# jMetal 4.3 User Manual

Antonio J. Nebro, Juan J. Durillo

*January 3, 2013*

# Contents

# List of Figures

# List of Tables

# Preface

This document contains the manual of jMetal, a framework for multi-objective optimization developed in the Computer Science Department of the University of Málaga.

The jMetal project began in 2006 with the idea of writing, from a former C++ package, a Java tool to be used in our research in multi-objective optimization with metaheuristics. We decided to put the package publicly available in November 2006, and it was allocated at SourceForge in November 2008 (`http://jmetal.sourceforge.net`). jMetal is an open source software, and it can be downloaded from `http://sourceforge.net/projects/jmetal`; as of today, it has been downloaded more than 6900 times.

Two versions of jMetal written in different languages are work in progress:

- jMetalCpp (`http://jmetalcpp.sourceforge.net`). This version is coded in C++ and it is available since February 2012. It implements about 70% of the Java implementation.
- jMetal.Net (`http://jmetalnet.sourceforge.net/`), which is implemented in C#. Several pre-releases are available since June 2011. The last release covers about 10% of the original Java version.

This manual covers the Java version and it is structured into eight chapters, covering issues such as installation, architecture description, examples of use, parallelism, a how-to's section, and a summary of versions and release notes.

# Chapter 1

# Overview

jMetal stands for *Metaheuristic Algorithms in Java*, and it is an object-oriented Java-based framework aimed at multi-objective optimization by using metaheuristics. jMetal provides a rich set of classes which can be used as the building blocks of multi-objective techniques; this way, by taking advantage of code-reusing, the algorithms share the same base components, such as implementations of genetic operators and density estimators, thus facilitating not only the development of new multi-objective techniques but also to carry out different kinds of experiments. The inclusion of a number of classical and state-of-the-art algorithms, many problems usually included in performance studies, and a set of quality indicators allow not only newcomers to study the basic principles of multi-objective optimization with metaheuristics but their application to solve real-world problems.

The jMetal project is continuously evolving. As we are researchers, not a software company, new versions are released when we require new features to be added into the software to carry out our research activities.

## 1.1 Motivation

When we started to work in metaheuristics for multi-objective optimization in 2004, we did not find any software package satisfying our needs. The implementation in C of NSGA-II, the most used multi-objective algorithm, publicly available[1], was difficult to be used as the basis of new algorithms, in part due to its lack of an object-oriented design. An interesting choice was (and still is) PISA [1], a C-based framework for multi-object optimization which is based on separating the algorithm specific part of an optimizer from the application-specific part. This is carried out by using a shared-file mechanism to communicate the module executing the application with the module running the metaheuristic. A drawback of PISA is that their internal design hinders to reuse code. From our point of view (we are computer science engineers), it became clear that it should be easier to develop our own tool starting from scratch that working with existing software. The result is the Java-based framework jMetal.

When we started to use jMetal in our research, we decided to make it available to the community of people interested in multi-objective optimization. It is licensed under the GNU Lesser General Public License, and it can be obtained freely from `http://jmetal.sourceforge.net`. During the development of jMetal, other Java-based software tools have been offered by other groups (e.g., EVA2[2], ECJ[3], OPT4J[4]). All these toolboxes can be useful enough for many researchers but, while jMetal is specifically oriented to multi-objective optimization with metaheuristics, most of existing frameworks are focused mainly on evolutionary algorithms, and many of them are centered in single-objective optimization, offering extensions to the multi-objective domain.

[1] NSGA-II: `http://www.iitk.ac.in/kangal/codes.shtml`
[2] EVA2: `http://www.ra.cs.uni-tuebingen.de/software/EvA2/`
[3] ECJ: `http://www.cs.gmu.edu/ eclab/projects/ecj/`
[4] `OPT4J: http://opt4j.sourceforge.net/`

## 1.2 Design goals

We imposed ourselves as design goals that jMetal should be simple and easy to use, portable (hence the choice of Java), flexible, and extensible. We detail these goals next:

- **Simplicity and easy-to-use**. These are the key goals: if they are not fulfilled, few people will use the software. The classes provided by jMetal follows the principle of that each component should only do one thing, and do it well. Thus, the base classes (`SolutionSet`, `Solution`, `Variable`, etc.) and their operations are intuitive and, as a consequence, easy to understand and use. Furthermore, the framework includes the implementation of many metaheuristics, which can be used as templates for developing new techniques.

- **Flexibility**. This is a generic goal. On the one hand, the software must incorporate a simple mechanism to execute the algorithms under different parameter settings, including algorithm-specific parameters as well as those related to the problem to solve. On the other hand, issues such as choosing a real or binary-coded representation and, accordingly, the concrete operators to use, should require minimum modifications in the programs.

- **Portability**. The framework and the algorithms developed with it should be executed in machines with different architectures and/or running distinct operating systems. The use of Java as programming language allows to fulfill this goal; furthermore, the programs do not need to be re-compiled to run in a different environment.

- **Extensibility**. New algorithms, operators, and problems should be easily added to the framework. This goal is achieved by using some mechanisms of Java, such as inheritance and late binding. For example, all the MOPs inherits from the class `Problem`, so a new problem can be created just by writing the methods specified by that class; once the class defining the new problem is compiled, nothing more has to be done: the late binding mechanism allows to load the code of the MOP only when this is requested by an algorithm. This way, jMetal allows to separate the algorithm-specific part from the application-specific part.

## 1.3 Summary of Features

A summary of jMetal main features is the following:

- Implementation of a number of classic and modern multi-objective optimization algorithms: NSGA-II [5], SPEA2 [38], PAES [17], PESA-II [2], OMOPSO [30], MOCell [25], AbYSS [27], MOEA/D [21], Densea [14], CellDE [10], GDE3 [18], FastPGA [12], IBEA [41], SMPSO [23], MOCHC [22], SMS-EMOA [11], SELF (work in progress).

- A rich set of test problems including:

  - Problem families: Zitzler-Deb-Thiele (ZDT) [37], Deb-Thiele-Laumanns-Zitzler (DTLZ) [4], Walking-Fish-Group (WFG) test problems [15]), CEC2009 (unconstrained problems) [35], and the Li-Zhang benchmark [21].
  - Classical problems: Kursawe [20], Fonseca [13], Schaffer [31].
  - Constrained problems: Srinivas[32], Tanaka [33], Osyczka2 [28], Constr_Ex [5], Golinski [19], Water [29].
  - Combinatorial problems: MOTSP.

- Implementation of a number of widely used quality indicators: Hypervolume [39], Spread [5], Generational Distance [34], Inverted Generational Distance [34], Epsilon [16].

- Different variable representations: binary, real, binary-coded real, integer, permutation.

- Validation of the implementation: we compared our implementations of NSGA-II and SPEA2 with the original versions, achieving competitive results [8].
- Support for performing experimental studies, including the automatic generation of
  - LaTeX tables with the results after applying quality indicators,
  - LaTeX tables summarizing statistical pairwise comparisons by using the Wilcoxon test to the obtained results, and
  - R (`http://www.r-project.org/`) boxplots summarizing those results.

  In addition, jMetal includes the possibility of using several threads for performing these kinds of experiments in such a way that several independent runs can be executed in parallel by using modern multi-core CPUs.
- A Web site (`http://jmetal.sourceforge.net`) containing the source codes, the user manual and, among other information, the Pareto fronts of the included MOPs, references to the implemented algorithms, and references to papers using jMetal.

## 1.4 Reference papers

If you want to cite jMetal, please use these references [9][6]:

```
@article{DN11,
author = "J. J. Durillo and A. J. Nebro",
title = "{jMetal}: A Java framework for multi-objective optimization",
journal = "Advances in Engineering Software",
volume = "42",
number = "10",
pages = " 760-771 ",
year = "2011",
note = "",
issn = "0965-9978",
doi = "DOI: 10.1016/j.advengsoft.2011.05.014",
url = "http://www.sciencedirect.com/science/article/pii/S0965997811001219",
}

@inproceedings{DNA10,
Address = {Barcelona, Spain},
Author = { J.J. Durillo and A.J. Nebro and E. Alba  },
Booktitle = {CEC 2010},
Month = {July},
Pages = {4138-4325},
OPTPublisher = {Springer Berlin / Heidelberg},
OPTSeries = {Lecture Notes in Computer Science},
Title = {The {jMetal} Framework for Multi-Objective Optimization: Design and Architecture},
OPTVolume = {5467},
Year = {2010}}
```

## 1.5 License

jMetal is licensed under the Creative Commons GNU Lesser General Public License License[5]

[5] `http://creativecommons.org/licenses/LGPL/2.1/`

# Chapter 2

# Installation

jMetal is written in Java, not requiring any other additional software. The requirement is to use Java JDK 1.5 or newer. The source code is bundled in a tar.gz package which can be download from SourceForge[1]. The jMetal Web page at SourceForge is: `http://jmetal.sourceforge.net`.

There exist several ways to work with Java programs; we briefly describe here how to compile and run algorithms developed with jMetal by using the command line in a text terminal, Netbeans[2], and Eclipse[3].

## 2.1 Unpacking the sources

Independently of your favorite way of working with Java, you have to decompress the tar.gz package and untar the resulting tarball. Using the command line, this can be done by typing:

```
gzip -d jmetal.tar.gz
tar xf jmetal.tar
```

Alternatively, you can type:

```
tar zxf jmetal.tar.gz
```

As a result, you will get a directory called `trunk` containing the jMetal package. Let us call this directory `JMETALHOME`.

## 2.2 Command line

If you intend to use jMetal from a text based terminal, please follow the following steps. We assume that you are using a bash shell in a Unix-like environment (e.g, Linux, MacOS X, or Cywgin under Windows).

### 2.2.1 Setting the environment variable `CLASSPATH`

To add directory `JMETALHOME` to the environment variable `CLASSPATH`, type:

```
export CLASSPATH=$CLASSPATH:$JMETALHOME
```

[1] `http://sourceforge.net/projects/jmetal`
[2] `http://www.netbeans.org/`
[3] `http://www.eclipse.org/`

### 2.2.2 Compiling the sources

Move to directory `JMETALHOME` and compile the sources. There are several ways to do that; we detail one of them:
**STEP 1**. Compile the problems

```
javac jmetal/problems/*.java
javac jmetal/problems/ZDT/*.java
javac jmetal/problems/DTLZ/*.java
javac jmetal/problems/WFG/*.java
```

**STEP 2**. Compile the algorithms

```
javac jmetal/metaheuristics/nsgaII/*.java
javac jmetal/metaheuristics/paes/*.java
javac jmetal/metaheuristics/spea2/*.java
javac jmetal/metaheuristics/mopso/*.java
javac jmetal/metaheuristics/mocell/*.java
javac jmetal/metaheuristics/abyss/*.java
```

Of course, you do not need to compile all of them; choose only those you are interested in.

### 2.2.3 Configuring and executing an algorithm

Let us suppose that we intend to use NSGA-II to solve a multi-objective optimization problem. There are several ways to accomplish this:

1. Configuring the algorithm by editing the `NSGA_main.java` program (see Section 3.3).
2. By using the `jmetal.experiments` package (see Chapter 4).

Here, we briefly describe the first option, consisting in editing file `NSGAII_main.java` belonging to the package `jmetal/metaheuristics/nsgaII`, recompiling, and executing it:

```
javac jmetal/metaheuristics/nsgaII/*.java
java jmetal.metaheuristics.nsgaII.NSGAII_main
```

As result, you will obtain to files: VAR, containing the values of the variables of the approximation set obtained, and FUN, which stores the corresponding values of the objective functions. Needless to say that you can change the names of these files by editing `NSGAII_main.java` .

## 2.3 Netbeans

We describe how to compile and use jMetal with NetBbeans 7.0.1.

### 2.3.1 Creating the project

1. Select *File* → *New Project*.
2. Choose *Java Project with Existing Sources* from the *General* category, and click the *Next* button.
3. Write a project name (e.g. jMetal) and choose the directory (folder) where you want to deploy the project. Check *Set as Main Project* and Click *Next*.
4. Click *Add Folder* to add the `JMETALHOME` directory to the source package folders. Click *Finish*

### 2.3.2 Configuring and executing an algorithm

We use as example the metaheuristic NSGA-II. To configure the algorithm, click in the *Files* tab in the IDE, and open the file *jMetal Source Packages* → *jmetal* → *metaheuristics* → *nsgaII* → *NSGAII_main.java*. Put the mouse pointer on the file name in the file tree to run the algorithm, click on the right button and choose *Run File*.

As a result, you obtain two files containing the Pareto optimal solutions and the Pareto front found by the metaheuristic. By default, these files are named VAR and FUN, respectively. They are located in the project folder.

## 2.4 Eclipse

We describe next how to compile and use jMetal using Eclipse 3.7.

### 2.4.1 Creating the project

1. Select *File* → *New* → *Java Project*.
2. Write a project name (e.g., jMetal) and click on the *Next* button.
3. Select *Link additional source* and browse to set `JMETALHOME` as linked folder location.
4. Click *Finish*.

### 2.4.2 Configuring and executing an algorithm

We use again NSGA-II as an example. To configure the algorithm, open the file `NSGAII_main.java` selecting it from the package `jmetal.metaheuristics.nsgaII` and modify the file accordingly to your preferences.

To run the algorithm, right click on `NSGAII_main.java` in the project tree or in blank part of the windows containing the file. Select *Run as* → *Java Application*. As a result, you obtain two files containing the Pareto optimal solutions and the Pareto front found by the algorithm. By default, these files are named VAR and FUN, respectively. They are located in the directory where the project workspace is stored (e.g., *$HOME/Documents/workspace/jmetal* in a Unix-based machine).

# Chapter 3

# Architecture

We use the Unified Modelling Language (UML) to describe the architecture and components of jMetal. A UML class diagram representing the main components and their relationships is depicted in Figure 3.1.

The diagram is a simplified version in order to make it understandable. The basic architecture of jMetal relies in that an `Algorithm` solves a `Problem` using one (and possibly more) `SolutionSet` and a set of `Operator` objects. We have used a generic terminology to name the classes in order to make them general enough to be used in any metaheuristic. In the context of evolutionary algorithms, populations and individuals correspond to `SolutionSet` and `Solution` jMetal objects, respectively; the same can be applied to particle swarm optimization algorithms concerning the concepts of swarm and particles.

## 3.1 Basic Components

In this section we describe the approaches taken in jMetal to implement solution encodings, operators, problems, and algorithms.

### 3.1.1 Encoding of Solutions

One of the first decisions that have to be taken when using metaheuristics is to define how to encode or represent the tentative solutions of the problem to solve. Representation strongly depends on the problem and determines the operations (e.g., recombination with other solutions, local search procedures, etc.) that can be applied. Thus, selecting a specific representation has a great impact on the behavior of metaheuristics and, hence, in the obtained results.

Fig. 3.2 depicts the basic components that are used for representing solutions into the framework. A `Solution` is composed of set of `Variable` objects, which can be of different types (binary, real, binary-coded real, integer, permutation, etc) plus an array to store the fitness values. With the idea of providing a flexible and extensible scheme, each `Solution` has associated a type (the `SolutionType` class in the figure). The solution type allows to define the variable types of the `Solution` and creating them, by using the `createVariables()` method. This is illustrated in Listing 3.1 which shows the code of the `RealSolutionType` class, used to characterize solutions composed only by real variables. jMetal provides similar solutions types to represent integer, binary, permutation, and other representations, as can be seen in Fig. 3.2.

The interesting point of using solution types is that it is very simple to define more complex representations, mixing different variable types. For example, if we need a new solution representation consisting in a real, an integer, and a permutation of integers, a new class extending `SolutionType` can be defined for representing the new type, where basically only the `createVariables()` method should be redefined. Listing 3.2 shows the code required for this new type of solution. This is explained in more detail in Section 6.3.

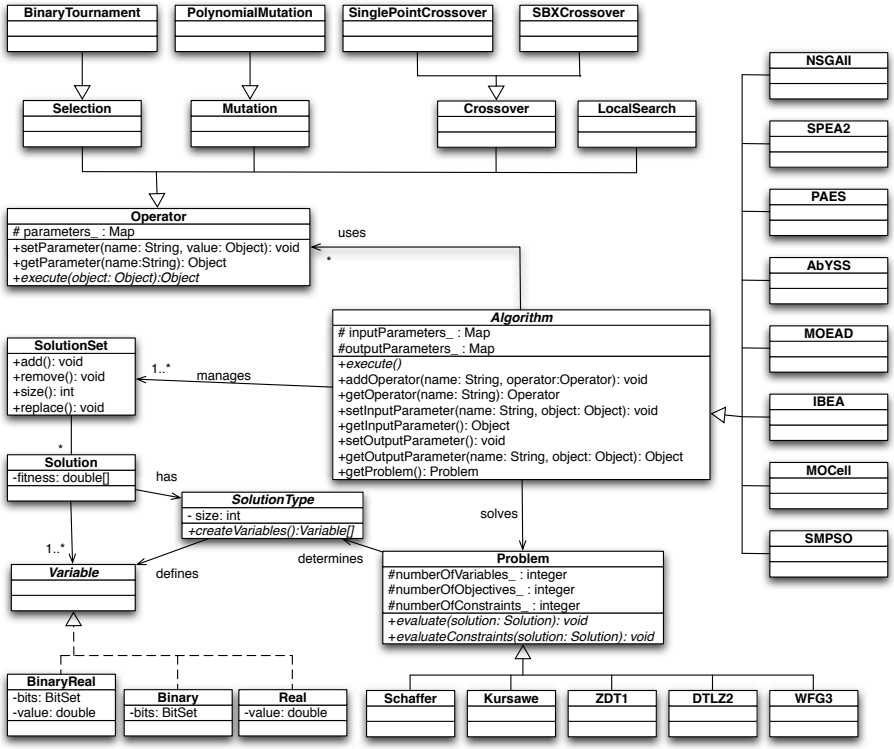


Figure 3.1: jMetal class diagram.

