Franz Wilhelmstötter

franz.wilhelmstoetter@gmail.com

https://jenetics.io

7.2.0-2023/08/30

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

1 Fundamentals .......................................................... 1
   1.1 Introduction ...................................................... 1
   1.2 Architecture ................................................... 4
   1.3 Base classes ..................................................... 5
      1.3.1 Domain classes ............................................. 6
         1.3.1.1 Gene .................................................. 6
         1.3.1.2 Chromosome .......................................... 7
         1.3.1.3 Genotype ............................................. 7
         1.3.1.4 Phenotype ........................................... 11
         1.3.1.5 Population .......................................... 11
      1.3.2 Operation classes ........................................... 12
         1.3.2.1 Selector .............................................. 12
         1.3.2.2 Alterer ............................................... 16
      1.3.3 Engine classes ............................................... 22
         1.3.3.1 Fitness function ..................................... 22
         1.3.3.2 Engine ............................................... 23
         1.3.3.3 Evolution ........................................... 25
         1.3.3.4 EvolutionStream ..................................... 26
         1.3.3.5 EvolutionResult ..................................... 27
         1.3.3.6 EvolutionStatistics ................................ 28
         1.3.3.7 Evaluator ............................................ 30
   1.4 Nuts and bolts .................................................... 31
      1.4.1 Concurrency ................................................ 31
         1.4.1.1 Basic configuration .................................. 32
         1.4.1.2 Concurrency tweaks .................................. 32
      1.4.2 Randomness ................................................ 34
      1.4.3 Serialization ............................................... 37
      1.4.4 Utility classes ............................................ 38

2 Advanced topics ....................................................... 41
   2.1 Extending Jenetics ................................................ 41
      2.1.1 Genes ..................................................... 41
      2.1.2 Chromosomes ............................................. 42
      2.1.3 Selectors ................................................ 44
      2.1.4 Alterers .................................................. 45
      2.1.5 Statistics ............................................... 45
      2.1.6 Engine .................................................... 46
   2.2 Encoding ............................................................ 47
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1 Real function</td>
<td>48</td>
</tr>
<tr>
<td>2.2.2 Scalar function</td>
<td>49</td>
</tr>
<tr>
<td>2.2.3 Vector function</td>
<td>49</td>
</tr>
<tr>
<td>2.2.4 Affine transformation</td>
<td>50</td>
</tr>
<tr>
<td>2.2.5 Graph</td>
<td>52</td>
</tr>
<tr>
<td>2.3 Codec</td>
<td>54</td>
</tr>
<tr>
<td>2.3.1 Scalar codec</td>
<td>55</td>
</tr>
<tr>
<td>2.3.2 Vector codec</td>
<td>55</td>
</tr>
<tr>
<td>2.3.3 Matrix codec</td>
<td>56</td>
</tr>
<tr>
<td>2.3.4 Subset codec</td>
<td>56</td>
</tr>
<tr>
<td>2.3.5 Permutation codec</td>
<td>58</td>
</tr>
<tr>
<td>2.3.6 Mapping codec</td>
<td>59</td>
</tr>
<tr>
<td>2.3.7 Composite codec</td>
<td>60</td>
</tr>
<tr>
<td>2.3.8 Invertible codec</td>
<td>61</td>
</tr>
<tr>
<td>2.4 Problem</td>
<td>62</td>
</tr>
<tr>
<td>2.5 Constraint</td>
<td>62</td>
</tr>
<tr>
<td>2.6 Termination</td>
<td>67</td>
</tr>
<tr>
<td>2.6.1 Fixed generation</td>
<td>68</td>
</tr>
<tr>
<td>2.6.2 Steady fitness</td>
<td>69</td>
</tr>
<tr>
<td>2.6.3 Evolution time</td>
<td>70</td>
</tr>
<tr>
<td>2.6.4 Fitness threshold</td>
<td>71</td>
</tr>
<tr>
<td>2.6.5 Fitness convergence</td>
<td>72</td>
</tr>
<tr>
<td>2.6.6 Population convergence</td>
<td>74</td>
</tr>
<tr>
<td>2.6.7 Gene convergence</td>
<td>75</td>
</tr>
<tr>
<td>2.7 Reproducibility</td>
<td>76</td>
</tr>
<tr>
<td>2.8 Evolution performance</td>
<td>76</td>
</tr>
<tr>
<td>2.9 Evolution strategies</td>
<td>77</td>
</tr>
<tr>
<td>2.9.1 $\mu, \lambda$ evolution strategy</td>
<td>77</td>
</tr>
<tr>
<td>2.9.2 $\mu + \lambda$ evolution strategy</td>
<td>78</td>
</tr>
<tr>
<td>2.10 Evolution interception</td>
<td>79</td>
</tr>
<tr>
<td>3 Modules</td>
<td>81</td>
</tr>
<tr>
<td>3.1 io.jenetics.ext</td>
<td>82</td>
</tr>
<tr>
<td>3.1.1 Data structures</td>
<td>82</td>
</tr>
<tr>
<td>3.1.1.1 Tree</td>
<td>82</td>
</tr>
<tr>
<td>3.1.1.2 Parentheses tree</td>
<td>83</td>
</tr>
<tr>
<td>3.1.1.3 Flat tree</td>
<td>84</td>
</tr>
<tr>
<td>3.1.1.4 Tree formatting</td>
<td>85</td>
</tr>
<tr>
<td>3.1.1.5 Tree reduction</td>
<td>86</td>
</tr>
<tr>
<td>3.1.2 Rewriting</td>
<td>86</td>
</tr>
<tr>
<td>3.1.2.1 Tree pattern</td>
<td>87</td>
</tr>
<tr>
<td>3.1.2.2 Tree rewriter</td>
<td>88</td>
</tr>
<tr>
<td>3.1.2.3 Tree rewrite rule</td>
<td>88</td>
</tr>
<tr>
<td>3.1.2.4 Tree rewrite system (TRS)</td>
<td>88</td>
</tr>
<tr>
<td>3.1.2.5 Constant expression rewriter</td>
<td>89</td>
</tr>
<tr>
<td>3.1.3 Genes</td>
<td>89</td>
</tr>
<tr>
<td>3.1.3.1 BigInteger gene</td>
<td>89</td>
</tr>
<tr>
<td>3.1.3.2 Tree gene</td>
<td>89</td>
</tr>
<tr>
<td>3.1.4 Operators</td>
<td>89</td>
</tr>
<tr>
<td>3.1.5 Weasel program</td>
<td>90</td>
</tr>
</tbody>
</table>
## CONTENTS

### 3.1 Modifying Engine
- 3.1.6 Modifying Engine
- 3.1.6.1 ConcatEngine
- 3.1.6.2 CyclicEngine

### 3.1.7 Multi-objective optimization
- 3.1.7.1 Pareto efficiency
- 3.1.7.2 Implementing classes
- 3.1.7.3 Termination
- 3.1.7.4 Mixed optimization

### 3.1.8 Grammatical evolution
- 3.1.8.1 Context-free grammar
- 3.1.8.2 Backus-Naur form
- 3.1.8.3 Sentence generation
- 3.1.8.4 Mapping
- 3.1.8.5 Implementing classes

### 3.2 io.jenetics.prog
- 3.2.1 Operations
- 3.2.2 Program creation
- 3.2.3 Program repair
- 3.2.4 Program pruning
- 3.2.5 Multi-root programs
- 3.2.6 Symbolic regression
  - 3.2.6.1 Loss function
  - 3.2.6.2 Complexity function
  - 3.2.6.3 Error function
  - 3.2.6.4 Sample points
  - 3.2.6.5 Regression problem
- 3.2.7 Boolean programs

### 3.3 io.jenetics.xml
- 3.3.1 XML writer
- 3.3.2 XML reader
- 3.3.3 Marshalling performance

### 3.4 io.jenetics.prngine

### 4 Internals
- 4.1 PRNG testing
- 4.2 Random seeding

### 5 Examples
- 5.1 Ones counting
- 5.2 Real function
- 5.3 Rastrigin function
- 5.4 0/1 Knapsack
- 5.5 Traveling salesman
- 5.6 Evolving images
- 5.7 Symbolic regression
- 5.8 Grammar based regression
- 5.9 DTLZ1

### 6 Build
Bibliography 154
Chapter 1

Fundamentals

Jenetics is an advanced Genetic Algorithm, Evolutionary Algorithm and Genetic Programming library, written in modern day Java. It is designed with a clear separation of the several algorithm concepts, e.g., Gene, Chromosome, Genotype, Phenotype, population and fitness Function. Jenetics allows you to minimize or maximize a given fitness function without tweaking it. In contrast to other GA implementations, the library uses the concept of an evolution stream (EvolutionStream) for executing the evolution steps. Since the EvolutionStream implements the Java Stream interface, it works smoothly with the rest of the Java Stream API. This chapter describes the design concepts and its implementation. It also gives some basic examples and best practice tips.

1.1 Introduction

Jenetics is a library, written in Java which provides Genetic algorithm (GA), Evolutionary algorithm (EA), Multi-objective optimization (MOO) and Genetic programming (GP) implementation. It has no runtime dependencies to other libraries, except the Java 17 runtime. Jenetics is available on the Maven central repository 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 [40], [29], [39], [26], [30] or [44]. For genetic programming you can have a look at [24] or [25].

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

```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.chromosome().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:
            \t" + 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 extends the Java Stream interface, it works smoothly with the rest of the Java Stream API. Now let’s have a closer look at listing 1.1 and discuss this simple program step by step:

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

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

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

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

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

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

```
| P_0 ← P_{initial} |
| F(P_0) |
| **while** !finished **do** |
| g ← g + 1 |
| S_g ← selectS(P_{g-1}) |
| O_g ← selectO(P_{g-1}) |
| O_g ← alter(O_g) |
| P_g ← filter[g_i ≥ g_{max}](S_g) + filter[g_i ≥ g_{max}](O_g) |
| F(P_g) |
```

Listing 1.2: Genetic algorithm

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

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

Figure 1.2.1: Evolution workflow

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

The evolution process terminates, if the EvolutionStream is truncated. The EvolutionStream truncation is controlled by the limit predicate. As long as the predicate returns true, the evolution is continued. 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 result. This is done in the same way as for the normal java.util.stream.Stream classes. With the additional limit(Predicate) method, it is possible to truncate the

\[^{5}\text{See section 2.6 on page 67 for a detailed description of the available termination strategies.}\]
EvolutionStream if some termination criteria is fulfilled. The separation of Engine and EvolutionStream is the separation of evolution definition and evolution execution.

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

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

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

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

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

1.3 Base classes

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

6The documentation of the whole API is part of the download package or can be viewed online: https://jenetics.io/javadoc/jenetics/7.2/index.html
1.3. BASE CLASSES

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

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

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

1.3.1 Domain classes

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

Figure 1.3.1: Domain model

Figure 1.3.1 shows the class diagram of the domain classes. The Gene is the base of the class structure. Genes are aggregated in Chromosomes, and 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

The basic building blocks of the Jenetics library contain the actual information of the encoded solution, the allele. Some of the implementations also contain domain information of the wrapped allele. This is the case for all BoundedGenes, 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

---


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

9[A constraint can restrict the space of valid values of a given problem domain. An example will be a DoubleGene, where the allowed minimal and maximal value of the double allele is part of the gene.](#)
1.3. BASE CLASSES  CHAPTER 1. FUNDAMENTALS

**Gene**s from the existing one and play a crucial role by the exploration of the problem space.

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

Listing 1.3: Gene interface

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

1.3.1.2 Chromosome

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

```java
public interface Chromosome<G>
    G get(int index);
    int length();
    Chromosome<G> newInstance(ISeq<G> genes);
}
```

Listing 1.4: Chromosome interface

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

```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 [range_min, range_max], equally distributed. Figure 1.3.2 shows the structure of a Chromosome with variable length.

1.3.1.3 Genotype

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

Figure 1.3.2: Chromosome structure

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

Figure 1.3.3: Genotype structure

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

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

As already mentioned, the Chromosomes of a Genotype don’t necessarily have to have the same size. It is only required that all genes are from the same type and the Genes within a Chromosome have the same constraints; e.g. the same minimal and maximal 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(0.1, 0.9, 5)
);
```
The code snippet in the listing above creates a Genotype with the same structure as shown in figure 1.3.3. In this example the DoubleGene has been chosen as the Gene type.

**Genotype vector** The Genotype is essentially a two-dimensional composition of 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 shows a Genotype vector in row major layout. A Genotype vector of length $n$ needs one Chromosome of length $n$. Each Gene of such a vector must obey the same constraints. E.g., for Genotype vectors containing NumericGenes, 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 alterers are more efficient in exploring the search domain.

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

The following code snippet shows the creation of a row major Genotype vector. All Alterers derived from the Recombinator do a fairly good job in exploring the problem space for a 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 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.
The code snippet below shows how to create a `Genotype` vector in column major layout. It’s a little bit more effort to create such a vector, since every `Gene` has to be wrapped into a separate `Chromosome`. The `DoubleChromosome` in the given example has a length of one, when the length parameter is omitted.

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

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

- `MeanAlterer`[^10]
- `IntermediateCrossover`[^12]

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

**Genotype scalar** A special case of a `Genotype` contains only one `Chromosome` with length one. The layout of such a `Genotype scalar` is shown in 1.3.6 Such `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`.

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

[^10]: 1.3.2.2 on page 21
[^11]: 1.3.2.2 on page 21
[^12]: 1.3.2.2 on page 21
1.3. BASE CLASSES

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

1.3.1.4 Phenotype

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

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

Listing 1.5: Phenotype class

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

1.3.1.5 Population

There is no special class which represents a population. It’s just a collection of Phenotypes. As a collection class, the ISeq interface is used. The ISeq interface allows for the expression of the immutability of the population at the type level and makes the code more readable. For a detailed description of this collection classes see section 1.4.4.
1.3.2 Operation classes

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

1.3.2.1 Selector

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

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

```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 offspring

\[ N_{O_g} = \|O_g\| = \text{rint}(\|P_g\| \cdot f_O) \]  \hspace{1cm} (1.3.2)

and the number of selected survivors

\[ N_{S_g} = \|S_g\| = \|P_g\| - \|O_g\|. \]  \hspace{1cm} (1.3.3)

The Jenetics library contains the following selector implementations:

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

Beside the well known standard selector implementation, the ProbabilitySelector is the base of a set of fitness proportional selectors.

Tournament selector In tournament selection the best individual from a random sample of 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 others \( -1 \) competitors. Note that the worst individual never survives, and the best individual wins in all the
tournaments in which it participates. The selection pressure can be varied by changing the tournament size, $s$. For large values of $s$, weak individuals have less chance of being selected. Compared with fitness proportional selectors, the tournament selector is often used in practice because of its lack of stochastic noise. Tournament selectors are also independent to the scaling of the genetic algorithm fitness function.

**Truncation selector** Individuals are sorted according to their fitness and only their 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.\[40\]

**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

$$
Overall \ fitness \ F = \sum_{i=0}^{N-1} f_i
$$

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

$$
P(i, k) = \binom{n}{k} P(i)^k (1 - P(i))^{n-k} \quad (1.3.5)
$$
where \( n \) is the overall number of selected individuals and \( k \) the number of individuals \( i \) in the set of selected individuals. The runtime complexity of the implemented probability selectors is \( O(n + \log(n)) \) instead of \( O(n^2) \) as for the naive approach: *A binary (index) search is performed on the summed probability array.*

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

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

(1.3.6)

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

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

(1.3.7)

where

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

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

**Linear-rank selector** The roulette-wheel selector will have problems when the fitness values differ very much. If the best *Chromosome* fitness is 90%, its circumference occupies 90% of roulette-wheel, and then other *Chromosomes* have too few chances to be selected. 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 \( n^- \) is the probability of the worst individual to be selected and \( n^+ \) 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:

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

(1.3.9)
where \( c \) must be within the range \([0, 1)\). A small value of \( c \) increases the probability of the best individual to be selected. If \( c \) is set to zero, the selection probability of the best individual is set to one. The selection probability of all other individuals is zero. A value near one equalizes the selection probabilities. This selector sorts the population in descending order before calculating the selection probabilities.

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

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

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

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

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 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, 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 rediscovering previously refused partial solutions. Individuals that are preserved through
elitism remain eligible for selection as parents of the next generation. Elitism is also related with memory: remember the best solution found so far. A problem with elitism is that it may cause the GA to converge to a local optimum, so pure elitism is a race to the nearest local optimum. The elite selector implementation of the Jenetics library also lets you specify the selector for the non-elitist 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 too close to one another. A genetic algorithm can improve the solution solely by the mutation operator. Even if most of the search is being performed by crossover, mutation can be vital to provide the diversity which crossover needs.

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

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

$$
\hat{\mu} = N_p \cdot N_g \cdot P(m)
$$

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

**Mutator** The mutator has to deal with the problem, that the genes are arranged in a hierarchical structure with three levels (see chapter 1.3.1.3). The mutator selects the gene which will be mutated in three steps:

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

The needed sub selection probabilities are set to
\[
P_G(m) = P_C(m) = P_g(m) = \sqrt[3]{P(m)}. \tag{1.3.13}
\]

**Gaussian mutator** The Gaussian mutator performs the mutation of number genes. 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
\[
\hat{\sigma}^2 = \left( \frac{g_{\text{max}} - g_{\text{min}}}{4} \right)^2 \tag{1.3.14}
\]
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) combines elements of existing solutions in order to create a new solution, with some of the properties of each parent. Recombination creates a new chromosome by combining parts of two (or more) parent chromosomes. This combination of chromosomes can be made by selecting one or more crossover points, splitting these chromosomes on the selected points, and merging those portions of different chromosomes to form new ones.

```cpp
1  void recombine(final MSeq<Phenotype<G, C>> pop) {
2    // Select the Genotypes for crossover.
3    final RandomGenerator random = RandomRegistry.random();
4    final int i1 = random.nextInt(pop.length());
5    final int i2 = random.nextInt(pop.length());
6    final Phenotype<G, C> pt1 = pop.get(i1);
7    final Phenotype<G, C> pt2 = pop.get(i2);
8    final Genotype<G> gt1 = pt1.genotype();
9    final Genotype<G> gt2 = pt2.genotype();
10   // Choosing the Chromosome for crossover.
11   final int chIndex =
12     random.nextInt(min(gt1.length(), gt2.length()));
13   final MSeq<Chromosome<G>> c1 = MSeq.of(gt1);
14   final MSeq<Chromosome<G>> c2 = MSeq.of(gt2);
15   final MSeq<G> genes1 = MSeq.of(c1.get(chIndex));
16   final MSeq<G> genes2 = MSeq.of(c2.get(chIndex));
17   // Perform the crossover.
18
19}
```
Listing 1.6: Chromosome selection for recombination

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

Because of the possible different chromosome length and/or chromosome constraints within a genotype, only chromosomes with the same genotype position are recombined (see listing 1.6).

