EvA2 Short Documentation

Fabian Becker, Marcel Kronfeld

Dept. of Cognitive Systems,
Prof. Dr. Andreas Zell,
University of Tübingen

URL: `http://www.ra.cs.uni-tuebingen.de/software/EvA2`

Last updated: July 8, 2013

Contents

Acknowledgements

We like to thank all former and current developers of EvA: Fabian Becker, Alexander Hasel, Roland Baumann, Karsten Jung, Jürgen Wakunda, Holger Ulmer, Felix Streichert, Christian Spieth, Hannes Planatscher, Andreas Dräger, Michael de Paly, Marcel Kronfeld and all former students involved in the work.

CHAPTER 1

Introduction

EvA2 (an Evolutionary Algorithms framework, revised version 2) is a comprehensive heuristic optimization framework with emphasis on Evolutionary Algorithms implemented in Java[1]. It is a revised version of the JavaEvA [**8**] optimization toolbox, which has been developed as a resumption of the former EvA software package [**9**].

EvA2 integrates several derivation free optimization methods, preferably population based, such as Evolution Strategies, Genetic Algorithms, Differential Evolution, Particle Swarm Optimization, as well as classical techniques such as multi-start Hill Climbing or Simulated Annealing.

EvA2 aims at two groups of users. Firstly, the applying user who does not know much about the theory of Evolutionary Algorithms, but wants to use them to solve a specific application problem. Secondly, the scientific user who wants to investigate the performance of different optimization algorithms or wants to compare the effect of alternative or specialized evolutionary or heuristic operators. The latter usually knows more about evolutionary or heuristic optimization and is able to extend EvA2 by adding specific optimization strategies or solution representations. Explicit usage examples for EvA2 are given in [**5**].

This document is, as the title says, not an extensive manual on the EvA2 framework, but instead a short introduction hoping to ease access to EvA2. Thus, the document is mainly sketched along use-cases and tries to deliver knowledge on a top-down basis, with most important things first and details where required. Still: as EvA, just as mostly any larger software package, can become tricky sometimes, it is not always possible to explain things without cross-references. We hope that this document will, anyways, be a valuable helper in working with EvA2.

The document contains, of course, a Quick Start guide (Sec. 2) also explaining the graphical user interface (GUI, Sec. 2.3). Sec. **??** contains hints on how to use EvA2 with external programs, e.g. MATLAB[2]. We provide a quick-and-simple way to add an application problem implementation in Sec. 3 and describe more details of the API in Sec. 4 to access further options and functionality. Finally, we propose some literature sources for readings on Evolutionary and Heuristic Optimization in Sec. 5 for the interested users.

[1] Oracle and *Java* are registered trademarks of Oracle and/or its affiliates.

[2] MATLAB is a registered trademark of The MathWorks, Inc. in the United States and other countries.

CHAPTER 2

Quick Start

The following sections give a short introduction in the main aspects of using EvA2, explaining the possibilities accessible by the GUI. Even if you want to use the API without the GUI, we recommend to try some optimization runs through the GUI first, as it will help to learn about the concepts used in the framework.

2.1. Running EvA2

To quickly test EvA 2, we recommend you download the jar-package *EvA2.jar* and start the GUI. The jar-file can be downloaded from the EvA2 homepage[1] [**1**].

To start under GNU/Linux, you can just type:

```
$ java -cp EvA2.jar eva2.client.EvAClient
```

Note that "`$`" stands for the command prompt, which needs not to be typed. In the same or a similar way, you can start it on all other platforms that support Java. The Java option *-jar* is also possible with the base package of EvA2, but does not allow adding further jar-packages on the classpath. Therefore, we encourage using the method given above.

If you want to work on the source code directly, note that it is also vital to copy the resource folder to the directory where the compiled class files are located. You can then again start the GUI or optimize through the API (Sec. 4). However we advise you to learn to know the GUI a little before digging in the source code.

2.2. Some Words on the Words

As EvA2 is mainly about Evolutionary and Heuristic Optimization, some of the terms and notions are borrowed from the area and used in this document. As they may not be familiar to all who want to use the framework, we give a short summary here.

We aim at optimizing a target function without knowing much about it, and find a certain position in the search space which minimizes the function, called the *solution*. During search, we use a specific search strategy, the *optimizer*, which usually looks at several positions in parallel. Those are all *potential solutions*, because we don't know the real one yet. For the potential solutions we evaluate the target function. The value received is often called *fitness* in analogy to Darwin's Theory of Evolution, where "the fitter ones survive". For the same reason, potential solutions are sometimes called *individuals*, and the set of potential solutions stored by the optimizer at a time may be called *the population*. Many of the implemented optimization strategies employ operators in analogy to natural *mutation*, *crossover* and *selection*.

[1] `http://www.ra.cs.uni-tuebingen.de/software/EvA2/`

There is nothing mystical about that, and of course the analogy is often exaggerated. Evolutionary Optimization is an algorithmic tool that serves mostly technical purposes. That it works in a computer is by no means a sign that we fully understand natural evolution or can prove anything about it. This said, of course, we would never doubt that natural evolution in fact works.

This document will not explain in detail how the implemented optimizers work, as there is enough literature out there handling these topics. We refer to Sec. 5 for suggestions on further reading.

2.3. Using the GUI

From the GUI, also called the EvA workbench, all important components of an optimization run can be accessed and configured. To change the optimization method, for example, click on the field labeled with "*optimizer*" and select the desired algorithm from the drop-down menu. Basically, you thereby select a Java class and create an instance, whose public properties are displayed in the window immediately with their standard values. For your optimization run, you may configure the parameter values directly through the input fields. A short description will be displayed by tip-text above the name of each parameter. If you just hit the "*Start*" button, an optimization will be started using the current settings.

The EvA GUI has two main tabs: the optimization parameter tab and the statistics tab (Fig. 2.3.1), the components of which will be summarized in the following.

2.3.1. The Workbench Window. The optimization parameters:

***Optimizer*:** Select the main optimization method. You can choose between classical as well as evolutionary and swarm-based optimization methods. For quick optimization, just use the standard values of the parameters and try several different optimizers.

***Post-processing parameters*:** In some cases, post processing of the results is desirable, e.g. if you want to improve the single found solution by small hill climbing steps, or if you want to retrieve more than one solution from a clustering optimization approach.

***Problem*:** The instance of the target function to be optimized is specified here. You can select from the benchmark problems delivered with the package or inherit from the problem class yourself (Sec. 4).