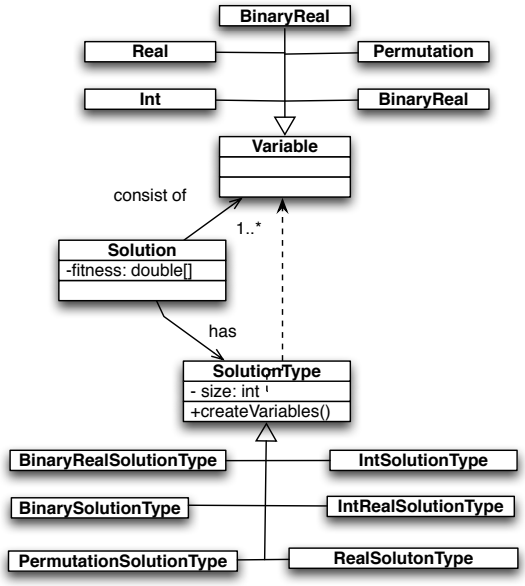


Figure 3.2: Elements describing solution representations into jMetal.

```java
//  RealSolutionType.java

package jmetal.encodings.solutionType;

import jmetal.core.Problem;
import jmetal.core.SolutionType;
import jmetal.core.Variable;
import jmetal.encodings.variable.Real;

/**
 * Class representing a solution type composed of real variables
 */
public class RealSolutionType extends SolutionType {

  /**
   * Constructor
   * @param problem
   * @throws ClassNotFoundException
   */
  public RealSolutionType(Problem problem) throws ClassNotFoundException {
    super(problem) ;
  } // Constructor

  /**
   * Creates the variables of the solution
   * @param decisionVariables
   */
  public Variable[] createVariables() {
    Variable[] variables = new Variable[problem_.getNumberOfVariables()];

    for (int var = 0; var < problem_.getNumberOfVariables(); var++)
      variables[var] = new Real(problem_.getLowerLimit(var),
          problem_.getUpperLimit(var));

    return variables ;
  } // createVariables
} // RealSolutionType
```

Listing 3.1: `RealSolutionType` class, which represents solutions composed of real variables

```java
  public Variable[] createVariables() {
    Variable[] variables = new Variable[3];

    variables[0] = new Real();
    variables[1] = new Int();
    variables[2] = new Permutation();

    return variables;
  } // createVariable
```

Listing 3.2: Code of the `createVariables()` method for creating solutions consisting on a Real, an Integer, and a Permutation

```
...
// Step 1: creating a map for the operator parameters
HashMap parameters = new HashMap() ;

// Step 2: configure the operator
parameters.put (``probability'', 0.9) ;
parameters.put (``distributionIndex'', 20.0) ;

// Step 3: create the operator
crossover = CrossoverFactory.getCrossoverOperator("SBXCrossover", parameters);

// Step 4: add the operator to an algorithm
algorithm.addOperator("crossover",crossover);
...
```

Listing 3.3: Configuring a SBX crossover operator.

Once we have the means to define or using existing solution representations, we can create solutions that can be grouped into `SolutionSet` objects (i.e., populations or swarms).

### 3.1.2 Operators

Metaheuristic techniques are based on modifying or generating new solutions from existing ones by means of the application of different operators. For example, EAs make use of crossover, mutation, and selection operators for modifying solutions. In jMetal, any operation altering or generating solutions (or sets of them) inherits from the `Operator` class, as can be seen in Fig. 3.1.

The framework already incorporates a number of operators, which can be classified into four different classes:

- *Crossover*. Represents the recombination or crossover operators used in EAs. Some of the included operators are the simulated binary (SBX) crossover [3] and the two-points crossover for real and binary encodings, respectively.
- *Mutation*. Represents the mutation operator used in EAs. Examples of included operators are polynomial mutation [3] (real encoding) and bit-flip mutation (binary encoding).
- *Selection*. This kind of operator is used for performing the selection procedures in many EAs. An example of selection operator is the binary tournament.
- *LocalSearch*. This class is intended for representing local search procedures. It contains an extra method for consulting how many evaluations have been performed after been applied.

Each operator contains the `setParameter()` and `getParameter()` methods, which are used for adding and accessing to an specific parameter of the operator. For example, the SBX crossover requires two parameters, a crossover probability (as most crossover operators) plus a value for the distribution index (specific of the operator), while a single point mutation operator only requires the mutation probability. The operators can also receive their parameters by passing them as an argument when the operator object is created.

It is worth noting that when an operator is applied on a given solution, the solution type of this one is known. Thus, we can define, for example, a unique two-points crossover operator that can be applied to binary and real solutions, using the solution type to select the appropriate code in each case.

To illustrate how operators are used and implemented in jMetal, let take as an example the SBX crossover operator. This has to parameters: the crossover probability and the distribution index. The way of creating and setting the operator is depicted in Listing 3.3. First, a Java `HashMap`, a map having

```
1  ...
2  public class MOMetaheuristic extends Algorithm {
3  ...
4  Operator crossover ;
5  crossover = operators_.get("crossover") ;
6  ...
7  Solution [] parents = new Solution[2] ;
8  parents[0] = (Solution) selectionOperator.execute(population);
9  parents[1] = (Solution) selectionOperator.execute(population);
10 Solution[] offSpring = (Solution[]) crossoverOperator.execute(parents);
11 ...
12 } // MOMetaheuristic
```

Listing 3.4: Using a SBX crossover operator inside an algorithm.

pairs (name, object), is created to store the operator parameters (line 3), which are set in lines 6-7; second, the operator is created (line 10) and, finally, it is added to an algorithm in line 13.

To make use of the operator inside a given algorithm, the following steps have to be carried out (see Listing 3.4). First, the algorithm must get the previously created operator (line 5), which is already to be used; second, the operator can be executed after invoking its `execute()` method with the corresponding parameters. In the case of the SBX crossover the parameters are two solutions previously obtained, typically after applying a selection operator (lines 8-9). Let us remark here that whenever a crossover operator is applied to a pair of solutions and the result is a pair another pair of solutions, the code in Listing 3.4 can remain as is, there is no need of made any modifications.

The implementation of the SBX crossover operator in jMetal is included in class `SBXCrossover` (see Listing 3.5). We can see that this class extends the `jmetal.operator.crossover` class (lines 3 and 5) and that the two parameters characterizing the operator(crossover probability and distribution index) are declared in lines 8 and 9. Let us pay attention now to lines 14-15. An operator can be applied to a given set of encodings, so the adopted approach is to indicate in a list the valid solution types. In the case of the SBX crossover, the operator is intended to `Real` and `ArrayReal` solution types, so they are included in the list called `VALID_TYPES`. Later, this list is used in the `execute()` method to check that the solutions to be combined have the correct representation.

The constructor (lines 19-26) merely gets the map received as argument and checks whether some of the parameters have to be set.

The `execute()` method receives as a parameter a generic Java `Object` (line 37), which must represent an array of two solutions, the parent solutions (line 38). We can see in lines 46-51 how the `VALID_TYPES` is used to check that the parent solutions have valid encodings. Finally, the method calls a function `doCrossover()` (line 54) which actually performs the crossover and returns an array with the two new generated solutions, with are the return object of the method (line 56).

### 3.1.3 Problems

In jMetal, all the problems inherits from class `Problem`. This class contains two basic methods: `evaluate()` and `evaluateConstraints()`. Both methods receive a `Solution` representing a candidate solution to the problem; the first one evaluates it, and the second one determines the overall constraint violation of this solution. All the problems have to define the `evaluate()` method, while only problems having side constraints need to define `evaluateConstraints()`. The constraint handling mechanism implemented by default is the one proposed in [5].

A key design feature in jMetal is that the problem defines the allowed solutions types that are suitable to solve it. Listing 3.6 shows the code used for implementing Kursawe's problem (irrelevant code is omitted). As we can observe observe, it extends class `Problem` (line 5). After that, a constructor method is defined for creating instances of this problem (lines 9-28), which has two parameters: a string containing a solution type identifier and the number of decision variables of the problem. As a general

```
1  //  SBXCrossover.java
2  ...
3  package jmetal.operators.crossover;
4  ...
5  public class SBXCrossover extends Crossover {
6    ...
7    public static final double ETA_C_DEFAULT_ = 20.0;
8    private Double crossoverProbability_ = null;
9    private double distributionIndex_ = ETA_C_DEFAULT_;
10
11   /**
12    * Valid solution types to apply this operator
13    */
14   private static List VALID_TYPES = Arrays.asList(RealSolutionType.class,
15                                                   ArrayRealSolutionType.class) ;
16  /**
17    * Constructor
18    */
19   public SBXCrossover(HashMap<String, Object> parameters) {
20     super (parameters) ;
21
22     if (parameters.get("probability") != null)
23       crossoverProbability_ = (Double) parameters.get("probability") ;
24     if (parameters.get("distributionIndex") != null)
25       distributionIndex_    = (Double) parameters.get("distributionIndex") ;
26   } // SBXCrossover
27
28   public Solution[] doCrossover(double probability,
29                                 Solution parent1,
30                                 Solution parent2) throws JMException {
31     ...
32   }
33
34  /**
35   * Executes the operation
36   */
37   public Object execute(Object object) throws JMException {
38     Solution [] parents = (Solution [])object;
39
40     if (parents.length != 2) {
41       Configuration.logger_.severe("SBXCrossover.execute: operator needs two " +
42           "parents");
43       ...
44     } // if
45
46     if (!(VALID_TYPES.contains(parents[0].getType().getClass())  &&
47           VALID_TYPES.contains(parents[1].getType().getClass())) ) {
48       Configuration.logger_.severe("SBXCrossover.execute: the solutions " +
49           "type " + parents[0].getType() + " is not allowed with this operator");
50        ...
51     } // if
52
53     Solution [] offSpring;
54     offSpring = doCrossover(crossoverProbability_, parents[0], parents[1]);
55
56     return offSpring;
57   } // execute
58 } // SBXCrossover
```

Listing 3.5: Implementation of the SBXCrossover operator.

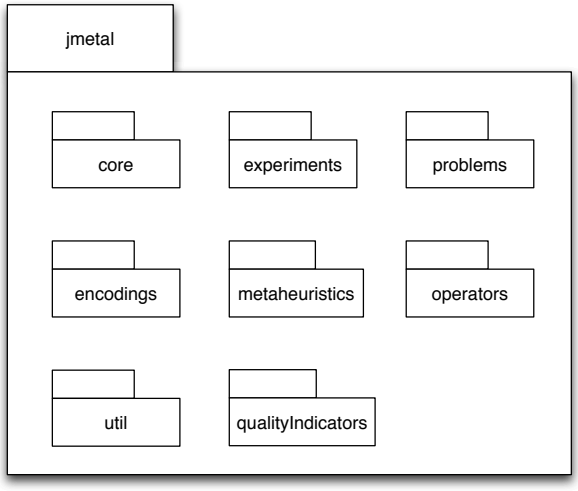


Figure 3.3: jMetal packages.

rule, all the problems should have as first parameter the string indicating the solution type. The basic features of the problem (number of variables, number of objectives, and number of constraints) are defined in lines 10-12. The limits of the values of the decision variables are set in lines 15-21. The sentences between lines 23-27 are used to specify that the allowed solution representations are binary-coded real and real, so the corresponding `SolutionType` objects are created and assigned to a state variable.

After the constructor, the `evaluate()` method is redefined (lines 33-43); in this method, after computing the two objective function values, they are stored into the solution by using the `setObjective` method of `Solution` (lines 41 and 42).

Many of commonly used benchmark problems are already included in jMetal. Examples are the ones proposed by Zitzler-Deb-Thiele (ZDT) [37], Deb-Thiele-Laumanns-Zitzler (DTLZ) [4], Walking-Fish-Group (WFG) test problems [15]), and the Li-Zhang benchmark [21].

### 3.1.4 Algorithms

The last core class in the UML diagram in Fig. 3.1 to comment is `Algorithm`, an abstract class which must be inherited by the metaheuristics included in the framework. In particular, the abstract method `execute()` must be implemented; this method is intended to run the algorithm, and it returns as a result a `SolutionSet`.

An instance object of `Algorithm` may require some application-specific parameters, that can be added and accessed by using the methods `addParameter()` and `getParameter()`, respectively. Similarly, an algorithm may also make use of some operators, so methods for incorporating operators (`addOperator()`) and to get them (`getOperator()`) are provided. A detailed example of algorithm can be found in Section 3.3, where the implementation of NSGA-II is explained.

Besides NSGA-II, jMetal includes the implementation of a number of both classic and modern multi-objective optimizers; some examples are: SPEA2 [38], PAES [17], OMOPSO [30], MOCell [24], AbYSS [27], MOEA/D [21], GDE3 [18], IBEA [41], or SMPSO [23].

## 3.2 jMetal Package Structure

jMetal is composed of six packages, which are depicted in Figure 3.3. The packages are `core`, `problems`, `metaheuristics`, `operators`, `encodings`, `qualityIndicators`, `util`, and `experiments` . We briefly describe them next:

```
1  //  Kursawe.java
2  ...
3  package jmetal.problems;
4  ...
5  public class Kursawe extends Problem {
6    /**
7     * Constructor.
8     */
9    public Kursawe(String solutionType, Integer numberOfVariables) throws
         ClassNotFoundException {
10     numberOfVariables_   = numberOfVariables.intValue() ;
11     numberOfObjectives_  = 2                             ;
12     numberOfConstraints_ = 0                             ;
13     problemName_         = "Kursawe"                     ;
14
15     upperLimit_ = new double[numberOfVariables_] ;
16     lowerLimit_ = new double[numberOfVariables_] ;
17
18     for (int i = 0; i < numberOfVariables_; i++) {
19       lowerLimit_[i] = -5.0 ;
20       upperLimit_[i] = 5.0  ;
21     } // for
22
23     if (solutionType.compareTo("BinaryReal") == 0)
24       solutionType_ = new BinaryRealSolutionType(this) ;
25     else if (solutionType.compareTo("Real") == 0)
26       solutionType_ = new RealSolutionType(this) ;
27     }
28   } // Kursawe
29
30   /**
31   * Evaluates a solution
32   */
33   public void evaluate(Solution solution) throws JMException {
34     double [] fx = new double[2] ; // function values
35
36     fx[0] = // f1 value
37     ...
38     fx[1] = // f2 value
39     ...
40
41     solution.setObjective(0, fx[0]);
42     solution.setObjective(1, fx[1]);
43   } // evaluate
44 } // Kursawe
```

Listing 3.6: Code of the class implementing problem Kursawe

- Package `core`: This package contains the basic ingredients to be used by the metaheuristics developed under jMetal. The main classes in this package have been commented in Section 3.1, which are: `Algorithm`, `Operator`, `Problem`, `Variable`, `Solution`, `SolutionSet`, and `SolutionType`. This package was named `jmetal.base` in versions prior to jMetal 4.0.

- Package `problems`: All the problems available in jMetal are included in this package. Here we can find well-known benchmarks (ZDT, DTLZ, and WFG) plus other more recent problem families (LZ07, CEC2009Competition). Furthermore, we can find many other problems (Fonseca, Kursawe, Schaffer, OKA2, etc.).

- Package `metaheuristics`: This package contains the metaheuristics implemented in jMetal. The list of techniques include NSGA-II, SPEA2, PAES, PESA-II, GDE3, FastPGA, MOCell, AbYSS, OMOPSO, IBEA, and MOEA/D. Although jMetal is aimed at multi-objective optimization, a number of single objective algorithms are included in the `jmetal.metaheuristics.singleObjective` package.

- Package `jmetal.operators`: This package contains different kinds of operator objects, including crossover, mutation, selection, and local search operators. We give next an example of an operator of each type:

  - `jmetal.operators.crossover.SBXCrossover`: This comparator takes also two solutions $S_1$ and $S_2$ and performs a simulated binary (SBX) crossover, returning as a result the two obtained offsprings.
  - `jmetal.operators.mutation.Polynomial`: Mutation operators typically are applied to single solutions, modifying them accordingly, and they return the mutated solution. In this case, the operator is a polynomial mutation.
  - `jmetal.operators.selection.BinaryTournament`: Selection comparators usually take as a parameter a solution set, returning a solution according to a criterium. In particular, this operator applies a binary tournament.
  - `jmetal.operators.localSearch.MutationLocalSearch`: These operators are intended to apply local search strategies to a given solution. The `MutationLocalSearch`, used in the AbYSS algorithm [27], requires as parameters a solution, a mutation operator, an integer $N$, and a `jmetal.base.archive` object; then the mutation operator is applied iteratively to improve the solution during $N$ rounds and the archive is used to store the found non-dominated solutions.

- Package `jmetal.encodings`: This package contains the basic variable representations and the solution types included in the framework. In jMetal version 4.0 the following clases are included:

  - Variables: `Binary`, `BinaryReal`, `BinaryReal` (binary coded real), `Int`, `Permutation`, `ArrayInt`, and `ArrayReal`.
  - Solution types: `BinarySolutionType`, `RealSolutionType`, `BinaryRealRealSolutionType`, `IntSolutionType`, `PermutationSolutionType`, `ArrayRealSolutionType`, `ArrayIntSolutionType`, `IntRealSolutionType` (combines `Int` and `Real` variables), and `ArrayRealAndBinarySolutionType` (combines `ArrayReal` and `Binary` variables).

- Package `util`: A number of utilities classes are included in this class, as a pseudorandom number generator, different types of archive, a neighborhood class to be used in cellular evolutionary algorithms, comparators, etc.

- Package `qualityIndicator`: To assess the performance of multi-objective metaheuristics, a number of quality indicators can be applied. The package contains currently six indicators:

  - Generational distance [34]

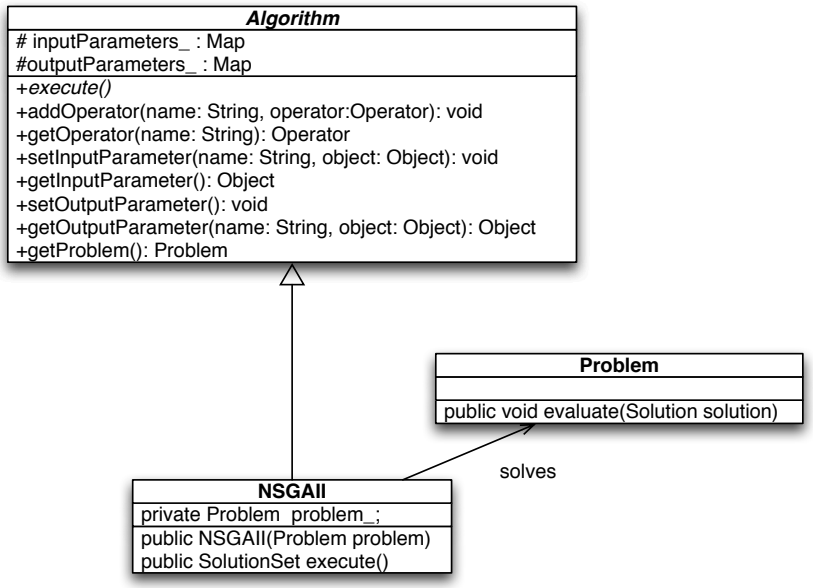


Figure 3.4: UML diagram of `NSGAII`.

