EvoJ Technical Guide

Release 3.0

Contents

1 Framework overview

The EvoJ is a Java framework which enables programmers and scientists to easily solve problems using Genetic Algorithm. The main feature of the framework is that you don't need to construct chromosomes manually – all you need to do is to declare a Java interface holding all the variables you need to solve your problems.

The key features of the framework are:

- simplicity – the EvoJ itself requires just a few lines of code allowing you to concentrate on the problem you solve
- no external dependencies – all you need is evoj-3.0.jar
- extensibility – you can adjust any stage of evolutionary algorithm
- native support for Artificial Neural networks
- native multithreading support for better performance
- declarative mutation control – use java annotations or property-like files to control how your variables mutate

2 Core principles

2.1 Automatic chromosome mapping

When you want to solve a problem using Genetic Algorithm you usually have some set of variables in mind. The genetic approach requires from you to organize these variables into a linear chromosome and then perform a set of evolutionary operations over such chromosomes.

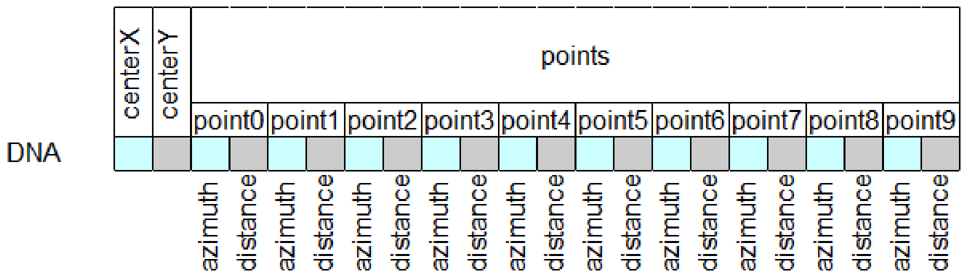
To simplify the mapping process and to allow programmers to conveniently use the capabilities of modern Java IDEs the EvoJ can treat Java interfaces as chromosomes.

Suppose we are solving a problem of finding an optimal polygon in polar coordinates, let's forget about a fitness function for now. In this case the variables we would need to describe the solution can be expressed as following Java interfaces.

```
public interface Polygon {

    float getCenterX();

    float getCenterY();

    @ListParams(length = "10")
    List<PolarPoint> getPoints();
}

public interface PolarPoint {

    float getAzimuth();

    float getRadius();
}
```

As you might see the polygon is described naturally as variables `centerX` and `centerY` and a `List` of 10 `PolarPoints`. This allows to calculate a fitness function for the solution in a natural way and facilitates easy refactoring and integrity control using IDE.

The EvoJ scans the interface and allocates an array of `Object` to hold all the variables. The array is then used as a chromosome in crossingover and mutation operations. The example above would automatically be maped into following array



However you will not see the chromosomes and genes instead of that you will be presented these interfaces each time the EvoJ will need an input from you, for example when you will be calculating the fitness-function of a solution.

2.2 Supported variable types

You can declare any number of properties in an interface. The properties may be of primitive, object types and Lists. The rules of declaring solution interfaces are:

1. The method names should start with “get” or “set” even if property is `boolean`
2. Getters should have no parameters and do not return void
3. Setters should have one parameter having exactly the same type as returned by complement getter method
4. There should be no setter methods without corresponding getter methods
5. It is allowed to declare getter methods without complement setter methods
6. Properties can be:
   a) any primitive type or corresponding object type (will be referred to as “simple type”)
   b) enum class. Enum properties are treated as simple types
   c) Interface conforming to these rules (will be referred to as “complex type”)
   d) List of simple or complex type
7. No setter methods for complex types or lists are allowed
8. List properties must have their length specified by `@ListParams` annotation or by context properties

2.2.1 Simple properties

Properties which getter method return enums, primitive types or their corresponding object types are referred to as “Simple properties”.

Invocation of getter method of simple property causes reading from DNA represented by

`Object` array containing solution data to be performed. This is not that fast operation, so, it is recommended to cache values read from simple properties.

Calling setter of simple property causes write to underlying Object array.

Simple properties may have their range specified using `@Range` annotation or via `@Min` and `@Max` detached annotations. If range is set without strict option, then it only affects the creation of solution-interface – all simple properties will be initiated with random values lying within specified range. However, due to mutation or direct invocation of setter method value of simple property may exceed specified range.

Specifying that range is strict causes EvoJ to check returned and stored values to be within specified range. If stored value is for example higher than the upper value limit, then upper limit value will be actually stored.

2.2.2 Interface properties

Declared properties may return interfaces conforming to above mentioned rules as well. Invocation of getter of interface property returns proxy object implementing that interface. Though declaration of setters for interface properties is allowed, invocation of them will cause exception. If you want to change state of interface property – call setter methods of it's simple properties.

2.2.3 List properties

List properties must have their element type to be specified as generic parameter. List may contain object representation of primitive types, interfaces, nested lists. These are examples of legal list property declarations:

```
List<Point> getPoints();
List<Integer> getValues();
List<List<Boolean>> getMatrix();
```

Getter of list property returns proxy list. Setters for list properties are allowed to declare but not allowed to invoke. Just like for interface properties. Similar rule is applied to get(int index) and set(int index, Object value) of List interface: it's allowed to invoke set only for lists of simple type values.

List properties must have their length specified using by `@ListParams` annotation or by detached annotations. Length of nested list can only be specified by detached annotation.

Adding and removing of elements of list property is not allowed and will cause `UnsupportedOperationException`.