***Random Seed*:** As most algorithms in EvA2 incorporate stochastic components, the random seed is critical for the specific outcome of a run. For replicable results, set a seed value > 0. To receive statistically relevant results, test several times with a seed of 0, which means that the system time is used as seed for each new run, or use the multi-run option.

***Termination Criterion*:** Set the criterion by which to stop an optimization run, e.g. stop after n fitness evaluations.

The Statistics parameters:

***Convergence Rate Threshold*:** Provided the target value is zero, convergence is assumed if a value smaller than this threshold is reached. For multi-run experiments, the number of hits is counted using this criterion.

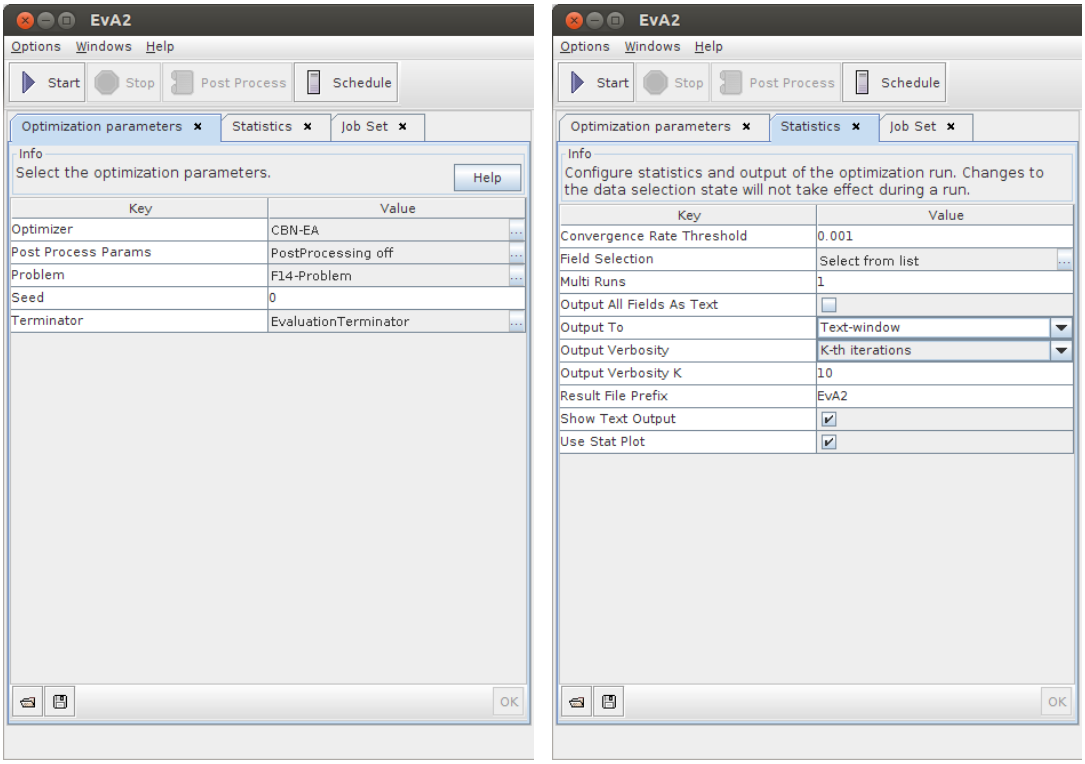

FIGURE 2.3.1. Screenshots of the workbench window, with optimization (left) and statistics (right) parameters.

***Field Selection*:** The data fields to be displayed can be selected. Typically, the current and best fitness are of most important, but other fields such as average distance of candidate solutions may be of interest. Since some data fields are complex types, such as arrays, they are not plotted in the graph window but dumped to the text window if marked. Depending on the problem and optimizer in use, different data fields may be available. Specific information on the data fields is given by tool tips in the application.

***Number of Multi-runs*:** To achieve statistically meaningful results on how well a certain optimizer works on a given problem, set this number to do several runs in a row. The plot will be averaged, while all intermediate data can be collected in an output file or text window.

***Output All Fields As Text*:** In some cases it is helpful to show only selected data fields graphically but dump all data fields to the text listeners, e.g., for external analysis.

***Output To*:** Textual information can be shown in a text box, or redirected to a file, or both. The output file will be stored to the current directory with a descriptive name containing the timestamp of the start of the run.

***Output Verbosity*:** Select the verbosity of the textual output, possible settings are "no output", "final results", "k-th iterations" and "all iterations". An iteration is usually a generational cycle, meaning that for "all iterations", intermediate data is printed after each generation.

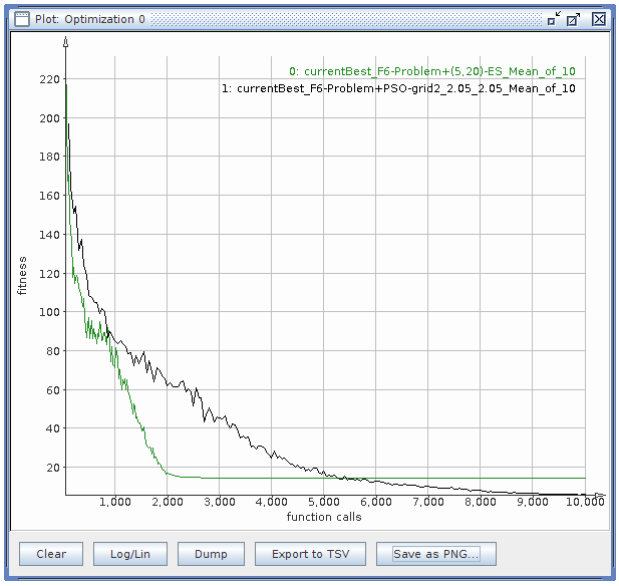

FIGURE 2.3.2. Plot window comparing a simple (5,20)-ES and PSO on Rastrigin's after 10 runs each.

***Verbosity Parameter k*:** Define the interval parameter for the "k-th iterations" setting of the output verbosity, by which intermediate data is printed.

Finally, there are three buttons on top. *"Description"* shows some very general information on the main EvA module. The "*Start*" button, as expected, starts an optimization run using the parameters set, or multiple runs sequentially if *multiRuns* is set higher than one. During optimization, the "*Stop*" button can abort the (multi-)run.