  - Inverted generational distance [34]
  - Additive epsilon [40]
  - Spread [5]
  - Generalized spread [36]
  - Hypervolume [39]

- Package `experiments`: This package contains a set of classes intended to carry out typical studies in multi-objective optimization. It is described in Chapter 4.

## 3.3 Case Study: NSGA-II

In this section, we describe the implementation of NSGA-II in jMetal. Under jMetal, a metaheuristic is composed of a class defining the algorithm itself and another class to execute it. This second class is used to specify the problem to solve, the operators to apply, the parameters of the algorithm, and whatever other parameters need to be set (since jMetal 2.0, we have introduced an alternative way, by using the package `jmetal.experiments`, as explained in Chapter 4). Let us call this two classes `NSGAII` and `NGAII_main`, respectively.

### 3.3.1 Class `NSGAII.java`

The UML diagram of the `NSGAII` class is depicted in Figure 3.4. As every metaheuristic developed in jMetal, `NSGAII` inherits from `Algorithm`. This class has an abstract method, `execute()`, that is called to run the algorithm and returns as a result a `SolutionSet` (typically, a population or archive containing the obtained approximation set). We can see that new operators can be added to an algorithm with the method `addOperation()`; these operations are accessed in the algorithm by invoking `getOperation()`. Similarly, we can pass parameters to an algorithm (methods `setInputParameter()` and `getInputParameter`), and an algorithm can return output results via `setOutputParemeters()` and `getOutputParameters`. `NSGAII` has a constructor which receives the problem to solve as a parameter,

```
1  //  NSGAII.java
2
3  package jmetal.metaheuristics.nsgaII;
4
5  import jmetal.core.*;
6
7  /**
8   * This class implements the NSGA-II algorithm.
9   */
10 public class NSGAII extends Algorithm {
11
12   /**
13    * Constructor
14    * @param problem Problem to solve
15    */
16   public NSGAII(Problem problem) {
17     super (problem) ;
18   } // NSGAII
19
20   /**
21    * Runs the NSGA-II algorithm.
22    */
23   public SolutionSet execute() {
24   } // execute
25 } // NSGA-II
```

Listing 3.7: Scheme of the implementation of class `NSGAII`

as well as the implementation on `execute()`. Next, we analyze the implementation of the `NSGAII` class in jMetal (file `jmetal/metaheuristics/nsgaII/NSGAII.java`). The basic code structure implementing the class is presented in listing 3.7.

Let us focus on the method `execute()` (see Listing 3.8). First, we comment the objects needed to implement the algorithm. The parameters to store the population size and the maximum number of evaluations are declared in lines 2-3. The next variable, `evaluations`, is a counter of the number of computed evaluations. The objects declared in lines 6-7 are needed to illustrate the use of quality indicators inside the algorithms; we will explain their use later; lines 10-12 contain the declaration of the populations needed to implement NSGA-II: the current population, an offspring population, and an auxiliary population used to join the other two. Next, we find the three genetic operators (lines 14-16) and a `Distance` object (from package `jmetal.util`), which will be used to calculate the crowding distance.

Once we have declared all the needed objects, we proceed to initialize them (Listing 3.9). The parameters `populationSize` and `maxEvaluations` are input parameters whose values are obtained in lines 23-24; the same applies to `indicators`, although this parameter is optional (the other two are required). The population and the counter of evaluations are initialized next (lines 28-29), and finally the mutation, crossover, and selection operators are obtained (lines 34-36).

The initial population is initialized in the loop included in Lisrting 3.10. We can observe how new solutions are created, evaluated, and inserted into the population.

The main loop of the algorithm is included in the piece of code contained in Listing 3.11. We can observe the inner loop performing the generations (lines 55-71), where the genetic operators are applied. The number of iterations of this loop is `populationSize/2` because it is assumed that the crossover returns two solutions; in the case of using a crossover operator returning only one solution, the sentence in line 55 should be modified accordingly.

After the offspring population has been filled, the next step in NSGA-II is to apply ranking and crowding to the union of the current and offspring populations to select the new individuals in the next generation. The code is included in Listing 3.12, which basically follows the algorithm described in [5].

```
1  public SolutionSet execute() throws JMException, ClassNotFoundException {
2    int populationSize;
3    int maxEvaluations;
4    int evaluations;
5
6    QualityIndicator indicators; // QualityIndicator object
7    int requiredEvaluations; // Use in the example of use of the
8                             // indicators object (see below)
9
10   SolutionSet population;
11   SolutionSet offspringPopulation;
12   SolutionSet union;
13
14   Operator mutationOperator;
15   Operator crossoverOperator;
16   Operator selectionOperator;
17
18   Distance distance = new Distance();
19   ...
```

Listing 3.8: `execute()` method: declaring objects.

```
20   ...
21
22   //Read the parameters
23   populationSize = ((Integer) getInputParameter("populationSize")).intValue();
24   maxEvaluations = ((Integer) getInputParameter("maxEvaluations")).intValue();
25   indicators = (QualityIndicator) getInputParameter("indicators");
26
27   //Initialize the variables
28   population = new SolutionSet(populationSize);
29   evaluations = 0;
30
31   requiredEvaluations = 0;
32
33   //Read the operators
34   mutationOperator = operators_.get("mutation");
35   crossoverOperator = operators_.get("crossover");
36   selectionOperator = operators_.get("selection");
37   ...
```

Listing 3.9: `execute()` method: initializing objects.

```
38   ...
39   // Create the initial solutionSet
40   Solution newSolution;
41   for (int i = 0; i < populationSize; i++) {
42     newSolution = new Solution(problem_);
43     problem_.evaluate(newSolution);
44     problem_.evaluateConstraints(newSolution);
45     evaluations++;
46     population.add(newSolution);
47   } // for
48   ...
```

Listing 3.10: `execute()` method: initializing the population.

```
49      // Generations
50      while (evaluations < maxEvaluations) {
51
52        // Create the offSpring solutionSet
53        offspringPopulation = new SolutionSet(populationSize);
54        Solution[] parents = new Solution[2];
55        for (int i = 0; i < (populationSize / 2); i++) {
56          if (evaluations < maxEvaluations) {
57            //obtain parents
58            parents[0] = (Solution) selectionOperator.execute(population);
59            parents[1] = (Solution) selectionOperator.execute(population);
60            Solution[] offSpring = (Solution[]) crossoverOperator.execute(parents);
61            mutationOperator.execute(offSpring[0]);
62            mutationOperator.execute(offSpring[1]);
63            problem_.evaluate(offSpring[0]);
64            problem_.evaluateConstraints(offSpring[0]);
65            problem_.evaluate(offSpring[1]);
66            problem_.evaluateConstraints(offSpring[1]);
67            offspringPopulation.add(offSpring[0]);
68            offspringPopulation.add(offSpring[1]);
69            evaluations += 2;
70          } // if
71        } // for
```

Listing 3.11: `execute()` method: main loop.

The piece of code in Listing 3.13 illustrates the use of quality indicators inside a metaheuristic. In particular, it shows the code we used in [26] to study the convergence speed of multi-objective metaheuristics. As we commented before, if the `indicator` object was specified as input parameter (otherwise, it would be null - line 120), we apply it to test whether the hypervolume of the new population, at the end of each generation, is equal of greater than the 98% of the hypervolume of the true Pareto front (see [26] for further details). In case of success, the variable `requiredEvaluations` is assigned the current number of function evaluations (line 124). Once this variable is not zero, we do not need to carry out the test any more; that is the reason of including the condition in line 121.

The last sentences of the `execute()` method are included in Listing 3.14. In line 129 we can observe that the variable requiredEvaluations is returned as an output parameter. Finally, we apply ranking to the resulting population to return only non-dominated solutions (lines 132-133).

### 3.3.2 Class `NSGAII_main`

In this section we describe the `NSGAII_main.java` program, used to ejecute NSGA-II. The file is located in `jmetal/metaheuristics/nsgaII`, as it is indicated in line 22 in the piece of code included in Listing 3.15, which contains the import section of the program. The logging classes (lines 39-40) are needed to use a logger object, which allows us to log error messages.

The code in Listing 3.16 contains the declaration of the `main()` method. In the implementation we provide, there are three ways of invoking the program:

- `jmetal.metaheuristics.nsgaII.NSGAII_main`: the program is invoked without arguments. In this case, a default problem is solved.
- `jmetal.metaheuristics.nsgaII.NSGAII_main problemName`: this is the choice to indicate the problem to solve. The problem name must fit with those in the package `jmetal.problems` (e.g., Kursawe, ZDT4, DTLZ5, WFG1, etc.).
- `jmetal.metaheuristics.nsgaII.NSGAII_main problemName paretoFronFile`: If we provide a file containing the Pareto front of the problem to solve, a `QualityIndicator` object will be created,

```
72
73        // Create the solutionSet union of solutionSet and offSpring
74        union = ((SolutionSet) population).union(offspringPopulation);
75
76        // Ranking the union
77        Ranking ranking = new Ranking(union);
78
79        int remain = populationSize;
80        int index = 0;
81        SolutionSet front = null;
82        population.clear();
83
84        // Obtain the next front
85        front = ranking.getSubfront(index);
86
87        while ((remain > 0) && (remain >= front.size())) {
88          //Assign crowding distance to individuals
89          distance.crowdingDistanceAssignment(front, problem_.getNumberOfObjectives());
90          //Add the individuals of this front
91          for (int k = 0; k < front.size(); k++) {
92            population.add(front.get(k));
93          } // for
94
95          //Decrement remain
96          remain = remain - front.size();
97
98          //Obtain the next front
99          index++;
100         if (remain > 0) {
101           front = ranking.getSubfront(index);
102         } // if
103       } // while
104
105       // Remain is less than front(index).size, insert only the best one
106       if (remain > 0) {  // front contains individuals to insert
107         distance.crowdingDistanceAssignment(front, problem_.getNumberOfObjectives());
108         front.sort(new CrowdingComparator());
109         for (int k = 0; k < remain; k++) {
110           population.add(front.get(k));
111         } // for
112
113         remain = 0;
114       } // if
```

Listing 3.12: `execute()` method: ranking and crowding.

```
115       ...
116       // This piece of code shows how to use the indicator object into the code
117       // of NSGA-II. In particular, it finds the number of evaluations required
118       // by the algorithm to obtain a Pareto front with a hypervolume higher
119       // than the hypervolume of the true Pareto front.
120       if ((indicators != null) &&
121         (requiredEvaluations == 0)) {
122         double HV = indicators.getHypervolume(population);
123         if (HV >= (0.98 * indicators.getTrueParetoFrontHypervolume())) {
124           requiredEvaluations = evaluations;
125         } // if
126       } // if
```

Listing 3.13: `execute()` method: using the hyper volume quality indicator.

```
127     ...
128     // Return as output parameter the required evaluations
129     setOutputParameter("evaluations", requiredEvaluations);
130
131     // Return the first non-dominated front
132     Ranking ranking = new Ranking(population);
133     return ranking.getSubfront(0);
134   } // execute
```

Listing 3.14: `execute()` method: end of the method.

```
1  //  NSGAII_main.java
2  //
3  //  Author:
4  //       Antonio J. Nebro <antonio@lcc.uma.es>
5  //       Juan J. Durillo <durillo@lcc.uma.es>
6  //
7  //  Copyright (c) 2011 Antonio J. Nebro, Juan J. Durillo
8  //
9  //  This program is free software: you can redistribute it and/or modify
10 //  it under the terms of the GNU Lesser General Public License as published by
11 //  the Free Software Foundation, either version 3 of the License, or
12 //  (at your option) any later version.
13 //
14 //  This program is distributed in the hope that it will be useful,
15 //  but WITHOUT ANY WARRANTY; without even the implied warranty of
16 //  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 //  GNU Lesser General Public License for more details.
18 //
19 //  You should have received a copy of the GNU Lesser General Public License
20 //  along with this program.  If not, see <http://www.gnu.org/licenses/>.
21
22 package jmetal.metaheuristics.nsgaII;
23
24 import jmetal.core.*;
25 import jmetal.operators.crossover.*;
26 import jmetal.operators.mutation.*;
27 import jmetal.operators.selection.*;
28 import jmetal.problems.*                    ;
29 import jmetal.problems.DTLZ.*;
30 import jmetal.problems.ZDT.*;
31 import jmetal.problems.WFG.*;
32 import jmetal.problems.LZ09.* ;
33
34 import jmetal.util.Configuration;
35 import jmetal.util.JMException;
36 import java.io.IOException;
37 import java.util.* ;
38
39 import java.util.logging.FileHandler;
40 import java.util.logging.Logger;
41
42 import jmetal.qualityIndicator.QualityIndicator;
```

Listing 3.15: `NSGAII_main`: importing packages.

```
44 public class NSGAII_main {
45   public static Logger      logger_ ;      // Logger object
46   public static FileHandler fileHandler_ ; // FileHandler object
47
48   /**
49    * @param args Command line arguments.
50    * @throws JMException
51    * @throws IOException
52    * @throws SecurityException
53    * Usage: three options
54    *      - jmetal.metaheuristics.nsgaII.NSGAII_main
55    *      - jmetal.metaheuristics.nsgaII.NSGAII_main problemName
56    *      - jmetal.metaheuristics.nsgaII.NSGAII_main problemName paretoFrontFile
57    */
58   public static void main(String [] args) throws
```

Listing 3.16: `NSGAII_main`: main method.

and the program will calculate a number of quality indicator values at the end of the execution of the algorithm. This option is also a requirement to used quality indicators inside the algorithms.

Listing 3.17 contains the code used to declare the objects required to execute the algorithm (lines 59-63). The logger object is initialized in lines 69-72, and the log messages will be written in a file named "NSGAII_main.log". The sentences included between lines 74 and 92 process the arguments of the `main()` method. The default problem to solve is indicated after line 85. The key point here is that, at end of this block of sentences, an instance of the `Problem` class must be obtained. This is the only argument needed to create an instance of the algorithm, as we can see in line 94. The next line contains the sentence that should be used if we intend to use the steady-state version of NSGA-II which is also included in jMetal.

Once an object representing the algorithm to run has been created, it must be configured. In the code included in Listing 3.18, the input parameters are set in lines 97-98, the crossover and mutation operators are specified in lines 101-109, and the selection operator is chosen in line 113. Once the operators have been specified, they are added to the algorithm object in lines 116-118. The sentence in line 121 sets the indicator object as input parameter.

When the algorithm has been configured, it is executed by invoking its `execute()` method (line 125 in Listing 3.19). When it has finished, the running time is reported, and the obtained solutions and their objectives values are stored in two files (lines 131 and 133). Finally, if the indicator object is not null, a number of quality indicators are calculated (lines 136-141) and printed, as well as the number of evaluations returned by the algorithm as an output parameter.

```
59      Problem   problem   ; // The problem to solve
60      Algorithm algorithm ; // The algorithm to use
61      Operator  crossover ; // Crossover operator
62      Operator  mutation  ; // Mutation operator
63      Operator  selection ; // Selection operator
64
65      HashMap  parameters ; // Operator parameters
66
67      QualityIndicator indicators ; // Object to get quality indicators
68
69      // Logger object and file to store log messages
70      logger_      = Configuration.logger_ ;
71      fileHandler_ = new FileHandler("NSGAII_main.log");
72      logger_.addHandler(fileHandler_) ;
73
74      indicators = null ;
75      if (args.length == 1) {
76        Object [] params = {"Real"};
77        problem = (new ProblemFactory()).getProblem(args[0],params);
78      } // if
79      else if (args.length == 2) {
80        Object [] params = {"Real"};
81        problem = (new ProblemFactory()).getProblem(args[0],params);
82        indicators = new QualityIndicator(problem, args[1]) ;
83      } // if
84      else { // Default problem
85        problem = new Kursawe("Real", 3);
86        //problem = new Kursawe("BinaryReal", 3);
87        //problem = new Water("Real");
88        //problem = new ZDT1("ArrayReal", 100);
89        //problem = new ConstrEx("Real");
90        //problem = new DTLZ1("Real");
91        //problem = new OKA2("Real") ;
92      } // else
93
94      algorithm = new NSGAII(problem);
95      //algorithm = new ssNSGAII(problem);
```

Listing 3.17: `NSGAII_main`: declaring objects, processing the arguments of `main()`, and creating the algorithm.

```
96      // Algorithm parameters
97      algorithm.setInputParameter("populationSize",100);
98      algorithm.setInputParameter("maxEvaluations",25000);
99
100     // Mutation and Crossover for Real codification
101     parameters = new HashMap() ;
102     parameters.put("probability", 0.9) ;
103     parameters.put("distributionIndex", 20.0) ;
104     crossover = CrossoverFactory.getCrossoverOperator("SBXCrossover", parameters);
105
106     parameters = new HashMap() ;
107     parameters.put("probability", 1.0/problem.getNumberOfVariables()) ;
108     parameters.put("distributionIndex", 20.0) ;
109     mutation = MutationFactory.getMutationOperator("PolynomialMutation", parameters);
110
111     // Selection Operator
112     parameters = null ;
113     selection = SelectionFactory.getSelectionOperator("BinaryTournament2", parameters) ;
114
115     // Add the operators to the algorithm
116     algorithm.addOperator("crossover",crossover);
117     algorithm.addOperator("mutation",mutation);
118     algorithm.addOperator("selection",selection);
119
120     // Add the indicator object to the algorithm
121     algorithm.setInputParameter("indicators", indicators) ;
```

Listing 3.18: `NSGAII_main`: configuring the algorithm

```
122     ...
123     // Execute the Algorithm
124     long initTime = System.currentTimeMillis();
125     SolutionSet population = algorithm.execute();
126     long estimatedTime = System.currentTimeMillis() - initTime;
127
128     // Result messages
129     logger_.info("Total execution time: "+estimatedTime + "ms");
130     logger_.info("Variables values have been writen to file VAR");
131     population.printVariablesToFile("VAR");
132     logger_.info("Objectives values have been writen to file FUN");
133     population.printObjectivesToFile("FUN");
134
135     if (indicators != null) {
136       logger_.info("Quality indicators") ;
137       logger_.info("Hypervolume: " + indicators.getHypervolume(population)) ;
138       logger_.info("GD         : " + indicators.getGD(population)) ;
139       logger_.info("IGD        : " + indicators.getIGD(population)) ;
140       logger_.info("Spread     : " + indicators.getSpread(population)) ;
141       logger_.info("Epsilon    : " + indicators.getEpsilon(population)) ;
142
143       int evaluations = ((Integer)algorithm.getOutputParameter("evaluations")).intValue
              ();
144       logger_.info("Speed      : " + evaluations + " evaluations") ;
145     } // if
146   } //main
147 } // NSGAII_main
```

Listing 3.19: `NSGAII_main`: running the algorithms and reporting results

# Chapter 4

# Experimentation with jMetal

In our research work, when we want to assess the performance of a multi-objective metaheuristic, we usually compare it with other algorithms over a set of benchmark problems. After choosing the test suites and the quality indicators to apply, we carry out a number of independent runs of each experiments and after that we analyze the results.