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

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

![Single-point crossover](image)

Figure 1.3.9: Single-point crossover

Figure 1.3.9 shows how the `SinglePointCrossover` class is performing the crossover for different crossover points. Sub figure `a)` shows the two chromosomes chosen for crossover. The examples in sub figures `b)` to `f)` illustrate the crossover results for indexes `0, 1, 3, 6 and 7`.

```java
   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.
   pop.set(i1, Phenotype.of(Genotype.of(c1.toISeq())));
   pop.set(i2, Phenotype.of(Genotype.of(c2.toISeq())));
```

18
Multi-point crossover  If the MultiPointCrossover class is created with one crossover point, it behaves exactly like the single-point crossover. The figures in 1.3.10 shows how the multi-point crossover works with two crossover points. Figure a) shows the two chromosomes chosen for crossover, b) shows the crossover result for the crossover points at index 0 and 4, c) uses crossover points at index 3 and 6 and d) at index 0 and 7.

![Figure 1.3.10: 2-point crossover](image)

Figure 1.3.10: 2-point crossover

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

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

The PMX is similar to the two-point crossover. A crossing region is chosen by selecting two crossing points (see figure 1.3.12 a)).

After performing the crossover we normally got two invalid chromosomes (figure 1.3.12 b)). Chromosome 1 contains the value 6 twice and misses the value 3. On the other side chromosome 2 contains the value 3 twice and misses the value 6. We can observe that this crossover is equivalent to the exchange of the values 3→6, 4→5 and 5→4. To repair the two chromosomes we have to
apply this exchange outside the crossing region (figure 1.3.12 b)). At the end figure 1.3.12 c) shows the repaired chromosome.

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

![Figure 1.3.13: Uniform crossover](image)

Figure 1.3.13 shows an example of a uniform crossover with four crossover points. A gene is swapped, if a uniformly created random number, $r \in [0, 1]$, is smaller than the swap probability, $p_S$. The following code snippet shows how these swap indexes are calculated, in a functional way.

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

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

```java
final var alterer = new CombineAlterer<DoubleGene, Double>(
    (g1, g2) -> g1.newInstance(g1.doubleValue() / g2.doubleValue())
);
Mean alterer  The Mean alterer works on genes which implement the Mean interface. All numeric genes implement this interface by calculating the arithmetic mean of two genes. This alterer is a specialization of the CombineAlterer.

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

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

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

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

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

\[13^{13}\] The line crossover, also known as line recombination, was originally described by Heinz Mühlenbein and Dirk Schlierkamp-Voosen. [31]
Partial alterer  Alterers are working on the whole population, which is effectively a sequence of genotypes. If your genotype consists of more than one chromosome, the alterer is applied to all chromosomes. There is no way to bind an alterer to a specific chromosome. The PartialAlterer class overcomes this shortcoming and allows you to define the chromosomes the wrapped Alterer is using.

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

The example above shows how to use the PartialAlterer. The wrapped Mutator will only operate on the chromosome with the index 0 and 3, the wrapped MeanAlterer will alter on the chromosome with index 1 and the LineCrossover will work on all chromosomes. A potential drawback of the PartialAlterer is a possible performance penalty. This is because the chromosomes must be sliced into different population sequences for each PartialAlterer. If this is an issue for the overall performance depends on the concrete application.

1.3.3  Engine classes

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

1.3.3.1  Fitness function

The fitness Function is also an important part when modeling a genetic algorithm. It takes a Genotype as argument and returns a fitness value. The returned fitness value must implement the Comparable interface. This allows the evolution Engine, respectively the selection operators, to select the offspring- and survivor population. Some selectors have stronger requirements for the fitness value than a Comparable, but these constraints are checked by the Java type system at compile time.
The fitness function must be deterministic. Whenever it is applied to the same genotype, it must return the same fitness value. Nondeterministic fitness functions can lead to unexpected behavior, since the calculated fitness value is cached by the phenotype.

The following example shows the simplest possible fitness function. This function just returns the allele of a 1x1 float genotype.

```java
public class Main {
    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 -> gt.gene().allele();
    }
}
```

The first type parameter of the Function defines the kind of Genotype from which the fitness value is calculated and the second type parameter determines the return type, which must at least implement the Comparable interface.

### 1.3.3.2 Engine

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

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

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

Listing 1.7: Engine class

Listing 1.7 shows the main methods of the Engine class. The Engine is used for performing the actual evolution of a given population. One evolution step is executed by calling the Engine.evolve method, which returns an EvolutionResult object. This object contains the evolved population plus
additional information like the killed and as invalid marked individuals. With the `stream()` method you create a new `EvolutionStream`, which is used for controlling the evolution process. For more information about the `EvolutionStream` see section [1.3.3.3].

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

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

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

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

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

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

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

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

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

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

**1.3.1.1 evolutionStream**

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

**optimize**

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

**populationSize**

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

**selector**

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

**survivorsSelector**

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

The 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 longer creates infinite EvolutionStreams, they are truncated by the limit predicate defined by the Engine. This feature is needed for concatenating evolution Engines (see section 3.1.6.1).

```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 PredicateSupplier instead of the Predicate itself. The reason for this is that some Predicates have to maintain internal state to work properly. This means, every time the Engine creates a new stream, it must also create a new limiting Predicate. The Engine::limit function will return an EvolutionStreamable instead of an Engine. This interface lets you create EvolutionStreams, which is what you usually want to do with the Engine.

### 1.3.3.3 Evolution

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

```java
@FunctionalInterface
    EvolutionResult<G, C> evolve(EvolutionStart<G, C> start);
}
```
1.3. BASE CLASSES

CHAPTER 1. FUNDAMENTALS

1.3.3.4 EvolutionStream

The EvolutionStream controls the execution of the evolution process and can be seen as a kind of executionhandle. 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.

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

Listing 1.9: EvolutionStream interface

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

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

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

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

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

---

14It 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/)
This factory method takes a start value, of type `EvolutionStart`, and an evolution `Function`. The evolution `Function` takes the start value and returns an `EvolutionResult` object. To make the runtime behavior more predictable, the start value is fetched/created lazily at the evolution start time.

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

### 1.3.3.5 EvolutionResult

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

```java
  public ISeq<Phenotype<G, C>> population();
  public long generation();
}
```

Listing 1.10: EvolutionResult class

Listing 1.3.3.5 shows the two most important properties, the `population` and the `generation` the result belongs to. These are also the two properties needed for the next evolution step. The `generation` is, of course, incremented by one. To make collecting the `EvolutionResult` object easier, it also implements the `Comparable` interface. Two `EvolutionResults` are compared by its best `Phenotype`, depending on the optimization direction. 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 (minimization or maximization).
- `toBestPhenotype()` This collector can be used if you are only interested in the best `Phenotype`.
- `toBestGenotype()` Use this collector if you only need the best `Genotype` of the `EvolutionStream`.

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

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

CHAPTER 1. FUNDAMENTALS

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

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

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

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

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

1.3.3.6 EvolutionStatistics

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

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

Listing 1.11: EvolutionStatistics usage

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

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

```
+---------------------------------------------------------------------------+
| Time statistics |
+---------------------------------------------------------------------------+
| Selection: sum=0.046538278000 s; mean=0.003878189833 s |
| Altering: sum=0.086155457000 s; mean=0.007179621417 s |
| Fitness calculation: sum=0.022901606000 s; mean=0.001908467167 s |
| Overall execution: sum=0.147298067000 s; mean=0.012274838917 s |
```
A typical output of an `EvolutionStatistics` object will look like the example above.

The `EvolutionStatistics` object is a simple way for inspecting the `EvolutionStream` after it is finished. It doesn’t give you an active 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 evolution engine.
    static final Engine<EnumGene<Point>, Double> ENGINE = Engine
        .builder(TSM::dist, CODEC)
        .optimize(Optimize.MINIMUM)
        .build();

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

    // You will be informed on new results. This allows to
    // react on new best phenotypes e.g. log it.
    private static void update(final EvolutionResult<EnumGene<Point>, Double> result) {
        if (best == null ||
            best.compareTo(result.bestPhenotype()) < 0) {
            best = result.bestPhenotype();
            System.out.print(result.generation() + " : ") ;
            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));
    }
}
```
Listing 1.12: Live evolution statistics

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

1.3.3.7 Evaluator

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

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

Listing 1.13: `Evaluator` interface

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

1. the size of the returned, evaluated, phenotype sequence must be exactly the size of the input phenotype sequence and

2. all phenotypes of the returned population must have a fitness value assigned. That means, the expression `pop.forAll(Phenotype::isEvaluated)` must be true.

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

```
final Function<? super Genotype<G>, ? extends C> fitness = ...;
final Evaluator<G, C> evaluator = population.map(pt -> pt.eval(fitness)).asISeq();
```
To use the fitness `Evaluator`, you have to use the `Engine.Builder` constructor directly, instead of one of the factory methods.

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

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

```java
static Future<Double> fitness(final double x) {
    return ...;
}
public static void main(final String[] args) {
    final Codec<Double, DoubleGene> codec = ...;
    final Evaluator<DoubleGene, Double> evaluator = Evaluators
        .async(Main::fitness, codec);
    final Engine<DoubleGene, Double> engine =
        new Engine.Builder<>(evaluator, codec.encoding())
            .build();
}
```

### 1.4 Nuts and bolts

#### 1.4.1 Concurrency

The Jenetics library parallelizes independent tasks whenever possible. Especially the evaluation of the fitness function is done concurrently. That means that the fitness function must be thread-safe and deterministic. The easiest way for achieving thread safety is to make the fitness function immutable and reentrant.

Since the number of individuals of one population, which determines the number of fitness functions to be evaluated, is usually much higher then the number of available CPU cores, the fitness evaluation is done in batches. This reduces the evaluation overhead, for lightweight fitness functions.

![Figure 1.4.1: Evaluation batch](image)

Figure 1.4.1 shows an example population with 12 individuals. The evaluation of the phenotype’s fitness functions are evaluated in batches with three elements. For this purpose, the individuals of one batch are wrapped into a `Runnable` object. The batch size is automatically adapted to the available number of CPU cores. It is assumed that the evaluation cost of one fitness function is quite small. If this assumption doesn’t hold, you can configure the the maximal number of batch elements with the `io.jenetics.concurrency.maxBatchSize` system property. The usage of this property is described in section 1.4.1.2.
1.4. NUTS AND BOLTS  CHAPTER 1. FUNDAMENTALS

1.4.1.1 Basic configuration

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

```java
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 a 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();
    }
}
```

If no Executor is given, Jenetics uses a common ForkJoinPool 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 subpopulations using the abstract RecursiveAction class. If a minimal subpopulation size is reached, the fitness values for this subpopulation are directly evaluated. The default value of this threshold is five and can be controlled via the io.jenetics.concurrent-.splitThreshold system property. Besides the splitThreshold, the size of

the evaluated subpopulation is dynamically determined by the `ForkJoinTask::getSurplusQueuedTaskCount` method\(^{16}\). If this value is greater than three, the fitness values of the current subpopulation are also evaluated immediately. The default value can be overridden by the `io.jenetics.concurrency.maxSurplusQueuedTaskCount` system property.

```java
```

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

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

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

```java
$ java -Dio.jenetics.concurrency.maxBatchSize=3 -cp jenetics-7.2.0.jar:app.jar com.foo.bar.MyJeneticsApp
```

Another source of underutilized CPUs are lock contentions. It is therefore strongly recommended to avoid locking and blocking calls in your fitness function.

---

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

\(^{17}\)The number of sub-populations actually depends on the number of available CPU cores, which are determined with `Runtime.availableProcessors()`.
at all. If blocking calls are unavoidable, consider using the managed block functionality of the ForkJoinPool.\(^{18}\)

1.4.2 Randomness

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

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

```java
final var rgf = RandomGeneratorFactory.getDefault();
final List<Genotype<DoubleGene>> genotypes =
  RandomRegistry.with(rgf.create(123), r -> {
    Genotype.of(DoubleChromosome.of(0.0, 100.0, 10))
      .instances()
      .limit(100)
      .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
final Engine<DoubleGene, Double> engine = ...;
// Create a new evolution stream with the given initial genotypes.
final 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 nondeterministic.

By default, Jenetics uses the L64X256MixRandom random generator, which is part of the standard OpenJDK platform. If you want to change this default for your whole application, without changing your code, you can do this via the io.jenetics.util.defaultRandomGenerator system property. The code example below shows how to replace the default random generator with L64X-1024MixRandom\(^{19}\).

```
$ java -Dio.jenetics.util.defaultRandomGenerator=
```

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

\(^{19}\)A list of all random generators defined in the OpenJDK platform, can be found here: [https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/random/package-summary.html](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/random/package-summary.html).
Setting the PRNG to a RandomGenerator object with a defined seed has the effect, that every evolution stream will produce the same result—in a single threaded environment.

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

The Fair Play property of a PRNG guarantees that the quality of the genetic algorithm (evolution stream) does not depend on the degree of parallelization.

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

**Random seeding** Every thread uses the same kind of PRNG but with a different seed. This is the default strategy used by the Jenetics library. Random seeding works well for the most problems but without theoretical foundation.\(^{20}\) The RandomRegistry is initialized with the Java L64X256MixRandom class.\(^{21}\)

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

**Block splitting** With this method each thread will be assigned a non-overlapping contiguous block of random numbers, which should be enough for the whole

---

\(^{20}\)This is also expressed by Donald Knuth’s advice: «Random number generators should not be chosen at random.»

\(^{21}\)RandomRegistry.random(RandomGeneratorFactory.of("L64X256MixRandom"))
runtime of the process. If the number of threads is not known in advance, the length of each block should be chosen much larger than the maximal expected number of threads. This strategy is used when using the LCG64ShiftRandom class. This class assigns every thread a block of $2^{56} \approx 7.2 \cdot 10^{16}$ random numbers. After 128 threads, the blocks are recycled, but with changed seed.

![Figure 1.4.2: Block splitting](image)

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

![Figure 1.4.3: Leapfrogging](image)

**LCG64ShiftRandom** The LCG64ShiftRandom class is a port of the trng::lcg64_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.

```java
public class LCG64ShiftRandom
    implements RandomGenerator.ArbitrarilyJumpableGenerator
{
    public void jump(final double step);
    public void jumpPowerOfTwo(final int s);
    ...
}
```

---

22 The LCG64ShiftRandom PRNG is part of the io.jenetics.prng module (see section 3.4 on page 120).

http://numbercrunch.de/trng/
1.4. NUTS AND BOLTS CHAPTER 1. FUNDAMENTALS

Listing 1.14: LCG64ShiftRandom class

Listing 1.14 shows the interface used for implementing the block splitting and leapfrog parallelization techniques. This methods have the following meaning:

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

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

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

1.4.3 Serialization

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

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

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

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

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

// Reading the population from disk.
```
1.4. NUTS AND BOLTS

1.4.4 Utility classes

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

*io.jenetics.util.BaseSeq*  This interface defines a minimal contract for sequential data, which can be accessed by its index or position. The algorithms, implemented by the *Jenetics* library, assumes that accessing elements of a *BaseSeq* is done in $O(1)$.

*Chromosome* and *Genotype* implement the *BaseSeq* interface. This expresses the intent that the *Chromosome* is a sequence of *Genes* and the *Genotype* is a sequence of *Chromosomes*.

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

![Seq class diagram](image-url)

**Figure 1.4.4: Seq class diagram**

**Figure [1.4.4]** shows the *Seq* class diagram with their most important methods. The interfaces *MSeq* and *ISeq* are mutable, respectively immutable specialization-
tions of the basis interface. Creating instances of the \texttt{Seq} interfaces is possible via the static factory methods of the interfaces.

\begin{verbatim}
// 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);
\end{verbatim}

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

\texttt{io.jenetics.util.ProxySorter} This is a special sorter, which allows you to sort even an immutable collection. As the name suggests, it doesn’t sort a given sequence directly. Instead it sorts or rearranges a \texttt{proxy int[]} array, which can then be used for accessing the original sequence in a sorted order. The main usage for this special sorter in \texttt{Jenetics} is where you need to access of the population in sorted order, but have to preserve the original order of the population.\footnote{For this specific problem you could also do this by copying the population and sorting the copy instead of the original. But using a sorted \texttt{proxy} array can lead to simpler code.}

Many of the algorithms, implemented in the \texttt{io.jenetics.ext.moea} package, uses the \texttt{ProxySorter}, which leads to simpler code at this places. How the proxy sorter is used can be seen in the following code snippet.

\begin{verbatim}
final double[] array = RandomGenerator.getDefault().
  .doubles(100)
  .toArray();
final int[] proxy = ProxySorter.sort(array);

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

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

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

\begin{verbatim}
final IntFunction<String> access = ...;
final int length = 100;
final int[] proxy = ProxySorter.sort(
  access, length,
  (a, i, j) -> a.apply(i).compareTo(a.apply(j)));
\end{verbatim}

The code snippet above shows how to sort an \texttt{IntFunction}. With the proxy array you are now able to access the \texttt{access} function in ascending order. The \texttt{ProxySorter} uses the Timsort algorithm for sorting the \texttt{proxy int[]} array.\footnote{https://en.wikipedia.org/wiki/Timsort}
io.jenetics.stat This package contains classes for calculating statistical moments. They are designed to work smoothly with the Java Stream API and are divided into mutable (number) consumers and immutable value classes, which holds the statistical moments. The additional classes calculate the

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

<table>
<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 a given DoubleGene stream.

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

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

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

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

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

---

\[26\]https://en.wikipedia.org/wiki/Quantile
Chapter 2

Advanced topics

This section describes some advanced topics for setting up an evolution Engine or EvolutionStream. It contains some problem encoding examples and how to override the default validation strategy of the given 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 a GA. It also comes with different implementations for the base data-types (genes and chromosomes) and operators (alterers and selectors). If it is still some functionality missing, this section describes how you can extend the existing classes. Most of the extensible classes are defined by an interface and have an abstract implementation which makes it easier to extend it.

2.1.1 Genes

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 Gene::allele method which will return the wrapped data, (2) the Gene::newInstance method for creating new, random instances of the gene—must be of the same type and have the same constraint—and (3) the Gene::isValid method which checks if the gene fulfill the expected constraints. The gene constraint might be violated after mutation and/or recombination. If you want to implement a new number-gene, e. g. a gene which holds complex values, you may want to extend it from the NumericGene interface.
The custom Genes and Chromosomes implementations must use the RandomGenerator available via the RandomRegistry::random method when implementing their factory methods. Otherwise it is not possible to seamlessly change the RandomGenerator by using the RandomRegistry::random(RandomGenerator) method.

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

