 + g(x_M)),
\]

\(\forall i \in [1, \ldots, n]: 0 \leq x_i \leq 1\)

The functional \(g(x_M)\) requires \(|x_M| = k\) variables and must take any function with \(g \geq 0\). Typically \(g\) is defined as:

\[
g(x_M) = 100 \left[ |x_M| + \left( x - \frac{1}{2} \right)^2 - \cos \left( 20\pi \left( x - \frac{1}{2} \right) \right) \right].
\]

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.

```java
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static java.lang.Math.pow;
import io.jenetics.DoubleGene;
import io.jenetics.Mutator;
import io.jenetics.Phenotype;
import io.jenetics.TournamentSelector;
import io.jenetics.engine.Codecs;
import io.jenetics.engine.Engine;
import io.jenetics.engine.Problem;
import io.jenetics.util.IntRange;
import io.jenetics.ext.SimulatedBinaryCrossover;
import io.jenetics.ext.moea.MOEA;
import io.jenetics.ext.moea.NSGA2Selector;
import io.jenetics.ext.moea.Vec;

public class DTLZ1 {
    private static final int VARIABLES = 4;
    private static final int OBJECTIVES = 3;
    private static final int K = VARIABLES - OBJECTIVES + 1;

    static final Problem< double[] , DoubleGene , Vec< double[] >> PROBLEM = Problem.of(
        DTLZ1::f ,
        Codecs.ofVector(DoubleRange.of(0, 1.0) , VARIABLES)
    );

    static Vec< double[] > f( final double[] x ) {
        double g = 0.0;
        for( int i = VARIABLES - K; i < VARIABLES; i++ ) {
            g += pow(x[i] - 0.5 , 2.0) - cos(20.0*PI*(x[i] - 0.5));
        }
        return g;
    }
}
```
The listing above shows the encoding of the DTLZ1 problem with the Jenetics library. Figure 5.8.1 on the following page shows the output DTLZ1 optimization.
Figure 5.8.1: Pareto front DTLZ1
Chapter 6

Build

For building the Jenetics library from source, download the most recent, stable package version from [https://github.com/jenetics/jenetics/releases](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 as build system and organizes the source into sub-projects (modules). Each sub-project is located in it’s 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).

- **jenetics.prog**: The module contains classes which allows to do genetic programming (GP). It seamlessly works with the existing Evolution-Stream and evolution Engine.

- **jenetics.xml**: XML marshalling module for the Jenetics base data structures.

---

1 [http://gradle.org/downloads](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.doc**: Contains the code of the web-site 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

```bash
$ cd <build-dir>
$ gradle jar
```

or

```bash
$ ./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.

\(^3\)https://github.com/jenetics/prngine
External library dependencies  The following external projects are used for running and/or building the Jenetics library.

- **TestNG**
  - License: Apache License, Version 2.0
  - Scope: test

- **Apache Commons Math**
  - Version: 3.6.1
  - License: Apache License, Version 2.0
  - Scope: test

- **EqualsVerifier**
  - Version: 2.5.2
  - Homepage: [http://jqno.nl/equalsverifier/](http://jqno.nl/equalsverifier/)
  - Download: [https://github.com/jqno/equalsverifier/releases](https://github.com/jqno/equalsverifier/releases)
  - License: Apache License, Version 2.0
  - Scope: test

- **Java2Html**
  - Version: 5.0
  - Homepage: [http://www.java2html.de/](http://www.java2html.de/)
  - Download: [http://www.java2html.de/java2html_50.zip](http://www.java2html.de/java2html_50.zip)
  - License: GPL or CPL1.0
  - Scope: javadoc

- **Gradle**
  - Version: 4.10
  - Homepage: [http://gradle.org/](http://gradle.org/)
  - Download: [http://services.gradle.org/distributions/gradle-4.10-bin.zip](http://services.gradle.org/distributions/gradle-4.10-bin.zip)
  - License: Apache License, Version 2.0
  - Scope: build

**Maven Central**  The whole Jenetics package can also be downloaded from the Maven Central repository [http://repo.maven.apache.org/maven2](http://repo.maven.apache.org/maven2)
pom.xml snippet for Maven