Typically, we follow these steps:

1. Configure the algorithms by setting the parameter values in an associated `Settings` object (see Subsection 4.1).
2. Optionally, configure the problems to solve. For example, the DTLZ problems are configured by default with three objectives, while the WFG ones are bi-objective. If we want to modify these default settings, we have to do it by changing them in the files defining the problems.
3. Execute a number of independent runs per each par (algorithm, problem).
4. Analyze the results. jMetal can generate Latex tables and R scripts to present the results and to provide statistical information.

To carry out these steps, we use the `jmetal.experiments` package, first available in jMetal 2.0, and this chapter is devoted mainly to explaining the use of thas package. First, we describe the structure of the `jmetal.experiments.Settings` class and how it can be used to configure NSGA-II; then, we analyze the `jmetal.experiments.Main` class. Finally, we illustrate with two examples the use of the `jmetal.experiments.Experiment` class.

## 4.1 The `jmetal.experiments.Settings` Class

The motivation of designing this class has to do with the fact that in the traditional approach, a jMetal metaheuristic is executed through a `main` class, as `NSGAII_main` in the case of NSGA-II (see Section 3.3). This class contains the configuration of the algorithm so, if we want to run the metaheuristic with different parameter settings, we have to modify that file each time. This may become cumbersome, and it is a consequence of that, by using `main` objects, we cannot reuse the configurations of the algorithms in an easy way.

To face this issue, we took the alternative approach of defining the configuration of a metaheuristic in an object which will contain the default settings and will allow to modify them. Listing 4.1 contains the code of the `jmetal.experiment.Settings` class. The main features of this class are:

- Its state is represented by a `Problem` object (line 9), the problem name (line 10), and a string to store the file containing the true Pareto front of the problem if quality indicators are to be applied (line 11).

```
1  //  Settings.java
2  ...
3  package jmetal.experiments;
4  ...
5  /**
6   * Class representing Settings objects.
7   */
8  public abstract class Settings {
9    protected Problem problem_ ;
10   protected String problemName_ ;
11   public String paretoFrontFile_ ;
12
13   /**
14    * Constructor
15    */
16   public Settings() {
17   } // Constructor
18
19   /**
20     * Constructor
21    */
22   public Settings (String problemName) {
23     problemName_ = problemName ;
24   } // Constructor
25
26  /**
27   * Default configure method
28   * @return A problem with the default configuration
29   * @throws jmetal.util.JMException
30   */
31   abstract public Algorithm configure() throws JMException ;
32
33  /**
34   * Configure method. Change the default configuration
35   */
36   public final Algorithm configure(HashMap settings) throws JMException {
37     ...
38   } // configure
39
40  /**
41   * Changes the problem to solve
42   * @param problem
43   */
44   void setProblem(Problem problem) {
45     problem_ = problem ;
46   } // setProblem
47
48  /**
49   * Returns the problem
50   */
51   Problem getProblem() {
52     return problem_ ;
53   }
54 } // Settings
```

Listing 4.1: `Settings` class

- The problem can be set either when creating the object (lines 22-24), either by using the method `setProblem()` (lines 51-52).
- The default settings are stablished in the `configure()` method (line 31). This method must be defined in the corresponding subclasses of `Settings`.
- The values of the parameters can be modified by using a Java `HashMap` object, passing it as an argument to second definition of the `configure()` method (line 36).

## 4.2 An example of `Setting` class: NSGA-II

To illustrate the use of the `Settings` class, we analyze the `NSGAII_Settings` class, which is in the package `jmetal.experiments.settings`. The idea is simple: to move the parameter settings in `NSGAII_main` (see Section 3.3.2) to `NSGAII_Settings`. This is depicted in Listing 4.2, where the parameters to be set are declared in lines 9-14. The class constructor (lines 20-37), taking as argument the problem to be solved, creates an instance of the problem (lines 23-39) and assigns the default parameter values (lines 30-36). We impose the requirement of that the parameters have to the public and their name must end with the underscore ('_') character; the reason has to do with the mechanism to modify the settings, as is explained below.

```
1  //  NSGAII_Settings.java
2  ...
3  package jmetal.experiments.settings;
4  ...
5  /**
6   * Settings class of algorithm NSGA-II (real encoding)
7   */
8  public class NSGAII_Settings extends Settings {
9    public int populationSize_                 ;
10   public int maxEvaluations_                 ;
11   public double mutationProbability_         ;
12   public double crossoverProbability_        ;
13   public double mutationDistributionIndex_   ;
14   public double crossoverDistributionIndex_  ;
15
16   /**
17    * Constructor
18    * @throws JMException
19    */
20   public NSGAII_Settings(String problem) throws JMException {
21     super(problem) ;
22
23     Object [] problemParams = {"Real"};
24     try {
25       problem_ = (new ProblemFactory()).getProblem(problemName_, problemParams);
26     } catch (JMException e) {
27       // TODO Auto-generated catch block
28       e.printStackTrace();
29     }
30     // Default settings
31     populationSize_                 = 100   ;
32     maxEvaluations_                 = 25000 ;
33     mutationProbability_            = 1.0/problem_.getNumberOfVariables() ;
34     crossoverProbability_           = 0.9   ;
35     mutationDistributionIndex_      = 20.0  ;
36     crossoverDistributionIndex_     = 20.0  ;
37   } // NSGAII_Settings
38   ...
```

Listing 4.2: `jmetal.experiments.settings.NSGAII_Settings`: Default settings and constructor.

```
39   ...
40   /**
41    * Configure NSGAII with user-defined parameter settings
42    * @return A NSGAII algorithm object
43    * @throws jmetal.util.JMException
44    */
45   public Algorithm configure() throws JMException {
46     Algorithm algorithm ;
47     Selection  selection ;
48     Crossover  crossover ;
49     Mutation   mutation  ;
```

```
50
51      HashMap  parameters ; // Operator parameters
52
53      QualityIndicator indicators ;
54
55      // Creating the algorithm. There are two choices: NSGAII and its steady-
56      // state variant ssNSGAII
57      algorithm = new NSGAII(problem_) ;
58      //algorithm = new ssNSGAII(problem_) ;
59
60      // Algorithm parameters
61      algorithm.setInputParameter("populationSize",populationSize_);
62      algorithm.setInputParameter("maxEvaluations",maxEvaluations_);
63
64      // Mutation and Crossover for Real codification
65      parameters = new HashMap() ;
66      parameters.put("probability", crossoverProbability_) ;
67      parameters.put("distributionIndex", crossoverProbability_) ;
68      crossover = CrossoverFactory.getCrossoverOperator("SBXCrossover", parameters);
69
70      parameters = new HashMap() ;
71      parameters.put("probability", mutationProbability_) ;
72      parameters.put("distributionIndex", mutationDistributionIndex_) ;
73      mutation = MutationFactory.getMutationOperator("PolynomialMutation", parameters);
74
75      // Selection Operator
76      parameters = null ;
77      selection = SelectionFactory.getSelectionOperator("BinaryTournament2", parameters) ;
78
79      // Add the operators to the algorithm
80      algorithm.addOperator("crossover",crossover);
81      algorithm.addOperator("mutation",mutation);
82      algorithm.addOperator("selection",selection);
83
84     // Creating the indicator object
85     if ((paretoFrontFile_!=null) && (!paretoFrontFile_.equals(""))) {
86        indicators = new QualityIndicator(problem_, paretoFrontFile_);
87        algorithm.setInputParameter("indicators", indicators) ;
88     } // if
89
90      return algorithm ;
91    } // configure
92  } // NSGAII_Settings
```

Listing 4.3: `jmetal.experiments.settings.NSGAII_Settings`: Configuring the algorithm.

The implementation of the `configure()` method is included in Listing 4.3, where we can observe that it contains basically the same code used in the `NSGAII_main` class to configure the algorithm.

To modify specific parameters, we make use of a Java `HashMap` object. The map is composed of pairs (key, value), where the key and the value are strings. The idea is that the state variables defined in the subclass of `Settings` are used as keys in the properties object. As commented before, those variables must be `public`, and their identifiers must end with the underscore ('_') character.

Let us illustrate this with some pieces of code:

- Creating an instance of NSGA_II with the default parameter settings by using class `NSGAII_Settings`:

```
1 Algorithm nsgaII = new NSGAII_Settings(problem) ;
```

- Let us modify the crossover probability, which is set in the `crossoverProbability_` (Listing 4.2, line 34) to 1.0 (the default value is 0.9):

```
1 HashMap parameters = new HashMap() ;
2 parameters.put("crossoverProbability_", 1.0) ;
```

```
3 Algorithm nsgaII = new NSGAII_Settings(problem).configure(parameters) ;
```

- The algorithm can be executed now:

```
1 SolutionSet resultPopulation = nsgaII.execute() ;
```

An example of using of this feature can be found in Subsection 4.4.1.

In jMetal 4.2, we provide setting classes to a number of metaheuristics in `jmetal.experiments.settings`: AbYSS, CellDE, GDE3, IBEA, MOCell, MOEAD, NSGA-II, OMOPSO, PAES, RandomSearch, SMPSO, SPEA2, and IBEA.

## 4.3 The jmetal.experiments.Main class

The use of Settings objects in jMetal allows to have a unique program to run the algorithms. This program is defined in class jmetal.metaheuristics.Main. If we take a look to this class (see Listing 4.4 the three ways to run the program (lines 15-17), where the only required argument is the algorithm name. This name must be the prefix of the corresponding settings class (e.g., NSGAII, GDE3, etc.). The second argument is the problem name (e.g., ZDT1, DTLZ3, etc.) and the third one is the name of the file containing the Pareto front of the problem. In case of indicating the three arguments, the program calculates and displays the value of a number of quality indicators (lines 45-58) that will be applied to the obtained front.

An example of use is the following:

```
% java jmetal.experiments.Main NSGAII ZDT1 ../paretoFronts/ZDT1.pf
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Total execution time: 3965ms
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Objectives values have been writen to file FUN
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Variables values have been writen to file VAR
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Quality indicators
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Hypervolume: 0.6590761194336173
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: GD         : 2.828645886294944E-4
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: IGD        : 2.1542653967708837E-4
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Spread     : 0.4153061260894926
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Epsilon    : 0.018577848537497554
05-dic-2008 15:22:34 jmetal.experiments.Main main
INFO: Speed      : 13300 evaluations
```

```java
//  Main.java
...
package jmetal.experiments;
...
/**
 * Class for running algorithms
 */
public class Main {
  /**
   * @param args Command line arguments.
   * @throws JMException
   * @throws IOException
   * @throws SecurityException
   * Usage: three options
   *      - jmetal.experiments.Main algorithmName
   *      - jmetal.experiments.Main algorithmName problemName
   *      - jmetal.experiments.Main algorithmName problemName paretoFrontFile
   * @throws ClassNotFoundException
   */
  public static void main(String [] args) throws
                                  JMException, ... {
    Problem   problem   ;         // The problem to solve
    Algorithm algorithm ;         // The algorithm to use

    QualityIndicator indicators ; // Object to get quality indicators

    Settings settings = null;

    String algorithmName   = "" ;
    String problemName     = "Kursawe" ; // Default problem
    String paretoFrontFile = "" ;
    ...
    // Execute the Algorithm
    long initTime = System.currentTimeMillis();
    SolutionSet population = algorithm.execute();
    long estimatedTime = System.currentTimeMillis() - initTime;

    // Result messages
    logger_.info("Total execution time: "+estimatedTime + "ms");
    logger_.info("Objectives values have been writen to file FUN");
    population.printObjectivesToFile("FUN");
    logger_.info("Variables values have been writen to file VAR");
    population.printVariablesToFile("VAR");

    if (indicators != null) {
      logger_.info("Quality indicators") ;
      logger_.info("Hypervolume: " + indicators.getHypervolume(population)) ;
      logger_.info("GD         : " + indicators.getGD(population)) ;
      logger_.info("IGD        : " + indicators.getIGD(population)) ;
      logger_.info("Spread     : " + indicators.getSpread(population)) ;
      logger_.info("Epsilon    : " + indicators.getEpsilon(population)) ;

      if (algorithm.getOutputParameter("evaluations") != null) {
        Integer evals = (Integer)algorithm.getOutputParameter("evaluations") ;
        int evaluations = (Integer)evals.intValue();
        logger_.info("Speed      : " + evaluations + " evaluations") ;
      } // if
    } // if
  } //Main
} // Main
```

Listing 4.4: `jmetal.experiments.Main` class

## 4.4 Experimentation Example: NSGAIIStudy

Since version 2.0, jMetal includes the `jmetal.experiments.Experiment` class, which is intended to help in making experimentation studies of algorithms. In its current state, it allows to indicate: the metaheuristics to run, the problems to solve, the quality indicators to apply, and the number of independent runs to carry out. As a result, it generates a directory with all the obtained approximation sets and quality indicators values and, depending on the user preferences:

- A latex file containing tables with means and medians of the obtained measures.
- R scripts to produce boxplots of the results.
- R scripts to generate latex tables with the application of the Wilcoxon statistical test to the results.

In this section, we illustrate how to use this class by detailing the code of `jmetal.experiments.NSGAIIStudy`, a subclass of `Experiment` aimed at studing the effect of varying the crossover probability in NSGA-II. In concrete, we want to study four probabiliy values: 1.0, 0.9, 0.8, and 0.7. Let us recall that this is only an example.

### 4.4.1 Defining the experiment

We enumerate the steps to follow in order to define our own `Experiment` subclass:

```
1  //  NSGAIIStudy.java
2  ...
3  package jmetal.experiments;
4  ...
5  /**
6   * Class implementing an example of experiment using NSGA-II as base algorithm.
7   * The experiment consisting in studying the effect of the crossover probability
8   * in NSGA-II.
9   */
10 public class NSGAIIStudy extends Experiment {
11   ...
```

1. A method called `algorithmSettings` must be implemented:

```
12   ...
13   /**
14    * Configures the algorithms in each independent run
15    * @param problem The problem to solve
16    * @param problemIndex
17    * @param algorithm Array containing the algorithms to run
18    * @throws ClassNotFoundException
19    */
20   public synchronized void algorithmSettings(String problemName,
21                                              int problemIndex,
22                                              Algorithm[] algorithm)
23     throws ClassNotFoundException {
24     ...
```

   This method is invoked automatically in each independent run, por each problem and algorithm. The key is that a `Settings` object with the desired parameterization has to be created in order to get the `Algorithm` to be executed:

```
25     ...
26     try {
27        int numberOfAlgorithms = algorithmNameList_.length;
28
29        HashMap[] parameters = new HashMap[numberOfAlgorithms];
30
```

```
31        for (int i = 0; i < numberOfAlgorithms; i++) {
32          parameters[i] = new HashMap();
33        } // for
34
35        parameters[0].put("crossoverProbability_", 1.0);
36        parameters[1].put("crossoverProbability_", 0.9);
37        parameters[2].put("crossoverProbability_", 0.8);
38        parameters[3].put("crossoverProbability_", 0.7);
39
40        if ((!paretoFrontFile_[problemIndex].equals("")) ||
41            (paretoFrontFile_[problemIndex] == null)) {
42          for (int i = 0; i < numberOfAlgorithms; i++)
43            parameters[i].put("paretoFrontFile_", paretoFrontFile_[problemIndex]);
44        } // if
45
46        for (int i = 0; i < numberOfAlgorithms; i++)
47          algorithm[i] = new NSGAII_Settings(problemName).configure(parameters[i]);
48        ...
49    } // algorithmSettings
50    ...
```

In this example, as we are interested in four configurations of NSGA-II, with four different crossover probabilities, we define a Java `HashMap` object per algorithm (line 29) to indicate the desired values (lines 35-38). The code between lines 40-44 is used to incorporate the names of the Pareto front files if they are specified. Finally, the `Algorithm` objects are created and configure and they are ready to be executed (lines 46,47).

2. Once we have defined the `algorithmSettings` method, we have to do the same with the `main` method. First, an object of the `NSGAIIStudy` must be created:

```
51    ...
52    public static void main(String[] args) throws JMException, IOException {
53    NSGAIIStudy exp = new NSGAIIStudy() ;
54    ...
```

3. We need to give a name to the experiment (note: take into account that this name will be used to generate Latex tables, so you should avoid using the underscore symbol '_'). In this case, we choose the same name of the class, "NSGAIIStudy".

```
55    ...
56    exp.experimentName_ = "NSGAIIStudy" ;
57    ...
```

4. We have to indicate: the names of the algorithms, the problems to solve, the names of the files containing the Pareto fronts, and a list of the quality indicators to apply:

```
58    ...
59    exp.algorithmNameList_ = new String[] {"NSGAIIa", "NSGAIIb", "NSGAIIc", "
          NSGAIId"} ;
60    exp.problemList_ = new String[] { "ZDT1", "ZDT2", "ZDT3", "ZDT4", "DTLZ1", "WFG2
          "} ;
61    exp.paretoFrontFile_ = new String[] {"ZDT1.pf", "ZDT2.pf", "ZDT3.pf", "ZDT4.pf",
           "DTLZ1.2D.pf", "WFG2.2D.pf"} ;
62    exp.indicatorList_ = new String[] {"HV", "SPREAD", "IGD", "EPSILON"} ;
63    ...
```

The algorithm names are merely tags that will be used to create the output directories and the tables. The problem names must be the same used in `jmetal.problems`. We must note that:

- The order of the names of the Pareto front files must be the same as the name of the problems in the problem list.

- If we use the names of the Pareto front files that can be found in the jMetal Web site, when indicating a DTLZ problem (as `DTLZ1`), we must indicate the 2D file (`DTLZ1.2D.pf`) if we intend to solve it using a bi-objective formulation. Furthermore, we have to modify the problem classes, as `DTLZ1.java`, to indicate two objectives.

The same holds if we want to solve the WFG problems: by default they are defined as bi-objective, so they have to be modified to solved them with more objectives.