```java
class LastMonday {
    // Creates new random 'LocalDate' objects.
    private static LocalDate nextMonday() {
        final var random = RandomRegistry.random();
        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 RandomGenerator, used for creating the new LocalDate objects, must be requested from the RandomRegistry. With the optional validation function, isValid, it is possible to reject Genes whose alleles don’t conform to some criteria. The simple usage of the AnyGene has also its downsides. Since the AnyGene instances are created from function objects, serialization is not supported by the AnyGene class. It is also not possible to use some Alterer implementations with the AnyGene, like:

- GaussianMutator,
- MeanAlterer and
- PartiallyMatchedCrossover

### 2.1.2 Chromosomes

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

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

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

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

Listing 2.2: AnyChromosome example

```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.gene().allele()
    );

    // Creates new random 'LocalDate' objects.
    private static LocalDate nextMonday() {
        final var random = RandomRegistry.random();
        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. The example tries to find a Monday with a maximal day of month. An interesting detail is, that an Codec definition is used for creating new Genotypes and

---

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

2See section 2.3 on page 54 for a more detailed Codec description.
for converting them back to \texttt{LocalDate} alleles. The convenient usage of the \texttt{AnyChromosome} has to be payed by the same restriction as for the \texttt{AnyGene}: no serialization support for the chromosome and not usable for all \texttt{Alterer} implementations.

### 2.1.3 Selectors

If you want to implement your own selection strategy you only have to implement the \texttt{Selector} interface with the \texttt{select} method.

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

Listing 2.3: Selector interface

The first parameter is the original \texttt{population} from which the sub-population is selected. The second parameter, \texttt{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, \texttt{opt}, determines the optimization strategy which must be used by the selector. This is exactly the point where it is decided whether the GA minimizes or maximizes the fitness function.

Before implementing a selector from scratch, consider extending your selector from the \texttt{ProbabilitySelector} (or any other available \texttt{Selector} implementation). It is worth the effort to try to express your selection strategy in terms of selection property \( P(i) \). Another way for reusing existing \texttt{Selector} implementation is by composition.

```java
    private final TruncationSelector<G, C> _elite = new TruncationSelector<>();
    private final TournamentSelector<G, C> _rest = new TournamentSelector<>(3);

    public EliteSelector() {
    }

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

Listing 2.4: EliteSelector class
2.1. EXTENDING JENETICS

CHAPTER 2. ADVANCED TOPICS

if (population.isEmpty() || count <= 0) {
    result = ISeq.empty();
} else {
    final int ec = min(count, _eliteCount);
    result = _elite.select(population, ec, opt);
    result = result.append(
        _rest.select(population, max(0, count - ec), opt)
    );
}
return result;

Listing 2.4: Elite selector

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

2.1.4 Alterers

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

@FunctionalInterface
class Alterer<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
{
    AltererResult<G, C> alter(
        Seq<Phenotype<G, C>> population,
        long generation
    );
}

Listing 2.5: Alterer interface

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

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

2.1.5 Statistics

During the developing phase of an application which uses the Jenetics library, additional statistical data about the evolution process is crucial. Such data can help to optimize the parametrization of the evolution Engine. A good starting point is to use the EvolutionStatistics class in the io.jenetics.engine
package (see listing1.11). If the data in the EvolutionStatistics class doesn’t fit your needs, you simply have to write your own statistics class. It is not possible to derive from the existing EvolutionStatistics class. This is not a real restriction, since you still can use the class by delegation. Just implement the Java Consumer<EvolutionResult<G, C>> interface.

2.1.6 Engine

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

Listing 2.6: Special evolution engine

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

3Also refer to section 1.3.3.4 on page 26 on how to create an EvolutionStream from an evolution Function.
2.2. ENCODING  

CHAPTER 2. ADVANCED TOPICS

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

### 2.2 Encoding

This section presents some encoding examples for common optimization problems. The encoding should be a complete, and minimal representation of the problem domain. An encoding is complete if it contains enough information to represent every solution to the problem. Whereas a minimal encoding contains only the information needed to represent a solution to the problem. If an encoding contains more information than is needed to uniquely identify solutions to the problem, the search space will be larger than necessary. In the best case, there is a one-to-one mapping from the Genotype space to problem domain. Whenever possible, the encoding should not represent infeasible solutions.

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

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

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

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

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

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

Listing 2.7: Adjustable evolution stream

The purpose of such an adjustment is to broaden the search if the population variance tends to be too small. This can reduce the risk of converging to a local minimum. If the population variance is too big, a different engine configuration can help to speed up the optimization.
represents an infeasible solution, care must be taken in the fitness function to give partial credit to the Genotype for its "good" genetic material while sufficiently penalizing it for being infeasible. Implementing a specialized Chromosome, which won’t create invalid encodings can be a solution to this problem. In general, it is much more desirable to design a representation that can only represent valid solutions so that the fitness function measures only fitness, not validity. An encoding that includes invalid individuals enlarges the search space and makes the search more costly. A deeper analysis of how to create encodings can be found in [36] and [35]. Some of the encodings represented in the following sections have been implemented by Jenetics, using the Codec 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, $f : \mathbb{R} \rightarrow \mathbb{R}$:

- 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 gene method, and convert it to a double.

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

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

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

---

4See section 2.3 on page 54
5Scalar chromosomes contains only one gene.
2.2. ENCODING

2.2.2 Scalar function

Optimizing a function, \( f(x_1, \ldots, x_n) \), 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.

```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 \( (x_1, \ldots, x_n) \) as double[] array.

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

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

The second encoding must be used if the minimum and maximum value of the variables, \( x_i \), can’t be the same for all. 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),
    DoubleChromosome.of(min2, max2),
    ...
    DoubleChromosome.of(minn, maxn)
);
```

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

```java
static double[] toScalars(final Genotype<DoubleGene> gt) {
    return gt.stream()
        .mapToDouble(c -> c.gene().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) -.get(j) method and than convert it to the desired numeric value, by calling intValue(), longValue() or doubleValue().

2.2.3 Vector function

A function, \( f(X_1, \ldots, X_n) \), of one or more variables whose range is \( m \)-dimensional, is encoded by \( m \) DoubleChromosomes of lengthn. The domain–minimum and maximum values of one variable \( X_i \) are the same in this encoding.
2.2. ENCODING

Chapter 2. Advanced Topics

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

```java
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.

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

2.2.4 Affine transformation

An affine transformation is usually performed by a matrix multiplication with a transformation matrix—in a homogeneous coordinates system. For a transformation in \( \mathbb{R}^2 \), we can define the matrix \( A \):

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

(2.2.1)

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

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

The drawback with this kind of encoding is, that we will create a lot of invalid (non-affine transformation matrices) during the evolution process, which must be detected and discarded. It is also difficult to find the right parameters for the `min` and `max` values of the `DoubleChromosomes`.

A better approach will be to encode the transformation parameters instead of the transformation matrix. The affine transformation can be expressed by the following parameters:

- \( s_x \) – the scale factor in \( x \) direction
- \( s_y \) – the scale factor in \( y \) direction
- \( t_x \) – the translation in \( x \) direction
- \( t_y \) – the translation in \( y \) direction

Add references:

- https://en.wikipedia.org/wiki/Affine_transformation
2.2. ENCODING

- $s_y$ – 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.

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

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

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

(2.2.2)

This corresponds to a 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:

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

    return new double[][] {
        {1, 0, tx},
        {0, 1, ty},
        {0, 0, 1}
    };
}
```
2.2. ENCODING

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)

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 `BitChromosomes` of

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

This property violates the *minimal* encoding requirement we mentioned at the beginning of section 2.2 on page 47. For simplicity reason this will be ignored for the undirected graph encoding.

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. ENCODING

Whether the node\( i \) is connected to node\( j \) can be easily checked by calling \( \text{gt.get(i-1).get(j-1).booleanValue()} \). 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()
        .mapToInt(gene -> gene.bit() ? 1 : 0)
        .toArray())
    .toArray(new int[][]::new);
```

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

**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.
2.3. CODEC

CHAPTER 2. ADVANCED TOPICS

```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);
```

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

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

2.3 Codec

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

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

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

Listing 2.8: Codec interface

Listing 2.8 shows the Codec interface. The `encoding` method returns the `Genotype` factory, which is used by the `Engine` for creating new `Genotypes`. The decoder `Function`, which is returned by the `decoder` method, transforms the `Genotype` to the argument type of the actual 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`.

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

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

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

11Section 2.2 on page 47 describes some possible encodings for common optimization problems.
When using a Codec instance, the fitness function solely contains code from your actual problem domain—no dependencies to classes of the Jenetics library.

```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 in the `io.jenetics.engine.Codecs` class. The following subsections describe the most important predefined Codecs.

### 2.3.1 Scalar codec

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

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

Listing 2.9: Codec factory method: ofScalar

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

```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();
        ...
    }
}
```

### 2.3.2 Vector codec

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

```java
static Codec<int[], IntegerGene>
ofVector(IntRange domain, int length) {
    return Codec.of(
        Genotype.of(IntegerChromosome.of(domain, length)),
        gt -> gt.chromosome()
            .as(IntegerChromosome.class)
            .toArray()
    );
}
```
The usage example of the vector Codec is almost the same as for the scalar Codec. As an additional parameter, we need to define the length of the desired array and we define our fitness function with an int[] array.

```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 Matrix codec

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

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

### 2.3.4 Subset codec

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

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

CHAPTER 2. ADVANCED TOPICS

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

Listing 2.12: Codec factory method: ofSubSet

The following usage example ofSubSet Codec shows a simplified version of the Knapsack problem (see section 5.4). We try to find a subset, from the given basic SET, where the sum of the values is as big as possible, but smaller or equal than 20.

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

    // Fitness function directly takes an 'int' value.
    static int fitness(ISeq<Integer> subset) {
        assert (subset.size() <= SET.size());
        final int size = subset.stream().collect(Collectors.summingInt(Integer::intValue));
        return size <= 20 ? size : 0;
    }

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

Fixed sized subsets  The second kind of subsetCodec 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
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() {
```
2.3. CODEC

2.3.5 Permutation codec

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

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

Listing 2.13: Codec factory method: `ofPermutation`

Listing 2.13 shows the implementation of a permutation Codec, where the order of the given alleles influences the value of the fitness function. An alternate formulation of the traveling salesman problem is shown in the following listing. It uses the permutation Codec in listing 2.13 and uses `io.jenetics.jpx.WayPoints`, from the JFX 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(Geoid.DEFAULT.toTourLength())
            .to(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());
    }
}
```

1https://github.com/jenetics/jpx
2.3.6 Mapping codec

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

```java
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 |source| > |target|, the returned mapping function is surjective, if |source| < |target|, the mapping is injective and if |source| = |target|, the created mapping is bijective. In every case the size of the encoded Map is |target|. Figure 2.3.1 shows the described different mapping types in graphical form.

![Mapping codec types](image)

**Figure 2.3.1: Mapping codec types**

With |source| = |target|, you will create a Codec for the assignment problem. The problem is defined by a number of workers and a number of jobs. Every worker can be assigned to perform any job. The cost for a worker may vary depending on the worker job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each worker in such a way which optimizes the total assignment costs. 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>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>
</tr>
<tr>
<td>Worker 2</td>
<td>1</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>Worker 3</td>
<td>6</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Worker 4</td>
<td>1</td>
<td>3</td>
<td>5</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 can find an optimal solution in $O(n^3)$ time. You should consider this deterministic algorithm before using a GA.

### 2.3.7 Composite codec

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

```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 2.14: Composite Codec factory method

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

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

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

```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),
            https://en.wikipedia.org/wiki/Hungarian_algorithm
        );
    }
}
```

---

16https://en.wikipedia.org/wiki/Hungarian_algorithm
2.3. CODEC  

 CHAPTER 2. ADVANCED TOPICS

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

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

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

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

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

### 2.3.8 Invertible codec

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

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

Listing 2.15: InvertibleCodec interface

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

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

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

Listing 2.16: Problem interface

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

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

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

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

2.5 Constraint

Constraints delimit the feasible space of solutions of an optimization problem and are considered in evolutionary algorithms [13, 27, 12, 28]. This influence the desirability of each possible solution. If the constraints are satisfied, the solution is accepted and it is called a feasible solution; otherwise the solution is removed or modified. For a fitness function, \( f(x) \), the constraints are usually given as a
list of inequalities,
\[ g_i(x) \leq 0, \quad (2.5.1) \]
and a list of equations,
\[ h_j(x) = 0. \quad (2.5.2) \]

Figure 2.5.1: Constrained 2-dimensional search space

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

Usually, a given problem should be encoded in a way, that it is not possible for the evolution Engine to create invalid individuals (Genotypes). Some possible encodings for common data structures are described in section 2.2. The Engine creates new individuals in the altering step, by rearranging (or creating new) Genes within a Chromosome. Since a Genotype is treated as valid if every single Gene in every Chromosome is valid, the validity property of the 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 ones. Although this behavior will work for most Genotypes, it is still possible that invalid individuals will be created during the evolution. If you need a more advanced validation strategy, the Constraint interface comes into play.
2.5. CONSTRAINT CHAPTER 2. ADVANCED TOPICS

Listing 2.17: Constraint interface

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

Listing 2.17 shows the definition of the Constraint interface. The test method of the interface checks the validity of the given Phenotype and the repair method creates a new individual using the invalid individual as template.

The RetryConstraint class is the default implementation of the Constraint interface. It implements the repair method by creating new Phenotypes until the created individual is valid. Although this approach seems a little bit simplistic, it has an important and desirable property: the repaired individuals follow the same distribution than the original. This means, that no part of the problem domain is left out or isovercrowed. The number of necessary retries is also not a problem, for normal constraints. For example, the probability that a randomly created point lies outside the unit circle is $1 - \frac{\pi}{4} \approx 0.2146$. This leads to a failure probability after 10 retries, which is the default value of the RetryConstraint, of $(1 - \frac{\pi}{4})^{10} \approx 0.00000207$. You can parameterize a different Constraint definition with the constraint method of the Engine.Builder.

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

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

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

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

\[
\begin{align*}
    x_1 + x_2 - 1 & \leq 0 \\
    x_2 \cdot x_3 - 0.5 & \leq 0.
\end{align*}
\]

A repairing Constraint implementation checks the validity of a Phenotype and
The implementation of the new depends on your actual encoding and might look like this

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

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

```java
Phenotype<DoubleGene, Double> newPhenotype(double[] r, long gen) {
```
2.5. CONSTRAINT CHAPTER 2. ADVANCED TOPICS

Figure 2.5.3: Only valid domain points

2
3
4
5
6
7
8
9
10
11

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

Figure 2.5.4: Split range domain

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

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

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

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

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

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

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

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

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

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

2.6 Termination
to the \texttt{limit} function, returns false, the \textit{EvolutionStream} is truncated. The \texttt{EvolutionStream.limit(r \to true)} will create an infinite evolution stream.

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

All termination strategies described in the following sections are part of the library and can be created by factory methods of the \texttt{io.jenetics.engine-.Limits} class. The termination strategies were tested by solving the Knapsack problem\footnote{The actual implementation used for the termination tests can be found in the Github repository: \url{https://github.com/jenetics/jenetics/blob/master/jenetics.example/src/main/java/io/jenetics/example/Knapsack.java}} (see section 5.4) with 250 items. This makes it a real problem with a search space size of $2^{250} \approx 10^{75}$ elements.

| Population size: | 150 |
| Survivors selector: | \texttt{TournamentSelector<>(5)} |
| Offspring selector: | \texttt{RouletteWheelSelector<>()} |
| Alterers: | \texttt{Mutator<>(0.03)} and \texttt{SinglePointCrossover<>(0.125)} |

Table 2.6.1: Knapsack evolution parameters

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

Some of the implemented termination strategies need to maintain an internal state. These strategies can’t be reused in different evolution streams. To be on the safe side, it is recommended to always create a \texttt{Predicate} instance for each stream. Calling \texttt{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 \textit{EvolutionStream}. It just uses the existing \texttt{limit} method of the Java \texttt{Stream} interface.

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

This kind of termination method can always be applied—usually additional with other evolution terminators—, to guarantee the truncation of the \textit{EvolutionStream} and to define an upper limit of the executed generations. Additionally, the \texttt{Limits::byFixedGeneration(long)} predicate can be used instead of the \texttt{Stream::limit(long)} method. This predicate is mainly there for the completion reason and behaves exactly as the \texttt{Stream::limit(long)} function, except
2.6. TERMINATION

for the number of evaluations performed by the resulting stream. The evaluation of the population is \( m \times \text{max generations} + 1 \). This is because the limiting predicate works on the \texttt{EvolutionResult} object, which guarantees to contain an \textit{evaluated} population. That means, that the population must be evaluated at least once, even for a generation limit of zero. If this is an unacceptable performance penalty, better use the \texttt{Stream::limit(long)} function instead.

![Fitness vs. Generation Graph](image)

**Figure 2.6.1: Fixed generation termination**

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

2.6.2 Steady fitness

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

```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;
}
```

69
public SteadyFitnessLimit(final int generations) {
    _generations = generations;
}

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

Listing 2.18: Steady fitness

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

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

### 2.6.3 Evolution time

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

```java
final Engine<DoubleGene, Double> engine = ...;
final EvolutionStream<DoubleGene, Double> stream = engine
    .stream()
    .limit(Limits.byExecutionTime(Duration.ofMillis(500)));
```
In the code example above, the `byExecutionTime(Duration)` method is used for creating the termination object. Another method, `byExecutionTime(Duration, InstantSource)`, lets you define the `java.time.InstantSource`, which is used for measure the execution time. Jenetics uses the nano precision clock `io.jenetics.util.NanoClock` for measuring the time. To have the possibility to define a different `InstantSource` implementation is especially useful for testing purposes.

