jMetal 3.0 User Manual

Antonio J. Nebro, Juan J. Durillo

Draft. date March 12, 2010

Contents

List of Figures

List of Tables

Preface

This document contains the draft manual of jMetal, a framework for multi-objective optimization developed in 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 metaheuristics. In November 2006 we decided to put the package publicly available in `http://neo.lcc.uma.es/metal/`, and we moved it to SourceForge in November 2008 (`http://jmetal.sourceforge.net`). As of today, it has been downloaded from SouceForge more than 1600 times. jMetal is an open source software, and it can be downloaded from `http://sourceforge.net/projects/jmetal`.

This manual is structured into seven chapters, covering issues such as installation, architecture description, examples of use, 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 the development, experimentation, and study of algorithms for solving multi-objective optimization problems (MOPs). jMetal provides a rich set of classes which can be used as the building blocks of multi-objective metaheuristics; 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 kind of experiments. The inclusion of a number of classical and state-of-the-art algorithms, many problems usually included in performance studies, and quality indicators allows also to study the basic principles of multi-objective optimization for newcomers.

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 metaheuristic algorithm, publicly available[1], it is 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 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. In general, we found that most of C/C++ multi-objective software packages are difficult to understand and use. 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 jMetal, Java-based framework designed for multi-objective optimization using metaheuristics.

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 toolboxs 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 Desing goals

We imposed ourselves as design goals that jMetal should be simply 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 modern multi-objective optimization algorithms: NSGA-II [5], SPEA2 [35], PAES [14], PESA-II [2], OMOPSO [27], MOCell [22], AbYSS [24], MOEA/D [18], Densea [11], CellDE [8], GDE3 [15], FastPGA [9], IBEA [38], SMPSO [20], MOCHC [19].

- A rich set of test problems including:

  - Problem families: Zitzler-Deb-Thiele (ZDT) [34], Deb-Thiele-Laumanns-Zitzler (DTLZ) [4], Walking-Fish-Group (WFG) test problems [12]), CEC2009 (unconstrained problems) [32], and the Li-Zhang benchmark [18].
  - Classical problems: Kursawe [17], Fonseca [10], Schaffer [28].
  - Constrained problems: Srinivas[29], Tanaka [30], Osyczka2 [25], Constr_Ex [5], Golinski [16], Water [26].

- Implementation of a number of widely used quality indicators: Hypervolume [36], Spread [5], Generational Distance [31], Inverted Generational Distance [31], Epsilon [13].

- 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 [7].

- 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 using modern multi-core CPUs.
- A Graphical User Interface (GUI) for giving support in solving problems and performing experimental studies.
- 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 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 `src` 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).
3. Making use of the jMetal Graphical User Interface (GUI) (see Section 4.8).

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 5.0.

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*. It is advisable to build the project before executing an algorithm the first time; to do that, select *Build → Build Main Project* (altenatively, push F11). Once the project has been built, put the mouse pointer on the file name in the file tree to run the algorithm, click on the left 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 directory chosen to deploy the project (directory `JMETALHOME`).

2.4 Eclipse

We describe next how to compile and use jMetal using Eclipse 3.4.

2.4.1 Creating the project

1. Select *File → New → Java Project*.
2. Write a project name (e.g. jMetal), select *Create project from existing source*, and write `JMETALHOME` as *Directory*. 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 `JMETALHOME`.

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 encondings, 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 behaviour 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 Fig. 3.3 which shows the code of the `RealSolutionType` class (we have omitted irrelevant details), 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. Fig. 3.4 shows the code required for this new type of solution.

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).