5. The next step is to indicate the output directory and the directory where the Pareto front files are located:

```
64    ...
65     exp.experimentBaseDirectory_ = "/Users/antonio/Softw/pruebas/jmetal/" +
66                                    exp.experimentName_;
67     exp.paretoFrontDirectory_ = "/Users/antonio/Softw/pruebas/data/paretoFronts";
68    ...
```

6. Once everything is configured, the array containing the `Settings` of the algorithms must be initialized:

```
69    ...
70    exp.algorithmSettings_ = new Settings[numberOfAlgorithms] ;
71    ...
```

7. The number of independent runs has to be specified (30 in this example):

```
72    ...
73   exp.independentRuns_ = 30 ;
74    ...
```

8. Finally, we execute the algorithms. The `runExperiment()` method has an optional parameter (the default value is 1) indicating the number of threads to be created to run the experiments (see Section 4.8 for further details):

```
75    ...
76    // Run the experiments
77    int numberOfThreads ;
78    exp.runExperiment(numberOfThreads = 4) ;
79   ...
```

9. Optionally, we may be interested in generating Latex tables and statistical information of the obtained results. Latex tables are produced by the following command:

```
80   ...
81   // Generate latex tables
82   exp.generateLatexTables() ;
83   ...
```

In case of being interested in getting boxplots, since jMetal 2.1 it is possible to obtain R scripts to generate them. In that case, you need to invoke the `generateRBoxplotScripts()` method:

```
84     ...
85     // Configure the R scripts to be generated
86     int rows  ;
87     int columns  ;
88     String prefix ;
89     String [] problems ;
90
91     rows = 2 ;
92     columns = 3 ;
```

```
93      prefix = new String("Problems");
94      problems = new String[]{"ZDT1", "ZDT2","ZDT3", "ZDT4", "DTLZ1", "WFG2"} ;
95
96      boolean notch ;
97      exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch = true, exp)
            ;
98      ...
```

This method generates R scripts which produce .eps files containing `rows` × `columns` boxplots of the list of problems passed as third parameter. It is necessary to explicitly indicate the problems to be consider in the boxplots because if there are too much problems, the resulting graphics will be very small and difficult to see. In this situation, several calls to `generateRBoxplotScripts()` can be included. The name of the scripts will start by the prefix specified in the fourth parameter plus the name of the quality indicator, ended with the suffix "Botxplot.R". The last parameter indicates whether notched boxplots should be generated or not.

Additionally, a method called `generateRWilcoxonScripts()` is available since jMetal 2.2. This method is intended to apply the Wilcoxon rank-sum test to the obtained results:

```
99    ...
100   exp.generateRWilcoxonScripts(problems, prefix, exp) ;
101   ...
```

For each indicator, a file with suffix "Wilcox.R" will be generated. Once each of these scripts is executed, a latex file will be yielded as output. Please, see next section for further details.

### 4.4.2 Running the experiments

To run the experiments, if we are using the command line we simply have to type (assuming the the `CLASSPATH` variable has been configurated):

```
java jmetal.experiments.NSGAIIStudy
```

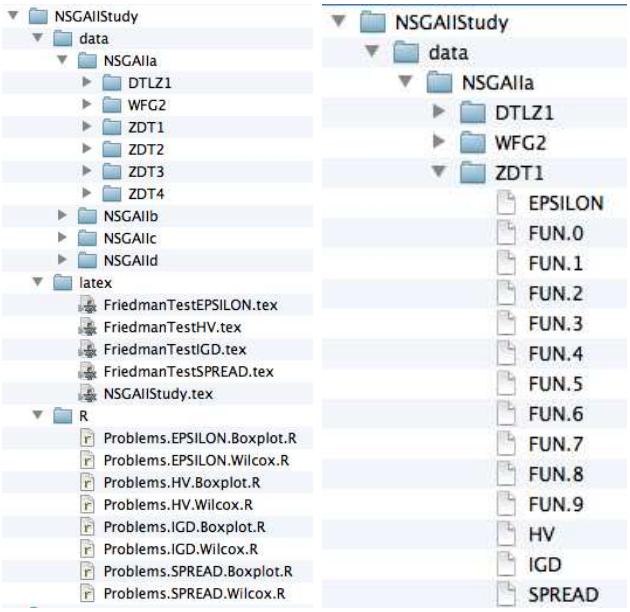


Figure 4.1: Output directories and files after running the experiment.

Table 4.1: HV. Mean and standard deviation

| | NSGAIIa | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|---|
| ZDT1 | $6.60e-01_{1.9e-04}$ | $6.59e-01_{4.5e-04}$ | $6.59e-01_{2.1e-04}$ | $6.59e-01_{3.5e-04}$ |
| ZDT2 | $3.27e-01_{3.1e-04}$ | $3.26e-01_{2.3e-04}$ | $3.26e-01_{3.7e-04}$ | $3.25e-01_{2.6e-04}$ |
| ZDT3 | $5.15e-01_{1.9e-04}$ | $5.15e-01_{1.3e-04}$ | $5.15e-01_{1.7e-04}$ | $5.14e-01_{2.6e-04}$ |
| ZDT4 | $6.57e-01_{4.1e-03}$ | $6.55e-01_{2.2e-03}$ | $6.52e-01_{5.6e-03}$ | $6.53e-01_{3.4e-03}$ |
| DTLZ1 | $4.90e-01_{3.8e-03}$ | $4.86e-01_{4.1e-03}$ | $4.85e-01_{4.6e-03}$ | $4.83e-01_{7.1e-03}$ |
| WFG2 | $5.62e-01_{1.7e-03}$ | $5.63e-01_{1.3e-03}$ | $5.62e-01_{1.5e-03}$ | $5.62e-01_{1.3e-03}$ |

Table 4.2: HV. Median and IQR

| | NSGAIIa | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|---|
| ZDT1 | $6.60e-01_{3.1e-04}$ | $6.59e-01_{5.3e-04}$ | $6.59e-01_{2.8e-04}$ | $6.59e-01_{5.2e-04}$ |
| ZDT2 | $3.26e-01_{3.3e-04}$ | $3.26e-01_{3.3e-04}$ | $3.25e-01_{7.3e-04}$ | $3.25e-01_{3.8e-04}$ |
| ZDT3 | $5.15e-01_{2.0e-04}$ | $5.15e-01_{2.2e-04}$ | $5.15e-01_{2.2e-04}$ | $5.14e-01_{4.3e-04}$ |
| ZDT4 | $6.58e-01_{4.3e-03}$ | $6.56e-01_{3.1e-03}$ | $6.54e-01_{3.0e-03}$ | $6.53e-01_{3.6e-03}$ |
| DTLZ1 | $4.91e-01_{3.6e-03}$ | $4.87e-01_{7.8e-03}$ | $4.86e-01_{8.3e-03}$ | $4.85e-01_{7.7e-03}$ |
| WFG2 | $5.62e-01_{3.5e-03}$ | $5.63e-01_{2.9e-03}$ | $5.62e-01_{2.9e-03}$ | $5.61e-01_{2.7e-03}$ |

After the execution of the algorithms, we obtain the directory tree depicted in Figure 4.1. The directories are:

- `data`: Output of the algorithms.
- `latex`: Latex file containing result tables.
- `R`: R scripts for generating statistical information.

### 4.4.3 Analyzing the output results

As it can be observed in Figure 4.1-left, the directory named `NSGAIIStudy` has three directories: `data`, `R`, and `latex`. The `data` directory contains (see Figure 4.1-right), for each algorithm, the files with the variable values (files `VAR.0`, `VAR.1`, ...) and function values (files `FUN.0`, `FUN.1`, ...) of the obtained approximation sets (we show four files instead of the 30 files), and the quality indicators of these solution sets are included in the files `HV`, `SPREAD`, `EPSILON`, and `IDG`.

As the `FUN.XX` files store the fronts of solutions computed by the algorithms, they can be plotted to observe the resulting approximation sets. Depending on the study you are interested in, you could also join all of them into a single file to obtain a reference set (after removing the dominated solutions).

The `latex` directory contains a Latex file with the name of the experiment, `NSGAIIStudy.tex`. You just need to compile the file with your favorite Latex tool. For example, you could simply type:

```
latex NSGAIIStudy.tex
dvipdf NSGAIIStudy.dvi
```

to get a pdf file. Alternatively, you could invoke the `pdflatex` command:

```
pdflatex NSGAIIStudy.tex
```

As an example of the obtained output, Table 4.1 includes the mean and standard deviation of the results after applying the hypervolume indicator, and the median and interquartile range (IQR) values are in Table 4.2. Both tables include similar values in all the problems but DTLZ1. The point here is that although NSGAIIa (crossover probability = 1.0) obtains the best values, we should ensure that the observed differences are statistically different. In this context is where the boxplots and the Wilcoxon test are of interest.

The `R` directory stores the R scripts. As commented before, the script names are composed of the indicated prefix ("Problems" in the example), the name of the quality indicator, having the "R" extension. Those ending in "Boxplot.R" yield as a results eps files containing boxplots of the values of

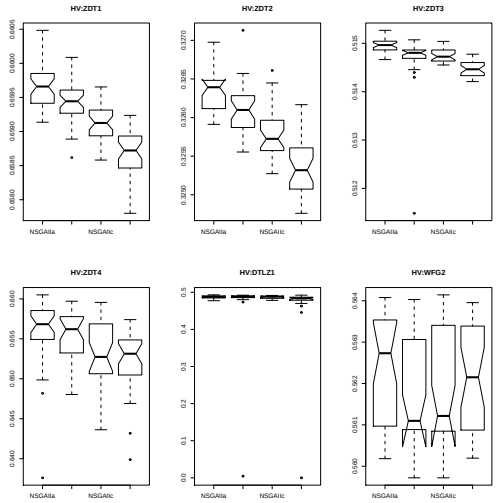


Figure 4.2: Boxplots of the values obtained after applying the hypervolume quality indicator (notch = true).

the indicators, while those ending in "Wilcox.R" contain the scripts to produce latex tables including the application of the Wilcoxon test.

To run the scripts, if you have properly installed R in your computer, you can type the following commands:

```
rscript ZDT.HV.Boxlplot.R
rscript ZDT.HV.Wilcox.R
rscript ZDT.EPSILON.Boxplot.R
rscript ZDT.EPSILON.Wilcox.R
...
```

Alternatively, if you are working with a UNIX machine, you can type:

```
for i in *.R ; do rscript $i 2>/dev/null ; done
```

As a result, you will get the same number of files, but with the .eps extension. Figure 4.2 shows the `Problems.HV.Boxplot.eps` file. Without entering into details about the results, in the notched boxplot, if two boxes' notches do not overlap then it is supposed with strong evidence that their medians differ, so we could conclude that NSGAIIa provides the best overall quality indicator values in the experiment with confidence. The we call the `generateRBoxplotScripts()` method with the notch parameter equal to false, the obtained result is included in Figure 4.3

Alternatively to using boxplots, the Wilcoxon rank-sum test can be used to determine the significance of the obtained results. To apply the Wilcoxon test to two distributions `a` and `b`, we use the R formula: `wilcox.test(a,b)`. The latex files produced when the "Wilcox.R" scripts are executed contains two types of tables: one per problem, and a global table summarizing the results. We include the tables of the first type corresponding to the hypervolume indicator in Tables 4.3 to 4.8. In each table, a ▲ symbol implies a $p$-value $< 0.05$, indicating than the null hyphotesis (the two distribution have the same median) is rejected; otherwise, a $\triangledown$ is used. Table 4.9 groups the other tables into one.

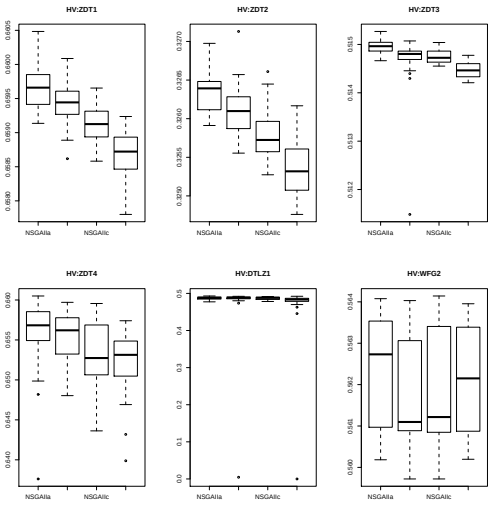


Figure 4.3: Boxplots of the values obtained after applying the hypervolume quality indicator (notch = false).

Table 4.3: ZDT1.HV.

| | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|
| NSGAIIa | ▲ | ▲ | ▲ |
| NSGAIIb | | ▲ | ▲ |
| NSGAIIc | | | ▲ |

Table 4.4: ZDT2 .HV.

| | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|
| NSGAIIa | ▲ | ▲ | ▲ |
| NSGAIIb | | ▲ | ▲ |
| NSGAIIc | | | ▲ |

Table 4.5: ZDT3.HV.

| | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|
| NSGAIIa | ▲ | ▲ | ▲ |
| NSGAIIb | | – | ▲ |
| NSGAIIc | | | ▲ |

Table 4.6: ZDT4 .HV.

| | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|
| NSGAIIa | – | ▲ | ▲ |
| NSGAIIb | | ▲ | ▲ |
| NSGAIIc | | | – |

Table 4.7: DTLZ1 .HV.

| | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|
| NSGAIIa | – | – | ▲ |
| NSGAIIb | | – | ▲ |
| NSGAIIc | | | ▲ |

Table 4.8: WFG2.HV.

| | NSGAIIb | NSGAIIc | NSGAIId |
|---|---|---|---|
| NSGAIIa | – | – | – |
| NSGAIIb | | – | – |
| NSGAIIc | | | – |

Table 4.9: ZDT1 ZDT2 ZDT3 ZDT4 DTLZ1 WFG2 .HV.

| | NSGAIIb | | | | | | NSGAIIc | | | | | | NSGAIId | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NSGAIIa | ▲ | ▲ | ▲ | – | – | – | ▲ | ▲ | ▲ | ▲ | – | – | ▲ | ▲ | ▲ | ▲ | ▲ | – |
| NSGAIIb | | | | | | | ▲ | ▲ | – | ▲ | – | – | ▲ | ▲ | ▲ | ▲ | ▲ | – |
| NSGAIIc | | | | | | | | | | | | | ▲ | ▲ | ▲ | – | ▲ | – |

## 4.5 Experimentation example: StandardStudy

In this section we describe another example of experiment. We have called it `StandardStudy` because it represents a kind of study we carry out frequently: comparing a number of different metaheuristics over the ZDT, DTLZ, and WFG benchmarks, making 100 independent runs.

The `algorithmSettings()` method (file: `jmetal.experiments.StandardStudy`) is included next:

```
1  /**
2   * Configures the algorithms in each independent run
3   * @param problemName The problem to solve
4   * @param problemIndex
5   * @throws ClassNotFoundException
6   */
7  public void algorithmSettings(String problemName,
8                                 int problemIndex,
9                                 Algorithm[] algorithm) throws ClassNotFoundException {
10   try {
11     int numberOfAlgorithms = algorithmNameList_.length;
12
13     HashMap[] parameters = new HashMap[numberOfAlgorithms];
14
15     for (int i = 0; i < numberOfAlgorithms; i++) {
16       parameters[i] = new HashMap();
17     } // for
18
19     if (!paretoFrontFile_[problemIndex].equals("")) {
20       for (int i = 0; i < numberOfAlgorithms; i++)
21         parameters[i].put("paretoFrontFile_", paretoFrontFile_[problemIndex]);
22       } // if
23
24       algorithm[0] = new NSGAII_Settings(problemName).configure(parameters[0]);
25       algorithm[1] = new SPEA2_Settings(problemName).configure(parameters[1]);
26       algorithm[2] = new MOCell_Settings(problemName).configure(parameters[2]);
27       algorithm[3] = new SMPSO_Settings(problemName).configure(parameters[3]);
28       algorithm[4] = new GDE3_Settings(problemName).configure(parameters[4]);
29     } catch (IllegalArgumentException ex) {
30     Logger.getLogger(StandardStudy.class.getName()).log(Level.SEVERE, null, ex);
31   } catch (IllegalAccessException ex) {
32     Logger.getLogger(StandardStudy.class.getName()).log(Level.SEVERE, null, ex);
33   } catch  (JMException ex) {
34     Logger.getLogger(StandardStudy.class.getName()).log(Level.SEVERE, null, ex);
35   }
36 } // algorithmSettings
```

We can observe that this method is simpler than in the case of `NSGAIIStudy`, because we assume that each algorithm is configured in its corresponding setting class. We test five metaheuristics: NSGAII, SPEA2, MOCell, SMPSO, and GDE3 (lines 24-28).