2.3 Minimal application

To demonstrate typical steps of solving of problems using EvoJ let's find a minimum of the following function:

$$Z(x,y) = 12 \cdot x^2 + 8 \cdot x + 9 \cdot y^2$$

The solution to our problem is a pair of values of variables x and y. Let's declare a Java interface holding our variables:

```
public interface Solution {

    @Range(min = "-10", max = "10")
    @MutationRange("0.2")
    double getX();

    @Range(min = "-10", max = "10")
    @MutationRange("0.2")
    double getY();
}
```

As you can see, the interface contains two getter methods – one for each variable. The methods are marked with annotations describing ranges of the corresponding variables and as well as the range in which a variable can change in a single act of mutation. The annotations will be discussed deeper in a dedicated chapter.

Next thing we need to do is to give EvoJ an instrument to estimate a fitness of a solution represented by the two above mentioned variables. We are going to do this by extending the `net.sourceforge.evoj.strategies.sorting.AbstractSimpleRating` class.

```
public class Rating extends AbstractSimpleRating<Solution, Double> {

    public static double calcFunction(Solution solution) {
        double x = solution.getX();
        double y = solution.getY();
        return 12 * x * x + 8 * x + 9 * y * y;
    }

    @Override
    public Double doCalcRating(Solution solution) {
        double fn = calcFunction(solution);
        if (Double.isNaN(fn)) {
            return null;
        } else {
            return -fn;
        }
    }
}
```

The abstract class we have just implemented has one method `doCalcRating` which takes an instance of the interface holding the problem-related variables as an argument and returns a `Comparable` value (`Double` in this case) as the result. The better the fitness of the solution – the higher the returned value should be.

As we are searching for the minimum of the function, the higher fitness will have the pairs of the variables which cause the function to evaluate to smaller values. Thus it seems logical to return negative value of the function as a solution rating.

In the corner case where function evaluates to `NaN` we return `null` which is treated by EvoJ as the smallest possible value of the solution rating, smaller than any non-null values.

As you might have noticed we operate the domain variables all the way, we do not deal with chromosomes and genes which are hidden by EvoJ.

The concept of rating as well as other ways to determine the fitness of the solution will be discussed in a corresponding chapter.

Alright, now we have everything to make it work. Let's create a class having it's main method implemented as follows:

```
DefaultPoolFactory pf = new DefaultPoolFactory();

GenePool<Solution> pool = pf.createPool(200, Solution.class, null);

Rating rating = new Rating();

DefaultHandler handler = new DefaultHandler(rating);

handler.iterate(pool, 300);

double val = Rating.calcFunction(pool.getBestSolution());

System.out.println("Detected minimum value: " + val);
```

Let's explain what is happening here. First we use the `DefaultPoolFactory` class to create a population of 200 random solutions stored in the instance of the `GenePool` class.

Then we create an instance of our Rating class. Which is then given as a parameter in the constructor of the `DefaultHandler` class. The Handlers in EvoJ embody the genetic algorithm itself, they perform all the evolutionary operations over the `GenePools`.

After creating the `DefaultHandler` we immediately perform 300 iterations of the genetic algorithm, which should be enough to find a solution to our simple problem.

Then we get a best found solution from the `GenePool` and print it.

In many cases your code will look like that except for the `Solution` interface and `Rating` class, cause these are always specific to the problem you want to solve.

3 Basic classes

3.1 Factories, Individuals, GenePools

The population of solutions in EvoJ is represented by the `GenePool` class which holds a list instances of objects implementing the solution specific interface. At the same time these objects implement the Individual interface which is used by strategy classes to access the low level representation of the solution – the DNA, `rating` and `age` properties.

The `GenePool` is virtually split into elite and non-elite parts. The elite part is the first individuals in the list. The number of the elite individuals is determined by the `eliteCount` property of the `GenePool`. There must be at least one elite individual. The elite individuals are then crossbred to produce the next generation of the solutions, the new born individuals replace the non-elite part of the `GenePool`.

In fact no new `Individual` objects are spawned, instead of that the DNA of the existing ones gets rewritten, the ages gets reset to 0 and the rating to null. This is done to improve performance and reduce the number of spawned object at each iteration since the number of iterations required to solve a problem is usually counted in thousands or even millions.

Since solutions in EvoJ are represented as Java interfaces we cannot instantiate them directly. For these purposes there exist `PoolFactories` and `Individual` factories. The former use the latter to fill the instances of `GenePool` with Individuals.

There is only one class implementing the `PoolFactory` interface – `DefaultPoolFactory` which should be normally used by programmers to instantiate the `GenePools`.

One does not need to deal with the `IndividualFactories` when solving conventional variable based problems. But when it comes to the Neural Networks one must use the `NeuralNetworkFactory` class which is described in the chapter dedicated to the neural networks.

3.2 Handlers

Handlers are the incarnation of the genetic algorithm itself. The `Handler` interface is defined as:

```
public interface PoolHandler<T> {

    void iterate(GenePool<T> genePool);

    void iterate(GenePool<T> genePool, int count);

    boolean iterate(GenePool<T> genePool, SuccessCondition success, long
maxIter);
}
```

The interface contains methods to perform a single iteration, a desired number of iterations and to iterate until some criteria defined by the implementation of the `SuccessCondition` interface is met or a defined number of iterations reached.

A typical iteration performed by Handlers consists of following steps:

1. Check that the given `GenePool` is sorted according to fitness-function (the best solutions come first. If not – calculate fitness and sort.
2. Produce new generation of solutions in the non-elite part of the `GenePool` using elite Individuals as parents
3. Mutate some of the new-born individuals
4. Increment age of the elite individuals
5. Sort the `GenePool` according to solution fitness

As you can see, after the iteration the `GenePool` is always sorted.