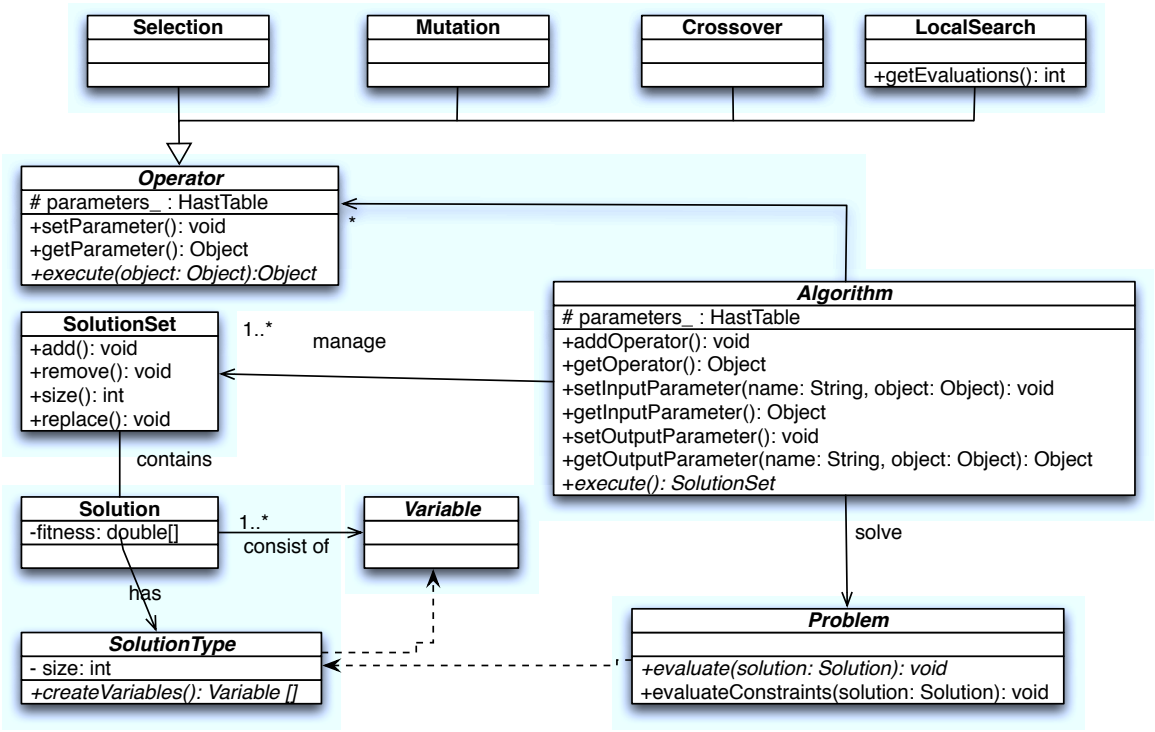

Figure 3.1: jMetal class diagram.

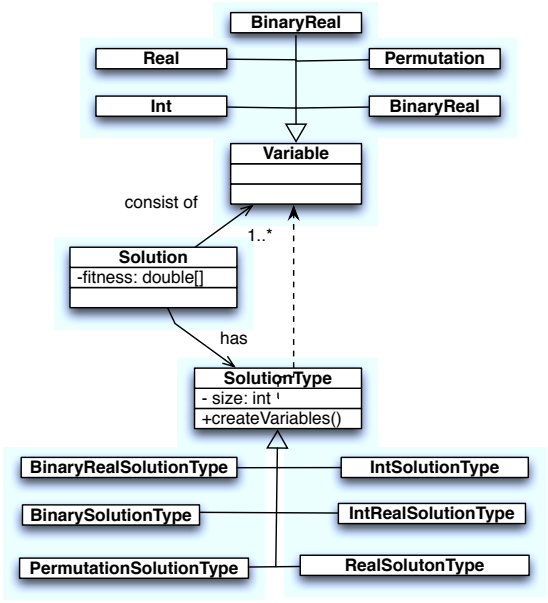

Figure 3.2: Elements describing solution representations into jMetal.

```
public class RealSolutionType extends SolutionType {
  // Constructor
  public RealSolutionType(Problem problem)  {
    ...
  } // Constructor

  public Variable[] createVariables() {
    Variable[] variables = new Variable[problem_.getNumberOfVariables()];

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

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

Figure 3.3: Code of the *RealSolutionType* class, which represents solutions of real variables.

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

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

    return variables;
  } // createVariable
```

Figure 3.4: Code of the `createVariables()` method for creating solutions consisting on a Real, an Integer, and a Permutation

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 operatos 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.

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.

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. Fig. 3.5 shows the code used for implementing the known Kursawe's problem (we have omitted again irrelevant code). As we can observe observe, it extends class `Problem` (line 1). After that, a constructor method is defined for creating instances of this problem (lines 3-16), which has a first parameter a string containing a solution type identifier. The basic features of the problem (number of variables, number of objectives, and number of constraints) are defined next (lines 4-6). The sentences between lines 9-12 are used to specify that the allowed solution representations are real and binary-coded real, so the corresponding `SolutionType` objects are created and assigned to a state variable.

After the constructor, the `evaluate()` method is redefined (lines 18-29); in this method, after computing the two objective function values, they are stored into the solution by using the `setObjective` method of `Solution` (lines 27 and 28).