2.3.2. The Plot Window. During the optimization run, the progress of the solution is plottet to a graph in a separate window (Fig. 2.3.2). Usually, the fitness of the best individual of every generation is drawn in Y-direction along the number of function calls in X-direction. Be aware that for multi-objective problems, only the first fitness dimension is shown. The 2-dimensional multi-objective problem classes have, however, a pareto-front viewer which displays the population in the two fitness dimensions sequentially. For higher fitness dimensions it is more practical to use external tools for visualization. Figure 2.3.2, by the way, shows the fitness progress averaged over 10 runs of a (5,20)-ES and a PSO strategy on Rastrigin's Problem: PSO converges slower, but finds better results on the long run, while ES settles earlier on higher plateaus.

The visible buttons have the following functions:

***Clear*:** Remove all graphs from the plot window.
***Log/Lin*:** Switch between linear and log-scaled view. Most benchmark problems in EvA are implemented with the minimum fitness at zero, so that the log-scale view allows to compare and analyze convergence behaviour in detail. Of course, if the target fitness may become zero or negative, log-scale view is impossible.
***Dump*:** Export the contained data to standard output. For each graph, a column is created in the same order they were generated.

***Export*:** Create the same output as *Dump* and save it to a file.
***Save as PNG*:** Create a PNG image of the plot window and save it to a file.

2.3.3. Basic Optimization using EvA2. To get a grip on EvA2 and what optimization means, it is best to run some experiments on the implemented standard benchmarks. To do that, start the GUI and select a benchmark problem, e.g. the F1-Problem consisting in a simple hyper-parabola. Leave the post-processing deactivated. Then choose an optimizer, such as Evolution Strategies with standard parameters, set the termination criterion to EvaluationTerminator with 10,000 fitness calls and push the "Start Optimization" button at the top of the window. Two additional windows will now open up: the plot window with a fitness graph, and a text box displaying the optimization progess in textual form. The final result will be printed into the text box at the end of the run, as well.

If you play around with some optimizer settings, e.g. you try different values for μ and λ or activate the plusStrategy checkbox, you will notice changing performance of the ES. On problems within discrete space, such as the B1-benchmark problem, for example, a Genetic Algorithm is often superior to an Evolution Strategy. You can try this if you clear the plot window, select the B1-problem and run the ES a few times. Now, switch to the Genetic Algorithm and run the optimization a few more times.

Notice, however, that by changing from the F1-problem to the B1-problem, the internal representation of individuals may change. As B1 is a typical binary problem, it uses GAIndividuals by default, which are based on binary vectors, while F1 uses double vectors. Be aware, that not all optimizers in EvA2 are built to work on all types of indviduals.

2.3.4. Post-Processing. To see how post processing works, you can select the *FM0Problem* from the problem list, which is a simple target function with a global and a local optimum. Select the *ClusterBasedNiching* algorithm as the optimizer. Now click on the *postProcessingParams* and activate them. For a clustering distance of $\sigma = 0.1$ and $\approx 5,000$ hill climbing steps, the optimizer should print out just a few solutions in the text box, the first of which hopefully are the optima near (1.7/0) and (-1.44/0).

Post-processing serves mainly two purposes: filter redundant solutions and refine the search results. Redundant solutions occur naturally in population-based heuristics. The optimizer handles several potential solutions in parallel, and it is hoped that they all converge on the global optimum during the run. Or for multi-modal problems which have several local optima, it can be desirable to have parts of the population converge in different areas of the solution space. In any case, one usually wants to retrieve *the* solution set or a refined global optimum. For this purpose, we employ a clustering approach which takes the whole solution set and merges similar solutions to an associated subset. For each of these bulks, only the best individual is returned in the filtered solution set.

Of course the size of this filtered set depends on the degree of convergence in the original set and on the clustering criterion. We employ density based clustering [**4**], which associates any two individuals which have a distance of less than the clustering parameter σ (Fig. 2.3.3). This is an intuitive approach that does not require a predefined number of clusters, in contrast to k-means, for example. By defining σ, you thus define the resolution you grant your solution set.

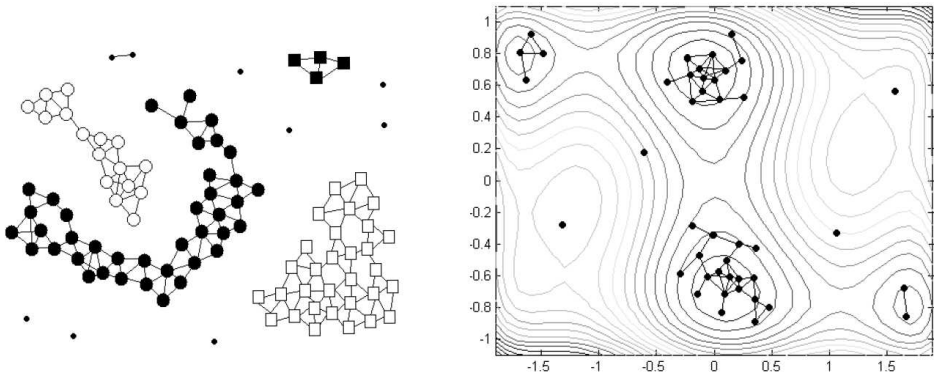

FIGURE 2.3.3. Examples for density based clustering (Streichert et al. [**7**]).

As the solution set always contains the last state of the heuristic optimization, one may hope that it is converged. But of course often it is not fully converged, or maybe the strategy even rediversifies the population from time to time, meaning that some part of the set it is converged while other individuals are freshly initialized and thus by no means optimal. So after filtering out redundancy, you might also want to refine the returned set a little. This can be done directly by Hill Climbing (HC) in the post-processing step by setting *postProcessSteps* to the number of evaluation calls you want to invest in the refinement.

If you set σ for clustering and performed hill climbing, then there will be another clustering step right after the HC process, to remove redundancy that emerged by the additional HC optimization.

2.4. Additional Packages

To add additional packages to use them with the EvA2 base package, you can just add them to the class path. For instance, to use the additional "Probs" package containing a larger set of benchmark problems, place them both in your working directory and type (GNU/Linux):

```
$ java -cp EvA2.jar:EvA2Probs.jar eva2.client.EvAClient
```

You should now be able to select from a larger set of optimization problems in the GUI. Note that different platforms use different characters as path separators (':' in GNU/Linux). To add your own classes to the EvA2 framework (see Sec. 3), you need to add your local development path to the classpath, for example:

```
$ java -cp EvA2.jar:EvA2Probs.jar:/home/username/OwnClassDir \
   eva2.client.EvAClient
```

Note that EvA2 will search all classpath entries for compatible classes, so you should only add those packages which are really required. For your own additional packages, this also means that, if you want your extensions (e.g. own problem implementations or operators) to be displayed in the EvA 2 GUI, they have to be assigned the same package tree as the associated EvA2 operators. Check Sec. 4 on more details.