The `main` method is included below, where we can observe the algorithm name list (lines 42-43), the problem list (lines 44-48), and the list of the names of the files containing the Pareto fronts (lines 49-56):

```
37 ...
38 public static void main(String[] args) throws JMException, IOException {
39   StandardStudy exp = new StandardStudy();
40
41   exp.experimentName_ = "StandardStudy";
42   exp.algorithmNameList_ = new String[]{
```

```
43                                      "NSGAII", "SPEA2", "MOCell", "SMPSO", "GDE3"};
44      exp.problemList_ = new String[]{"ZDT1", "ZDT2","ZDT3", "ZDT4","ZDT6",
45                                      "WFG1","WFG2","WFG3","WFG4","WFG5","WFG6",
46                                      "WFG7","WFG8","WFG9",
47                                      "DTLZ1","DTLZ2","DTLZ3","DTLZ4","DTLZ5",
48                                      "DTLZ6","DTLZ7"};
49      exp.paretoFrontFile_ = new String[]{"ZDT1.pf", "ZDT2.pf","ZDT3.pf",
50                                      "ZDT4.pf","ZDT6.pf",
51                                      "WFG1.2D.pf","WFG2.2D.pf","WFG3.2D.pf",
52                                      "WFG4.2D.pf","WFG5.2D.pf","WFG6.2D.pf",
53                                      "WFG7.2D.pf","WFG8.2D.pf","WFG9.2D.pf",
54                                      "DTLZ1.2D.pf","DTLZ2.2D.pf","DTLZ3.2D.pf",
55                                      "DTLZ4.2D.pf","DTLZ5.2D.pf","DTLZ6.2D.pf",
56                                      "DTLZ7.2D.pf"};
57    ...
```

The rest of the code is similar to `NSGAIIStudy`: the list of indicators is included in line 59, the directory to write the results and the one containing the Pareto fronts are specified next (lines 63-65), the number of independent runs is indicated in line 69, and the method to run the algorithm is called (lines 72-73):

```
58      ...
59      exp.indicatorList_ = new String[]{"HV", "SPREAD", "EPSILON"};
60
61      int numberOfAlgorithms = exp.algorithmNameList_.length;
62
63      exp.experimentBaseDirectory_ = "/Users/antonio/Softw/pruebas/jmetal/" +
64                                     exp.experimentName_;
65      exp.paretoFrontDirectory_ = "/Users/antonio/Softw/pruebas/data/paretoFronts";
66
67      exp.algorithmSettings_ = new Settings[numberOfAlgorithms];
68
69      exp.independentRuns_ = 100;
70
71      // Run the experiments
72      int numberOfThreads ;
73      exp.runExperiment(numberOfThreads = 4) ;
74    ...
```

Finally, we generate the Latex tables and generate the R scripts. Note that we invoke three times the methods `generateRBoxplotsScript()` and `generateRWilcoxonScript()`, one per problem family. The reason is that otherwise the resulting graphs and tables would not fit into an A4 page:

```
75      ...
76      // Generate latex tables
77      exp.generateLatexTables() ;
78
79      // Configure the R scripts to be generated
80      int rows  ;
81      int columns  ;
82      String prefix ;
83      String [] problems ;
84      boolean notch ;
85
86      // Configuring scripts for ZDT
87      rows = 3 ;
88      columns = 2 ;
89      prefix = new String("ZDT");
90      problems = new String[]{"ZDT1", "ZDT2","ZDT3", "ZDT4","ZDT6"} ;
91
92      exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch = false, exp) ;
93      exp.generateRWilcoxonScripts(problems, prefix, exp) ;
94
95      // Configure scripts for DTLZ
96      rows = 3 ;
```

```
97      columns = 3 ;
98      prefix = new String("DTLZ");
99      problems = new String[]{"DTLZ1","DTLZ2","DTLZ3","DTLZ4","DTLZ5",
100                                      "DTLZ6","DTLZ7"} ;
101
102     exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch=false, exp) ;
103     exp.generateRWilcoxonScripts(problems, prefix, exp) ;
104
105     // Configure scripts for WFG
106     rows = 3 ;
107     columns = 3 ;
108     prefix = new String("WFG");
109     problems = new String[]{"WFG1","WFG2","WFG3","WFG4","WFG5","WFG6",
110                             "WFG7","WFG8","WFG9"} ;
111
112     exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch=false, exp) ;
113     exp.generateRWilcoxonScripts(problems, prefix, exp) ;
114     ...
```

## 4.6 Experiments when the Pareto fronts of the problems are unknown

When solving real-world problems, the Pareto fronts of them usually are unknown, so applying the quality indicators available in jMetal is not possible. An usual approach to cope with this issue is to construct a reference front by collecting all the results of all the runs of all the algorithms (see Section 6.4). This way, we have a way to compare the relative performance of the algorithms.

In jMetal 4.2, we provide an automatic way to obtain the reference fronts into an experiment. From the user point of view, the approach is simple: just take an experiment class and leave the information of the Pareto front files out. In the case of the `StandardStudy` class, detailed in the previous section, the changes to do are indicated in the following piece of code:

```
37  ...
38  public static void main(String[] args) throws JMException, IOException {
39    StandardStudy exp = new StandardStudy();
40
41    exp.experimentName_ = "StandardStudy";
42    exp.algorithmNameList_ = new String[]{
43                                "NSGAII", "SPEA2", "MOCell", "SMPSO", "GDE3"};
44    exp.problemList_ = new String[]{"ZDT1", "ZDT2","ZDT3", "ZDT4","ZDT6",
45                                    "WFG1","WFG2","WFG3","WFG4","WFG5","WFG6",
46                                    "WFG7","WFG8","WFG9",
47                                    "DTLZ1","DTLZ2","DTLZ3","DTLZ4","DTLZ5",
48                                    "DTLZ6","DTLZ7"};
49    exp.paretoFrontFile_ = new String[18]; // Space allocation for 18 fronts
50  ...
51    exp.paretoFrontDirectory_ = "" ; // This directory must be empty
52  ...
```

We provide a `StandardStudy2` class including these sentences.

When the experiment is executed, a new directory called `referenceFront` appears in the same place of the `R`, `latex`, and `data` directories depicted in Figure 4.1. That directory contains files with a reference front per problem. These front are used in the experiment to apply the quality indicators[1].

[1] This is a contribution of Jorge Rodríguez

## 4.7 Using quality indicators

When any of the algorithms provided by jMetal are executed to solve a problem, two files containing the approximations of the optimal solutions and the Pareto front are returned (by default, these files are called `VAR` and `FUN`, respectively). Typically, the file `FUN` is used to apply some quality indicators (hypervolume, spread, etc.) offline.

In jMetal 1.8, a new class `jmetal.qualityIndicatorQualityIndicator` was introduced. This class is intended to facilitate the use of quality indicators inside the programs. We use mainly this class in the `algorithm_main` and `algorithm_Settings` classes. For example, let us take a look to the `jmetal/metaheuristics/moead/MOEAD_main.java` file, which executes the algorithm MOEA/D-DE. Here we can find the following lines of code:

- Importing the required class:

```
32 import jmetal.qualityIndicator.QualityIndicator;
```

- Declaring an object of the class:

```
57 QualityIndicator indicators ; // Object to get quality indicators
```

- The object is created using the third argument from the command line, which should contain the file storing the Pareto from of the problem to solve:

```
72 indicators = new QualityIndicator(problem, args[1]) ;
```

- Using the indicators object:

```
117     if (indicators != null) {
118       logger_.info("Quality indicators") ;
119       logger_.info("Hypervolume: " + indicators.getHypervolume(population)) ;
120       logger_.info("GD         : " + indicators.getGD(population)) ;
121       logger_.info("IGD        : " + indicators.getIGD(population)) ;
122       logger_.info("Spread     : " + indicators.getSpread(population)) ;
123       logger_.info("Epsilon    : " + indicators.getEpsilon(population)) ;
124     } // if
```

As it can be seen, the `QualityIndicator` object is applied to the population returned by the algorithm. This way, the program returns the values of the desired quality indicators of the obtained solution set.

Another example of using `QualityIndicator` objects was introduced in Section 3.3.1, where the use of the hypervolume inside NSGA-II to measure the convergence speed of the algorithm was detailed.

## 4.8 Running experiments in parallel

In jMetal 2.2 we introduced a first approximation to make use of current multi-core processors to speed-up the execution of experiments, by using Java threads for that purpose. In jMetal 4.2 a new parallel scheme has been adopted to enhance the performance of the running of the experiments.

We have tested this new feature by running the `NSGAIIStudy` experiment (see Section 4.4). The computer is a MacBook with a Core i7 2.2GHz processor and 8 GB of RAM, running Mac OS X 10.7.5 (11G63); the Java version is "Java(TM) SE Runtime Environment (build 1.6.0_37-b06-434)". The computing time using one thread is roughly 3.5 minutes, while using eight threads 2.4 minutes. The speed-up is far from lineal, but we have to take into account that each run of the algorithms is around 0.5 seconds; it is expected that dealing with problems having more time-consuming functions, or algorithms executing more than 25,000 function evaluations, the speed-up should be more noticeable.

# Chapter 5

# Parallel Algorithms

Since version 4.3, jMetal provides a basic support for developing parallel metaheuristics. This chapter describes our first approximation to this issue, which is currently focused in allowing the parallel evaluation of solutions taking advantage of the multicore feature of modern processors.

## 5.1 The `IParallelEvaluator` Interface

The taken approach to evaluate solutions in parallel is simple: the solutions to be evaluated are inserted into a list which is submitted to a *parallel evaluator* object that will make the parallel computation. This kind of objects are represented by the `IParallelEvaluator` interface, located in the `jmetal.util.parallel` package.

```
1  //  IParallelEvaluator.java
2  ...
3  package jmetal.util.parallel;
4
5  import java.util.List;
6
7  import jmetal.core.Problem;
8  import jmetal.core.Solution;
9
10 /**
11  * @author Antonio J. Nebro
12  * Interface representing classes for evaluating solutions in parallel
13  * The procedure is:
14  * 1- create the parallel evaluator with startEvaluator()
15  * 2- add solutions for being evaluated with addSolutionforEvaluation()
16  * 3- evaluate the solutions with parallelEvaluation()
17  * 4- shutdown the parallel evaluator with stopEvaluator()
18  */
19
20 public interface IParallelEvaluator {
21   public void startEvaluator(Problem problem) ;
22   public void addSolutionForEvaluation(Solution solution) ;
23   public List<Solution> parallelEvaluation() ;
24   public void stopEvaluator() ;
25 }
```

Listing 5.1: `IParallelEvaluator` interface

The code of `IParallelEvaluator` is included in Listing 5.1, and contains four methods:

- `startEvaluator()`: initializes and starts the parallel evaluator object.

- `addSolutionForEvaluation()`: sends a solution to the evaluator object. This solution will be queued in an internal list.
- `parallelEvaluation()`: all the solutions in the internal list are evaluated in parallel, and a list containing them is returned.
- `stopEvaluator()`: the parallel evaluator is stopped.

The `IParallelEvaluator` interface allows many possible implementations. In jMetal 4.3 we provide the `MultithreadedEvaluator` class, which is designed to make use of the processors/cores which are available on most computers nowadays. The constructor of this class is detailed in Listing 5.2. It takes as argument an integer value indicating the desired number of threads to be used. If this argument takes the value 0 then the number of processors of the system is used, according to the value returned by the Java `Runtime.getRuntime().availableProcessors()` method.

```
1  //  MultithreadedEvaluator.java
2  ...
3  package jmetal.util.parallel;
4  ...
5  public class MultithreadedEvaluator implements IParallelEvaluator {
6    ...
7      /**
8       * Constructor
9       * @param threads
10      */
11     public MultithreadedEvaluator(int threads) {
12       numberOfThreads_ = threads ;
13       if (threads == 0)
14         numberOfThreads_ = Runtime.getRuntime().availableProcessors() ;
15       else if (threads < 0) {
16          Configuration.logger_.severe("MultithreadedEvaluator: the number of threads" +
17          " cannot be negative number " + threads);
18       }
19       else {
20         numberOfThreads_ = threads ;
21     }
22   }
23
24   public void startEvaluator(Problem problem) { ...
25   public void addSolutionForEvaluation(Solution solution) { ...
26   public List<Solution> parallelEvaluation() { ...
27   public void stopEvaluator() { ...
28 }
```

Listing 5.2: `MultithreadedEvaluator` class

## 5.2 Evaluating Solutions In Parallel in NSGA-II: pNSGAII

In this section, we illustrate the use of the `IParallelEvaluator` in pNSGAII, a version of NSGA-II using this interface.

We start by showing how to instantiate the parallel evaluator, as is done in the pNSGAII_main class (see Listing 5.3, line 7). Then, the evaluator is passed as second argument to the `pNSGAII` class constructor.

```
1  //  pNSGAII_main.java
2  ...
3  public class pNSGAII_main {
4      ...
5      public static void main(String [] args) { ...
6        ...
```

```
7         int threads = 4 ; // 0 - use all the available cores
8         IParallelEvaluator parallelEvaluator = new MultithreadedEvaluator(threads) ;
9
10        algorithm = new pNSGAII(problem, parallelEvaluator);
11        ...
12      }
13  }
```

Listing 5.3: `pNSGAII_main` class

The `pNSGAII` class contains a state variable to reference the parallel evaluator, as shown in line 7 in Listing 5.4. It is initialized in the class constructor (line 17).

```
1  //  pNSGAII.java
2  ...
3  package jmetal.metaheuristics.nsgaII;
4  ...
5  public class pNSGAII extends Algorithm {
6
7    IParallelEvaluator parallelEvaluator_ ;
8
9    /**
10    * Constructor
11    * @param problem Problem to solve
12    * @param evaluator Parallel evaluator
13    */
14   public pNSGAII(Problem problem, IParallelEvaluator evaluator) {
15     super (problem) ;
16
17     parallelEvaluator_ = evaluator ;
18   } // pNSGAII
19   ...
20   public SolutionSet execute() throws JMException, ClassNotFoundException {
21   ...
```

Listing 5.4: `pNSGAII` class. Constructor

The parallel evaluator is started in line 22 in the code included in Listing 5.5. The method `startEvaluator()` takes as parameter the problem being solved, which is necessary for the further evaluation of the solutions. The initial population is initialized in theee steps. First, in the loop starting in line 26, every new instantiated solution (line 27) is sent to the evaluator (line 28); second, the `parallelEvaluation()` method of the parallel evaluator is invoked (line 31); finally, the evaluated solutions are inserted into de population (lines 32-35).

```
19   ...
20   public SolutionSet execute() throws JMException, ClassNotFoundException {
21     ...
22     parallelEvaluator_.startEvaluator(problem_) ;
23     ...
24     // Create the initial solutionSet
25     Solution newSolution;
26     for (int i = 0; i < populationSize; i++) {
27       newSolution = new Solution(problem_);
28       parallelEvaluator_.addSolutionForEvaluation(newSolution) ;
29     }
30
31     List<Solution> solutionList = parallelEvaluator_.parallelEvaluation() ;
32     for (Solution solution : solutionList) {
33       population.add(solution) ;
34       evaluations ++ ;
35     }
36     ...
```

Listing 5.5: `pNSGAII` class. Initializing initial population

Table 5.1: Solving ZDT1 with NSGA-II and pNSGAII with 1, 8, 32, 128, and 512 threads (times in milliseconds).

| | NSGAII | 1T | 8T | 32T | 128T | 512T |
|---|---|---|---|---|---|---|
| ZDT1 | 670 | 730 | 750 | 770 | 900 | 950 |

The same scheme is applied to evaluate in parallel the solutions created after applying the crossover and mutation operators, as it can be observed in the piece of code included in Listing 5.6.

```
36    ...
37    // Generations
38    while (evaluations < maxEvaluations) {
39      // Create the offSpring solutionSet
40      offspringPopulation = new SolutionSet(populationSize);
41      Solution[] parents = new Solution[2];
42      for (int i = 0; i < (populationSize / 2); i++) {
43        if (evaluations < maxEvaluations) {
44          //obtain parents
45          parents[0] = (Solution) selectionOperator.execute(population);
46          parents[1] = (Solution) selectionOperator.execute(population);
47          Solution[] offSpring = (Solution[]) crossoverOperator.execute(parents);
48          mutationOperator.execute(offSpring[0]);
49          mutationOperator.execute(offSpring[1]);
50          parallelEvaluator_.addSolutionForEvaluation(offSpring[0]) ;
51          parallelEvaluator_.addSolutionForEvaluation(offSpring[1]) ;
52        } // if
53      } // for
54
55      List<Solution> solutions = parallelEvaluator_.parallelEvaluation() ;
56
57      for(Solution solution : solutions) {
58        offspringPopulation.add(solution);
59        evaluations++;
60      }
61      ...
```

Listing 5.6: `pNSGAII` class. Evaluating solutions in parallel in the main loop of NSGA-II

Besides pNSGAII, a parallel version of the SMPSO algorithm, named pSMPSO (included in the `jmetal.metaheuristics.smpso` package), is provided in jMetal 4.3.

## 5.3 About Parallel Performance

In this section, we include some performance figures to give an idea of the benefits of using the multi-threaded parallel evaluator. The tests are executed in a MacBook Pro with a 2.2 GHz Intel Core i7 and 8 GB 13333 MHz DDR3; the operating system is Mac OS X Lion 10.7.5 (11G63) and the Java version is 1.6.0_37.

To determine the overhead of using the parallel evaluator, we execute NSGA-II to solve the ZDT1 problem using standard settings (25,000 function evaluations) and pNSGAII with 1, 8, 32, 128, and 256 threads. The reported times in Table 5.1 are the rough mean of a few independent runs. We can observe that the overhead of using 1 thread in pNSGAII versus NSGA-II is about 70 ms, which can be considered as very low taking into account that we are solving a benchmark problem. As the i7 processor has four cores and each one incorporates two hyperthreads, the total number of cores reported by the Java `Runtime.getRuntime().availableProcessors()` method is 8. As a consequence, the performance should degrade when using more than 8 threads, which is corroborated by the times reported in Table 5.1, although the penalty when using up to 512 threads is about 300 ms and 200 ms compared to the sequential NSGA-II and pNSGA-II with 8 threads, respectively.

Table 5.2: Solving ZDT1b with NSGA-II and pNSGAII with 1, 8, 32, 128, and 512 threads (times in milliseconds). ZDT1b is a the problem as ZDT1 but including a idle loop in the evaluation function to increase its computing time.

| | NSGAII | 1T | 8T | 32T | 128T | 512T |
|---|---|---|---|---|---|---|
| ZDT1b | 86,500 | 87,000 | 24,000 | 24,000 | 24,000 | 24,000 |

Next we increase computing time to evaluate solutions to ZDT1 by adding the next loop into the evaluation function:

```
1 for (long i = 0 ; i < 10000000; i++) ;
```

We called this problem ZDT1b, and the computing times of the NSGA-II variants are included in Table 5.2. Now, NSGA-II requires 86.5 seconds to perform 25,000 evaluations, while the multithreaded versions from 8 threads take roughly 24 seconds, which means a speed-up of 3.6 (i.e., an efficiency of 0.45). The reason to not achieving a higher speed-up is that we are evaluating the individuals in parallel, but the ranking and crowding procedures are carried out sequentially.

# Chapter 6

# How-to's

This chapter is devoted to containing answers to some questions that may emerge when working with jMetal.

## 6.1 How to use binary representations in jMetal

All the examples we have presented in the manual are related to optimizing continuous problems using a real representation of the solutions. In this section we include an example of using a binary coding representation. To illustrate this, we use the `jmetal.experiments.settings.NSGAIIBinary_Settigns` class.