There are two classes which implement the Handler interface: `DefaultHandler` and `MultithreadedHandler`. These two do very same things in terms of genetic algorithm, the only difference is that the latter uses advantages of modern multicore processors and calculates things in a desired number of threads. The multithreading matters are discussed in the corresponding chapter.

Every operation of the genetic algorithm performed by the handlers is hidden behind the corresponding strategy interface. There exist default strategies for selection, DNA recombination and mutation, which can be customized by programmers by re-implementing of the corresponding interfaces. However this is not mandatory.

The only strategy programmer must implement is the `PoolSorter` interface, since this is specific to the problem being solved. There exist helper-classes like `AbstractSimpleRating` which simplify implementation of this strategy.

The startegies must be given as parameters in the constructors of the handlers. If you would like to use a default version of a strategy then just give null as the corresponding parameter.

3.3 Strategies

3.3.1 Fitness

In the EvoJ the fitness of the individual is represented by the position it takes in the `GenePool`. The closer to the beginning of the pool – the higher fitness has an individual.

Handlers use the `PoolSorter` interface whenever they need individuals in a `GenePool` to be ordered by their fitness.

Then the position of the individual is used by the implementations of the `SelectionStrategy` to choose parents for the individual of the next generation.

Despite the `PoolSorter` interface is the only mandatory interface to implement by a programmer there is no need to implement it directly. There is a set of helper-classes which simplify this process in common cases.

There exist two general approaches to determining of the fitness of the solution:

- rating based - when you can think of a function which defines a fitness of a solution as a `Comparable` value
- comparison based – when it's hard to express a fitness of a solution as a number (or any other `Comparable` value), but it's easy to compare two solutions together and say which of the two is better.

3.3.1.1 Rating based

The main helper-class for implementation of the rating based fitness-function is the `AbstractSimpleRating` class, which you have seen in the sample program. This class is abstract and requires implementation of the `doCalcRating` method:

```
protected abstract E doCalcRating(T solution);
```

The implementation of this method should estimate a given solution and return a `Comparable` value. The contract requires that the better a solution is – the greater the returned value should be.

In most cases `Double` or `Long` numbers can be used as rating values. However if you perform multifactor estimation of a solution them you might want to create your own implementation of the `Comparable` interface which suits your needs better.

One nice feature of the `AbstractSimpleRating` class is that it can help you to get rid of too old solutions among the elite. Having too many old solutions may lead in some cases to convergence to local extremum instead of global one.

You can specify the maximal number of iterations an individual can survive by setting of maxAge property of the AbstractSimpleRating class. Setting it to 0 will cause the elite to be completely replaced by the newborn individuals. The old individuals are automatically given a null-rating which will cause them to slide down to the bottom of the gene pool and be replaced at the next iteration.

You can give the instance of your implementation of the AbstractSimpleRating class as a parameter in the constructors of the DefaultHandler and the MultithreadedHandler classes.

3.3.1.2 Comparison based

If you can't find a function to express a fitness of an individual as a Comparable value, you can simply implement the Comparator interface according to general contract of the Comparator class to compare two individuals.

You can give the instance of your implementation of you custom implementation of the Comparator class as a parameter in the constructors of the DefaultHandler and the MultithreadedHandler classes.

3.3.2 Mutation

The second important strategy you may want to adjust is the MutationStrategy. The interface is defined as

```
public interface MutationStrategy {
    void mutate(Individual i);
}
```

This strategy is applied to every newborn individual. It is up to implementer to decide which elements to mutate, by which amount, etc.

You can cast the given individual to your solution-interface and perform a mutation in a domain specific way.

Usually there is no need to implement this interface as the most common use cases are covered by the DefaultMutationStrategy class. Which performs mutations according to the mutation configuration specified by the annotations and context properties. More than that, you can specify how many newborn individuals to mutate, and how many elements of the DNA to mutate through the constructor parameters.

The instance of the DefaultMutationStrategy is created as follows:

```
DefaultMutationStrategy m = new DefaultMutationStrategy(0.1, 0.3, 0.8);
```

The first parameter of the constructor specifies the probability that the new born individual will mutate at all. The second parameter specifies how many mutable variables will be affected by mutation, if a number less than 1 is given, then it is treated as fraction of all mutable elements. The last parameter specifies the probability that instead of mutating according to `@MutationRange` annotation the chosen variable will be given a random value within it's Range, i.e. the value will be reset.

3.3.3 Selection

The purpose of the `SelectionStrategy` interface is to chose a pair of parents to be used to produce a newborn individual. A new pair is chosen for every newborn individual.

The interface is declared as follows:

```
public interface SelectionStrategy {
    void performSelection(List<Individual> pool, int[] pair);
}
```

Here the pool parameter is the elite part of the gene pool being processed. The implementation must pick two individuals and specify their indices in the pair array.

The selection strategy is no used if `eliteCount` of the `GenePool` is set to 1 or 2, cause in these cases the parents are obvious.

There are two built-in selection strategies: `CompetitiveSelectionStrategy` and `EqualChanceSelectionStrategy`. The `CompetitiveSelectionStrategy` picks individuals with the respect to their place in the pool, the better ones are chosen more frequently. This strategy is the default one.

The `EqualChanceSelectionStrategy` choses parents with equal probability disregarding their position in the pool.