CHAPTER 3

Quickly Adding a Problem Class

It is easy to integrate your own Java-coded problem in EvA2 and optimize it using the GUI. Preconditions for the quick way are only the two following ones:

- A potential solution to your problem can be coded as a double vector or a BitSet.
- You can implement the target function such that it produces a double valued fitness vector from the double vector or BitSet.

If these simple conditions are met, you have the advantage of being able to use the full "Java-Power" to implement a target function and optimize it in EvA2. Just follow these steps:

- Create an empty class (let's say, `ExampleProblem`) and assign it to the package `simpleprobs`. Put it in a directory called "simpleprobs" within your working directory.
- Have the class inherit from `simpleprobs.SimpleDoubleProblem` or, depending on which datatype you want to use, from `simpleprobs.SimpleBinaryProblem`. Both base types can be used directly from the EvA2 jar-file (which of course needs to be on the java classpath - in Eclipse, for instance, add the EvA2 jar as "External jar" to the Java build path through the project settings).
- Implement the method `public int getProblemDimension() {...}` whithin your ExampleProblem, which returns the number of dimensions of the solution space, i.e. the length of the x vector or size of the `BitSet`, respectively. The problem dimension may be a variable defined in your class, but it must not change during an optimization run.
- Implement the method `public double[] eval(double[] x)` for double coded problems or `public double[] eval(BitSet bs)` for binary coded problems, whithin your ExampleProblem, where the fitness of a potential solution is calculated.
- Start the EvA2 GUI. *Make sure that your working directory is in the Java classpath*! From the problem list, select the *SimpleProblemWrapper* class as optimization problem. The wrapper class allows you to select your ExampleProblem as target function. If you implemented a double valued problem, you may also set a default range parameter, defining the positive and negative bound of the solution space allowed. Now select your prefered optimization method and start the optimization.

CHAPTER 4

Using the EvA2 API

This chapter describes how to incorporate EvA2 into existing projects through the Java API.

4.1. Accessing Standard Optimizers

A standard optimizer is typically a well-knwon optimization algorithm with operators and parameters predefined in a way so that a user may start it out-of-the-box and expect good optimization results in general. Of course, every expert may have an own notion on what parameters are preferable in general. Therefore we give two customization examples using the API. In any case, a necessary step is to extend a EvA2 base class constituting the target function. You may, again, use the simpleprobs package described in Sec. 3. However, this makes it necessary to use the wrapper class `SimpleProblemWrapper`, which is great for direct GUI usage but may make programming a bit more complicated. We therefore, at this point, recommend to go one step higher in hierarchy and extend `AbstractProblemDouble` or `AbstractProblemBinary`.

4.1.1. The Abstract Problem Classes. The problem class subtree encapsulates properties of target functions that can be directly optimized by the EvA2 framework and starts at the `AbstractOptimizationProblem` class. The basic properties of the problem class are functional:

(1) The problem is itself initialized,
(2) the problem knows how to initialize a set of potential solutions, and
(3) the problem evaluates a set of potential solutions.

To implement your own target function in EvA2, we recommend inheriting from `AbstractProblemDouble` or `AbstractProblemBinary`, depending on your preferred data representation. Integer and program-data-based problems are currently not covered by this document. Notice that if you want your problem implementation to be displayed and manageable by the GUI in the same way the predefined problems are handled, you have to assign it the same package tree as its parent class (e.g. `eva2.server.go.problems` for new problem implementations).

The class `AbstractProblemDouble`. The `AbstractProblemDouble` class in the package `eva2.server.go.problems` encapsulates methods useful to implement double valued target functions. Let's assume you want to implement a function $f_{\text{target}}(x) = y$ with $f_{\text{target}} : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Important for double-valued problems is the range of the solution space, defining in any dimension the minimum and maximum value allowed. By setting the `defaultRange` member variable (method `setDefaultRange`) in your constructor, you can easily define a symmetric range $[-dR,\ dR]^n$. If you need a different range definition, you may overload the methods

`getRangeLowerBound(int dim)` and `getRangeUpperBound(int dim)` which are to return the upper or lower range limit for a given dimension.

The `AbstractProblemDouble` class also contains a noise parameter and an individual template. The *noise* can be used to add gaussian fluctuation to any target function. If *noise* $>$ 0, it defines the standard deviation of the random disturbance of $f_{\text{target}}(x)$. The individual template defines the data representation for potential solutions. It also contains evolutionary operators which work directly on the representation, see Sec. 4.3.2 for a usage example.

We will now give short comments on further important member functions:

`public Object clone()`: Object copy operator. We recommend implementing a copy constructor `public YourProblem(YourProblem o)` and refering to it from within the `clone()` method. You may call `cloneObjects` from AbstractProblemDouble to copy the super members. Make sure that the copy constructor copies all necessary member variables you added to your class.

`public void evaluate(AbstractEAIndividual individual)`: Main evaluation method. We recommend not to override it.

`public abstract double[] eval(double[] x)`: Essential evaluation method. Override it to implement the desired target function $f_{target}(x)$. Make sure that the delivered solution vector is always of the same dimensionality m. Even if your problem is one-dimensional, return an array of length 1 with the single fitness value as the only field.

`public abstract int getProblemDimension()`: Return the problem dimension in solution space. Make sure it is constantly equal to n during an optimization run. If you implement a corresponding method `public void setProblemDimension(int n)` and define your class within the same package, you may change the dimension through the GUI.

`public void initProblem()`: Called before optimization in general. If you define member variables of your class which are not constant, we recommend setting them to initial values in this function. If you override, make sure you call the super-method from within your version.

`public void initPopulation(Population population)`: This method is called before optimization with a certain population. It initializes the population to the problem specific range using the individual template and calls a method of the `Population` collection to initialize the individuals randomly whithin the range. If you want to alter the way the individuals are initialized, we recommend overriding this method. Call the super-method from your implementation and make changes to the individuals afterwards.

For an example on how to extend `AbstractProblemDouble` you can look at the `F1Problem` class which implements a simple parabola function.