Let us start by commenting the piece of code below. The code is very simple, and we can observe how the `BinaryReal` encoding is selected (line 22), so the problem is configured to use binary coded real solutions (line 24). The mutation probability is set as $1/L$ (line 34), where $L$ is the number of bits of the solution.

```
1  //  NSGAIIBinary_Settings.java
2  ...
3  package jmetal.experiments.settings;
4  ...
5  /**
6   * Settings class of algorithm NSGA-II (binary encoding)
7   */
8  public class NSGAIIBinary_Settings extends Settings {
9
10   int populationSize_  ;
11   int maxEvaluations_  ;
12
13   double mutationProbability_  ;
14   double crossoverProbability_ ;
15
16   /**
17    * Constructor
18    */
19   public NSGAIIBinary_Settings(String problem) {
20     super(problem) ;
21
22     Object [] problemParams = {"BinaryReal"};
23     try {
24       problem_ = (new ProblemFactory()).getProblem(problemName_, problemParams);
25     } catch (JMException e) {
26       // TODO Auto-generated catch block
27       e.printStackTrace();
28     }
29
```

```
30      // Default settings
31      populationSize_ = 100    ;
32      maxEvaluations_ = 25000 ;
33
34      mutationProbability_  = 1.0/problem_.getNumberOfBits();
35      crossoverProbability_ = 0.9 ;
36    } // NSGAIIBinary_Settings
37    ...
```

In the `configure()` method, we choose the single point and bit-flip crossover and mutation operators (lines 62 -68):

```
38  /**
39    * Configure NSGAII with user-defined parameter settings
40    * @return A NSGAII algorithm object
41    * @throws jmetal.util.JMException
42    */
43   public Algorithm configure() throws JMException {
44     Algorithm algorithm ;
45     Operator  selection ;
46     Operator  crossover ;
47     Operator  mutation  ;
48
49     QualityIndicator indicators ;
50
51     HashMap  parameters ; // Operator parameters
52
53     // Creating the problem
54     algorithm = new NSGAII(problem_) ;
55
56     // Algorithm parameters
57     algorithm.setInputParameter("populationSize", populationSize_);
58     algorithm.setInputParameter("maxEvaluations", maxEvaluations_);
59
60
61     // Mutation and Crossover Binary codification
62     parameters = new HashMap() ;
63     parameters.put("probability", 0.9) ;
64     crossover = CrossoverFactory.getCrossoverOperator("SinglePointCrossover", parameters
           );
65
66     parameters = new HashMap() ;
67     parameters.put("probability", 1.0/problem_.getNumberOfBits()) ;
68     mutation = MutationFactory.getMutationOperator("BitFlipMutation",parameters);
69
70     // Selection Operator
71     parameters = null ;
72     selection = SelectionFactory.getSelectionOperator("BinaryTournament2", parameters) ;
73
74     // Add the operators to the algorithm
75     algorithm.addOperator("crossover",crossover);
76     algorithm.addOperator("mutation",mutation);
77     algorithm.addOperator("selection",selection);
78
79    // Creating the indicator object
80     if ((paretoFrontFile_!=null) && (!paretoFrontFile_.equals(""))) {
81       indicators = new QualityIndicator(problem_, paretoFrontFile_);
82       algorithm.setInputParameter("indicators", indicators) ;
83    } // if
84     return algorithm ;
85   } // configure
```

If we want to use NSGA-II to solve the ZDT1 problem by using a binary coded real representation we simply need to execute this command: `java jmetal.experiments.Main NSGAIIBinary ZDT4`. If the problem only allows a binary encoding (e.g., the ZDT5 problem), then the line 22 must be modified

as follows:

```
Object [] problemParams = "Binary".
```

## 6.2 How to use the Mersenne Twister pseudorandom number generator?

The default pseudorandom number generator can be changed by the Mersenne Twister algorithm[1].

The way to do it is quite simple. The `jmetal.util.PseudoRandom` class has a method called `setRandomGenerator()`, so to use the Mersenne Twister just add the following sentence at the beginning of the execution of an algorithm:

```
jmetal.util.PseudoRandom.setRandomGenerator(new MersenneTwisterFast()) ;
```

## 6.3 How to create a new solution type having mixed variables?

jMetal provides many built-in solutions types (`RealSolutionType`, `BinarySolutionType`, `IntSolutionType`, `PermutationSolutionType`, etc.). A solution type specifies the types of the variables a given solution can have, thus defining the solution encoding. In general, the variables of most of the provided solution types are of the same class; thus, a solution type `RealSolutionType` refers to solutions composed of real variables, the `IntSolutionType` incorporates integer variables, and so on. However, defining solutions types having different variable types is very simple in jMetal; in fact, it is not different from defining one containing variables of the same type.

In jMetal 4.0 we provide two solution types having mixed variables `IntRealSolutionType` and `ArrayRealAndBinarySolutionType`; the first one contains integer and real variables, and the second one represents solutions having an array of real values plus a binary string. The code of the `IntRealSolutionType` is included in Listing 6.1.

```
1  //  IntRealSolutionType.java
2  ...
3  package jmetal.encodings.solutionType;
4  ...
5  /**
6   * Class representing  a solution type including two variables: an integer
7   * and a real.
8   */
9  public class IntRealSolutionType extends SolutionType {
10   private int intVariables_ ;
11   private int realVariables_ ;
12
13   /**
14    * Constructor
15    */
16   public IntRealSolutionType(Problem problem, int intVariables, int realVariables)
         throws ClassNotFoundException {
17     super(problem) ;
18     intVariables_ = intVariables ;
19     realVariables_ = realVariables ;
20   } // Constructor
21
22   /**
23    * Creates the variables of the solution
24    * @param decisionVariables
25    * @throws ClassNotFoundException
26    */
27   public Variable[] createVariables() throws ClassNotFoundException {
```

[1] This is a contribution of Jean-Laurent Hippolyte

```
28      Variable [] variables = new Variable[problem_.getNumberOfVariables()];
29
30      for (int var = 0; var < intVariables_; var++)
31        variables[var] = new Int((int)problem_.getLowerLimit(var), (int)problem_.
              getUpperLimit(var));
32
33      for (int var = intVariables_; var < (intVariables_ + realVariables_); var++)
34          variables[var] = new Real(problem_.getLowerLimit(var), problem_.getUpperLimit(
                var));
35
36      return variables ;
37    } // createVariables
38  } // IntRealSolutionType
```

Listing 6.1: `IntRealSolutionType` class

In jMetal 4.0 we provide two solution types having mixed variables `IntRealSolutionType` and `ArrayRealAndBinarySolutionType`; the first one contains integer and real variables, and the second one represents solutions having an array of real values plus a binary string. The code of the `IntRealSolutionType` is included in Listing 6.1. We can observe that the number of integers and real variables is indicated in the class constructor (lines 16-20), and when the `createVariables()` method is called, the required variables are created. The `ArrayRealAndBinarySolutionType` class is even simpler (see 6.2).

```
1  //  ArrayRealAndBinarySolutionType.java
2  ...
3  package jmetal.encodings.solutionType;
4  ...
5   /**
6   * Class representing the solution type of solutions composed of array of reals
7   * and a binary string.
8   * ASSUMPTIONs:
9   * - The numberOfVariables_ field in class Problem must contain the number
10  *   of real variables. This field is used to apply real operators (e.g.,
11  *   mutation probability)
12  * - The upperLimit_ and lowerLimit_ arrays must have the length indicated
13  *   by numberOfVariables_.
14  */
15 public class ArrayRealAndBinarySolutionType extends SolutionType {
16   private int binaryStringLength_ ;
17   private int numberOfRealVariables_ ;
18   /**
19    * Constructor
20    * @param problem
21    * @param realVariables Number of real variables
22    * @param binaryStringLength Length of the binary string
23    * @throws ClassNotFoundException
24    */
25   public ArrayRealAndBinarySolutionType(Problem problem,
26                                         int realVariables,
27                                         int binaryStringLength)
28   throws ClassNotFoundException {
29     super(problem) ;
30     binaryStringLength_    = binaryStringLength ;
31     numberOfRealVariables_ = realVariables ;
32   } // Constructor
33
34   /**
35    * Creates the variables of the solution
36    * @param decisionVariables
37    * @throws ClassNotFoundException
38    */
39   public Variable[] createVariables() throws ClassNotFoundException {
40     Variable [] variables = new Variable[2];
```

```
41
42      variables[0] = new ArrayReal(numberOfRealVariables_, problem_);
43      variables[1] = new Binary(binaryStringLength_);
44      return variables ;
45    } // createVariables
46  } // ArrayRealAndBinarySolutionType
```

Listing 6.2: `listing:ArrayRealAndBinarySolutionType` class

As any other solution type, the key point is that we can define operators to be applied to them. As we observed in the description of the SBX crossover (see Listing 3.5), we can specify in the operator the list of valid types which it can be applied to. in jMetal 4.0 we supply two operators to this solution type:

- `SBXSinglePointCrossover`: applies a SBX crossover to the real variables and a single point crossover to the binary part.
- `PolynomialBitFlipMutation`: the real part of the solution is mutated with a polynomial mutation and a bit flip is applied to the binary string.

If we take a look to the implementation of these two operator we can observe that they do not differ from any other operator, as the SBX crossover detailed in Section 3.1.2.

The solution types and operators cited in this section can be used as templates to develop your own solution types and associated operators if they are not available in jMetal. If you do so and think that your new classes can be useful to other researchers, please feel free to contact us to include them in jMetal.

## 6.4 How to obtain the non-dominated solutions from a file?

Sometimes we face the problem of having a file containing both dominated and non-dominated solutions and we need to get only the non-dominated ones. To achieve this, we provide the `jmetal.util.ExtractParetoFront` utility.

To illustrate the use of this tool, let us suppose that we made 100 independent runs of an algorithm to solve the a given problem. If the result Pareto front approximations are stored in files named FUN.0, FUN.1, ..., FUN.99, we can obtain a file containing all the found solutions, if we are working a UNIX machine, with this sentence:

```
% for i in FUN.* ; do cat $i >> front ; done
```

If we run the utility without any parameters we get the following messages:

```
%  java jmetal.util.ExtractParetoFront
Wrong number of arguments:
Sintaxt: java ExtractParetoFront <file> <dimensions>
  <file> is a file containing points
  <dimensions> represents the number of dimensions of the problem
```

Thus, to select the non-dominated solutions from the previously file we have to execute the utility as follows (we assume that the number of objectives of the problem is 2):

```
%  java jmetal.util.ExtractParetoFront front 2
```

As a result, the program generates a file named "front.pf".

This utility is useful when we run a number of algorithms to solve a problem whose Pareto front is unknown. This way, a reference Pareto front can be easily by joining all the results of all the executions and then selecting the non-dominated solutions.

## 6.5 How to get the latest development version of jMetal?

jMetal is an Open Source software, and its current development version is always available in SourceForge. Since jMetal 4.2 we left Subversion out as control version system, using Git instead.

To get the source code, just enter this Web page:

`http://sourceforge.net/p/jmetal/git/ci/8c9d6c99a76f303e0d190a2172f7c8064e4bb4bb/tree/`.

Here, you can find the instructions to clone the git repository.

# Chapter 7

# What about's

This chapter contains answers to some questions which have not been dealt with before in the manual.

## 7.1 What about developing single-objective metaheuristics with jMetal?

As jMetal is intended to MO optimization, it is clear that to solve SOP problems you could define problems having one objective function and use some of the available MO metaheuristics. However, this could not be the best choice; there are many frameworks available for SO optimization (e.g., EO, Open Beagle, JavaEva, etc.), so you might consider them first before jMetal.

Anyway, developing a SO version of a MO metaheuristic is not difficult: in fact, it is very simple. In jMetal 4.1 we offer the following evolutionary algorithms:

- `gGA`: generational genetic algorithm (GA)
- `ssGA`: steady-state GA
- `scGA`: synchronous cellular GA (cGA)
- `acGA`: asynchronous cGA

Additionally, SO versions of differential evolution, evolution strategies, and PSO are also available.

## 7.2 What about optimized variables and solution types?

When we deal with problems having a few number of variables, the general scheme of creating solutions is reasonably efficient. However, if we have a problem having hundreds or thousands of decision variables, the scheme is inefficient in terms of storage and computing time. For example, if the number of decision variables of the problem is 1000, each solution will contains 1000 Java objects of class `jmetal.base.variable.Real`, one per each `Variable` object, each one storing its proper lower and upper bound values. This wastes memory, but it also implies that manipulating solutions (e.g., creating and copying them) is also computationally expensive.

To cope with this issue, we have defined what we have called "optimization types". The idea is simple: instead of using solutions with an array of N `Real` objects, we will use solutions with an array of N real values. In jMetal 3.0 we incorporated two optimization types based on this idea: `ArrayReal` and `ArrayInt`.

Using optimization types brings some difficulties that have to be solved. Thus, we have now the of using a set of N decision variables, or one decision variable composed of an array of N values, which affects the way variable types are initialized and used. We have solved these problems by using wrapper

objects, which are included in `jmetal.util.wrapper`; in particular, we will show next how to use the `XReal` wrapper.

Let us start by showing the class implementing the Schaffer problem:

```
1  //  Schaffer.java
2  ...
3  package jmetal.problems;
4  ...
5  /**
6   * Class representing problem Schaffer
7   */
8  public class Schaffer extends Problem {
9
10  /**
11   * Constructor.
12   * Creates a default instance of problem Schaffer
13   * @param solutionType The solution type must "Real" or "BinaryReal".
14   */
15   public Schaffer(String solutionType) throws ClassNotFoundException {
16     numberOfVariables_   = 1;
17     numberOfObjectives_  = 2;
18     numberOfConstraints_ = 0;
19     problemName_         = "Schaffer";
20
21     lowerLimit_ = new double[numberOfVariables_];
22     upperLimit_ = new double[numberOfVariables_];
23     lowerLimit_[0] = -100000;
24     upperLimit_[0] =  100000;
25
26     if (solutionType.compareTo("BinaryReal") == 0)
27       solutionType_ = new BinaryRealSolutionType(this) ;
28     else if (solutionType.compareTo("Real") == 0)
29       solutionType_ = new RealSolutionType(this) ;
30     else {
31       System.out.println("Error: solution type " + solutionType + " invalid") ;
32       System.exit(-1) ;
33     }
34   } //Schaffer
35
36
37   /**
38   * Evaluates a solution
39   * @param solution The solution to evaluate
40    * @throws JMException
41   */
42   public void evaluate(Solution solution) throws JMException {
43     Variable[] variable  = solution.getDecisionVariables();
44
45     double [] f = new double[numberOfObjectives_];
46     f[0] = variable[0].getValue() * variable[0].getValue();
47
48     f[1] = (variable[0].getValue() - 2.0) *
49            (variable[0].getValue() - 2.0);
50
51     solution.setObjective(0,f[0]);
52     solution.setObjective(1,f[1]);
53   } //evaluate
54 } //Schaffer
```

The class constructor contains at the end a group of sentences indicating the allowd solution types that can be used to solve the problem (`BinaryRealSolutionType` and `RealSolutionType`. The `evaluate()` method directly accesses the variables to evaluate the solutions. Schaffer's problem is an example of problem that do not need to use optimized types, given that it has only a variable.

Let us consider now problems which can have many variables: some examples are the ZDT, DTLZ,

WFG benchmark problems, and Kursawe's problem. We use this last one as an example. Its constructor is included next:

```
1  public Kursawe(String solutionType, Integer numberOfVariables) throws
       ClassNotFoundException {
2    numberOfVariables_   = numberOfVariables.intValue() ;
3    numberOfObjectives_  = 2                            ;
4    numberOfConstraints_ = 0                            ;
5    problemName_         = "Kursawe"                    ;
6
7    upperLimit_ = new double[numberOfVariables_] ;
8    lowerLimit_ = new double[numberOfVariables_] ;
9
10   for (int i = 0; i < numberOfVariables_; i++) {
11     lowerLimit_[i] = -5.0 ;
12     upperLimit_[i] = 5.0  ;
13   } // for
14
15   if (solutionType.compareTo("BinaryReal") == 0)
16     solutionType_ = new BinaryRealSolutionType(this) ;
17   else if (solutionType.compareTo("Real") == 0)
18     solutionType_ = new RealSolutionType(this) ;
19   else if (solutionType.compareTo("ArrayReal") == 0)
20     solutionType_ = new ArrayRealSolutionType(this) ;
21   else {
22     System.out.println("Error: solution type " + solutionType + " invalid") ;
23     System.exit(-1) ;
24   }
25 } // Kursawe
```

We can observe that at the end of the constructor, we have added the `ArrayRealSolutionType` as a third choice of solution representation to represent the problem. The point now is that accessing directly the decision variables of the problem is cumbersome, because we must distinguish what kind of solution type we are used. The use of the `XReal` wrapper simplifies this task, as we can see in the `evaluate()` method:

```
1  public void evaluate(Solution solution) throws JMException {
2    XReal vars = new XReal(solution) ;
3
4    double aux, xi, xj           ; // auxiliary variables
5    double [] fx = new double[2] ; // function values
6    double [] x = new double[numberOfVariables_] ;
7    for (int i = 0 ; i < numberOfVariables_; i++)
8      x[i] = vars.getValue(i) ;
9
10   fx[0] = 0.0 ;
11   for (int var = 0; var < numberOfVariables_ - 1; var++) {
12     xi = x[var] *  x[var];
13     xj = x[var+1] * x[var+1] ;
14     aux = (-0.2) * Math.sqrt(xi + xj);
15     fx[0] += (-10.0) * Math.exp(aux);
16   } // for
17
18   fx[1] = 0.0;
19
20   for (int var = 0; var < numberOfVariables_ ; var++) {
21     fx[1] += Math.pow(Math.abs(x[var]), 0.8) +
22           5.0 * Math.sin(Math.pow(x[var], 3.0));
23   } // for
24
25   solution.setObjective(0, fx[0]);
26   solution.setObjective(1, fx[1]);
27 } // evaluate
```

Now, the wrapper encapsulates the access to the solutions, by using the `getValue(index)` method. We must note that using the `XReal` wrapper implies that all the operators working with real values must use it too (e.g., the real crossover and mutation operators). Attention must be paid when requesting information about parameters of the problems, as the number of variables. This information is obtained typically by invoking the `getNumberOfVariables()` on the problem to be solved, which in turn returns the value of the state variable `numberOfVariables_`. However, while this works properly when using `RealSolutionType`, that method returns a value of 1 when using `ArrayRealSolutionType`. Let us recall that we are replacing N variables by one variable composed of an array of size N. To avoid this issue, the `ArrayRealSolutionType()` method of class XReal must be used.

To give an idea of the kind of benetifs of using the optimized type `ArrayReal`, we have executed NSGA-II to solve the ZDT1 problem with 1000 and 5000 variables (the default value is 30). The target computer is a MacBook with 2GHz Intel Core 2 Duo, 4GB 1067 MHZ DDR3 RAM, running Snow Leopard; the version of the JDK is 1.6.0_17. The computing times of the algorithm when using the `RealSolutionType` and `ArrayRealSolutionType` solutions types when solving the problem with 1000 variables are 12.5s and 11.4s, respectively; in the case of the problem with 5000 variables, the times are 90s and 69s, respectively.

On the other hand, if we configure ZDT1 with 10,000 variables, the program fails reporting an out-of-memory error when using `RealSolutionType`, while it runs properly when using the optimized type. The error memory can be fixed easily by including the proper flags when launching the Java virtual machine (e.g., `java -Xmx512M java.experiments.Main NSGAII ZDT1`), but this is an example illustrating that the memory savings resulting of using an optimized type can be significant.