Figure 2.6.3 shows the evaluated generations depending on the execution time. Except for very small execution times, the evaluated generations per time unit stays quite stable.\footnote{While running the tests, all other CPU intensive process has been stopped. The measuring started after a warm-up phase.} That means that a doubling of the execution time will double the number of evolved generations.

### 2.6.4 Fitness threshold

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

```java
final Engine<DoubleGene, Double> engine = ...
final EvolutionStream<DoubleGene, Double> stream = engine.stream().limit(Limits.byFitnessThreshold(10.5)).limit(5000);
```
When limiting the `EvolutionStream` by a fitness threshold, you have to have knowledge about the expected maximal fitness. This can be the case if you are minimizing an error function with a known optimal value of zero. If there is no such knowledge, it is advisable to add an additional fixed sized generation limit as a safety net.

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

### 2.6.5 Fitness convergence

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

```java
public static <N extends Number & Comparable<? super N>>
Predicate<EvolutionResult<?, N>> byFitnessConvergence(
    final int shortFilterSize,
    final int longFilterSize,
)
```
2.6. TERMINATION  

CHAPTER 2. ADVANCED TOPICS

Figure 2.6.4: Fitness threshold termination

```
5     final BiPredicate<DoubleMoments, DoubleMoments> proceed
6 ) ;
```

Listing 2.19: General fitness convergence

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

```
1 public static <N extends Number & Comparable<? super N>>
2     Predicate<EvolutionResult <?, N>> byFitnessConvergence(
3         final int shortFilterSize ,
4         final int longFilterSize ,
5         final double epsilon
6 ) ;
```

Listing 2.20: Mean fitness convergence

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

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

(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

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

(2.6.2)

\[19\] https://en.wikipedia.org/wiki/Moving_average
is fulfilled, the\texttt{EvolutionStream} is truncated. Where $\delta$ is defined as follows:

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

```java
final Engine<DoubleGene, Double> engine = ...

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

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

![Figure 2.6.5: Fitness convergence termination: $N_S = 10, N_L = 30$](image)

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

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

### 2.6.6 Population convergence

This termination method stops the evolution when the population is deemed as converged. A population is deemed as converged when the average fitness across the current population is less than a user specified percentage away from the best fitness of the current population. The population is deemed as converged and the \texttt{EvolutionStream} is truncated if
2.6. TERMINATION

2.6.7 Gene convergence

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

### 2.7 Reproducibility

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

```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 2.21: Reproducible evolution Engine

Listing 2.21 shows the basic setup of such a reproducible evolution Engine. Firstly, you have to make sure that all evolution steps are executed serially. This is done by configuring a single threaded executor. In the simplest case the evolution is performed solely on the main thread—Runnable::run. If the evolution Engine uses more than one worker thread, the reproducibility is no longer guaranteed.

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

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

### 2.8 Evolution performance

This section contains an empirical proof, that evolutionary selectors deliver significantly better fitness results than a random search. The MonteCarloSelector is used for creating the comparison (random search) fitness values.
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

---

20The termination tests are using a TournamentSelector, with tournament-size 5, for selecting the survivors, and a RouletteWheelSelector for selecting the offspring.
21See 1.3.2.1 on page 13
22See 1.3.2.2 on page 16
the $\mu$ fittest individuals gets to produce $\lambda$ children through an ordinary mutation. The newly created children just replaces the discarded parents.\[26\]

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 $\lambda = 20$, then we have a(5, 20)-ES.

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

Listing 2.22: $(\mu, \lambda)$ Engine configuration

Listing 2.22 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: \[26\]

- **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 push the algorithm towards exploitative search, because only the best individuals are used for reproduction.\[23\]
- **Mutation probability** $p$: A high mutation probability pushes the algorithm toward a fairly random search, regardless of the selectivity of $\mu$.

### 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 compared 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. \[20\] Jenetics works with a constant population size and it is therefore not possible to implement an increasing population size. Besides this restriction, the `Engine` configuration for the $(\mu + \lambda)$-ES is shown in listing 2.23.

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

\[23\] As you can see in listing 2.22, 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.
2.10. EVOLUTION INTERCEPTION CHAPTER 2. ADVANCED TOPICS

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

2.10 Evolution interception

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

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

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

```java
final Engine<DoubleGene, Double> engine = Engine.builder(problem)
    .interceptor(EvolutionInterceptor.ofAfter(this::mapping))
    .build();
```

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

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

```java
final Engine<DoubleGene, Double> engine = Engine.builder(problem)
    .interceptor(EvolutionResult.toUniquePopulation())
    .build();
```

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

Modules

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.jenetics.base</td>
<td>io.jenetics:jenetics:7.2.0</td>
</tr>
<tr>
<td>io.jenetics.ext</td>
<td>io.jenetics:jenetics.ext:7.2.0</td>
</tr>
<tr>
<td>io.jenetics.prog</td>
<td>io.jenetics:jenetics.prog:7.2.0</td>
</tr>
<tr>
<td>io.jenetics.xml</td>
<td>io.jenetics:jenetics.xml:7.2.0</td>
</tr>
<tr>
<td>io.jenetics.prngine</td>
<td>io.jenetics:prngine:2.0.0</td>
</tr>
</tbody>
</table>

Table 3.0.1: Jenetics modules

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

Figure 3.0.1: Module graph

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

The io.jenetics.ext module implements additional nonstandard genes and evolutionary operations. It also contains data structures which are used by these additional genes and operations.

3.1.1 Data structures

3.1.1.1 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>> {
    V value();
    Optional<T> parent();
    T childAt(int index);
    int childCount();
}
```

Listing 3.1: Tree interface

Listing [3.1] shows the Tree interface with its basic abstract tree methods. All other needed tree methods, e.g. for node traversal and search, are implemented by default methods, which are derived from these four abstract tree methods. An mutable default implementation of the Tree interface is given by the TreeNode class.

![Example tree](image.png)

Figure 3.1.1: Example tree

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

```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)))
  .attach(8)
```

82
3.1. IO.JENETICS.EXT

CHAPTER 3. MODULES

3.1.1.2 Parentheses tree

A parentheses tree\textsuperscript{2} is a serialized representation of a tree and is a simplified form of the Newick tree format\textsuperscript{3}. The parentheses tree representation of the tree in figure 3.1.1 will look like the following string:

\[
0(1(4,5),2(6),3(7(10,11),8,9))
\]

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

The NodeValue in the diagram is the string representation of the Tree::value object.

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:

\[
(\text{root})(\text{"}\ (\text{,})\text{,})\text{)
\]

\textsuperscript{2}\url{https://www.i-programmer.info/programming/theory/3458-parentheses-are-trees.html}

\textsuperscript{3}\url{http://evolution.genetics.washington.edu/phylip/newicktree.html}
3.1. IO.JENETICS.EXT

CHAPTER 3. MODULES

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.

3.1.1.3 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 3.2). 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> {
    int childOffset();
    default ISeq<T> flattenedNodes() {...};
}
```

Listing 3.3: FlatTree interface

Listing 3.3 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 3.1.3: Example FlatTree](https://en.wikipedia.org/wiki/Binary_tree#Arrays)

There exists mapping schemes for perfect binary trees, which allows a bijective mapping from tree to array without additional storage need [https://en.wikipedia.org/wiki/Binary_tree#Arrays](https://en.wikipedia.org/wiki/Binary_tree#Arrays). For general trees with arbitrary child count, such simple mapping doesn’t exist. [https://en.wikipedia.org/wiki/Breadth-first_search](https://en.wikipedia.org/wiki/Breadth-first_search)
Figure 3.1.3 illustrates the flattened example tree shown in figure 3.1.1. The curved arrows denote the child offset of a given parent node and the curly braces denote 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);
```

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

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

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

### 3.1.1.4 Tree formatting

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

- **TreeFormatter.PARENTHESES** Converts a tree to its default parentheses format. This is the default format and is also used by the `Tree::toString` method.
- **TreeFormatter.TREE** Creates a verbose tree string, which spans multiple lines, e. g.

```
    div
      cos
        1.3
      cos
        3.14
```

- **TreeFormatter.DOT** Creates a tree string in the dot format, which can be used to create nice graphs with Graphviz.

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

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

[1] https://www.graphviz.org/
3.1.1.5 Tree reduction

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

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

Listing 3.4: `Tree::reduce` method

Listing 3.4 shows the signature of the `reduce` method. The `neutral` array is used in the `reducer` function for leaf elements. After the reduction process, one element is returned, which might be `null` for empty trees. The following code snippet shows how to use the `Tree::reduce` method for evaluating a simple arithmetic expression tree.

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

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

3.1.2 Rewriting

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

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

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

Definition. (Tree rewrite system): A tree rewrite system is a set, \( \mathcal{R} \), of rewrite rules, \( l \rightarrow r \).
In contrast to string rewriting systems, whose objects are flat sequences of symbols, the objects a term rewriting system works on, i.e. the terms, form a term algebra. A term can be visualized as a tree of symbols, the set of admitted symbols being fixed by a given signature.

### 3.1.2.1 Tree pattern

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

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

Listing 3.5: `TreePattern` class

Listing 3.5 shows the constructor and main methods of the `TreePattern`. The `filters` method is used when used for the left-hand side and the `expand` method for the right-hand side. How to create a simple tree pattern is shown in the code snippet below.

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

You can see that the variable `x` will match for arbitrary sub trees. For more complicated patterns it is quite cumbersome to create it via a Decl tree. Usually you will create a `TreePattern` object by compiling a proper pattern string. For creating the same pattern as in the example above you can write `TreePattern.compile("add($x,1)")`. The base syntax for the tree pattern follows the parentheses tree DSL described in 3.1.1.2. It only differs in the declaration of tree variables, which start with a `$` and must be a valid Java identifier. If you want to match non-string trees you must specify an additional mapper function with the `compile` method.

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

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

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

[https://en.wikipedia.org/wiki/Algebraic_data_type](https://en.wikipedia.org/wiki/Algebraic_data_type)
3.1.2.2 Tree rewriter

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

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

Listing 3.6: TreeRewriter interface

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

3.1.2.3 Tree rewrite rule

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

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

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

3.1.2.4 Tree rewrite system (TRS)

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

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

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

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

Since the given tree rewrite system is terminating, we can safely apply the `TRS` to `add(S(0),S(mul(S(0),S(S(0))))), which will then be rewritten to `S(S(S(S(0))))`. 88
### 3.1.2.5 Constant expression rewriter

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

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

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

### 3.1.3 Genes

#### 3.1.3.1 BigInteger gene

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

#### 3.1.3.2 Tree gene

The `TreeGene` interface extends the `FlatTree` interface and serves as basis for the `ProgramGene`, used for genetic programming. Its tree nodes are stored in the corresponding `TreeChromosome`. How the tree hierarchy is flattened and mapped to an array is described in section 3.1.1.3.

### 3.1.4 Operators

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

**Single-node crossover** The `SingleNodeCrossover` class works on `TreeChromosomes`. It swaps two, randomly chosen, nodes from two tree chromosomes. Figure [3.1.4] shows how the single-node crossover works. In this example node 3 of the first tree is swapped with node 9 of the second tree.

**Reverse sequence mutator (RSM)** The `RSMutator` chooses two positions \(i\) and \(j\) randomly. The gene order in a chromosome will then be reversed between these two points. This mutation operator can also be used for combinatorial problems, where no duplicated genes within a chromosome are allowed, e.g. for the TSP.[1]
Hybridizing PSM and RSM (HPRM) The `HPRMutator` constructs an offspring from a pair of parents by hybridizing two mutation operators, PSM (SwapMutator) and RSM. Its main application is for combinatorial problems, like the TSP.[2]

3.1.5 Weasel program

The Weasel program[4] is a thought experiment from Richard Dawkins, in which he tries to illustrate the function of genetic mutation and selection[4] 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” 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?[4]

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

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

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

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

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

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

    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();
    }
}
Listing 3.7: Weasel program

Listing 3.7 shows how to implement the `WeaselProgram` with `Jenetics`. Step (1) and (2) of the algorithm is done implicitly when the initial population is created. The third step is done by the `WeaselMutator`, with mutation probability of 0.05. Step (4) is done by the `WeaselSelector` together with the configured offspring fraction of one. The `EvolutionStream` is limited by the `Limits.byFitnessThreshold`, which is set to `score_max - 1`. In the current example this value is set to `TARGET.length() - 1 = 27`.

The (shortened) output of the Weasel program (listing 3.7) shows, that the optimal solution is reached in generation 46.

### 3.1.6 Modifying Engine

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

```java
public interface EvolutionStreamable<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    EvolutionStream<G, C>
        stream(Supplier<EvolutionStart<G, C>> start);
```
EvolutionStream\langle G, C \rangle \text{ stream}(\text{EvolutionInit}\langle G \rangle \text{ init});

EvolutionStreamable\langle G, C \rangle \text{ limit}(\text{Supplier}\langle \text{Predicate}\langle ? \text{ super EvolutionResult}\langle G, C \rangle \rangle \rangle p))

Listing 3.8: EvolutionStreamable interface

Listing 3.8 shows the main methods of the EvolutionStreamable interface. The existing stream methods take an initial value, which allows to concatenate different engines. With the limit method it is possible to limit the size of the created EvolutionStream instances. The io.jenetics.ext module contains additional classes which allows for concatenating evolution Engines with different configurations, which will then create one varying EvolutionStream. This additional Engine classes are:

1. ConcatEngine
2. CyclicEngine.

3.1.6.1 ConcatEngine

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

Figure 3.1.5: Engine concatenation

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

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

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

```java
final Engine\langle DoubleGene, Double \rangle engine1 = ...;
final Engine\langle DoubleGene, Double \rangle engine2 = ...;
```
3.1. IO.JENETICS.EXT  

CHAPTER 3. MODULES

3.1.6.2 CyclicEngine

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

![Figure 3.1.6: CyclicEngine](image)

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.

3.1.7 Multi-objective optimization

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

a vector of decision variables which satisfies constraints and optimizes

a vector function whose elements represent the objective functions.
These functions form a mathematical description of performance criteria which are usually in conflict with each other. Hence, the term “optimize” means finding such a solution which would give the values of all the objective functions acceptable to the decision maker.\[34\]

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

### 3.1.7.1 Pareto efficiency

Pareto efficiency is named after the Italian economist and political scientist Vilfredo Pareto\[19\] 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 exists, 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\[10\]):** A solution \(x\), is said to be Pareto optimal iff 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\[10\]):** A vector \(u = (u_1,...,u_k)\) is said to dominate another vector \(v = (v_1,...,v_k)\) (denoted \(u \sqsupset v\)) if \(u_i \geq v_i\), i.e., \(\forall i \in \{1,...,k\}: u_i \geq v_i \& \exists i \in \{1,...,k\} : u_i > v_i\).

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

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

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

### 3.1.7.2 Implementing classes

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

\[10\]https://en.wikipedia.org/wiki/Vilfredo_Pareto
Figure 3.1.7: Circle points

Figure 3.1.8: Maximizing Pareto front

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

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

Listing 3.9: Vec interface
Listing 3.9 shows the necessary methods of the `Vec` interface. These methods are sufficient to do all the optimization calculations. The `data()` method returns the underlying vector type, like `double[]` or `int[]`. With the `ElementComparator`, which is returned by the `comparator()` method, it is possible to compare single elements of the vector type `T`. This is similar to the `ElementDistance` function, returned by the `distance()` method, which calculates the distance of two vector elements. The last method, `dominance()`, returns the Pareto dominance comparator, `≻`. Since it is quite a bothersome to implement all these needed methods, the `Vec` interface comes with a set of factory methods, which allows for creating `Vec` instance for some primitive array types.

```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.

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

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

```java
final ISeq<Phenotype<DoubleGene, Vec<double[]>>> paretoSet =
    engine.stream();
```
Since there exists a potential infinite number of Pareto optimal solutions, you have to define desired number of set elements. This is done with an `IntRange` object, where you can specify the minimal and maximal set size. The example above will return a Pareto size with size in the range of \([30, 50]\). For reducing the Pareto set size, the distance between two vector elements is taken into account. Points which lie very close to each other are removed. This leads to a result, where the Pareto optimal solutions are, more or less, evenly distributed over the whole Pareto front. The crowding distance\(^{11}\) measure is used for calculating the proximity of two points and it is described in\(^{[10]}\) and\(^{[18]}\).

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

```java
final Problem<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\(^{3.1.8}\) you can see how the resulting solution will look like. There is almost no difference in creating an evolution `Engine` for single- or multi-objective optimization. You only have to take care to choose the right `Selector`. Not all `Selectors` will work for multi-objective optimization. This include all `Selectors` which needs a `Number` fitness type and where the population needs to be sorted\(^{12}\). The `Selector` which works fine in a multi-objective setup is the `TournamentSelector`. Additionally you can use one of the special MO selectors: `NSGA2Selector` and `UFTournamentSelector`.

**NSGA2 selector** This selector selects the first elements of the population, which has been sorted by the Crowded-comparison operator (equation\(^{3.1.1}\)), as described in\(^{[10]}\):

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

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

\(^{11}\)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.\(^{https://www.igi-global.com/dictionary/crowding-distance/42740}\)

\(^{12}\)Since the \(\succeq\) relation doesn’t define a total order, sorting the population will lead to an `IllegalArgumentException` at runtime.
Unique fitness tournament selector The selection of unique fitnesses lifts the selection bias towards over represented fitnesses by reducing multiple solutions sharing the same fitness to a single point in the objective space. It is therefore no longer required to assign a crowding distance of zero to individual of equal fitness as the selection operator correctly enforces diversity preservation by picking unique points in the objective space.[18]

Since the multi-objective optimization (MOO) classes are an extensions to the existing evolutionEngine, 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 4.2 for comparing the Jenetics evolution flavor.

3.1.7.3 Termination

Most of the existing termination strategies, implemented in the Limits class, presumes a total order of the fitness values. This assumption holds for single-objective optimization problems, but not for multi-objective problems. Only termination strategies which don’t rely on the total order of the fitness value, can be safely used. The following termination strategies can be used for multi-objective problems:

- Limits::byFixedGeneration,
- Limits::byExecutionTime and
- Limits::byGeneConvergence.

All other strategies doesn’t have a well defined termination behavior.

3.1.7.4 Mixed optimization

Till now, we have only considered MOO problems, where all objectives where either minimized or maximized. This property might be to restrictive for some problem classes. If you have MOO problem with three objectives, for example, where objective one and three must be minimized and objective two has to be maximized, you need some additional mechanisms for doing this. Defining the optimization direction of the Engine is not sufficient. The fitness result Vec has to be configured accordingly. This can be done by using the most generic factory method of the Vec interface. Since this is quite bothersome, the VecFactory can be used for this task. Listing 3.10 shows the main method of the interface. The additional static factory methods has been omitted.