3.3.4 Replication

The `ReplicationStrategy` interface is declared as:

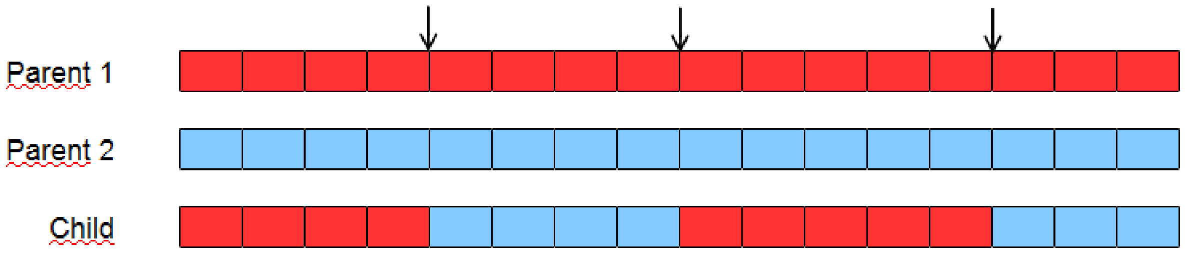
```
public interface ReplicationStrategy {
    void setChildDNA(Individual child, Individual parent1, Individual 
parent2);
}
```

The contract of the interface is to set the child DNA by intermixing the DNAs of the parent1 and the parent2 Individuals.

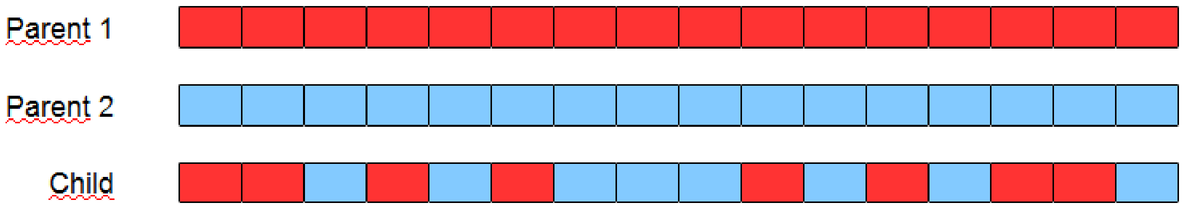
The built-in implementations of the strategy are the `MultisplitCrossingover` and the `RandomInterleaveCrossingover`.

The `MultisplitCrossingover` splits the DNA into sections at desired number of random sptlit-points (given through a constructor parameter) and produces the new DNA by interleaving sections of DNA from both parents.

The image below demonstrates how the `MultisplitCrossingover` class works. The arrows denote the randomly chosen split-points.



The `RandomInterleaveCrossingover` simply goes through the elements of the DNA and randomly decides from which parent to take the corresponding element. The scheme is given below:



The default replication strategy is the `MultisplitCrossingover` with a single split point.

There is rarely a need to implement a custom replication strategy.

4 Declarative configuration

EvoJ gives you the way to flexibly manage the process of evolution. You can specfy

- ranges of variable values
- the amount of mutation a variable receives in each act of mutation
- lengths of lists (this one you MUST do)
- relative probabilities of mutations of variable (in other words you can make the variable X mutate twice more frequent than the variable Y).

4.1 Annotations

Using annotations is the basic way to configure EvoJ just in place. The annotations must be always applied to getter methods.

The annotations may have parameters which are always strings. This is done to enable use of context properties which are described in a corresponding chapter.

4.1.1 @Range

The Range annotation is used to specify the range of values a variable can have. The default implementation of the `IndividualFactory` interface use this annotation to set initial values of the variables.

The annotation is used as follows

```
@Range(min = "-10", max = "6", strict = "true")
short getAge();

@Range(min = "-10", max = "6")
byte getUnrestrictedAge();
```

If the option `strict=”true”` is specified then the corresponding variable can never exceed the specified range, even you will try to set it using setter method. If the value being set exceeds the specified strict range then the nearest boundary value will be actually set.

The values of the min and max parameters must be valid string representations of the return type of the getter method, or the reference to an existing context property which has a valid value. The Range annotation works for enums as well.

4.1.2 @MutationRange

The `MutationRange` annotation specifies how far from its current value a variable can mutate. This annotation is used by the `DefaultMutationStrategy`.

The examples of annotation usage can be seen below:

```
@MutationRange("12.2")
double getX();

@Range(min = "-10", max = "10")
@MutationRange(value="20%" gaussian=”false”)
double getY();
```

There are two ways to specify the radius in which a value can fluctuate: absolute and relative. In the example the variable X can mutate within +/- 12.2 from the value it had before mutation. For example if the variable X=3, then after the mutation it's value will be somewhere from -9.2 to 15.2.

This is an example of absolute mutation range.

The example of relative mutation is the Y variable. As stated by the annotation it can mutate within 20% of it's range. Since the Range has the size of 10-(-10)=20, then the 20% of 20 is 4. Hence the Y variable can mutate within 4 units.

By default the mutations have gaussian distribution – the larger mutations happen less frequently than the smaller ones. You can enforce the uniform distribution by setting gaussian property to false, as shown in the example above for the Y variable

4.1.3 @MutationAffinity

The `MutationAffinity` annotation is an instrument to control the relative probabilities of mutations of variables. The `DefaultMutationStrategy` analyzes the mutation affinities of all the variables – simple and complex ones, a chooses which variable to mutate according to this analysis.

By default all simple variables have mutation affinity of 1, which causes them to be chosen for mutation with equal probability. All the complex variables like lists and interfaces have default mutation affinity of 0, this prevents them from mutating as the whole.

The mutation affinity is not inherited, in other words the fact that the parent complex variable has mutation affinity of 0 does not prevent the child variables from mutation.