Many of commonly used benchmark problems are already included in jMetal. Examples are the ones proposed by Zitzler-Deb-Thiele (ZDT) [34], Deb-Thiele-Laumanns-Zitzler (DTLZ) [4], Walking-Fish-Group (WFG) test problems [12]), and the Li-Zhang benchmark [18].

```
1. public class Kursawe extends Problem {
2. // Constructor
3.   public Kursawe(String solutionType, Integer numberOfVariables) {
4.      numberOfVariables_   = numberOfVariables ;
5.      numberOfObjectives_  = 2                 ;
6.      numberOfConstraints_ = 0                 ;
7.      problemName_         = "Kursawe"         ;
8.
9.      if (solutionType.compareTo("BinaryReal") == 0)
10.       solutionType_ = new BinaryRealSolutionType(this) ;
11.     else if (solutionType.compareTo("Real") == 0)
12.       solutionType_ = new RealSolutionType(this) ;
16.  } // Kursawe
17.
18.  /**
19.  * Evaluates a solution
20.  */
21.  public void evaluate(Solution solution)  {
22.    double f1 = 0.0, f2 = 0.0;
23.    // computing f1 value
24.    ...
25.    // computing f2 value
26.    ...
27.    solution.setObjective(0, f1);
28.    solution.setObjective(1, f2);
29.  } // evaluate
30. } // Kursawe
```

Figure 3.5: Code of the class implementing problem Kursawe.

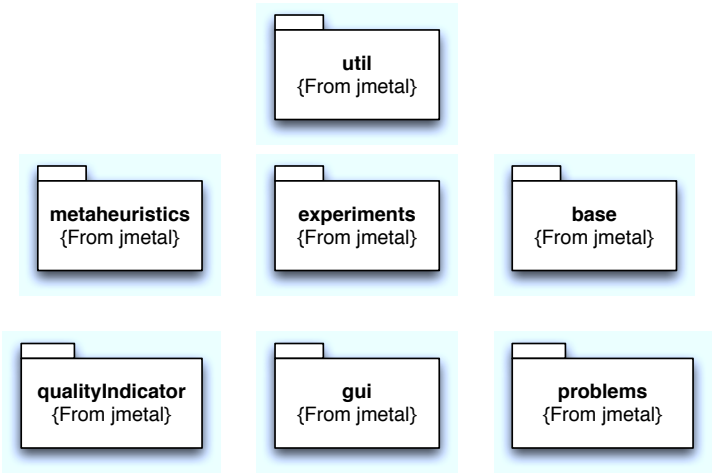

Figure 3.6: jMetal packages.

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 [35], PAES [14], OMOPSO [27], MOCell [21], AbYSS [24], MOEA/D [18], GDE3 [15], IBEA [38], or SMPSO [20].

3.2 jMetal Package Structure

jMetal is composed of six packages, which are depicted in Figure 3.6. The packages are `base`, `problems`, `metaheuristics`, `qualityIndicators`, `util`, `experiments`, and `gui`. We briefly describe them next.

3.2.1 Package `base`

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. The full components are included in Figure 3.7.

We can observe that `jmetal.base` contains the following packages:

- `jmetal.base.operator`: This package contains different kinds of operator objects, including comparators, crossover, mutation, selection, and local search operators. We give next an example of an operator of each type:
  - `jmetal.base.operator.comparator.DominanceComparator`: This comparator takes two solutions S_1 and S_2 and returns -1 if S_1 dominates S_2, 1 if S_2 dominates S_1, and 0 if both solutions are non-dominated.

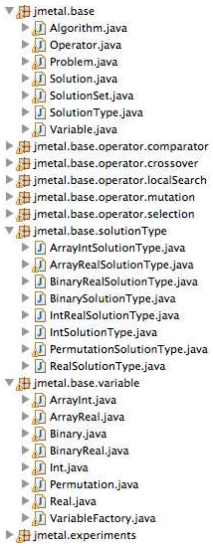

Figure 3.7: Package `jmetal.base`.

  - `jmetal.base.operator.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.base.operator.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.base.operator.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.base.operator.localSearch.MutationLocalSearch`: These operators are intended to apply local search strategies to a given solution. The `MutationLocalSearch`, used in the AbYSS algorithm [24], 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.

- `jmetal.base.solutionType`: Here we find the solution types available in the framework, such as `BinarySolutionType`, `RealSolutionType`, `IntSolutionType`, etc.

- `jmetal.base.variable`: The different representations of decision variables are included in this package. Currently, the representations are `Binary`, `BinaryReal` (binary-coded real), `Int`, `Permutation`, `ArrayInt`, and `ArrayReal`.

3.2.2 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). Futhermore, we can find many other problems (Fonseca, Kursawe, Schaffer, OKA2, etc.)

3.2.3 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, Densea, and MOEA/D.

Although jMetal is aimed at multi-objective optimization, a number of single objective algorithms are implemented in the `jmetal.metaheuristics.singleObjective` package.