```java
@FunctionalInterface
public interface VecFactory<T> {
    Vec<T> newVec(final T array);
}
```

Listing 3.10: VecFactory interface

Instead of creating the solution Vec instances directly, the fitness function must create it with a properly configured VecFactory instance.
3.1.8 Grammatical evolution

One of the difficulties when defining a codec\(^{13}\) for a given problem, is the creation of invalid solution candidates or individuals. Using constraints is one way of handling this problem, which is described in section\(^{2.5}\) on page 62\(^{13}\). Another possibility is to assign bad fitness values to invalid individuals. This might work to some degree, but at the cost of additional CPU and evolution time. In the best case, your codec is creating no invalid solutions at all. Grammatical evolution (GE) can help you to achieve this. GE was introduced by Michael O’Neill and Conor Ryan\(^{37, 32, 38}\) for solving genetic programming (GP)\(^{14}\) problems. Although the initial realm for GE was creating/evolving programs in arbitrary languages, it is not restricted to this area. GE can be used for every problems, where the problem domain can be expressed as sentences of a context-free language (CFL)\(^{15}\).

GE requires an additional mapping step for creating a sentence (program) from a given CFG. Diagram 3.1.10 on the following page shows the principal decoding process of GE. A BitChromosome or IntegerChromosome determines the rules which are used for creating the sentences. The grammar is usually given in Backus-Naur form (BNF). The result is the construction of a syntactically correct program from a binary string which can then be evaluated by a fitness function. In the following sections, the building blocks of GE are described in a greater detail.

\(^{13}\) See section\(^{2.3}\) on page 54\(^{13}\).

\(^{14}\) See section\(^{3.2}\) on page 108\(^{14}\).

3.1.8.1 Context-free grammar

A context-free grammar (CFG) basically consists of a finite set of grammar rules. The grammar rules consists of two kind of symbols: 1) the terminals, which are the symbols of the alphabet underlying the languages, and 2) the non-terminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \rightarrow \alpha$, where $A$ is a non-terminal, and $\alpha$ is a terminal or non-terminal symbol. CFGs are used to generate strings (sentences), rather than recognize strings.

Definition. (CFG): A context-free grammar, $G$, is defined by the tuple $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, S)$, where

1. $\mathcal{N}$ is a finite set of non-terminal symbols called the variables. Every symbol represents a different type of phrase in the sentence.

2. $\mathcal{T}$ is a finite set of terminal symbols, not element of $\mathcal{N}$. Terminals make up the actual content of the sentences defined by $G$. The terminals are the alphabet of the language.

3. $\mathcal{R}$ is a finite relation in $\mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$. Members of $\mathcal{R}$ are the rewrite rules of the grammar.

4. $S \in \mathcal{N}$ is the start symbol and represents the whole sentence.

3.1.8.2 Backus-Naur form

The Backus-Naur form (BNF) is a traditional form to represent a CFG. It is used to formally define the grammar of a language, so that there is no ambiguity as to what is allowed and what is not. BNF uses a range of symbols and
expressions to create production rules. The names of the production rules are put within angle brackets, \(<...>\), and the alternatives are separated by vertical bars, |. A simple BNF production rule might look like this:

\[
<\text{num}> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

In this example, \(\text{num}\) is the name of the production rule and the values 1..9 represent the terminal symbols, associated with the rule. If a non-terminal symbol appears on the right hand side, it means that there will be another production rule (or a set of rules) to define its replacement.

\[
<\text{expr}> ::= <\text{num}> | <\text{var}> | <\text{expr}> <\text{op}> <\text{expr}>
\]

This shows that \(\text{expr}\) is either an \(\text{num}\) or \(\text{var}\) or two \(\text{expr}\), concatenated by an \(\text{op}\). All components are non-terminal, therefore further production rules are required.

\[
<\text{op}> ::= + | - | * | /
<\text{var}> ::= x | y
\]

This two rules defines four \(\text{op}\) symbols, \{\(x, -, +, /\}\), and two \(\text{var}\) terminals, \{\(x, y\}\}. Production rules for the syntax of a language might come as a large set of BNF statements that specify how every aspect of the language is defined. Every non-terminal symbol on the right hand side of a production rule, must have a rule that has the symbol on the left side. This continues until everything can be specified in relation to terminal symbols. Here is the whole grammar, which will define a language for simple arithmetic expressions:

\[
<\text{expr}> ::= <\text{num}> | <\text{var}> | <\text{expr}> <\text{op}> <\text{expr}>
<\text{op}> ::= + | - | * | /
<\text{var}> ::= x | y
<\text{num}> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

The symbol of the first rule, \(\text{expr}\), will automatically serve as start symbol of the grammar, \(G\), defined by the given BNF.

### 3.1.8.3 Sentence generation

After we have defined the CFG, we want to create a valid (random) sentences from our grammar. To do so, we will introduce a numbering schema for the alternative expressions for every rule. Such a schema, for our simple arithmetic grammar is shown below:

\[
[0] \quad [1] \quad [2]
<\text{expr}> ::= <\text{num}> | <\text{var}> | <\text{expr}> <\text{op}> <\text{expr}>
[0] \quad [1] \quad [2] \quad [3]
<\text{op}> ::= + | - | * | /
[0] \quad [1]
<\text{var}> ::= x | y
<\text{num}> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

The following example shows snapshots of the symbol list, \(L\), for a possible sentence creation run, according to the described leftmost derivation algorithm [3.1] on the next page.
Algorithm 3.1 Leftmost derivation

1. \( L \leftarrow [S] \)
   Initialize the symbol list, \( L \), with the start symbol, \( S \).

2. \( n \leftarrow L[\min\{i\}] \in N \)
   Pick the leftmost non-terminal symbol, \( L[\min\{i\}] \), from the current sentence list, \( L \).

3. \( E \leftarrow R(n)[\text{rand}] \)
   Get the rule, \( R(n) \), for the chosen non-terminal, \( n \), and select a random rule alternative, \( E \). \( E \) will contain one or more terminal and/or non-terminal symbols.

4. \( L[\min\{i\}] \leftarrow E \)
   Replace the chosen variable, \( n \), with the selected symbols, \( E \).

5. Repeat steps 2..4 until the symbol list, \( L \), contains only terminals.

1. \( L = [\langle expr\rangle] \). Initialize the sentence list with the start symbol.

2. \( L = [\langle expr\rangle, \langle op\rangle, \langle expr\rangle] \). Select the first non-terminal symbol in the list, \( \langle expr\rangle \), and replace it with an randomly chosen rule alternative. Alternative \([2]\), \( \langle expr\rangle \langle op\rangle \langle expr\rangle \), is chosen and replaces the variable \( \langle expr\rangle \).

3. \( L = [\langle num\rangle, \langle op\rangle, \langle expr\rangle] \). Replacing the first \( \langle expr\rangle \) with alternative \([0]\), \( \langle num\rangle \).

4. \( L = [5, \langle op\rangle, \langle expr\rangle] \). Replacing \( \langle num\rangle \) with 5.

5. \( L = [5, -, \langle expr\rangle] \)

6. \( L = [5, -, \langle var\rangle] \)

7. \( L = [5, -, x] \)

After the 6th iteration the symbol list contains only terminal symbols and the generation process stops. If we convert the symbol list into a string, we get an algebraic expression, \( 5 - x \), which is an element of the CFL, defined by our simple CFG. Algorithm 3.2 on the following page shows the rightmost derivation method, which is a variation of the leftmost derivation algorithm. The only difference is, that instead of the processing the symbol list from left to right, it is processed from right to left.

3.1.8.4 Mapping

Now we have everything in place to do the last step. How to create a sentence from a given grammar? The original paper [37] uses a bit string for creating a sentence from a given chromosome and called this process mapping. Since we have already described how to create a sentence from a grammar, we must describe the last missing part of the process, and this missing part is the selection of alternative symbols from a given rule. This is step three in the described algorithms 3.1 on...
Algorithm 3.2 Rightmost derivation

1. $L \leftarrow [S]$
   Initialize the symbol list, $L$, with the start symbol, $S$.

2. $n \leftarrow L[\max\{i\}] \in \mathcal{N}$
   Pick the rightmost non-terminal symbol, $L[\max\{i\}]$, from the current sentence list, $L$.

3. $E \leftarrow R(n)\ [\text{rand}]$
   Get the rule $R(n)$, for the chosen non-terminal, $n$, and select a random rule alternative, $E$.
   $E$ will contain one or more terminal and/or non-terminal symbols.

4. $L[\min\{i\}] \leftarrow E$
   Replace the chosen variable, $n$, with the selected symbols, $E$.

5. Repeat steps 2..4 until the symbol list, $L$, contains only terminals.

Instead of selecting a random alternative, the index of the selected alternative must be determined by the given chromosome.

Figure 3.1.11: GE mapping

Figure 3.1.11 shows he GE mapping process. The binary string is split into 8-bit junks and interpreted as unsigned integers, with values in the range of [0, 256). This 8-bit junks are called codons. Whenever a new rule alternative must be selected, a new codon is read from the chromosome. A simple modulo operation is used to get an index within the desired range. If more codons are needed during the mapping process than available in the input chromosome, the reading of the codons is wrapped over and starts at the beginning of the chromosome again.
3.1.8.5 Implementing classes

This section describes the main classes which are part of the implementation of the GE. All the described classes are part of the \texttt{io.jenetics.ext} module and reside in the \texttt{io.jenetics.ext.grammar} package.

\textbf{Cfg} \hspace{1em} This class represents a context-free grammar as described in section 3.1.8.1 on page 101.

\textbf{Bnf} \hspace{1em} Although it is possible to create instances of the \texttt{Cfg} class manually, it is in most cases easier to define it as BNF string. This class contains methods for parsing and formatting CFGs from and to BNF strings.

\textbf{SymbolIndex} \hspace{1em} The \texttt{SymbolIndex} interface defines the strategy, which alternative symbols of a given rule are selected during the sentence generation process. The classical algorithm uses codons, encoded in an binary string.

\textbf{Codons} \hspace{1em} This class is an implementation of the classical symbol selection algorithms as described in the original paper. It implements the \texttt{SymbolIndex} interface.

\textbf{Generator} \hspace{1em} The \texttt{Generator} interface is an abstraction of the sentence generation algorithm as such as algorithm 3.1 on page 103 or 3.2 on the preceding page.

\textbf{Mappings} \hspace{1em} The \texttt{Mappings} class contains factory methods for creating different mapping strategies. Since the mapping process, as shown in figure 3.1.11 on the previous page, is implemented as \texttt{Codec}\footnote{See section 2.3 on page 54}, the factory methods will return \texttt{Codec} instances for the \texttt{BitChromosome} and \texttt{IntegerChromosome}.

Let’s start with the \texttt{Cfg} class and how to create grammars. The \texttt{Cfg} class comes with a set of factory methods, which lets you create \texttt{Cfg} objects quite easily. Our already known arithmetic expression grammar can be created with the following code. The used factory methods, \texttt{N}, \texttt{T}, \texttt{E}, \texttt{R}, were statically imported.

```java
final Cfg<String> cfg = Cfg.of(
    R("expr",
        E(N("num")), E(N("var")),
        E(N("expr"), N("op"), N("expr"))
    ),
    R("op", E(T("+")), E(T("-")), E(T("*")), E(T("/"))),
    R("var", E(T("x")), E(T("y"))),
    R("num",
        E(T("1")), E(T("2")), E(T("3")),
        E(T("4")), E(T("5")), E(T("6")),
        E(T("7")), E(T("8")), E(T("9"))
    )
);
```

As alternative, the above CFG can also been creating via a string in BNF form, as described in section 3.1.8.2 on page 101. The code snippet below shows how to do this.

```java
final Cfg<String> cfg = Bnf.parse(""
    <expr> ::= <num> | <var> | <expr> <op> <expr>
    <op> ::= + | - | * | /
    <var> ::= x | y
"");
```

\footnote{See section 2.3 on page 54}
The `SymbolIndex` interface is responsible for selecting the index of the rule alternative to choose. The reason for this interface is to decouple the selection algorithm from the data structure, which determines the selection; it should decouple the algorithm from the underlying `BitChromosome` or `IntegerChromosome`.

```java
@FunctionalInterface
public interface SymbolIndex {
    int next(Cfg.Rule<?> rule, int bound);
}
```

Listing 3.11: `SymbolIndex` interface

Listing 3.11 shows the single method of the `SymbolIndex` interface. It takes the rule, for which to select the alternatives and the desired index bounds and returns the selected symbol index. The `rule` parameter of the next function allows to use different index selection strategies, or different index sources, for the different rules. This interface makes it possible to let the sentence creation to be controlled by an `IntegerChromosome` instead the classical bit string.

The `Codons` class implements the `SymbolIndex` interface and can be created from a `BitChromosome` and `IntegerChromosome` instances. `Codons` created from a `BitChromosome` gives you an implementation of the classic symbol selection algorithm.

```java
// Create codons backed up by binary string.
var bch = BitChromosome.of(100*8);
var bcodons = Codons.ofBitGenes(bch);

// Create codons backed up by integer string.
var ich = IntegerChromosome.of(IntRange.of(0, 256), 100);
var icodons = Codons.ofIntegerGenes(ich);
```

The code snippet above shows two ways for creating essentially the same codons. In the first variant, the codons are read from a `BitChromosome` with the length of 800, which results in 100 different indexes readable from the created `Codons` object. In the second variant an `IntegerChromosome`, with a value range of [0, 256] and a length of 100, is used for the `Codons` object. Using an `IntegerChromosome` instead of a `BitChromosome` gives you a greater flexibility, as it allows you to specify an explicit range for the codon values.

Implementations of the functional `Generator` interface are responsible for creating sentences from a given grammar. It takes an `Cfg` object as input and returns the created result sentence of type `R`. This genericity lets you use the same interface for different result type, like `List<Symbol<String>>` or `Tree<Symbol<String>>`.

```java
@FunctionalInterface
public interface Generator<T, R> {
    R generate(Cfg<? extends T> cfg);
}
```

Listing 3.12: `Generator` interface

Listing 3.12 shows the definition of the `Generator` interface. It takes an `Cfg` object as input and returns the created result sentence of type `R`. The kind of the created result is not determined by interfaces. It can be a random instance of the input
grammar, a reproducible, deterministic result, or even always the same result. It is the responsibility of the implementations to determine which kind of result to return. The reason for this is, not make assumptions about the sentence creation and bake this assumptions into the API; this avoid a leaky abstraction.

There are currently two implementation of the `Generator` interface: 1) the `SentenceGenerator`, which generates a list of terminal symbols (sentences) from a given grammar, and 2) the `DerivationTreeGenerator`, generating a derivation tree, or parse tree.

```java
// Define sentence generator for creating random sentences.
var generator = new SentenceGenerator<String>(
    SymbolIndex.of(RandomGenerator.getDefault()),
    1_000
); // Create random sentence for given CFG.
List<Cfg.Terminal<String>> sentence = generator.generate(cfg);
String string = sentence.stream()
    .map(Cfg.Symbol::name)
    .collect(Collectors.joining());
```

In the code snippet above you can see how to create a `SentenceGenerator` which will create random sentences with a maximal length of 1000. The generated sentence will be a list of terminal symbols. But it is quite easy to convert it to a string. The `DerivationTreeGenerator` work can be used similarly. Instead of a list of terminals, it creates a `Tree<Symbol<String>, ?>` instead.

The factory methods of the `Mappings` class puts it all together and let you easily specify the needed parts for the mapping process. Jenetics already has an interface for expressing such a mapping process, the `Codec` interface. So no additional interface or class was introduce and the `Mappings` factories will return `Codec` instances instead. In the code snippet below you can see how to create the classical mapping, with a single `BitChromosome` used for the rule symbol selection. It is created via the `Mappers::singleBitChromosomeMapper` method.

```java
final Codec<List<Terminal<String>>, BitGene> codec =
    Mappers.singleBitChromosomeMapper(
        cfg,
        // Length of the used BitChromosome.
        100*8,
        // The used generator, created from SymbolIndex.
        index -> new SentenceGenerator<>(index, 1_000)
    );
```

For creating this codec you must specify the CFG, the length of the chromosome and the sentence `Generator`. The `Generator` is given as factory function, which takes an `SymbolIndex` at input. The codec, created in the code snippet below, is essentially the same as created with the `Mappers::singleBitChromosomeMapper` method. It only differs in the way the codons are created. Using an `IntegerChromosome` as codons source gives you a greater flexibility and allows to change the value range of the created codons.

```java
final Codec<List<Terminal<String>>, IntegerGene> codec =
    Mappers.singleIntegerChromosomeMapper(
        cfg,
        // Value range of created codons.
        IntRange.of(0, 256),
```

\[^{19}\text{See section 2.3 on page 54}\]

107
Beside the classical mapping algorithms, **Jenetics** contains also a novel method for genotype to sentence mapping. This new approach uses a separate **Integer-Chromosome** for every **Cfg.Rule**. This allows to align the range of the chromosome with the number of alternatives of the rule. With this property, no modulo operation is needed for creating codons with the correct bounds. No additional modulo operation means, that all alternatives have the same probability for being selected, for randomly created chromosomes. All rule alternatives have the same changes of being selected. There is no accidental bias towards a given rule alternative or symbol.