The class `AbstractProblemBinary`. The binary variant is widely analogous to `AbstractProblemDouble`. The main differences are that there is no range definition (the range is always $0^n - 1^n$) and that the template is of a different type, namely `GAIndividualBinaryData`, and

Algorithm 1 Simple `OptimizerFactory` usage example.

```
1  package eva2.examples;
2
3  import eva2.OptimizerFactory;
4  import eva2.optimization.problems.F1Problem;
5
6  public class TestingF1PSO {
7
8      public static void main(String[] args) {
9          F1Problem f1 = new F1Problem();
10         // start a PSO with a runtime of 50000 evaluations
11         OptimizerFactory.setEvaluationTerminator(50000);
12         double[] sol = OptimizerFactory.optimizeToDouble(OptimizerFactory.PSO, f1, null);
13         System.out.println(OptimizerFactory.terminatedBecause() + "\nFound solution: ");
14         for (int i = 0; i < f1.getProblemDimension(); i++) {
15             System.out.print(sol[i] + " ");
16         }
17         System.out.println();
18     }
19 }
```

delivers a Java `BitSet` as data representation instead of a double vector. The `eval` function signature changes accordingly, and to implement a target function $g_{\text{target}}(x) = y$ with $g_{\text{target}} : \{0,1\}^n \to R^m$, you need to work on a `BitSet`[1] object.

> `public abstract double[] eval(BitSet bs):` Essential evaluation method. Override it to implement the desired target function $g_{\text{target}}(b)$. Make sure that the delivered solution vector is always of the same dimensionality m. Even if your problem is only one-dimensional, return an array of length 1 with the single fitness value as only field.

For an example on how to implement binary functions, look at the `B1Problem` class which realizes a simple minimize bits problem.

4.1.2. The `OptimizerFactory`. To access default optimization algorithms easily, we have defined an `OptimizerFactory` class. It allows to specify an optimization algorithm by an ID number and mainly takes a problem class and an optional output file as input. For example, if you have a double-valued problem class and just want to retrieve one solution, use the `optimizeToDouble(final int optType, AbstractOptimizationProblem problem, String outputFilePrefix)` method, which returns the solution as a double vector.

Table 1 gives an overview over the currently accessible optimization strategies. You may use `showOptimizers()` to get a string with a summary of the implemented algorithms with ID associations. A short example is shown in Listing 1, where PSO is used to optimize the simple hyper parabola (in 10 dimensions by default). The PSO with $50,000$ evaluations should find a solution very close to zero, e.g. absolutely below 10^{-30} in every dimension.

[1] Concerning `BitSet`: it may be valuable to note that when looping over a `BitSet`, it is preferable to use the self defined problem dimension as index limit instead of `size()` or `length()` methods of `BitSet`.

ID	Short Description
STD_ES	A standard (15,50)-Evolution Strategy.
CMA_ES	(15,50)-Evolution Strategy with Covariance Matrix Adaptation.
STD_GA	Standard Genetic Algorithm with elitism
PSO	Particle Swarm Optimization with constriction.
DE	Differential Evolution
TRIBES	Tribes: an adaptive PSO
RANDOM	Random search (Monte-Carlo)
HILLCL	Multi-start hill climbing
CL_HILLCL	Clustering multi-start hill climbing
CBN_ES	Clustering-based Niching ES

TABLE 1. Overview over algorithms accessible through OptimizerFactory.

4.2. Termination Criteria

To configure the termination criteria of an optimization process started through the OptimizerFactory, use the setTerminator family. The default terminator stops at a maximal number of fitness evaluations, e.g. $10,000$. To change the number of evaluations performed to, for example, $50,000$, call `OptimizerFactory.setEvaluationTerminator(50000)` before starting the optimization, as in Alg. 1.

For more flexible termination criteria, there are several Terminator parameter classes you can use and set them directly using `OptimizerFactory.setTerminator(term)`. Available are the following variants:

`EvaluationTerminator`: Construct with a maximal number of evaluations. Parameters:
- `maxEval`: integer number of evaluations to be performed at maximum.

`GenerationTerminator`: Terminate after a given number of generations. As not all algorithms use constant population sizes, we suggest to use EvaluationTerminator preferably. Parameters:
- `gens`: integer number of generations to be performed at maximum.

`FitnessValueTerminator`: A minimum fitness value (vector) must be reached to terminate. Construct with a double array representing the target fitness value. Parameters:
- `v`: double array fitness vector to be reached.

`CombinedTerminator`: For effective optimization, one might want to combine several criteria in a boolean way, i.e. terminate if $20,000$ evaluations have been performed OR the best fitness doesn't change for $1,000$ evaluations. To allow for this, we provide the `CombinedTerminator` class which just takes to terminator instances and combines them logically. Thus, terminators can be nested as required. Parameters:
- `t1`: First terminator to combine.
- `t2`: Second terminator to combine.
- `bAnd`: Boolean flag indicating whether to use conjunctive (AND, for bAnd == true) or disjunctive (OR, for bAnd == false) combination.

A specific class of terminators checks for convergence of a population measure, such as the average distance of individuals within the population. Common to these terminators is the required definition of convergence. Generally, convergence is assumed if the measure $m(P(t))$

has not changed more than a certain threshold θ during a fixed time period t_S. The stagnation time t_S may either be taken as fitness calls (calls to the target function) or generations (iterations of the optimizer), however the population measure is calculated only after full generations.

The convergence threshold may either be an absolute value, a change of the absolute value compared to $m(P(t - t_S))$ or a change of the relative ratio of the value $m(P(t - t_S))$. Additionally, stagnation may be determined if no decrease by more than θ has occurred, or if no change occured (bidirectional: neither decrease nor increase by more than θ). In the case of bidirectional check, termination occurs if the population measure changes less than θ for the stagnation period t_S: $\forall\, i \in \{0, 1, ...p - 1\} : m(x_{t-t_S}) - \delta \leq m(x_{t-i}) \leq m(x_{t-t_S}) + \delta$, where $m(x_t)$ stands for the population measure of individual x at generation t. The allowed boundary δ is given with the fixed threshold in the absolute change case ($\delta = \theta$) or calculated from $m(x_{t-t_S})$ in the relative change case: $\delta = \theta \cdot m(x_{t-t_S})$. In the absolute value case, $m(x_{t-i})$ is directly compared to θ: $m(x_{t-i}) \leq \theta$ must hold for t_S iterations.