# Chapter 8

# Versions and Release Notes

This manual starts with jMetal 2.0, released on December 2008. We detail next the release notes, new features, and changes in the manual in the current release, jMetal 4.3, from the previous versions.

## 8.1 Version 4.3 ($3^{rd}$ January 2013)

**Release notes**

In this release we include a basic support to implement parallel multi-objective metaheuristics by allowing the parallel evaluation of solutions. To parallel version of NSGA-II and SMPSO, named pNSGAII and pSMPSO, respectively, have been developed using this feature.

**New features**

- New package `jmetal.util.parallel`, including the `IParallelEvaluator` interface and the `MultithreadedEvaluat` class.
- Two new algorithms: pNSGAII and pSMPSO, including the `pNSGAII_main`, `pNSGAII_Settings`, `pSMPSO_main`, and `pSMPSO_Setting` classes.
- Two new problems: `Bihn2` and `FourBarTruss`.

**Additions and Changes to the Manual**

- A new chapter has been added (Chapter 5).

## 8.2 Version 4.2 ($14^{th}$ November 2012)

**Release notes**

In this release a new algorithm has been included, a new problem has been added, some methods have been improved, a new random number generator is available, and a new utility is provided.

**New features**

- The MOTSP problem, a multi-objective version of the TSP.
- The NSGAII_MOTSP_main class, aimed to solve the MOTSP problem.

- A new random number generator based on the Mersenne Twister (see Subsection 6.2)[1].
- The `Experiment` class can obtain reference fronts when they are not available in advance[2]. See Section 4.6.
- A new utility, called `jmetal.util.ExtractParetoFront`, has been added to extract the non-dominated solutions front a file containing both dominated and non-dominated solutions. See Section 6.4.

#### Bugs

Some minor bugs have been fixed.

#### Additions and Changes to the Manual

- Section 6.2 has been added.
- Section 6.4 has been added.
- Section 6.5 has been modified.

#### Performance improvements

The following improvements affecting performance has been included:

- The fast non-dominated sorting algorithm has been optimized[3].
- The thread model in the Experiment class has been changed to allow a better use of multicore CPUs[4].

## 8.3 Version 4.0 ($10^{th}$ November 2011)

#### Release notes

In this release the package structure has been modified, the `Operator` class has been redefined, some methods have been improved, and some bugs have been fixed.

#### New features

- The former `jmetal.base` package has been renamed as `jmetal.core`. The reason is to try to keep the same package structure as in the C# version of jMetal we are developing[5]; `base` is a reserved keyword in C#.
- The package structure has been modified. Now, the former package `jmetal.base.operator` becomes `jmetal.operators`, and `jmetal.base.variable` and `jmetal.base.solutionType` are now `jmetal.encodings.variable` and `jmetal.encodings.solutionType`.
- New encoding: `ArrayRealAndBinarySolutionType` allows a representation combining a array of reals and a binary string.

[1] Contribution of Jean Laurent Hippolyte
[2] Contribution of Jorge Rodríguez
[3] Contribution of Guillaume Jacquenot
[4] Contribution of Jorge Rodríguez
[5] `http://jmetalnet.sourceforge.net`

- Two new operators: `PolinomialBitFlipMutation` (Package: `jmetal.operators.mutation`) and `SBXSinglePointCrossver` (Package: `jmetal.operators.crossover`), intended to be applied to `ArrayRealAndBinarySolutionType` solutions.
- The operators can be configured now when their constructor is invoked (see Subsection 3.1.2, which includes the example of the SBX crossover in Listing 3.3).

#### Removed features

The GUI is not available in this release.

#### Bugs

Many minor bugs have been fixed.

#### Additions and Changes to the Manual

- Section 6 has been updated.
- Section 3 has been updated.

#### Performance improvements

Many sorting methods used in the framework have been optimized, and the fact that the new approach to create operators is more efficient (previously, all the operator parameters where checked each time on operator was invoked) has lead to significant performance improvements. To give an idea of the benefits of using the new version in terms of computing time, we include two examples. The target computer is a MacBook with 2GHz Intel Core 2 Duo, 4GB 1067 MHZ DDR3 RAM, running Mac OS X Lion 10.7.2 (11C74); the version of the JDK is 1.6.0_26.

In the first example, we run NSGA-II to solve problem Kursawe using typical settings (population size: 100, number of evaluations: 25000). The times obtained when running the algorithms with version 3.1 and 4.0 of jMetal are about 2.6s and 1.8s, respectively. In the second example, we execute MOEAD to solve the LZ09_F1 problem with standard settings (population size: 300, number of evaluations: 300000), getting times in the order of 14.7s and 3.3s , respectively. That is, the time reductions are about the 30% and the 77% in the two examples, respectively.

## 8.4 Version 3.1 ($1^{st}$ October 2010)

#### Release notes

This release includes new algorithms (single and multi-objective) and some bugs have been fixed.

#### New features

- A new solution type: `ArrayRealAndBinarySolutionType` (Package: `jmetal.base.solutionType`).
- Two new operators: `PolinomialBitFlipMutation` (Package: `jmetal.base.operator.mutation`) and `SBXSinglePointCrossver` (Package: `jmetal.base.operator.crossover`), intended to be applied to `ArrayRealAndBinarySolutionType` solutions.
- The SMS-EMOA algorithm (contributed by Simon Wessing).
- A single-objective version of a PSO algorithm.
- MOEA/D-DRA, a version of MOEA/D presented in CEC09.

### Bugs

Bugs in the following packages and classes have been fixed:

- Class `jmetal.base.operator.crossover.PMXCrossover`

### Additions and Changes to the Manual

- Section 1 has been modified.

## 8.5 Version 3.0 ($28^{th}$ February 2010)

### Release notes

This release contains changes in the architecture of the framework, affecting the way the solution representations are enconded (by using solution types). Other significant contribution the `jmetal.gui` package, which includes two graphical tools.

### New features

- A new approach to define solution representations (Section 3.1.1).
- Two new variable representations: ArrayInt and ArrayReal (packages: `jmetal.base.variable.ArrayInt` and `jmetal.base.variable.ArrayReal`).
- Two wrapper classes, `XReal` and `XInt`, to encapsulate the access to the different included representations of real and integer types, respectively.
- Two graphical tools: the Simple Execution Support GUI (`jmetal.gui.SimpleExecutionSupportGUI`) and the Experiment Support GUI (`jmetal.gui.ExperimentsSupportGUI`).
- Single-objective versions of a number of genetic algorithms (steady-state, generational, synchronous cellular, asynchronous cellular), differential evolution, and evolution strategies (elitist and non-elitist).
- A parallel version of MOEA/D.

### Bugs

Bugs in the following packages and classes have been fixed:

- Class `jmetal.metaheuristics.moead.MOEAD`

### Additions and Changes to the Manual

- Section 3 has been modified
- Added Chapter 4
- Added Chapter 6
- Added Chapter 7
- Chapter FAQ has been removed

## 8.6 Version 2.2 (28$^{nd}$ May 2009)

### Release notes

This release contains as main contributions two new algorithms (random search and a steady-state version of NSGA-II), an update of the experiments package, and several bugs has been fixed.

### New features

- A random search algorithm (package: `jmetal.metaheuristic.randomSearch`).
- A steady-state version of NSGA-II (class: `jmetal.metaheuristic.nsgaII.ssNSGAII`). To configure and running the algorithm, the `jmetal.metaheuristic.nsgaII.NSGAII_main` can be used, indicating `ssNSGAII` instead of `NSGAII` in line 84; alternatively, a `ssNSGAII_Settingss` class is available in `jmetal.experiments.settings`. We used this algorithm in [7].
- The `experiments` package allows to generate latex tables including the application of the Wilcoxon statistical test to the results of jMetal experiments. Additionally, it can be indicated whether to generate notched or not notched boxplots (Chapter 4).
- A first approximation to the use of threads to run experiments in parallel has been included (Section 4.8).

### Bugs

Bugs in the following packages have been fixed:

- `jmetal.problems.ConstrEx`.
- `jmetal.metaheuristics.paes.Paes_main`.
- `jmetal.metaheuristics.moead.MOEAD`.

### Additions and Changes to the Manual

- Chapter 4

## 8.7 Version 2.1 (23$^{rd}$ February 2009)

### Release notes

This release contains as main contribution the support of automatically generating R[6] scripts, which when compiled produce figures representing boxplots of the results.

### New features

- Class `jmetal.experiments.Experiment`: method `generateRScripts()`.
- The IBEA algorithm [41] (package: `jmetal.metaheuristic.ibea`). This algorithm is included for testing purposes; we have not validated the implementation yet.

### Bugs

- A bug in the `jmetal.base.operator.crossover.SinglePointCrossover` class has been fixed.

[6] `http://www.r-project.org/`

## 8.8 Version 2.0 ($23^{rd}$ December 2008)

### Release notes

This release contains as main contribution the package `jmetal.experiments`, which contains a set of classes intended to facilitate carrying out performance studies with the algorithms included in the framework. As a consequence, a new class `jmetal.experiments.Settings` has been defined to allow to set the parameters of the metaheuristics in a separate class, so that the configurations of the algorithms can be reused easily. In versions previous to jMetal 2.0, the settings were specified in a `main` Java program associated to each technique, what made the reusing of the algorithm configurations difficult.

### New features

- Package `jmetal.experiments`.
- The Additive Epsilon indicator (package: `jmetal.qualityIndicators`).
- CEC2008 benchmark (package: `jmetal.problems`).

### Known Bugs

### Additions and Changes to the Manual

# Bibliography


[1] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA — a platform and programming language independent interface for search algorithms. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494 – 508, Berlin, 2003. Springer.

[2] D.W. Corne, N.R. Jerram, J.D. Knowles, and M.J. Oates. PESA-II: Region-based selection in evolutionary multiobjective optimization. In *Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 283–290. Morgan Kaufmann, 2001.

[3] K. Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001.

[4] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. Scalable test problems for evolutionary multiobjective optimization. In Ajith Abraham, Lakhmi Jain, and Robert Goldberg, editors, *Evolutionary Multiobjective Optimization. Theoretical Advances and Applications*, pages 105–145. Springer, USA, 2005.

[5] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[6] J.J. Durillo, A.J. Nebro, and E. Alba. The jMetal framework for multi-objective optimization: Design and architecture. In *CEC 2010*, pages 4138–4325, Barcelona, Spain, July 2010.

[7] J.J. Durillo, A.J. Nebro, F. Luna, and E. Alba. On the effect of the steady-state selection scheme in multi-objective genetic algorithms. In *5th International Conference, EMO 2009*, volume 5467 of *Lecture Notes in Computer Science*, pages 183–197, Nantes, France, April 2009. Springer Berlin / Heidelberg.

[8] J.J. Durillo, A.J. Nebro, F. Luna, B. Dorronsoro, and E. Alba. jMetal: a Java framework for developing multi-objective optimization metaheuristics. Technical Report ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos, 2006.

[9] Juan J. Durillo and Antonio J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760 – 771, 2011.

[10] Juan J. Durillo, Antonio J. Nebro, Francisco Luna, and Enrique Alba. Solving three-objective optimization problems using a new hybrid cellular genetic algorithm. In G. Rudolph, T. Jensen, S. Lucas, C. Poloni, and N. Beume, editors, *Parallel Problem Solving from Nature - PPSN X*, volume 5199 of *Lecture Notes in Computer Science*, pages 661–670. Springer, 2008.

[11] M. Emmerich, N. Beume, and B. Naujoks. An emo algorithm using the hypervolume measure as selection criterion. In C.A. Coello, A. Hernández, and E. Zitler, editors, *Third International Conference on Evolutionary MultiCriterion Optimization, EMO 2005*, volume 3410 of *LNCS*, pages 62–76. Springer, 2005.

[12] H. Eskandari, C. D. Geiger, and G. B. Lamont. FastPGA: A dynamic population sizing approach for solving expensive multiobjective optimization problems. In S. Obayashi, K. Deb, C. Poloni, T. Hiroyasu, and T. Murata, editors, *Evolutionary Multi-Criterion Optimization. 4th International Conference, EMO 2007*, volume 4403 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2007.

[13] C.M. Fonseca and P.J. Flemming. Multiobjective optimization and multiple constraint handling with evolutionary algorithms - part ii: Application example. *IEEE Transactions on System, Man, and Cybernetics*, 28:38–47, 1998.

[14] D. Greiner, J.M. Emperador, G. Winter, and B. Galván. Improving computational mechanics optimum design using helper objectives: An application in frame bar structures. In S. Obayashi, K. Deb, C. Poloni, T. Hiroyasu, and T. Murata, editors, *Fourth International Conference on Evolutionary MultiCriterion Optimization, EMO 2007*, volume 4403 of *Lecture Notes in Computer Science*, pages 575–589, Berlin, Germany, 2006. Springer.

[15] S. Huband, P. Hingston, L. Barone, and L. While. A review of multiobjective test problems and a scalable test problem toolkit. *IEEE Transactions on Evolutionary Computation*, 10(5):477–506, October 2006.

[16] J. Knowles, L. Thiele, and E. Zitzler. A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. Technical Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, 2006.

[17] J. D. Knowles and D. W. Corne. Approximating the nondominated front using the pareto archived evolution strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

[18] S. Kukkonen and J. Lampinen. GDE3: The third evolution step of generalized differential evolution. In *IEEE Congress on Evolutionary Computation (CEC'2005)*, pages 443 – 450, 2005.

[19] A. Kurpati, S. Azarm, and J. Wu. Constraint handling improvements for multi-objective genetic algorithms. *Structural and Multidisciplinary Optimization*, 23(3):204–213, 2002.

[20] F. Kursawe. A variant of evolution strategies for vector optimization. In H.P. Schwefel and R. Männer, editors, *Parallel Problem Solving for Nature*, pages 193–197, Berlin, Germany, 1990. Springer-Verlag.

[21] H. Li and Q. Zhang. Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii. *IEEE Transactions on Evolutionary Computation*, 12(2):284–302, April 2009.

[22] A.J. Nebro, E. Alba, G. Molina, F. Chicano, F. Luna, and J.J. Durillo. Optimal antenna placement using a new multi-objective chc algorithm. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 876–883, New York, NY, USA, 2007. ACM Press.

[23] A.J. Nebro, J.J. Durillo, J. García-Nieto, C.A. Coello Coello, F. Luna, and E. Alba. Smpso: A new pso-based metaheuristic for multi-objective optimization. In *2009 IEEE Symposium on Computational Intelligence in Multicriteria Decision-Making (MCDM 2009)*, pages 66–73. IEEE Press, 2009.

[24] A.J. Nebro, J.J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. Design issues in a multiobjective cellular genetic algorithm. In S. Obayashi, K. Deb, C. Poloni, T. Hiroyasu, and T. Murata, editors, *Evolutionary Multi-Criterion Optimization. 4th International Conference, EMO 2007*, volume 4403 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2007.

[25] AJ Nebro, JJ Durillo, F Luna, B Dorronsoro, and E Alba. Mocell: A cellular genetic algorithm for multiobjective optimization. *Int. J. Intell. Syst.*, 24(7):726–746, 2009.

[26] Antonio J. Nebro, Juan J. Durillo, C.A. Coello Coello, Francisco Luna, and Enrique Alba. Design issues in a study of convergence speed in multi-objective metaheuristics. In G. Rudolph, T. Jensen, S. Lucas, C. Poloni, and N. Beume, editors, *Parallel Problem Solving from Nature - PPSN X*, volume 5199 of *Lecture Notes in Computer Science*, pages 763–772. Springer, 2008.

[27] Antonio J. Nebro, Francisco Luna, Enrique Alba, Bernabé Dorronsoro, Juan J. Durillo, and Andreas Beham. AbYSS: Adapting Scatter Search to Multiobjective Optimization. *IEEE Transactions on Evolutionary Computation*, 12(4), August 2008.

[28] A. Osyczka and S. Kundo. A new method to solve generalized multicriteria optimization problems using a simple genetic algorithm. *Structural Optimization*, 10:94–99, 1995.

[29] T. Ray, K. Tai, and K.C. Seow. An Evolutionary Algorithm for Multiobjective Optimization. *Engineering Optimization*, 33(3):399–424, 2001.

[30] M. Reyes and C.A. Coello Coello. Improving PSO-based multi-objective optimization using crowding, mutation and $\epsilon$-dominance. In C.A. Coello, A. Hernández, and E. Zitler, editors, *Third International Conference on Evolutionary MultiCriterion Optimization, EMO 2005*, volume 3410 of *LNCS*, pages 509–519. Springer, 2005.

[31] J.D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In J.J. Grefensttete, editor, *First International Conference on Genetic Algorithms*, pages 93–100, Hillsdale, NJ, 1987.

[32] N. Srinivas and K. Deb. Multiobjective function optimization using nondominated sorting genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1995.

[33] M. Tanaka, H. Watanabe, Y. Furukawa, and T. Tanino. Ga-based decision support system for multicriteria optimization. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume 2, pages 1556–1561, 1995.

[34] D. A. Van Veldhuizen and G. B. Lamont. Multiobjective Evolutionary Algorithm Research: A History and Analysis. Technical Report TR-98-03, Dept. Elec. Comput. Eng., Graduate School of Eng., Air Force Inst. Technol., Wright-Patterson, AFB, OH, 1998.

[35] Q. Zhang, A. Zhou, S. Z. Zhao, P. N. Suganthan, W. Liu, and S. Tiwari. Multiobjective optimization test instances for the cec 2009 special session and competition. Technical Report CES-487, University of Essex and Nanyang Technological University, Essex, UK and Singapore, September 2008.

[36] A. Zhou, Y. Jin, Q. Zhang, B. Sendhoff, and E. Tsang. Combining model-based and genetics-based offspring generation for multi-objective optimization using a convergence criterion. In *2006 IEEE Congress on Evolutionary Computation*, pages 3234–3241, 2006.

[37] E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195, Summer 2000.

[38] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. In K. Giannakoglou, D. Tsahalis, J. Periaux, P. Papailou, and T. Fogarty, editors, *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, pages 95–100, Athens, Greece, 2002.

[39] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

[40] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, and V.G. Da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7:117–132, 2003.

[41] Eckart Zitzler and Simon Künzli. Indicator-based selection in multiobjective search. In Xin Yao et al., editors, *Parallel Problem Solving from Nature (PPSN VIII)*, pages 832–842, Berlin, Germany, 2004. Springer-Verlag.