```java
final Codec<List<Terminal<String>>, IntegerGene> codec =
    Mappers.multiIntegerChromosomeMapper(
        // Chromosome length depends of number of alternatives.
        cfg, // Chromosome length depends of number of alternatives.
        rule -> IntRange.of(rule.alternatives().size() * 25),
        index -> new SentenceGenerator<>(index, 1_000)
    );
```

The codes snippet above shows how to create a mapping, which uses a genotype with one chromosome for every rule in the given CFG. Our example CFG

```plaintext
<expr> ::= <num> | <var> | <expr> <op> <expr>
<op> ::= + | - | * | /
<var> ::= x | y
<num> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

will use a **Genotype** with the following structure for the encoding of the codons.

```java
Genotype.of(
    IntegerChromosome.of(IntRange.of(0, 3), 3 * 25), // <expr>
    IntegerChromosome.of(IntRange.of(0, 4), 4 * 25), // <op>
    IntegerChromosome.of(IntRange.of(0, 2), 2 * 25), // var
    IntegerChromosome.of(IntRange.of(0, 9), 9 * 25) // <num>
);
```

As you can see, the number range for each rule chromosome reflects the number of available rule alternatives and the chromosome length can be expressed as a multiple of the number of alternatives of the corresponding rule. If you don’t need this flexibility, it is of course also possible to use constant chromosome lengths. Just use a length function like this: `rule -> IntRange.of(1_000)`.

## 3.2 io.jenetics.prog

In artificial intelligence, **genetic programming** (GP) is a technique whereby computer programs are encoded as a set of genes that are then modified (evolved) using an evolutionary algorithm (often a genetic algorithm). The **io.jenetics**...
3.2. IO.JENETICS.PROG

The .prog module contains classes which enables the Jenetics library doing GP. It introduces a ProgramGene and ProgramChromosome pair, which serves as the main data structure for genetic programs. A ProgramGene is essentially a tree (AST\(^\text{21}\)) of operations (Op) stored in a ProgramChromosome\(^\text{22}\).

3.2.1 Operations

When creating own genetic programs, it is not necessary to derive classes from the ProgramGene or ProgramChromosome. The intended extension point is the Op interface.

> The extension point for own GP implementations is the Op interface. There is in general no need for extending the ProgramChromosome class.

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

Listing 3.13: GP Op interface

The generic type of the Op interface (see listing 3.13) 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. These 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.

\(^\text{21}\)https://en.wikipedia.org/wiki/Abstract_syntax_tree

\(^\text{22}\)When implementing the GP module, the emphasis was to not create a parallel world of genes and chromosomes. It was a requirement, that the existing Alterer and Selector classes could also be used for the new GP classes. This has been achieved by flattening the AST of a genetic program to fit into the 1-dimensional (flat) structure of a chromosome.
3.2. IO.JENETICS.PROG  

**Var** The **Var** operation defines a variable of a program, which is set from outside when it is evaluated.

```java
final Var<Double> x = Var.of("x", 0);
final Var<Double> y = Var.of("y", 1);
final Var<Double> z = Var.of("z", 2);
final ISeq<Op<Double>> terminals = ISeq.of(x, y, z);
```

The terminal operations defined in the listing above can be used for defining a program which takes a 3-dimensional vector as input parameters, \(x, y,\) and \(z,\) with the argument indices 0, 1, and 2. If you have again a look at the **apply** method of the operation interface, you can see that this method takes an object array of type \(T\). The variable \(x\) will return the first element of the input arguments, because it has been created with index 0.

**Const** The **Const** operation will always return the same, constant value when evaluated.

```java
final Const<Double> one = Const.of(1.0);
final Const<Double> pi = Const.of("PI", Math.PI);
```

You can create a constant operation in two flavors: with a value only, and with a dedicated name. If a constant has a name, the symbolic name is used, instead of the value, when the program tree is printed.

**EphemeralConst** An ephemeral constant is a terminal operation, which encapsulates a value that is generated at run time from the **Supplier** it is created from. Ephemeral constants allows you to have terminals that don’t have all the same values. To create an ephemeral constant that takes its random value in \([0, 1)\) you will write the following code.

```java
final Op<Double> rand1 = EphemeralConst.of(RandomRegistry.random()::nextDouble);
final Op<Double> rand2 = EphemeralConst.of("R", RandomRegistry.random()::nextDouble);
```

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

### 3.2.2 Program creation

The **ProgramChromosome** comes with some factory methods, which lets you easily create program trees with a given depth and a given set of operations and terminals.

```java
final int depth = 5;
final ISeq<Op<Double>> operations = ISeq.of(...);
final ISeq<Op<Double>> terminals = ISeq.of(...);
final ProgramChromosome<Double> program = ProgramChromosome.of(depth, operations, terminals);
```

The code snippet above will create a **perfect** program tree 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.

\(^{23}\)All leafs of a perfect tree have the same depth and all internal nodes have degree **Op.arity**.
The created program tree is perfect, which means that all leaf nodes have the same depth. If new trees needs to be created during evolution, they will be created with the depth, operations and terminals defined by the template program tree.

During the evolution phase, the size of the ProgramChromosome can grow and shrink. The SingleNodeCrossover, which is part of the jenetics.ext module is responsible for this change in the program size. When a smaller sub tree is exchanged with a bigger sub tree, the size of the first tree will grow and the size of the second tree will shrink. This can lead to undesirable large programs. Because of this reason, it is possible to create a ProgramChromosome with an additional validation predicate.

```java
final ProgramChromosome<Double> program = ProgramChromosome.of(
    depth, ch -> ch.root().size() <= 50,
    operations, terminals
);
```

The predicate, ch -> ch.root().size() <= 50, marks all programs with more then 50 nodes as invalid. Invalid chromosomes will then be replaced by newly created one. When defining a validation predicate, you have to take care, that the desired depth and the validation predicate matches. If the given program tree depth is too big, e.g. 51, every newly created program will be immediately marked as invalid. This is because a tree with depth 51 will have for sure more than 50 nodes.

The evolution Engine used for solving GP problems is created the same way as for normal GA problems. Also the execution of the EvolutionStream stays the same. The first Gene of the collected final Genotype represents the evolved program, which can be used to calculate function values from arbitrary arguments.

```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())
    .gene();

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

For a complete GP example have a look at the examples in chapter5.7. The code example above also shows, that the program is represented by the first gene (aka root gene) of the ProgramChromosome. Since the ProgramGene implements the Tree<Op<A>, ProgramGene<A>>, interface, it smoothly integrates with existing tree algorithms. Some possible program gene assignments are shown in the code snippet below, which will compile without warnings or additional casts.

```java
final ProgramChromosome<Double> chromosome = ...
```

111
3.2.3 Program repair

The specialized crossover class, `SingleNodeCrossover`, for a `TreeGene` guarantees that the program tree after the alter operation is still valid. It obeys the tree structure of the `Gene`. General alterers, not written for `ProgramGene` of `TreeGene` classes, will most likely destroy the tree property of the altered chromosome. There are essentially two possibility for handling invalid tree chromosomes:

1. Marking the `Chromosome` as invalid. This possibility is easier to achieve, but would also lead to a large number of invalid `Chromosomes`, which must be recreated. When recreating invalid `Chromosomes` we will also lose possible solutions.

2. Trying to repair the invalid `Chromosome`. This is the approach the `Jenetics` library has chosen. The repair process reuses the operations in a `Program-Chromosome` and rebuilds the tree property by using the operation arity.

`Jenetics` allows for the usage of arbitrary `Alterer` implementations. Even alterers not implemented for `ProgramGenes` `Gens` destroyed by such alterers are repaired.

3.2.4 Program pruning

When you are solving symbolic regression problems, the mathematical expression trees, created during the evolution process, can become quite big. From the diversity point of view, this might be not that bad, but it comes with additional computation cost. With the `MathRewriteAlterer` you are able to simplify some portion of the population in each generation. The rewrite alterer uses the `defaultTreeRewriter` defined by the `MathExpr.REWRITER` field. It is also possible to create a `MathRewriteAlterer` instance with your own `TreeRewriter`.

```java
final Engine<ProgramGene<Double>, Double> engine = Engine
    .builder(Main::error, program)
    .minimizing()
    .alterers(
        newSingleNodeCrossover<>(),
        new Mutator<>(),
        new MathRewriteAlterer<> (0.5))
    .build();
```

²⁴See section 3.1.2 on page 86 for a detailed description of the implemented tree rewrite system.
In the example above, half of the expression trees are simplified in each generation. If you want to prune the final result, you can do this with the `MathExpr::rewrite` method, which uses the `MathExpr.REWRITER` tree rewriter for the rewrite task.

```java
final ProgramGene<Double> program = engine.stream()
    .limit(3000)
    .collect(EvolutionResult.toBestGenotype());
final TreeNode<Op<Double>> tree = TreeNode.ofTree(program);
MathExpr.rewrite(tree);
```

The algorithm used for pruning the expression tree, currently only uses some basic mathematical identities, like $x + 0 = x$, $x \cdot 1 = x$, or $x \cdot 0 = 0$. More advanced simplification algorithms may be implemented in the future. The `MathExpr` helper class can also be used for creating mathematical expression trees from the usual textual representation.

```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.

### 3.2.5 Multi-root programs

The given examples, so far, where using a single `ProgramChromosome` for modeling the program. Since the `Genotype` is able to hold more than one `Chromosome`, it is possible to create more than one program root. These programs are evaluated concurrently.

```java
final Codec<ISeq<Function<Double [] , Double>>, ProgramGene<Double>> codec = Codec.of(
    Genotype.of(
        // First 'program'.
        ProgramChromosome.of(
            4, ch -> ch.root().size() <= 30,
            operations, terminals
        ),
        // Second 'program'.
        ProgramChromosome.of(
            5, ch -> ch.root().size() <= 50,
            operations, terminals
        ),
        gt -> gt.stream()
            .map(Chromosome::gene)
            .collect(ISeq.toISeq())
    );
```

The code snippet above shows how to create a codec with two independent program roots. These programs are then mapped, in the fitness function, to the combined fitness value. It is also possible to use different operations and terminals for each `ProgramChromosome`. 

113
3.2.6 Symbolic regression

Symbolic regression is a specific type of regression analyses, where the search space consists of mathematical expressions. The task is to find a model, which fits a given data set in terms of accuracy and simplicity. In a classical approach, you will try to optimize the parameters of a predefined function type, e.g. a polynomial of grade $n$:

$$f(x) = \sum_{k=0}^{n} a_k x^k.$$  

The encoding would only be a `DoubleChromosome` of length $n + 1$, where the `Gene` at position $k \in [0, ..., n]$ represents the factor $a_k$ of the polynomial. If the type of mathematical function is not known in advance, GP can be used finding a function which is composed out of a given set of primitives.

Symbolic regression involves finding a mathematical expression, in symbolic form, that provides a good, best, or perfect fit between a given finite sampling of values of the independent variables and the associated values of the dependent variables.\(^\text{[24]}\)

Since symbolic regression is quite a common task in GP, `Jenetics` comes with classes and interfaces, supporting the implementation of such problems. These classes are defined in the `io.jenetics.prog.regression` package. The following sections describes these classes and interfaces and its usage. A complete symbolic regression example is given in section 5.7.

3.2.6.1 Loss function

The loss function measures how good the evolved program (tree) predicts the expected outcome or data set. If the prediction deviates too much from the expected data, the loss function will cough up a larger number. Loss functions are classified into two major categories, depending on the type of the learning task—regression losses and classification losses. In the following paragraphs, only loss functions suitable for regression problems will be described.

**Mean squared error**  The mean squared error is the default loss function used for regression analysis. It is also known as quadratic loss or $L2$ loss and is calculated as the average of the squared differences between the predicted and actual values.

$$MSE = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2,$$  \hspace{1cm} (3.2.1)

where $y_i$ denotes the expected function value and $\hat{y}_i$ the calculated (estimated) value for data point $i$. The result is always positive and the perfect value is 0. The squaring means that larger mistakes result in more errors than smaller mistakes, meaning that the model penalizes larger mistakes. The mean squared error is the preferred loss function for regression problems.

**Mean absolute error**  The mean absolute error, also known as $L1$ loss, is calculated as the average of the absolute difference between the expected and
calculated values.

\[ MAE = \frac{1}{n} \sum_{i=0}^{n-1} |y_i - \tilde{y}_i| \]  

(3.2.2)

This loss function is suitable for regression problems where the distribution of the target variable may be mostly Gaussian, but may have outliers, e.g. large or small values far from the mean value. This means that the MAE is more robust than the MSE, which is useful if the sample data is corrupted with outliers.

The MAE is more robust to outliers, but its derivatives are not continuous, making it less efficient to find the correct solution. The MSE is sensitive to corrupt data, but finds more stable and closed form solutions.

The interface used for calculating the loss between calculated and expected values is shown in listing 3.14.

```java
@FunctionalInterface
class LossFunction<T> {
    double apply(T[] calculated, T[] expected);
}
```

Listing 3.14: `LossFunction` interface

### 3.2.6.2 Complexity function

The complexity function measures the complexity of the evolved tree. If you have two programs with the same loss value, you usually want the simpler program to survive. A simple complexity measure is the number of nodes a program tree consists of. You can obtain such a measure by the `ofNodeCount(int)` factory method of the `Complexity` interface. The complexity measure, \( C(P) \), is defined as

\[ C(P) = 1 - \sqrt{1 - \frac{\min(N(P), N_{max})^2}{N_{max}^2}}, \]  

(3.2.3)

where \( N(P) \) is the number of nodes the program, \( P \), consists of and \( N_{max} \) the maximal allowed program nodes. If the number of program nodes is equal or greater than the maximal node number, \( C(P) \), will return 1.

The graph in figure 3.2.1 shows how the program complexity increases with the number of nodes. For the example graph the maximal node count was set to 28.

```java
@FunctionalInterface
class Complexity<T> {
    double apply(Tree<T> extends Op<T>, T[] program);
}
```

Listing 3.15: `Complexity` interface

Listing 3.15 shows the interface which calculates the complexity measure of a given program tree.
3.2.6.3 Error function

The error function combines the loss function and the complexity function into one error measure. It is used as fitness function which the GP is minimizing.

```java
@FunctionalInterface
public interface Error<T> {
    double apply(
        Tree<? extends Op<T>, ?> program,
        T[] calculated,
        T[] expected
    );
}
```

Listing 3.16: Error interface

Listing 3.16 shows the error (fitness) function used for evolving symbolic regression problems. Instead of implementing the error function from scratch, you will probably want to use one of the factory methods for creating it from one of the predefined LossFunction and Complexity measure.

```java
final Error<Double> error1 = Error.of(LossFunction::mse);
final Error<Double> error2 = Error.of(  
    LossFunction::mae,  
    Complexity.ofNodeCount(28)  
);  
final Error<Double> error3 = Error.of(  
    LossFunction::mse,  
    Complexity.ofNodeCount(28),  
    (loss, complexity) -> loss + loss * complexity  
);
```

The code snippet above shows the three possibilities to create an error function by using the predefined loss functions and complexity measure. `error1` is created by using the mean squared error, MES.`error2` and `error3` defines the same error function. The only difference is that `error3` defines the loss-complexity composition function explicitly.
### 3.2.6.4 Sample points

Solving regression problems requires to compare the current solution (program tree) with a set of sample points, which represents the original function to be approximated. The `Sample` interface represents such sample point. It actually maps an n-dimensional point of domain $\mathbb{D}$, to an one-dimensional point of the same domain: $\mathbb{D}^n \rightarrow \mathbb{D}$.

```java
public interface Sample<T> {
    int arity();
    T argAt(int index);
    T result();
}
```

Listing 3.17: Sample interface

The arity of the sample point returns the dimension, $n$. To make it easier to create `double` sample points, some factory methods are also given in the `Sample` interface.

```java
final Sample<Double> sample1 = Sample.ofDouble(0.0, 0.0);
final Sample<Double> sample2 = Sample.ofDouble(1.0, 1.0);
final Sample<Double> sample3 = Sample.ofDouble(2.0, 2.0);
```

The code snippet above shows how to create three sample points for a function $f: \mathbb{R} \rightarrow \mathbb{R}$.

### 3.2.6.5 Regression problem

The `Regression` class is the only concrete type of the public API of the `regression` package. It integrates the interfaces, described in the last sections, into one problem definition.

```java
public final class Regression<T> implements Problem<Tree<Op<T>, ?>, ProgramGene<T>, Double> {
    ...
}
```

As you can see in the code snippet above, the `Regression` class implements the `Problem` interface and can be therefore easily used in setting up an appropriate evolution `Engine`. A full such regression example can be found in section 5.7.

### 3.2.7 Boolean programs

The default data type for doing symbolic regression is the `Double` class. This is supported by a standard set of mathematical operations, defined in the `MathOp` class. Since the GP operations are not restricted to any particular type, the boolean operations, defined in the `BoolOp` class, can be used for defining boolean programs.

### 3.3 io.jenetics.xml

The `io.jenetics.xml` module allows for writing and reading `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.

### 3.3.1 XML writer

The main entry point for writing XML files is the typed `XMLWriter` interface. Listing 3.18 shows the interface of the `XMLWriter`.

```java
@FunctionalInterface
public interface Writer<T> {
    void write(XMLStreamWriter xml, T data)
            throws XMLStreamException;

    static <T> Writer<T> attr(String name);
    static <T> Writer<T> attr(String name, Object value);
    static <T> Writer<T> text();

    static <T> Writer<T> elem(String name, Writer<? super T>... children);
    static <T> Writer<Iterable<T>> elems(Writer<? super T> writer);
}
```

Listing 3.18: `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.

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

3.3.2 XML reader

Reading and writing XML files uses the same concepts. For reading XML there is an abstract `Reader` class, which can be easily composed. The main method of the Reader class can be seen in listing 3.19.
### 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 3.3.1 shows the required space of the marshaled genotypes for different marshalling methods: (a) Java serialization, (b) JAXB\textsuperscript{25} serialization and (c) XML Writer.

Using the Java serialization will create the smallest files and the XML Writer of the io.jenetics.xml module will create files roughly 75% the size of the JAXB serialized genotypes. The size of the marshaled objects also influences the write performance. As you can see in diagram 3.3.1 the Java serialization is the fastest marshalling method, followed by the JAXB marshalling. The XML Writer is the slowest one, but still comparable to the JAXB method.

\textsuperscript{25}The JAXB marshalling has been removed in version 4.0. It is still part of the table for comparison with the new XML marshalling.
### 3.4. 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 implements the Java RandomGenerator.

---

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

27 https://de.wikipedia.org/wiki/Monte-Carlo-Simulation
interface, which makes it easily usable in other projects.