The following Terminators are based on this convergence criterion:

`FitnessConvergenceTerminator`: Terminate as soon as there is hardly any improvement (or change) in the fitness of the best individual for a certain period of time. In a multi-objective case, the 2-norm of the fitness vector is calculated (however a pareto metric should be prefered in these cases). Parameters:

- `checkType`: check for decrease only of the measure or bidirectional change, where the measure may not leave in both directions to determine stagnation.
- `convergenceCondition`: check for relative change, absolute change or absolute value reached along the stagnation time.
- `convergenceThreshold`: double valued fitness threshold θ.
- `stagnationMeasure`: interprete the stagnation time as number of function calls or generations.
- `stagnationTime`: integer length of the stagnation time.

`PhenotypeConvergenceTerminator`: In analogy to the `FitnessConvergenceTerminator`, terminate as soon as there is hardly any change in the best phenotype, meaning that $m(P) = x_{best}$ for the best indiviual $x_{best} \in P(t)$. Parameters: see `FitnessConvergenceTerminator`.

`DiversityTerminator`: Terminate as soon as the average/minimal/maximal distance of individuals in the population stagnates. Thus, $m(P)$ calculates distances on all individuals in P at every iteration, which may be time-consuming. Parameters: see `FitnessConvergenceTerminator`, with additions:

- `criterion`: Check the minimal, maximal or average distance as diversity measure on P.
- `metric`: The metric to use for distance calculation.

`ParetoMetricTerminator:` Terminate if the Pareto front of a multi-objective optimization problem stagnates. To measure progress on pareto fronts, several metrics can be used. Note that this only works with multi-objective problem instances. Parameters: see `FitnessConvergenceTerminator`, with additions:

- `paretoMetric`: The metric to use to measure change of the Pareto front.
- `useCurrentPop`: Apply the measure to the current population or the current Pareto front stored by the optimizer.

Some notes on terminators:

- Note that the termination criterion is always checked after one iteration, i.e. one generation. This means that, if the number of evaluations is not a multiple of the population size or the population size is variable, the maximum number of evaluations may be slightly exceeded. The same holds for the stagnation time in convergence terminators.
- Concerning convergence terminators: Note that for flat plateaus in fitness space, the fitness may hardly change while there is still progress in the solution space. On the other hand, note that highly nonlinear problems may hardly change in phenotype but still change considerably in fitness. When using convergence terminators, we suggest to set the stagnation period sufficiently high. Also, take care when checking for relative convergence with measures tending towards zero, since relative fluctuations with very small values may be high without showing real progress.

For a usage example that creates and sets a terminator which stops if either $20,000$ evaluations have been performed or both best fitness and phenotype stagnate for $1,000$ evaluations, see Listing 2. Note that "convergence" in this context refers to stagnation After a combined termination, one might want to know why the run actually stopped. The method `terminatedBecause()` implemented in `OptimizerFactory` as well as any terminator object will return a String object describing the exact reason, while the method `lastEvalsPerformed()` informs on how many evaluations were required. The PSO in the given example should require about $5,000 - 7,000$ evaluations to meet the two convergence criteria.

4.3. Customizing the Optimization

Beyond standard optimizers, you may wish to customize optimization parameters manually or iterate over different settings in a loop. To do this, you need to access the optimization parameter structure `GOParameters` and alter its values before starting optimization. A `GOParameters` instance contains all settings required for an optimization run: target function, optimizer, random seed, post-processing options and termination criterion; basically all that is also set through the workbench window of the GUI (Sec. 2.3.1). To alter specific settings, it is sufficient to alter a `GOParameters` instance generated by the `OptimizerFactory` and then start the optimization using this altered parameter instance.

4.3.1. Accessing Parameters. Look at Listing 3 for an example on how to customize optimization parameters. To find out about which operators are implemented and usable, it is again easiest to check out in the GUI, where there are also short descriptions available.

Note that in example 3, there will actually be $1,050$ evaluations performed, which is the first multiple of the population size of 150 that exceeds the maximum evaluation limit of $1,000$ set. To change the population size before a run, simply set a new population of the target size. Any population will be initialized to contain the given number of individuals by the problem class, usually in a random distribution over the problem range.

Algorithm 2 Terminator combination example.

```
1  package eva2.examples;
2
3  import eva2.OptimizerFactory;
4  import eva2.optimization.operators.terminators.CombinedTerminator;
5  import eva2.optimization.operators.terminators.EvaluationTerminator;
6  import eva2.optimization.operators.terminators.FitnessConvergenceTerminator;
7  import eva2.optimization.operators.terminators.PhenotypeConvergenceTerminator;
8  import eva2.optimization.operators.terminators.PopulationMeasureTerminator.ChangeTypeEnum;
9  import eva2.optimization.operators.terminators.PopulationMeasureTerminator.DirectionTypeEnum;
10 import eva2.optimization.operators.terminators.PopulationMeasureTerminator.StagnationTypeEnum;
11 import eva2.optimization.problems.F1Problem;
12
13 public class TerminatorExample {
14
15     public static void main(String[] args) {
16         F1Problem f1 = new F1Problem();
17         double[] sol;
18         // A combined terminator for fitness and phenotype convergence
19         CombinedTerminator convT = new CombinedTerminator(
20                 // fitness-based stagnation period, absolute threshold, consider stagnation
21                 // in both direction (per dim.) or w.r.t. minimization only
22                 new FitnessConvergenceTerminator(0.0001, 1000, StagnationTypeEnum.fitnessCallBased, ChangeTypeEnum.absoluteChange, DirectionT
23                 new PhenotypeConvergenceTerminator(0.0001, 1000, StagnationTypeEnum.fitnessCallBased, ChangeTypeEnum.absoluteChange, Directi
24                 CombinedTerminator.AND);
25         // Adding an evaluation terminator with OR to the convergence criterion
26         OptimizerFactory.setTerminator(new CombinedTerminator(
27                 new EvaluationTerminator(20000),
28                 convT,
29                 CombinedTerminator.OR));
30         sol = OptimizerFactory.optimizeToDouble(OptimizerFactory.PSO, f1, null);
31         System.out.println(OptimizerFactory.lastEvalsPerformed()
32                 + "␣evals␣performed.␣"
33                 + OptimizerFactory.terminatedBecause()
34                 + "␣Found␣solution:␣");
35         for (int i = 0; i < f1.getProblemDimension(); i++) {
36             System.out.print(sol[i] + "␣");
37         }
38         System.out.println();
39     }
40 ;
41 }
```

4.3.2. Setting Evolutionary Operators. In Listing 4, a (1+5) CMA-ES is configured and run on a simple bimodal target function with the global optimum near (1.7/0) and a local one near (-1.44/0). The (1+5)-CMA-ES is powerful and will find the global optimum most of the time. Sometimes, however, due to its relatively high selection pressure and elitistic strategy, it will converge in the local optimum, depending on the random initialization.