To help you understand the concept of mutation affinity let's analyze some examples.

```
public interface Point {

    float getX();

    float getY();
}
```

Here the both variables have default affinity of 1, thus they will mutate with equal probability.

```
public interface Point {

    @MutationAffinity("2")
    float getX();

    float getY();
}
```

In this case the X variable has the mutation affinity 2, thus it will mutate twice more frequent than Y. This means that the probability the X variable will mutate is 2/3=0.66.

Let's get back to the example from the second chapter:

```
public interface Polygon {

    float getCenterX();

    float getCenterY();

    @ListParams(length = "10")
    List<PolarPoint> getPoints();
}

public interface PolarPoint {

    float getAzimuth();

    float getRadius();
}
```

The individual built using this model will have 22 simple variables: `centerX`, `centerY` plus 10 polar points having 2 variables each. Thus the probability of any of these variables is 1/22. This also means that with the probability of 20/22=91% the mutation will happen in ONE of the list of the points. To be more precise the mutation will affect either azimuth or radius in one of the points.

Suppose we want to balance the probabilities and make the center coordinates to mutate with the same probability as the complex property points.

```
public interface Polygon {

    @MutationAffinity(“10”)
    float getCenterX();

    @MutationAffinity(“10”)
    float getCenterY();

    @ListParams(length = "10")
    List<PolarPoint> getPoints();
}
```

Now summary affinity of the all mutable variables is 40. The probabilities to mutate are

- centerX – 10/40=25%
- centerY – 10/40=25%
- any polar point – 20/40=50%

Let's have another example of usage of `MutationAffinity` annotation.

```
public interface Polygon {

    @MutationAffinity(“10”)
    float getCenterX();

    @MutationAffinity(“10”)
    float getCenterY();

    @MutationAffinity(“1”)
    @ListParams(length = "10")
    List<PolarPoint> getPoints();
}
```

Here we have put the `MutationAffinity` on a complex variable points. Now this variable has got a chance to mutate as the whole. Let's count the probabilities again. The sum of affinities in the system is 41 now. Thus probabilities to mutate are:

- centerX – 10/41=24.4%
- centerY – 10/41=24.4%
- ANY polar point – 20/41=48.8%
- EVERY element in ALL polar points – 1/41=2.4%

And the last example of `MutationAffinity` usage

```
public interface Polygon {

    float getCenterX();

    float getCenterY();

    @MutationAffinity(“1”)
    @ListParams(length = "10")
    List<PolarPoint> getPoints();
}

public interface PolarPoint {

    @MutationAffinity(“0”)
    float getAzimuth();

    @MutationAffinity(“0”)
    float getRadius();
}
```

Here we have set mutation affinity of 0 to the polar point variables and mutation affinity of 1 to the complex property points. This prevents the components of the polar points to be chosen for mutation, the only option now for the list point to mutate is to mutate all together. Let's calculate the probabilities to mutate, having in mind that sum of affinities in the system is now 3.

- centerX – 1/3=33.3%
- centerY – 1/3=33.3%
- EVERY element in ALL polar points – 1/3=33.3%

4.1.4 @Immutable

The `Immutable` annotation is similar to `MutationAffinity(“0”)` except for it is inherited. That is if you for example put `Immutable` annotation on a `List<Point>` then neither of the child variables will ever mutate.

4.1.5 @ListParams

Since EvoJ need to allocate enough space to hold all the variables it needs to know the lengths of the List properties, this makes this annotation mandatory for the list properties.

Using this annotation you can specify also ranges, mutation ranges and mutation affinities of the simple list elements.

4.2 Context

It is not always convenient to lock the ranges and other parameters at the compile time. This is where the context feature comes to help.

You can specify parameter of any annotation as an arbitrary string representing a property name. Later in the runtime you can set any desired values of these properties.

Let's have an example. Suppose we have an interface:

```
public interface Point {

    @Range(min = "point.x.min", max = "point.x.max")
    double getX();

    @Range(min = "point.y.min", max = "point.y.max")
    float getY();
}
```

Please not that instead of the concrete range values we have used context parameter names. This allows to postpone the exact value determination until runtime.

```
Map<String, String> context = new HashMap<String, String>();
context.put("point.x.min", "-10");
context.put("point.x.max", "10");
context.put("point.y.min", "-12");
context.put("point.y.max", "12");
DefaultPoolFactory dpf = new DefaultPoolFactory();
dpf.createPool(10, Point.class, context);
```

Here we fill map with concrete values of the context parameters and the provide the context map to the pool factory.

Any annotation can be parametrized in this way. The names of context parameters can be chosen arbitrary.

This approach allows to calculate certain parameters of a model in a runtime.

4.3 Detached annotations

The further development of the context properties idea is the idea of detached annotations.

The detached annotations are specially constructed property names which can be mapped to the solution variables using syntax resembling commonly known dot notation.

Detached annotations is the only way to parametrize the nested lists.

Let's rewrite an example from chapter 2.1 without any annotations.

```
public interface Polygon {

    float getCenterX();

    float getCenterY();

    List<PolarPoint> getPoints();
}

public interface PolarPoint {

    float getAzimuth();

    float getRadius();
}
```

Now let's build a property file with detached annotations for the above mentioned interfaces.

```
centerX@Min=-5
centerX@Max=5
centerY@Min=-5
centerY@Max=5
points@Length=10
```

Things look simple so far. Now let's configure the elements of the points list.

```
points.item.azimuth@Min=0
points.item.azimuth@Max=359
points.item.distance@Min=0
points.item.distance@Max=100
```

So we traverse the property tree separating the path elements using dot. Please not that we use item keyword to denote the list element. Now we can put these properties into a context map in either way.