The pseudo random number generators of the `io.jenetics.prngine` module are not cryptographically strong PRNGs.

The `io.jenetics.prngine` module consists of the following PRNG implementations:

**KISS32Random** Implementation of an simple PRNG as proposed in *Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications* (JKISS32, page 3) David Jones, UCL Bioinformatics Group.\[22\] The period of this PRNG is \( \approx 2.6 \cdot 10^{36} \).

**KISS64Random** Implementation of an simple PRNG as proposed in *Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications* (JKISS64, page 10) David Jones, UCL Bioinformatics Group.\[22\] The PRNG has a period of \( \approx 1.8 \cdot 10^{75} \).

**LCG64ShiftRandom** This class implements a linear congruential PRNG with additional bit-shift transition. It is a port of the `trng::lcg64_shift` PRNG class of the TRNG library created by Heiko Bauke.\[28\]

---

\[28\] https://github.com/jenetics/trng4
3.4. JOJENETICS.PRNGINE

CHAPTER 3. MODULES

MT19937_32Random This is a 32-bit version of Mersenne Twister pseudo random number generator.\[29\]

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.\[30\])

XOR64ShiftRandom This generator was discovered and characterized by George Marsaglia [Xorshift RNGs]. In just three XORs and three shifts (generally fast operations) it produces a full period of \(2^{64} - 1\) on 64 bits. (The missing value is zero, which perpetuates itself and must be avoided.)

All implemented PRNGs have been tested with the dieharder test suite. Table 3.4.1 shows the statistical performance of the implemented PRNGs, including the Java RandomGenerator, L64X256MixRandom, which is the default generator used by the library.

<table>
<thead>
<tr>
<th>PRNG</th>
<th>Passed</th>
<th>Weak</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>KISS32Random</td>
<td>113</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>KISS64Random</td>
<td>109</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>LCG64ShiftRandom</td>
<td>111</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>MT19937_32Random</td>
<td>111</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>MT19937_64Random</td>
<td>108</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>XOR32ShiftRandom</td>
<td>103</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>XOR64ShiftRandom</td>
<td>113</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>L64X256MixRandom</td>
<td>111</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4.1: Dieharder results

The second important performance measure for PRNGs is the number of random number it is able to create per second.\[31\] Table 3.4.2 shows the PRN creation speed for all implemented generators.

---

\[29\]https://en.wikipedia.org/wiki/Mersenne_Twister
\[30\]http://digitalcommons.wayne.edu/jmasm/vol2/iss1/2/
\[31\]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 3.4.2: 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>L64X256MixRandom</td>
<td>162</td>
<td>136</td>
<td>121</td>
<td>166</td>
</tr>
</tbody>
</table>
Appendix
Chapter 4
Internals

This section contains internal implementation details which doesn’t fit in one of the previous sections. They are not essential for using the library, but would give the user a deeper insight in some design decisions made when implementing the library. It also introduces tools and classes which were developed for testing purpose. These classes are not exported and not part of the official API.

4.1 PRNG testing

Jenetics uses the dieharder\(^1\) (command line) tool for testing the randomness of the used PRNGs. dieharder is a random number generator (RNG) testing suite. It is intended to test generators, not files of possibly random numbers. Since dieharder needs a huge amount of random data for testing the quality of a RNG, it is usually advisable to pipe the random numbers to the dieharder process:

```
$ cat /dev/urandom | dieharder -g 200 -a
```

The example above demonstrates how to stream a raw binary stream of bits to the stdin (raw) interface of dieharder. With the DieHarder class, which is part of the io.jenetics.prngine.internal package, it is easily possible to test PRNGs extending the java.util.random.RandomGenerator interface. The only requirement is, that the PRNG must be default-constructible and part of the classpath.

```
$ java -cp prngine-2.0.0.jar\n   io.jenetics.prngine.internal.DieHarder\n   <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.

---
4.2 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 nanoSeed() {
    return System.nanoTime();
}
```

Before applying this method throughout the whole library, I decided to perform some statistical tests. For this purpose I treated the seed method itself as PRNG and analyzed the created long values with the DieHarder class. The

In the listing above, a part of the created dieharder report is shown. For testing the LCG64ShiftRandom class, which is part of the io.jenetics.prngine module, the following command can be called:

```
$ java -cp prngine-2.0.0.jar
   io.jenetics.prngine.internal.DieHarder
  LCG64ShiftRandom -a
```

Table 4.1.1 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>111</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1.1: LCG64ShiftRandom quality

4.2 Random seeding

4.2.1 Random seed

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.
4.2. RANDOM SEEDING

nanoSeed method has been wrapped into the `io.jenetics.prngine.internal.NanoTimeRandom` class. Assuming that the `dieharder` tool is in the search path, calling

```
$ java -cp prngine-2.0.0.jar
  io.jenetics.prngine.internal.DieHarder
  NanoTimeRandom -a
```

will perform the statistical tests for the nano timerandom engine. The statistical quality is rather bad: every single test failed. Table 4.2.1 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 4.2.1: 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 objectSeed() {
    final long a = new Object().hashCode();
    final long b = new Object().hashCode();
    return mixStafford13(a << 32 | b);
}
```

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

```
$ java -cp prngine-2.0.0.jar
  io.jenetics.prngine.internal.DieHarder
  ObjectHashRandom -a
```

Table 4.2.2 shows the summary of the `dieharder` report which is already excellent.

After additional experimentation, a combination of the nano time seed and the object hash seeding seems to be the right solution. The rational behind this was, that the PRNG seed shouldn’t rely on a single source of entropy.

---

4The detailed test report can be found in the source of the `NanoTimeRandom` class: https://github.com/jenetics/prngine/blob/master/prngine/src/main/java/io/jenetics/prngine/internal/NanoTimeRandom.java

4.2. RANDOM SEEDING

Table 4.2.2: Object hash seeding quality

<table>
<thead>
<tr>
<th>Passed tests</th>
<th>Weak tests</th>
<th>Failed tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>113</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The code in listing 4.1 shows how the nano time seed is mixed with the object seed. The mix method was inspired by the mixing step of the lcg64_shift\(^6\) random engine, which has been reimplemented in the LCG64ShiftRandom class. Running the tests with

```
$ java -cp prngine-2.0.0.jar
   io.jenetics.prngine.internal.DieHarder
SeedRandom -a
```

leads to the statistics summary\(^7\) which is shown in table 4.2.3.

<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 4.2.3: Combined random seeding quality

The statistical performance of this seeding is better, according to the die-harder\(^7\) test suite, than some of the real random engines, including the default Java Random engine. Using the proposed seed method is in any case preferable to the simple System.nanoTime() call.

Open questions

- How does this method perform on operating systems other than Linux?
- How does this method perform on other JVM implementations?

\(^6\)This class is part of the TRNG library

\(^7\)Full report
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.

```
$ ./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 5.6.

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.EvolutionStatistics;

public class OnesCounting {
    // This method calculates the fitness for a given genotype.
    private static Integer count(final Genotype<BitGene> gt) {
        return gt.chromosome().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  \hfill \textsc{chapter 5. examples}

\begin{verbatim}
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);
}
\end{verbatim}

The \texttt{Genotype} in this example consists of one \texttt{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 \texttt{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.\footnote{For the other default values (population size, maximal age, ...) have a look at the Javadoc\url{https://jenetics.io/javadoc/jenetics/7.2/index.html}}

\begin{verbatim}
+---------------------------------------------------------------------------+
| Time statistics             |                                   |
+---------------------------------------------------------------------------+
| Selection: sum=0.016580144000 s; mean=0.001381678667 s                  |
| Altering: sum=0.096904159000 s; mean=0.008075346583 s                  |
| Fitness calculation: sum=0.022894318000 s; mean=0.001907859833 s       |
| Overall execution: sum=0.136575323000 s; mean=0.011381276917 s        |
+---------------------------------------------------------------------------+
| Evolution statistics        |                                   |
+---------------------------------------------------------------------------+
| Generations: 12             |                                   |
| Altered: sum=40,487; mean=3373.9166666667                              |
| Killed: sum=0; mean=0.0000000000                                       |
| Invalids: sum=0; mean=0.0000000000                                     |
+---------------------------------------------------------------------------+
| Population statistics       |                                   |
+---------------------------------------------------------------------------+
|                                |                                   |
+---------------------------------------------------------------------------+
\end{verbatim}
The given example will print the overall timing statistics onto the console. In the Evolution statistics section you can see that it actually takes 15 generations to fulfill the termination criteria—finding no better result after 7 consecutive generations.

5.2 Real function

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

\[ f(x) = \cos \left( \frac{1}{2} + \sin (x) \right) \cdot \cos (x). \] (5.2.1)

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.
import io.jenetics.util.DoubleRange;

public class RealFunction {

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

    public static void main(final String[] args) {
        final Engine<DoubleGene, Double> engine = Engine.<
            RealFunction::fitness,
            Codecs.ofScalar(DoubleRange.of(0.0, 2.0*Math.PI)).
            populationSize(500).
            optimize(Optimize.MINIMUM).
            alterers(new Mutator<>(0.03),
                new MeanAlterer<>(0.6)).
            builder()
            .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 $[0, 2\pi]$. 

<table>
<thead>
<tr>
<th>Time statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection: sum=0.04406459000; mean=0.003066974095 s</td>
</tr>
<tr>
<td>Altering: sum=0.07015838200; mean=0.003340875333 s</td>
</tr>
<tr>
<td>Fitness calculation: sum=0.05045264700; mean=0.002402507000 s</td>
</tr>
<tr>
<td>Overall execution: sum=0.16983515400; mean=0.008087388286 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Evolution statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generations: 21</td>
</tr>
<tr>
<td>Altered: sum=3.897; mean=185.571428571</td>
</tr>
<tr>
<td>Killed: sum=0; mean=0.000000000</td>
</tr>
<tr>
<td>Invalids: sum=0; mean=0.000000000</td>
</tr>
</tbody>
</table>
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 is often used to test the optimization performance of genetic algorithm.

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

(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 \( f(x) = 0 \).

![Rastrigin function plot](https://en.wikipedia.org/wiki/Rastrigin_function)

Figure 5.3.1: Rastrigin function
The following listing shows the Engine setup for solving the Rastrigin function, which is very similar to the setup for the real function in section 5.2. Beside the different fitness function, the Codec for double vectors is used, instead of the double scalar Codec.

```java
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(double[] x) {
        double value = A*N;
        for (int i = 0; i < N; ++i) {
            value += x[i]*x[i] - A*cos(2.0*Math.PI*x[i]);
        }
        return value;
    }

    public static void main(String[] args) {
        final Engine<DoubleGene, Double> engine = Engine
            .builder()
            .ofVector(DoubleRange.of(-R, R), N)
            .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 its size and value, is given. The task is to select a disjoint subset so that the total size does not exceed the knapsack size. For solving the 0/1 knapsack problem we define a `BitChromosome`, one bit for each item. If the $i$th bit is set to one the $i$th item is selected.

```java
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import io.jenetics.BitGene;
import io.jenetics.Mutator;
import io.jenetics.Phenotype;
import io.jenetics.RouletteWheelSelector;
import io.jenetics.SinglePointCrossover;
import io.jenetics.TournamentSelector;
import io.jenetics.engine.Codec;
import io.jenetics.engine.Codecs;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionStatistics;
import io.jenetics.util.ISeq;
import io.jenetics.util.RandomRegistry;

// The main class.
public class Knapsack {
    // This class represents a knapsack item, with a specific
    // 'size' and 'value'.
    final static class Item {
        public final double size;
```
```java
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 var r = RandomRegistry.random();
    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());

    // Defining the codec.
    final Codec<ISeq<Item>, BitGene> codec = Codecs.ofSubSet(items);

    // Configure and build the evolution engine.
    final Engine<BitGene, Double> engine = Engine
        .builder(fitness(kssize), codec)
        .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, ?>
```
5.4. 0/1 KNAPSACK  

CHAPTER 5. EXAMPLES

```java
statistics = EvolutionStatistics.ofNumber();

final Phenotype<BitGene, Double> best = engine.stream()
    .limit(bySteadyFitness(7))
    .limit(100)
    .peek(statistics)
    .collect(toBestPhenotype());

final ISeq<Item> knapsack = codec.decode(best.genotype());

System.out.println(statistics);
System.out.println(best);
System.out.println("\n\n");
System.out.printf("Genotype of best item: %s\n", best.genotype());

final double fillSize = knapsack.stream()
    .mapToDouble(it -> it.size)
    .sum();

System.out.printf("%.2f%% filled.%n", 100 * fillSize / ksize);
```

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

<table>
<thead>
<tr>
<th>Time statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection: sum=0.044465978000 s; mean=0.005558247250 s</td>
</tr>
<tr>
<td>Altering: sum=0.067385211000 s; mean=0.008423151375 s</td>
</tr>
<tr>
<td>Fitness calculation: sum=0.037208189000 s; mean=0.004651023625 s</td>
</tr>
<tr>
<td>Overall execution: sum=0.126468539000 s; mean=0.015808567375 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Evolution statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generations: 8</td>
</tr>
<tr>
<td>Altered: sum=4,842; mean=605.250000000</td>
</tr>
<tr>
<td>Killed: sum=0; mean=0.000000000</td>
</tr>
<tr>
<td>Invalids: sum=0; mean=0.000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Population statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age: max=7; mean=1.387500; var=2.780039</td>
</tr>
<tr>
<td>Fitness:</td>
</tr>
<tr>
<td>min = 0.000000000000</td>
</tr>
<tr>
<td>max = 542.363235999342</td>
</tr>
<tr>
<td>mean = 436.098248628661</td>
</tr>
<tr>
<td>var = 11431.801291812390</td>
</tr>
<tr>
<td>std = 106.919601999878</td>
</tr>
</tbody>
</table>
```

[01111011[10111101] -> 542.3632359993417

137
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 paths for \( N \) cities would lead to \( N! \) checks to find the shortest one.

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

```java
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static java.lang.Math.sin;
import static java.lang.System.out;
import static java.util.Objects.requireNonNull;
import static io.jenetics.engine.EvolutionResult.toBestPhenotype;
import static io.jenetics.engine.Limits.bySteadyFitness;
import java.util.function.Function;
import java.util.stream.IntStream;
import io.jenetics.EnumGene;
import io.jenetics.Optimize;
import io.jenetics.PartiallyMatchedCrossover;
import io.jenetics.Phenotype;
import io.jenetics.SwapMutator;
import io.jenetics.engine.Codec;
import io.jenetics.engine.Codecs;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionStatistics;
import io.jenetics.engine.Problem;
import io.jenetics.util.ISeq;
import io.jenetics.util.MSeq;
import io.jenetics.util.RandomRegistry;

public class TravelingSalesman
    implements Problem<ISeq<double[]>, EnumGene<double[]>, Double>
{
    private final ISeq<double[]> _points;

    // Create new TSP problem instance with given way points.
    public TravelingSalesman(ISeq<double[]>> points) {
        _points = requireNonNull(points);
    }

    @Override
    public Function<ISeq<double[]>>, Double> fitness() {
        return p -> IntStream.range(0, p.length())
            .mapToDouble(i -> {
                final double p1 = p.get(i);
                final double p2 = p.get((i + 1) % p.size());
                return hypot(p1[0] - p2[0], p1[1] - p2[1]);
            }).sum();
    }
}
```


138
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});
    }

    // Shuffling of the created points.
    final var random = RandomRegistry.random();
    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());
}
5.6. EVOLVING IMAGES

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

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

Original idea by Roger Johansson
http://rogeralsing.com/2008/12/07/

[genetic-programming-evolution-of-mona-lisa]
$ ./gradlew jar

you can start the example by calling

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

![Evolving images UI]

Figure 5.6.1: Evolving images UI

Figure 5.6.1 shows the GUI after evolving the default image for about 4,000 generations. With the "Open" button it is possible to load other images for *polygonization*. The "Save" button allows to store *polygonized* images in PNG format to disk. At the button of the UI, you can change some of the GA parameters of the example:

**Population size** The number of individual in the population.

**Tournament size** The example uses a *TournamentSelector* for selecting the offspring population. This parameter lets you set the number of individuals used for the tournament step.

**Mutation rate** The probability that a polygon *component* (color or vertex position) is altered.

**Mutation magnitude** In case a polygon *component* is going to be mutated, its value will be randomly modified in the uniform range of \([-m, +m]\).

**Polygon length** The number of edges (or vertices) of the created polygons.
5.7. SYMBOLIC REGRESSION

Polygon count  The number of polygons of one individual (Genotype).

Reference image size  To improve the processing speed, the fitness of a given
polygon set (individual) is not calculated with the full sized image. Instead
a scaled reference image with the given size is used. A smaller reference
image will speed up the calculation, but will also reduce the accuracy.