Lines 23-26 of Listing 4 show how to access evolutionary operators directly. What happens here is that the template individual delivered with the problem class (because the problem defines the representation) is modified to use the given mutation and crossover operator and probabilities. Typical for ES, the mutation probability p_m is relatively high, while the crossover probability p_C is rather low. One could also use `fm0.getIndividualTemplate().setMutationOperator(...)` etc. to set the operators and probabilities one by one just as through the GUI. Notice that not all implemented heuristics make use of individual evolutionary operators. Several come with their own operator definitions, such as DE and PSO, for example. The Hill Climbers, on the other hand, do use individual mutation but override individual probabilities to $p_m = 1$ and $p_c = 0$ by definition.

Algorithm 3 Customized optimization example.

```
1  package eva2.examples;
2  import eva2.OptimizerFactory;
3  import eva2.optimization.operators.selection.SelectXProbRouletteWheel;
4  import eva2.optimization.operators.terminators.EvaluationTerminator;
5  import eva2.optimization.populations.Population;
6  import eva2.optimization.problems.B1Problem;
7  import eva2.optimization.strategies.GeneticAlgorithm;
8  import eva2.optimization.modules.GOParameters;
9  import java.util.BitSet;
10
11 public class TestingGAB1 {
12         public static void main(String[] args) {
13                 B1Problem b1 = new B1Problem();
14                 BitSet sol;
15                 // default go-parameter instance with a GA
16                 GOParameters gaParams = OptimizerFactory.standardGA(b1);
17                 // add an evaluation terminator
18                 gaParams.setTerminator(new EvaluationTerminator(1000));
19                 // set a specific random seed
20                 gaParams.setSeed(2342);
21
22                 // access the GA
23                 GeneticAlgorithm ga = (GeneticAlgorithm)gaParams.getOptimizer();
24                 ga.setElitism(false);
25                 ga.setParentSelection(new SelectXProbRouletteWheel()); // roulette wheel selection
26                 ga.setPopulation(new Population(150));  // population size 150
27
28                 // run optimization and print intermediate results to a file with given prefix
29                 sol = OptimizerFactory.optimizeToBinary(gaParams, "ga-opt-results");
30                 System.out.println(OptimizerFactory.terminatedBecause() + "\nFound␣solution:␣");
31                 for (int i=0; i<b1.getProblemDimension(); i++) {
32             System.out.print(sol.get(i)+"␣");
33         }
34                 System.out.println();
35         };
36 }
```

4.3.3. A Multi-Modal Example with Post-Processing. When looking at the last example in 4.3.2 working on a bimodal function, one might ask how to retrieve more than just one optimum of the target function. In EvA2 there are some optimizers implemented which are specialized on this task. Listing 5 shows an example using the clustering-based niching EA (CBN-EA) [**7**] and post-processing to identify both optima of the target function.

Notice that two post-processing cycles are performed (lines 26 and 33). Any repeated post-processing iteration performed on the `OptimizerFactory` uses the same initial state. This means that if there is no new optimization cycle, any new post-processing will work on the same set of solutions, namely the result population of the last optimization.

This is useful, for example, to search for optima using different resolutions iteratively. In our example, we however just demonstrate the different results without (line 26) and with clustering (line 33). If you run the example a few times, it will happen quite often that after the first post-processing with clustering only, more than the two optima are returned, while after the second step with hill climbing, this will happen rather seldomly. All in all, CBN locates the two optima in most of the cases and post-processing helps to identify the real hot spot. In Fig. 4.3.1, two graphs show the target function and the states of an exemplary CBN-run after 500 (left) and 1,500 evaluations (right).

Algorithm 4 Setting up a (1+5) CMA-ES.

```
1  package eva2.examples;
2
3  import eva2.OptimizerFactory;
4  import eva2.optimization.individuals.AbstractEAIndividual;
5  import eva2.optimization.operators.crossover.CrossoverESDefault;
6  import eva2.optimization.operators.mutation.MutateESCovarianceMatrixAdaption;
7  import eva2.optimization.operators.terminators.EvaluationTerminator;
8  import eva2.optimization.problems.FM0Problem;
9  import eva2.optimization.strategies.EvolutionStrategies;
10 import eva2.optimization.modules.GOParameters;
11
12 public class TestingPlusCmaEs {
13
14     public static void main(String[] args) {
15         // a simple bimodal target function, two optima near (1.7,0) and (-1.44/0)
16         FM0Problem fm0 = new FM0Problem();
17         AbstractEAIndividual bestIndy;
18         // create standard ES parameters
19         GOParameters esParams = OptimizerFactory.standardES(fm0);
20         esParams.setTerminator(new EvaluationTerminator(2000));
21         // set a random seed based on system time
22         esParams.setSeed(0);
23
24         // set evolutionary operators and probabilities
25         AbstractEAIndividual.setOperators(
26                 fm0.getIndividualTemplate(),
27                 new MutateESCovarianceMatrixAdaption(true), 0.9,
28                 new CrossoverESDefault(), 0.1);
29
30         // access the ES
31         EvolutionStrategies es = (EvolutionStrategies) esParams.getOptimizer();
32         // set a (1+5) selection strategy
33         es.setMu(1);
34         es.setLambda(5);
35         es.setPlusStrategy(true);
36
37         // run optimization and retrieve winner individual
38         bestIndy = (AbstractEAIndividual) OptimizerFactory.optimizeToInd(esParams, null);
39         System.out.println(esParams.getTerminator().lastTerminationMessage() + "\nFound solution: "
40                 + AbstractEAIndividual.getDefaultDataString(bestIndy));
41     }
42 ;
43 }
```

The FM0Problem is of course very simple, 2-dimensional and having only two optima in the defined range. For harder problems, a few thousand evaluations will not suffice, and for highly multi-modal target functions, e.g. if there are thousands or tens of thousands of local optima, things get really tough. The current implementation of CBN is able to find more optima than the population size defined, because it is able to reinitialize a converged cluster during a run, saving a representative to an archive. But for problems with a lot of deceptive optima, it might also be a good strategy to concentrate on finding one global optimum, and, as it won't be found in most of the runs, look at the results of the single-run solution set.