Using of detached annotations is the only way to specify lengths of nested lists. Suppose we have following property:

```
List<List<List<Integer>>> getIntCube()
```

Now we must specify the lengths of all the lists in order to allow EvoJ to calculate the size of DNA to allocate. Although we can specify the length of the external list using `@ListParams` annotation, but the only way to specify the lengths of the nested lists is using of detached annotations. Let's build a property file configuring the above property.

```
intCube@Length=5
intCube.item@Length=5
intCube.item.item@Length=5

intCube.item.item.item@Min=-10
intCube.item.item.item@Max=10
```

The first two lines configure the lengths of all the lists. The last two lines specify the range of the elements of the most nested list.

The full list of supported detached annotation is given below.

Detached annotation	Meaning
`Min`	min property of `@Range` annotation
`Max`	max property of `@Range` annotation
`StrictRange`	strict property of `@Range` annotation
`Immutable`	the same as the `@Immutable` annotation
`MutationAffinity`	the same as the `@MutationAffinity` annotation
`MutationRange`	the same as the `@MutationRange` annotation
`GaussianMutation`	`gaussian` property of `@MutationRange` annotation
`Length`	`length` property of `@ListParams`

You can use a mixture of the Java annotations and detached annotations. However when applied to the same property the detached annotations will prevail.

5 Multithreading

5.1 Multithreaded handlers

The operations of the genetic algorithm can be very effectively parallelized. Here does the `MultithreadedHandler` class come to help.

This class conforms to the general contract of the Handler. Constructors of the `MultithreadedHandler` look alike to those of `DefaultHandler` class, but they have one additional parameter – number of threads.

Let's rewrite the example we have given in the chapter 2.3 using the `MultithreadedHandler`.

```
DefaultPoolFactory pf = new DefaultPoolFactory();

GenePool<Solution> pool = pf.createPool(200, Solution.class, null);

Rating rating = new Rating();

MultithreadedHandler handler = new MultithreadedHandler(2, rating);
try {
    handler.iterate(pool, 300);
} finally {
    handler.shutdown();
}

double val = Rating.calcFunction(pool.getBestSolution());

System.out.println("Detected minimum value: " + val);
```

This code snippet looks almost the same as the original example, except for the need to call the `handler.shutdown()` method when the `MultithreadedHandler` is no more needed. Call to this method terminates all worker threads allowing the application to exit gracefully.

When performing the step of the evolutionary algorithm the `MultithreadedHandler` splits the `GenePool` into possibly equal chunks depending on the number of the worker threads and processes each chunk parallelly. Thus when dealing with the `MultithreadedHandler` you must carefully implement any strategies you'd like to override, this especially concerns the way you calculate the fitness of the individuals.

Using more threads than the number of physical cores your processor has (this does not count HyperThreading cores) does not increase performance.

For optimal resource usage the size of the non-elite part of a `GenePool` must be a multiply of the number of threads you uses, otherwise some of the worker threads will get a lesser chunks of work and will finish processing earlier than the other threads and will be idle wasting the computing power you may have utilized.

Using of multiple threads imposes side costs for issuing a tasks for each thread and waiting for them to finish execution. It some cases this may cause the `MultithreadedHandler` to perform worse than a single-threaded `DefaultHandler`.

In fact the example of `MultithreadedHandler` usage we have shown above will work slower than the original single-threaded example. This happens because the pool is too small and the rating calculation procedure is too fast, this leads to the situation where the worker thread finishes execution of the task earlier than the next one is put into the task queue, thus the task execution becomes effectively single-threaded.

It is recommended to compare the performance of the `MultithreadedHandler` against the `DefaultHandler` on a short span of iterations to decide which one is better for the particular case.

5.2 Parallel handlers

Another approach to parallelism is the `ParallelHandler` concept. Despite it's name the `ParallelHandler` class does not implement the HandlerInterface. Instead of that it takes an array of `Handlers` and an array of `GenePools` and processes each pool with respective Handler.

In the realm of evolutionary algorithms this approach is called Island Model, as the `GenePool`'s undergo isolated evolution without exchanging of genes. This helps to deal with the local minima problem, by allowing to cover more of the solution space.

Later you can interbreed the `GenePools` using `PoolInterbreeder` class which will be discussed later. This may help to find an even better solution.

Another strategy to use `ParallelHandler` in junction with `PoolInterbreeder` is to process two pools: one with higher mutation, the other with lower. The pool with higher mutation will do the wide search, trying to jump out the local extremum – lets call this experimental pool. The pool with the lower level of mutation on contrary will move towards the nearest extremum – let's call it preservation pool. The idea is to interbreed preservation pool with experimental pool each time it reaches better result than preservation pool. And clone (or interbreed with itself) the preservation pool into experimental pool if the latter did not outperform the former within certain amount of iterations.

6 Interbreeding pools

As mentioned in the previous chapter, you can perform parallel evolution of several `GenePools`. After a reasonable amount of iterations you will get several evolved solutions. As in each case the evolution goes in unpredictable direction these solutions may be quite diverse.

As in the real nature hybridization of far-related species may produce an offspring which outperforms both parents, so in in our case the interbreeding of independent `GenePools` may give us a solution better then any of the ancestors.

This is where the `PoolInterbreeder` class comes to help. The class can perofrm two basic operations both of which require two `GenePools` as a source of genes.

The first operation takes elite part of the two given `GenePools` and uses these as parents to fill the non-elite parts of both pools. Every new born individual has one parent from the first pool and another – from the second.

The second operation uses elite individuals from the two parent `GenePool` to recreate all the individuals in the third `GenePool`.