```
<dependency>
    <groupId>io.jenetics</groupId>
    <artifactId>module-name</artifactId>
    <version>4.3.0</version>
</dependency>
```

Gradle

`'io.jenetics:module-name:4.3.0'`

License  The library itself is licensed under the [Apache License, Version 2.0](http://www.apache.org/licenses/LICENSE-2.0).

Copyright 2007-2018 Franz Wilhelmstötter
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
Bibliography


Index

0/1 Knapsack, 110
2-point crossover, 19
3-point crossover, 20

AdaptiveEngine, 86
Allele, 6, 9
Alterer, 16, 43
AnyChromosome, 41
AnyGene, 40
Apache Commons Math, 126
Architecture, 4
Assignment problem, 56
Base classes, 5
BigIntegerGene, 81
Block splitting, 34
Boltzmann selector, 15
Build, 124
Gradle, 124
gradlew, 124
Chromosome, 7, 8, 40
recombination, 18
scalar, 45
variable length, 7
Codec, 51
Composite, 56
Mapping, 56
Permutation, 55
Scalar, 52
Subset, 56
Vector, 55
Compile, 125
Composite codec, 56
ConcatEngine, 85
Concurrency, 30
configuration, 30
maxBatchSize, 32
maxSurplusQueuedTaskCount, 31
splitThreshold, 31
tweaks, 31

Crossover
2-point crossover, 19
3-point crossover, 20
Intermediate crossover, 21
Line crossover, 21
Multiple-point crossover, 18
Partially-matched crossover, 19
Simulated binary crossover, 51
Single-point crossover, 18, 19
Uniform crossover, 19, 20
CyclicEngine, 86
Dieharder, 73
Directed graph, 50
Distinct population, 71
Domain classes, 6
Domain model, 6
Download, 124
DTLZ1, 120
Elite selector, 16, 43
Elitism, 16, 43
Encoding, 45
Affine transformation, 17
Directed graph, 50
Graph, 49
Real function, 45
Scalar function, 46
Undirected graph, 49
Vector function, 47
Weighted graph, 50
Engine, 22, 44
Evaluator, 29
reproducible, 68
Engine classes, 21
Ephemeral constant, 94
ES, 69
Evolution
Engine, 22
interception, 71
performance, 69
reproducible, 68
Stream, 21
Evolution strategy, 69
(μ + λ)-ES, 70
(μ, λ)-ES, 70
Evolution time, 63
Evolution workflow, 4
EvolutionResult, 26
mapper, 71
EvolutionStatistics, 27
EvolutionStream, 25
EvolutionStreamable, 85
Evolving images, 115, 116
Examples, 104
0/1 Knapsack, 110
Evolving images, 115, 116
Ones counting, 104
Rastrigin function, 108
Real function, 106
Traveling salesman, 112
Exponential-rank selector, 15
Fitness convergence, 65
Fitness function, 22
Fitness threshold, 64
Fixed generation, 61
FlatTree, 80
Gaussian mutator, 17
Gene, 6, 39
validation, 7
Gene convergence, 67
Genetic algorithm, 9
Genetic programming, 92, 118
Const, 94
Operations, 93
Program creation, 94
Program pruning, 96
Program repair, 95
Var, 93
Genotype, 8
scalar, 10
Validation, 24
vector, 9
Git repository, 124
GP, 92
Gradle, 124
gradlew, 124
Graph, 49
Hello World, 2
Installation, 124
Interception, 71
Internals, 73
io.jenetics.ext, 78
io.jenetics.prg, 99
io.jenetics.prog, 92
io.jenetics.xml, 96
Java property
maxBatchSize, 82
maxSurplusQueuedTaskCount, 31
splitThreshold, 31
Java2Html, 126
LCG64ShiftRandom, 43
Leapfrog, 34
License, 127
Linear-rank selector, 14
Mapping codec, 56
Mean alterer, 29
Modifying Engine, 84
Modules, 77
io.jenetics.ext, 78
io.jenetics.prg, 99
io.jenetics.prog, 92
io.jenetics.xml, 96
Mona Lisa, 117
Monte Carlo selector, 13, 69
MOO, 88
Multi-objective optimization, 88
Multiple-point crossover, 18
Mutation, 16
Mutator, 17
NSGA2 selector, 92
Ones counting, 104
Operation classes, 12
Package structure, 5
Parentheses tree, 79
Pareto dominance, 88
Pareto efficiency, 88
Partially-matched crossover, 19
Permutation codec, 55
Phenotype, 11
INDEX

Validation, 24
Population, 6
  unique, 25
Population convergence, 66
PRNG, 32
  Block splitting, 34
  LCG64ShiftRandom, 34
  Leapfrog, 34
  Parameterization, 34
  Random seeding, 33
PRNG testing, 73
Probability selector, 13
Problem, 58
Quantile, 38
Random, 32
  Engine, 32
  LCG64ShiftRandom, 34
  Registry, 32
  seeding, 74
  testing, 73
Random seeding, 33
Randomness, 32
  Rastrigin function, 108
  Real function, 106
  Recombination, 17
  Reproducibility, 68
  Roulette-wheel selector, 14
SBX, 81
  Scalar chromosome, 45
  Scalar codec, 52
  Scalar genotype, 45
  Seeding, 74
  Selector, 12
    Elite, 43
  Seq, 56
  Serialization, 35
  Simulated binary crossover, 81
  Single-node crossover, 81
  Single-point crossover, 18
  Source code, 124
  Statistics, 37
  Steady fitness, 61
  Stochastic-universal selector, 15
  Subset codec, 53
  Swap mutator, 17
  Symbolic regression, 118
Termination, 60
Evolution time, 63
  Fitness convergence, 65
  Fitness threshold, 64
  Fixed generation, 61
  Gene convergence, 67
  Population convergence, 66
  Steady fitness, 61
TestNG, 126
  Tournament selector, 13
  Traveling salesman, 112
  Tree, 78
  TreeGene, 81
  Truncation selector, 13
  Undirected graph, 49
  Uniform crossover, 20
  Unique fitness tournament selector, 92
  Unique population, 71
  Validation, 7, 24
  Vec, 89
  Vector codec, 53
  Weasel program, 82
  WeaselMutator, 83
  WeaselSelector, 83
  Weighted graph, 50

133