It is also possible to run and configure the Evolving Images example from the
command line. This allows for performing long running evolution experiments and
save polygon images every n generations—specified with the --image-generation
parameter.

```
$ ./jrun io.jenetics.example.image.EvolvingImages evolve
   --engine-properties engine.properties
   --input-image monalisa.png
   --output-dir evolving-images
   --generations 10000
   --image-generation 100
```

Every command line argument has proper default values, so that it is possible
to start it without parameters. Listing 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
```

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
```

5.7 Symbolic regression

The following example shows how to set up and solve a symbolic regression
problem with the help of GP and Jenetics. The data set used for the example
was created with the polynomial $4x^3 - 3x^2 + x$. This allows us to check
the quality of the function found by the GP. Setting up a GP requires a little bit
more effort then the setup of a GA. First, you have to define the set of atomic mathematical operations, the GP is working with. These operations influence the search space and is a kind of *a priori* knowledge put into the GP. As a second step you have to define the terminal operations. Terminals are either constants or variables. The number of variables defines the domain dimension of the fitness function.

```java
import static io.jenetics.util.RandomRegistry.random;
import java.util.List;
import io.jenetics.Mutator;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionResult;
import io.jenetics.engine.Limits;
import io.jenetics.util.ISeq;
import io.jenetics.extSingleNodeCrossover;
import io.jenetics.ext.util.TreeNode;
import io.jenetics.prog.ProgramGene;
import io.jenetics.prog.op.EphemeralConst;
import io.jenetics.prog.op.MathExpr;
import io.jenetics.prog.op.MathOp;
import io.jenetics.prog.op.Op;
import io.jenetics.prog.op.Var;
import io.jenetics.prog.regression.Error;
import io.jenetics.prog.regression.LossFunction;
import io.jenetics.prog.regression.Regression;
import io.jenetics.prog.regression.Sample;

public class SymbolicRegression {

```

Figure 5.6.2: Evolving *Mona Lisa* images
// Definition of the allowed operations.
private static final ISeq<Op<Double>> OPS =
    ISeq.of(MathOp.ADD, MathOp.SUB, MathOp.MUL);

// Definition of the terminals.
private static final ISeq<Op<Double>> TMS = ISeq.of(
    Var.of("x", 0),
    EphemeralConst.of(() -> (double)random().nextInt(10)));

// Lookup table for {\code 4\times^3 - 3\times^2 + x}
static final List<Sample<Double>> SAMPLES = List.of(
    Sample.ofDouble(-1.0, -8.0000),
    Sample.ofDouble(-0.9, -6.2460),
    Sample.ofDouble(-0.8, -4.7680),
    Sample.ofDouble(-0.7, -3.5420),
    Sample.ofDouble(-0.6, -2.5440),
    Sample.ofDouble(-0.5, -1.7500),
    Sample.ofDouble(-0.4, -1.1360),
    Sample.ofDouble(-0.3, -0.6780),
    Sample.ofDouble(-0.2, -0.3520),
    Sample.ofDouble(-0.1, -0.1340),
    Sample.ofDouble(0.0, 0.0000),
    Sample.ofDouble(0.1, 0.0740),
    Sample.ofDouble(0.2, 0.1120),
    Sample.ofDouble(0.3, 0.1380),
    Sample.ofDouble(0.4, 0.1760),
    Sample.ofDouble(0.5, 0.2500),
    Sample.ofDouble(0.6, 0.3840),
    Sample.ofDouble(0.7, 0.6020),
    Sample.ofDouble(0.8, 0.9280),
    Sample.ofDouble(0.9, 1.3860),
    Sample.ofDouble(1.0, 2.0000)
);

static final Regression<Double> REGRESSION =
    Regression.of(
        Regression.codecOf(
            OPS, TMS, 5,
            t -> t.gene().size() < 30
        ),
        Error.of(LossFunction::mse),
        SAMPLES
    );

public static void main(final String[] args) {
    final Engine<ProgramGene<Double>, Double> engine = Engine
        .builder(REGRESSION)
        .minimizing()
        .alterers(
            new SingleNodeCrossover<>(0.1),
            new Mutator<>())
        .build();

    final EvolutionResult<ProgramGene<Double>, Double> er =
        engine.stream()
            .limit(Limits.byFitnessThreshold(0.01))
            .collect(EvolutionResult.toBestEvolutionResult());

    final ProgramGene<Double> program = er.bestPhenotype()
        .genotype();
The error function uses the mean squared error loss function and no additional tree complexity metric. One output of a GP run is shown in figure 5.7.1. If we simplify this program tree, we will get exactly the polynomial which created the sample data.

![Symbolic regression polynomial](image.png)

**Figure 5.7.1: Symbolic regression polynomial**

### 5.8 Grammar based regression

The same example as in given in section 5.7 on page 142 can be done with grammatical evolution. How to do this is shown in the following example.

---

7. https://en.wikipedia.org/wiki/Mean_squared_error
8. See section 3.1.8 on page 100
public class GrammarBasedRegression implements Problem<Tree<Op<Double>, ?>, IntegerGene, Double> {
    private static final Cfg<String> GRAMMAR = Bnf.parse(""
        <expr> ::= x | <num> | <expr> <op> <expr>
        <op> ::= + | − | ∗ | /
        <num> ::= 2 | 3 | 4
"";

    private static final Codec<Tree<Op<Double>, ?>, IntegerGene> CODEC = Mappers.multiIntegerChromosomeMapper(
        GRAMMAR,
        // The length of the chromosome is 25 times the length
        // of the alternatives of a given rule. Every rule
        // gets its own chromosome. It would also be possible
        // to define variable chromosome length with the
        // returned integer range.
        rule -> IntRange.of(rule.alternatives().size() * 25),
        // The used generator defines the generated data type,
        // which is `List<Terminal<String>>`.
        index -> new SentenceGenerator<>(index, 50)
    )
        // Map the type of the codec from `List<Terminal<String>>`
        // to `String`
        .map(s -> s.stream().map(Symbol::name).collect(joining()))
}
```java
// Map the type of the codec from 'String' to
// 'Tree<Op<Double>, ?>'
.map(e -> e.isEmpty() ? TreeNode.of(Const.of(0.0)) : parseTree(e));

private static final Error<Double> ERROR = Error.of(LossFunction::mse);

private final Sampling<Double> _sampling;

public GrammarBasedRegression(Sampling<Double> sampling) {
    _sampling = requireNonNull(sampling);
}

public GrammarBasedRegression(List<Sample<Double>> samples) {
    this(Sampling.of(samples));
}

@Override
public Codec<Tree<Op<Double>, ?>, IntegerGene> codec() {
    return CODEC;
}

@Override
public Function<Tree<Op<Double>, ?>, Double> fitness() {
    return program -> {
        final Result<Double> result = _sampling.eval(program);
        return ERROR.apply(program, result.calculated(), result.expected());
    };
}

public static void main(final String[] args) {
    final var regression = new GrammarBasedRegression(
        SymbolicRegression.SAMPLES
    );

    final Engine<IntegerGene, Double> engine = Engine
        .builder(regression)
        .alterers(
            new SwapMutator<>(),
            new SinglePointCrossover<>())
        .minimizing()
        .build();

    final EvolutionResult<IntegerGene, Double> result = engine
        .stream()
        .limit(Limits.byFitnessThreshold(0.05))
        .collect(EvolutionResult.toBestEvolutionResult());

    final Phenotype<IntegerGene, Double> best = result.bestPhenotype();

    final Tree<Op<Double>, ?> program = regression.decode(best.genotype());

    System.out.println("Generations: " + result.totalGenerations());
    System.out.println("Function: " + new MathExpr(program).simplify());
```
5.9. DTLZ1

Deb, Thiele, Laumanns and Zitzler have proposed a set of generational MOPs for testing and comparing MOEAs. This suite of benchmarks attempts to define generic MOEA test problems that are scalable to a user defined number of objectives. Because of the last names of its creators, this test suite is known as DTLZ (Deb-Thiele-Laumanns-Zitzler).

DTLZ1 is an $M$-objective problem with linear Pareto-optimal front:

\[
\begin{align*}
  f_1(x) &= \frac{1}{2}x_1x_2 \cdots x_{M-1} (1 + g(x_M)), \\
  f_2(x) &= \frac{1}{2}x_1x_2 \cdots (1 - x_{M-1}) (1 + g(x_M)), \\
  &\vdots \\
  f_{M-1}(x) &= \frac{1}{2}x_1 (1 - x_2) (1 + g(x_M)), \\
  f_M(x) &= \frac{1}{2} (1 - x_1) (1 + g(x_M)),
\end{align*}
\]

$\forall i \in [1,..n] : 0 \leq x_i \leq 1$

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.DoubleRange;
import io.jenetics.util.ISeq;
import io.jenetics.util.IntRange;
import io.jenetics.ext.SimulatedBinaryCrossover;
import io.jenetics.ext.moea.MOEA;
```
The listing above shows the encoding of the DTLZ1 problem with the Jenetics library. Figure 5.9.1 on the next page shows the Pareto-optimal front of the DTLZ1 optimization.
Figure 5.9.1: 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\(^1\) as build system and organizes the source into sub-projects (modules).\(^2\) Each sub-project is located in its own sub-directory.

Published projects

- **jenetics**: This project contains the source code and tests for the Jenetics base-module.

- **jenetics.ext**: This module contains additional non-standard GA operations and data types. It also contains classes for solving multi-objective problems (MOEA). Additional classes for defining tree rewrite systems are also part of this module.

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

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

\(^1\)[http://gradle.org/downloads]

\(^2\)If you are calling the `gradlew` script (instead of gradle), which are part of the downloaded package, the proper Gradle version is automatically downloaded and you don’t have to install Gradle explicitly.
• **prngine**: PRNGine is a pseudo-random number generator library for sequential and parallel Monte Carlo simulations. Since this library has no dependencies to one of the other projects, it has its own repository with independent versioning.

**Non-published projects**

- **jenetics.example**: This project contains example code for the *base*-module.
- **jenetics.incubator**: This project contains experimental code which might find its way into one of the *official* modules.
- **jenetics.doc**: Contains the code of the website and this manual.
- **jenetics.tool**: This module contains classes used for doing integration testing and algorithmic performance testing. It is also used for creating GA performance measures and creating diagrams from the performance measures.

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

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

For building the library from the source, call

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

or

```
$ ./gradlew jar
```

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

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

[https://github.com/jenetics/prngine](https://github.com/jenetics/prngine)
pom.xml snippet for Maven

```xml
<dependency>
    <groupId>io.jenetics</groupId>
    <artifactId>module-name</artifactId>
    <version>7.2.0</version>
</dependency>
```

Gradle

`'io.jenetics:module-name:7.2.0'`

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

Copyright 2007-2023 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, 135
2-point crossover, 19
3-point crossover, 19

Allele, 6, 41
Alterer, 16, 45
AnyChromosome, 43
AnyGene, 42
Architecture, 4
Assignment problem, 59

Backus-Naur form, 101
Base classes, 5
BigIntegerGene, 89
Block splitting, 35
BNF, 101
Boltzmann selector, 15
Build, 151
Gradle, 151
gradlew, 151

CFG, 101
Chromosome, 7, 8, 42
recombination, 18
scalar, 48
variable length, 7

Clock, 24
Code, 54
Composite, 60
Mapping, 59
Matrix, 56
Permutation, 58
Scalar, 55
Subset, 56
Vector, 55
Combine alterer, 20
Compile, 152
Complexity function, 115
Composite codec, 60
ConcatEngine, 93
Concurrency, 31

configuration, 92
maxBatchSize, 33
maxSurplusQueuedTaskCount, 33
splitThreshold, 32
tweaks, 92

Constraint, 24, 62
Context-free grammar, 101
Crossover
2-point crossover, 19
3-point crossover, 19
Intermediate crossover, 21
Line crossover, 21
Multiple-point crossover, 19
Partially-matched crossover, 20
Simulated binary crossover, 89
Single-point crossover, 18
Uniform crossover, 20

CyclicEngine, 94

Dieharder, 125
Directed graph, 53
Distinct population, 79
Domain classes, 6
Domain model, 6
Download, 151
DTLZ1, 148

Elite selector, 15, 45
Elitism, 15, 45

Encoding, 47
Affine transformation, 50
Directed graph, 53
Graph, 52
Real function, 18
Scalar function, 49
Undirected graph, 52
Vector function, 49
Weighted graph, 53

Engine, 23, 46
Evaluator, 30
<table>
<thead>
<tr>
<th>Location</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.jenetics.ext</td>
<td>82</td>
</tr>
<tr>
<td>io.jenetics.prngine</td>
<td>120</td>
</tr>
<tr>
<td>io.jenetics.prog</td>
<td>108</td>
</tr>
<tr>
<td>io.jenetics.xml</td>
<td>117</td>
</tr>
<tr>
<td>Mona Lisa</td>
<td>143</td>
</tr>
<tr>
<td>Monte Carlo selector</td>
<td>13, 76</td>
</tr>
<tr>
<td>MOO</td>
<td>94</td>
</tr>
<tr>
<td>Mixed optimization</td>
<td>99</td>
</tr>
<tr>
<td>MSE</td>
<td>114</td>
</tr>
<tr>
<td>Multi-objective optimization</td>
<td>94</td>
</tr>
<tr>
<td>Mixed optimization</td>
<td>99</td>
</tr>
<tr>
<td>Multiple-point crossover</td>
<td>19</td>
</tr>
<tr>
<td>Mutation</td>
<td>16</td>
</tr>
<tr>
<td>Mutator</td>
<td>16</td>
</tr>
<tr>
<td>NSGA2 selector</td>
<td>98</td>
</tr>
<tr>
<td>Ones counting</td>
<td>129</td>
</tr>
<tr>
<td>Operation classes</td>
<td>12</td>
</tr>
<tr>
<td>Package structure</td>
<td>5</td>
</tr>
<tr>
<td>Parentheses tree</td>
<td>59</td>
</tr>
<tr>
<td>Pareto dominance</td>
<td>93</td>
</tr>
<tr>
<td>Pareto efficiency</td>
<td>95</td>
</tr>
<tr>
<td>Partial alterer</td>
<td>22</td>
</tr>
<tr>
<td>Partially-matched crossover</td>
<td>19</td>
</tr>
<tr>
<td>Permutation codec</td>
<td>58</td>
</tr>
<tr>
<td>Phenotype</td>
<td>11</td>
</tr>
<tr>
<td>Population</td>
<td>6, 11</td>
</tr>
<tr>
<td>Population convergence</td>
<td>74</td>
</tr>
<tr>
<td>PRNG</td>
<td>34</td>
</tr>
<tr>
<td>Block splitting</td>
<td>35</td>
</tr>
<tr>
<td>LCG64ShiftRandom</td>
<td>36</td>
</tr>
<tr>
<td>Leapfrog</td>
<td>36</td>
</tr>
<tr>
<td>Parameterization</td>
<td>35</td>
</tr>
<tr>
<td>Random seeding</td>
<td>33</td>
</tr>
<tr>
<td>PRNG testing</td>
<td>125</td>
</tr>
<tr>
<td>Probability selector</td>
<td>13</td>
</tr>
<tr>
<td>Problem</td>
<td>62</td>
</tr>
<tr>
<td>Quantile</td>
<td>40</td>
</tr>
<tr>
<td>Random</td>
<td>34</td>
</tr>
<tr>
<td>Generator</td>
<td>34</td>
</tr>
<tr>
<td>LCG64ShiftRandom</td>
<td>36</td>
</tr>
<tr>
<td>Registry</td>
<td>33</td>
</tr>
<tr>
<td>seeding</td>
<td>126</td>
</tr>
<tr>
<td>testing</td>
<td>125</td>
</tr>
<tr>
<td>Random seeding</td>
<td>35</td>
</tr>
<tr>
<td>RandomGenerator</td>
<td>54</td>
</tr>
<tr>
<td>Randomness</td>
<td>34</td>
</tr>
<tr>
<td>Rastrigin function</td>
<td>133</td>
</tr>
<tr>
<td>Real function</td>
<td>131</td>
</tr>
<tr>
<td>Recombination</td>
<td>17</td>
</tr>
<tr>
<td>Regression problem</td>
<td>117</td>
</tr>
<tr>
<td>Reproducibility</td>
<td>76</td>
</tr>
<tr>
<td>Rewrite rule</td>
<td>86</td>
</tr>
<tr>
<td>Rewrite system</td>
<td>86</td>
</tr>
<tr>
<td>Rewriting</td>
<td>86</td>
</tr>
<tr>
<td>Roulette-wheel selector</td>
<td>14</td>
</tr>
<tr>
<td>Sample point</td>
<td>117</td>
</tr>
<tr>
<td>SBX</td>
<td>89</td>
</tr>
<tr>
<td>Scalar chromosome</td>
<td>48</td>
</tr>
<tr>
<td>Scalar codec</td>
<td>55</td>
</tr>
<tr>
<td>Scalar genotype</td>
<td>48</td>
</tr>
<tr>
<td>Seeding</td>
<td>126</td>
</tr>
<tr>
<td>Selector</td>
<td>12, 44</td>
</tr>
<tr>
<td>Elite</td>
<td>45</td>
</tr>
<tr>
<td>Seq</td>
<td>38</td>
</tr>
<tr>
<td>Serialization</td>
<td>37</td>
</tr>
<tr>
<td>Simulated binary crossover</td>
<td>89</td>
</tr>
<tr>
<td>Single-node crossover</td>
<td>89</td>
</tr>
<tr>
<td>Single-point crossover</td>
<td>18</td>
</tr>
<tr>
<td>Source code</td>
<td>151</td>
</tr>
<tr>
<td>Statistics</td>
<td>49</td>
</tr>
<tr>
<td>Steady fitness</td>
<td>69</td>
</tr>
<tr>
<td>Stochastic-universal selector</td>
<td>15</td>
</tr>
<tr>
<td>Subset codec</td>
<td>56</td>
</tr>
<tr>
<td>Swap mutator</td>
<td>17</td>
</tr>
<tr>
<td>Symbolic regression</td>
<td>114</td>
</tr>
<tr>
<td>SymbolIndex</td>
<td>106</td>
</tr>
<tr>
<td>Term rewrite rule</td>
<td>88</td>
</tr>
<tr>
<td>Term rewrite system</td>
<td>88</td>
</tr>
<tr>
<td>Termination</td>
<td>67, 99</td>
</tr>
<tr>
<td>Evolution time</td>
<td>70</td>
</tr>
<tr>
<td>Fitness convergence</td>
<td>72</td>
</tr>
<tr>
<td>Fitness threshold</td>
<td>71</td>
</tr>
<tr>
<td>Fixed generation</td>
<td>68</td>
</tr>
<tr>
<td>Gene convergence</td>
<td>73</td>
</tr>
<tr>
<td>Population convergence</td>
<td>74</td>
</tr>
<tr>
<td>Steady fitness</td>
<td>69</td>
</tr>
<tr>
<td>Tournament selector</td>
<td>12</td>
</tr>
<tr>
<td>Traveling salesman</td>
<td>138</td>
</tr>
<tr>
<td>Tree</td>
<td>82</td>
</tr>
<tr>
<td>pattern</td>
<td>87</td>
</tr>
<tr>
<td>reduce</td>
<td>86</td>
</tr>
<tr>
<td>rewrite rule</td>
<td>86</td>
</tr>
<tr>
<td>rewrite system</td>
<td>86</td>
</tr>
<tr>
<td>Tree pattern</td>
<td>87</td>
</tr>
</tbody>
</table>
Tree rewrite rule, 86, 88
Tree rewrite system, 86, 88
Tree rewriter, 88
TreeGene, 89
Truncation selector, 13

Undirected graph, 52
Uniform crossover, 20
Unique fitness tournament selector, 99
Unique population, 79

Validation, 7
Vec, 96
VecFactory, 99
Vector codec, 55

Weasel program, 90
WeaselMutator, 91
WeaselSelector, 91
Weighted graph, 59