Algorithm 5 Solving a bimodal function with CBN and post-processing.

```
1  package eva2.examples;
2  import eva2.OptimizerFactory;
3  import eva2.optimization.individuals.AbstractEAIndividual;
4  import eva2.optimization.operators.postprocess.PostProcessParams;
5  import eva2.optimization.operators.terminators.EvaluationTerminator;
6  import eva2.optimization.problems.FM0Problem;
7  import eva2.optimization.modules.GOParameters;
8  import java.util.List;
9
10 public class TestingCbnPostProc {
11         public static void main(String[] args) {
12                 // a simple bimodal target function, two optima near (1.7,0) and (-1.44/0)
13                 FM0Problem fm0 = new FM0Problem();
14                 AbstractEAIndividual best;
15                 List<AbstractEAIndividual> ppSols;
16
17                 GOParameters esParams = OptimizerFactory.standardCbnES(fm0);
18                 esParams.setTerminator(new EvaluationTerminator(2000));
19                 esParams.setSeed(0);
20                 best = (AbstractEAIndividual)OptimizerFactory.optimizeToInd(esParams, null);
21
22                 System.out.println(esParams.getTerminator().lastTerminationMessage() + "\nFound solution: "
23                                 + AbstractEAIndividual.getDefaultDataString(best));
24
25                 // post-process with clustering only
26                 ppSols = OptimizerFactory.postProcessIndVec(new PostProcessParams(0, 0.1, 5));
27                 System.out.println("After clustering: ");
28                 for (AbstractEAIndividual indy : ppSols) {
29                         System.out.println(AbstractEAIndividual.getDefaultDataString(indy));
30                 }
31
32                 // post-process with clustering and hill climbing
33                 ppSols = OptimizerFactory.postProcessIndVec(new PostProcessParams(1000, 0.1, 5));
34                 System.out.println("After clustering and local refinement: ");
35                 for (AbstractEAIndividual indy : ppSols) {
36                         System.out.println(AbstractEAIndividual.getDefaultDataString(indy));
37                 }
38         };
39 }
```

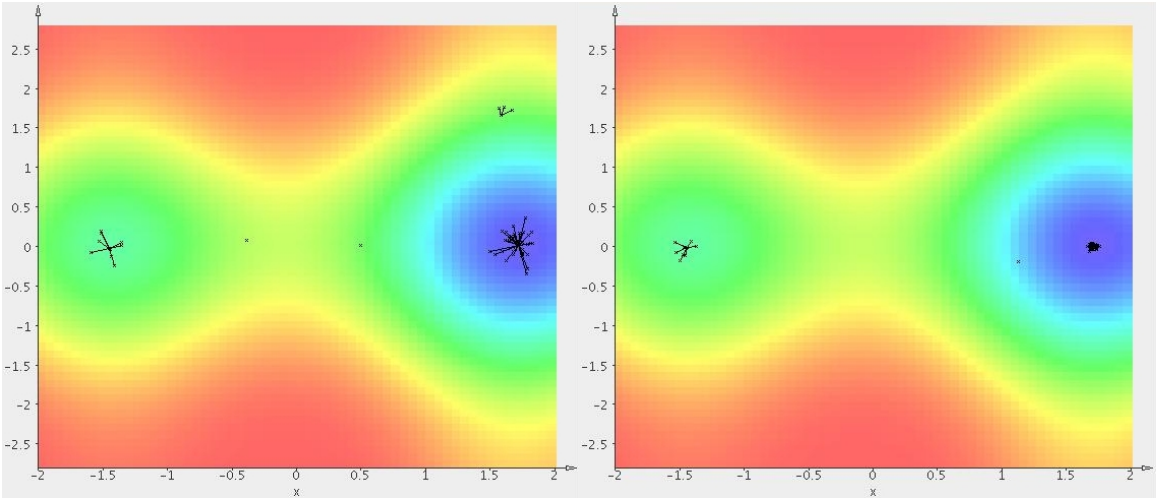

FIGURE 4.3.1. Exemplary states of CBN on the simple bimodal FM0Problem.

CHAPTER 5

Further Reading

As noted before, this introduction does not cover algorithmic basics. We recommend the introduction of Engelbrecht [**3**] for newer strategies (DE, PSO) and Bäck et al. [**2**] for classical Evolutionary Algorithms. The applications note from 2010 gives some short and simple use cases for the framework [**5**]. To go into details of EvA2, a look on the technical report of 2005 [**8**], though slightly outdated, will still be helpful, as well as the thesis of Felix Streichert [**6**], one of the main authors of JavaEvA.

Bibliography

[1] EvA2 Project Homepage. `http://www.ra.cs.uni-tuebingen.de/software/EvA2`. Accessed in February 2011.

[2] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, 1999.

[3] Andries Engelbrecht. *Computational Intelligence: An Introduction, 2nd Edition*. Halsted Press/Wiley, New York, NY, USA, 2007.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xu Xiaowei. A density-based algorithm for discovering clusters in large spatial databases with noise. In *2nd International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.

[5] Marcel Kronfeld, Hannes Planatscher, and Andreas Zell. The EvA2 optimization framework. In Christian Blum and Roberto Battiti, editors, *Learning and Intelligent Optimization Conference, Special Session on Software for Optimization (LION-SWOP)*, number 6073 in Lecture Notes in Computer Science, LNCS, pages 247–250, Venice, Italy, January 2010. Springer Verlag.

[6] Felix Streichert. *Evolutionary Algorithms in Multi-Modal and Multi-Objective Environments*. PhD thesis, University of Tübingen, Germany, 2007.

[7] Felix Streichert, Gunnar Stein, Holger Ulmer, and Andreas Zell. A clustering based niching method for evolutionary algorithms. In *Evolution Artificielle, Proceedings of the 6th International Conference (EA 2003)*, pages 293–304, Marseille, France, 2003. Springer.

[8] Felix Streichert and Holger Ulmer. JavaEvA - a java based framework for evolutionary algorithms. Technical Report WSI-2005-06, Universitätsbibliothek Tübingen, Germany, 2005.

[9] Jürgen Wakunda and Andreas Zell. EvA - a tool for optimization with evolutionary algorithms. In *Proceedings of the 23rd EUROMICRO Conference*, Budapest, Hungary, 1997.