The both operations use `ReplicationStrategy` and `SelectionStrategy` to cross the DNA of the parents.

The interbreeding of pools may bring no profit if the DNA of the individuals is not homological.

7 Persistence

You can save and load the `GenePools` or the single Individuals using the `PoolIO` class.

The usage of the class is pretty straight forward. However when you load the pool or individual you need to know what you are loading, thus you need to provide a solution interface or an `IndividualFactory` to instantiate the individuals in the loaded pool.

Below are examples of loading and saving pools. Please note that no one of the methods does close the given streams.

```
PoolIO.savePool(outputStream, pool);

GenePool<ExampleInterface> loadPool = PoolIO.loadPool(inputStream,
ExampleInterface.class);
```

The example above demonstrates loading and saving of simple GenePools. While saving of GenePools of neural network is very similar, loading of such pools is a bit more complicated, since it's only DNA and context parameters are saved, thus we need to provide an information about the network structure.

```
LayerModel layer1 = new LayerModelConfiguration(10).withEmbeddingMode(4,
2).getModel();

LayerModel layer2 = new LayerModelConfiguration(2).getModel();

NeuralNetworkFactory neuralNetworkFactory = new NeuralNetworkFactory(new
NeuralNetworkModel(8, null, layer1, layer2));

GenePool<NeuralNetwork> loadPool = PoolIO.loadPool(bis,
neuralNetworkFactory);
```

In the example above first we create the `NeuralNetworkFactory` – an instance of the `IndividualFactory` which is then used by `PoolIO` class to instantiate the individuals of the pool before loading their DNA.

8 Neural Networks

Evolutionary algorithms have proven their effectiveness for training of Artificial Neural Networks. Evolutionary approach is especially effective when we deal with the case where we don't have predefined set of inputs and desired outputs, for example evolving of Artificial Intellect.

Since version 3.0 EvoJ provides classes for working with Artificial Neural Networks.

8.1 Basics

8.1.1 Creation

Just like regular EvoJ individuals the Neural networks should be instantiated via factories. Then `NeuralNetworkFactory` class instantiates individuals holding the genes of the neural networks and exposing the `NeuralNetwork` interface.

Below there is an example of creating of a `GenePool` of simple neural networks.

```
LayerModel layer1 = new LayerModelConfiguration(10).getModel();
LayerModel layer2 = new LayerModelConfiguration(5).getModel();
NeuralNetworkModel model = new NeuralNetworkModel(12, null, layer1,
layer2);
NeuralNetworkFactory factory = new NeuralNetworkFactory(model);
DefaultPoolFactory dpf = new DefaultPoolFactory();
GenePool<NeuralNetwork> pool = dpf.createPool(30, factory);
```

The example above creates a `GenePool` of neural networks having 12 inputs, one hidden layer with 10 neurons and one output layer with 5 neurons.

The elements of the `GenePool` implement also the Individual interface. This allows standard genetic strategies (replication, selection, mutation) to work for neural networks too.

8.1.2 Calculations

Below is an example of performing calculations using neural networks.

```
network.readDNA();
float[] input = {0, 1};
float[] outputs = network.getOutputs(input);
```

To improve performance the neural network classes cache the values of DNA in the local variables, thus these need to be reread prior to doing any calculations each time the DNA changes.

8.1.3 Training

If you are going to train neural network based on some predefined set of input values along with desired output values, then you can use the `NeuroNetRating` class which extends above mentioned `AbstractSimpleRating`.

The `NeuroNetRating` class works in conjunction with `Tutorial` class which holds pairs of input and output arrays. Below is an example of using of the classes.

```
float[][] inputs = new float[][]{
    {1, 0, 0, 0, 0, 0, 0}, {0, 1, 0, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 1, 0, 0, 0}, {0, 0, 0, 0, 1, 0, 0}, {0, 0, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0, 0, 1}
};
float[][] outputs = new float[][]{
    {1, 0, 0}, {0, 1, 0}, {1, 1, 0}, {0, 0, 1}, {1, 0, 1}, {0, 1, 1},
    {1, 1, 1},};

Tutorial tutorial = new Tutorial();
for (int i = 0; i < 7; i++) {
    tutorial.addEntry(inputs[i], outputs[i]);
}

NeuroNetRating neuroRating = new NeuroNetRating(tutorial);

DefaultHandler handler = new DefaultHandler(neuroRating);
```

The `NeuroNetRating` class displays the neural network all the training cases and calculates the negative sum of square error as Rating.

If you wish to customize the learning process then you can just subclass the `AbstractSimpleRating` class or any other fitness related interface and perform estimation any way you need. Below is a hint how to do this.

```
public class Rating extends AbstractSimpleRating<NeuralNetwork, Double>
{

    @Override
    protected Double doCalcRating(NeuralNetwork solution) {
        solution.readDNA();
        return calculateFitness(solution);
    }
}
```

One important thing to note here is the call to `solution.readDNA()` prior to any calculations.

8.2 Configuring

As you have seen in the example above the `NeuralNetworkFactory` requires a list of `LayerModel` objects describing layers of the neural network.

The layer model object should be created via usage of `LayerModelConfiguration` class. Which uses fluent style to adjust properties of a layer being configured.

Here are the examples of layer configuration.

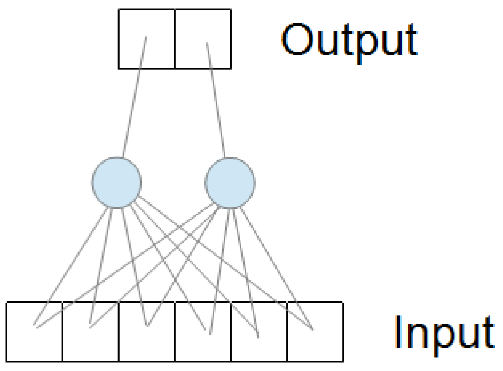
```
LayerModel model1 = new
LayerModelConfiguration(10).withRecurrence().withNeuroFunction(new
HyperbolicTangentFunction()).getModel();

LayerModel model2 = new LayerModelConfiguration(5).withNeuroFunction(new
SoftmaxFunction()).getModel();
```

Using this fluent approach you can specify ranges of weights, biases, initial states, layer functions and layer types.

8.3 Layer types

8.3.1 Simple layers

The most basic feed-forward layer.



Any layer by default is simple, unless you call methods `withEmbedding` or `withRecurrence` of the `LayerModelConfiguration` class.

The number of inputs is determined automatically as a number of outputs from previous layer.

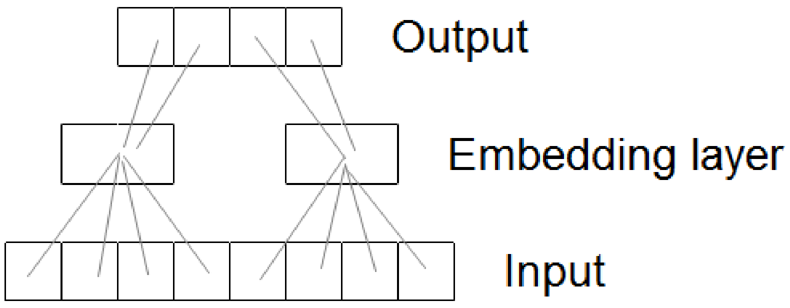
If you want to create a simple layer, just do not call any layer type changing methods during layer model creation.

The example below configures a simple layer of 5 neurons with softmax activation function

```
LayerModel model2 = new LayerModelConfiguration(5).withNeuroFunction(new
SoftmaxFunction()).getModel();
```

8.3.2 Embedding layers

Embedding layers is a special type of layers. The scheme demonstrating the idea of embedding layers is given below.



As you can see, the embedding layer is in fact simple layer repeated horizontally desired number of times. Each copy of simple layer takes it's own subset of inputs and produces a subset of outputs.

This is useful when the input is composed of similar entities, like letters or notes. The embedding layer allows to detect features of each input element using same logic.

The example below creates a model of an embedding layer.

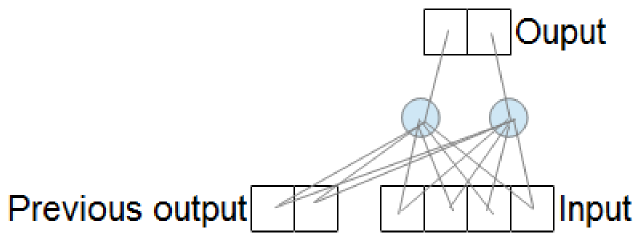
```
LayerModel mdl = new LayerModelConfiguration(2).withEmbeddingMode(4,
3).getModel();
```

This code creates a model of an embedding layer consisting of 2 neurons, repeated 3 times, each neuron requires 4 inputs. Thus the whole layer will produce 6 output values and require 12 input values.

Unlike the simple layer the embedding layer cannot adjust itself to any number of inputs. Thus the underlying layer must provide exactly the number of inputs per neuron of an embedding layer times the layer is repeated

8.3.3 Recurrent layers

The recurrent layers calculate their state using not only the input from the lower layer, but also information about their state at the previous iteration. The scheme of a recurrent layer is similar to a scheme of a simple layer, the only difference is that the array of inputs is extended by a set of outputs from a previous state.



At the first iteration the previous input values is taken from a set of initial values which are also stored in the DNA.

An example of creation of recurrent layer model is given below.

```
LayerModel recurrentModel = new
LayerModelConfiguration(12).withRecurrence().getModel();
```

The example creates the recurrent layer of 12 neurons.

Like the simple layer the recurrent layers can adjust to any number of input values.

8.4 Functions

The output values of a neural layer are determined by the `NeuroFunction` class. The default function for the layers is the `SigmoidFunction` class. There exist also the `SoftmaxFunction`, `LinearFunction` and `HyperbolicTangent` implementations.

If you'd like to implement your own function please take into consideration that there are helper classes to extend instead of the `NeuroFunction` interface itself.

8.5 Internal structure

All the layers internally are built alike – each neuron has a set of weights and a bias.

The recurrent layer also has it's initial state which is used when the layer used for the first time after creation or reset.

In order to improve performance the layers do not reallocate the output arrays each time, instead of that the same instance of the array is returned each time you need to get an output. This makes the neural networks in EvoJ not thread-safe.

Also for the sake of performance the parameter like weights, bias, etc. must be read into layer cache from DNA each time the DNA changes – after replication or mutation. Thus you need to call `NeuralNetwork.readDNA()` method in order to refresh internal cache.

If the network contains at least one recurrent layer then you need to call the `resetState()` method in order to bring the layers into initial state.

8.6 Performance considerations

The activation functions extensively use exponents, logarithms and hyperbolic tangents. Unfortunately these functions are very slow in Java since they are implemented in code instead of using of co-processor instructions, this was made to ensure the bit-to-bit precision and predictable calculation result at all platforms and processors. This issue drastically slows down the calculation of the above mentioned functions.

It's highly recommended to reimplement standard functions using Java libraries which provide fast computations or even use JNI to calculate the mentioned functions using co-processor instructions.