3.2.4 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, etc.

3.2.5 Package `gui`

This package contains classes related to jMetal's GUI, which is described in Section 4.8.

3.2.6 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 [31]
- Inverted generational distance [31]
- Additive epsilon [37]
- Spread [5]
- Generalized spread [33]
- Hypervolume [36]

3.2.7 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 3.8. 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 we can add to an algorithm new operations with the method `addOperation()`; these operations are accessed in the algorithm by invoking `getOperation()`. Similarly, we can pass parameters to an algorithm (methods `setInputParameter()`

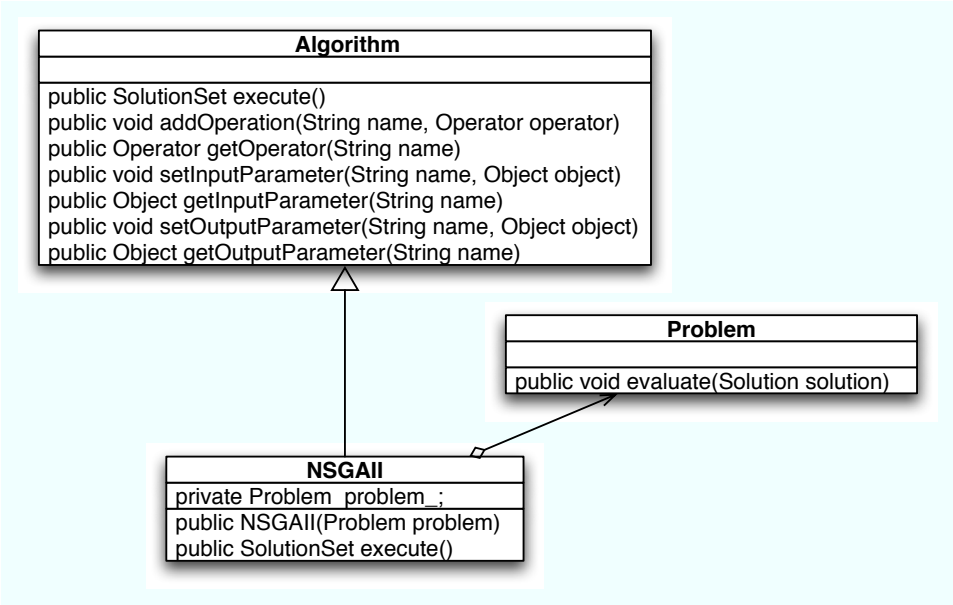

Figure 3.8: UML diagram of `NSGAII`.

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, as well as the implementation on `execute()`. Next, we analize the implementation of the `NSGAII` class in jMetal (file `jmetal/metaheuristics/nsgaii/NSGAII.java`). The basic code structure implementing the class is presented in Figure 3.9.

Let us focus on the method `execute()` (see Figure 3.10). First, we comment the objects needed to implement the algorithm. The parameters to specify the population size and the maximum number of evaluations are declared in lines 37-38. The next variable, `evaluations`, is a counter of the number of computed evaluations. The objects declared in lines 41-42 are needed to illustrate the use of quality indicators inside the algorithms; we will explain their use later; lines 45-47 contain the declaration of the populations needed to implement NSGA-II: the current population, an offspring population, an auxiliar population used to join the other two. Next, we find the three genetic operators (lines 49-51) and a `Distance` object (from package `jmetal.util`), which implements the crowding distance.

Once we have declared all the needed objects, we proceed to initialize them (Figure 3.11). The parameters `populationSize` and `maxEvaluations` are input parameters whose values are obtained in lines 56-57; 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 61-62), and finally the mutation, crossover, and selection operators are obtained (lines 67-69).

The initial population is initialized in the loop included in Figure 3.12. 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 Figure 3.13. We can observe the inner loop performing the generations (lines 87-103), 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 87 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 Figure 3.14, which basically follows the algorithm described in [5].

The piece of code in Figure 3.15 illustrates the use of quality indicators inside a metaheuristic.

```
2 * NsgaII.java
6 package jmetal.metaheuristics.nsgaII;
7
8 import jmetal.base.*;
10
13 * This class implements the NSGA-II algorithm.
15 public class NSGAII extends Algorithm {
16
18 * stores the problem to solve
20 private Problem problem_;
21
22 /**
23 * Constructor
24 * @param problem Problem to solve
25 */
26 public NSGAII(Problem problem){
29
30 * Runs of the NSGA-II algorithm.
36 public SolutionSet execute() throws JMException {
169 } // NSGA-II
```

Figure 3.9: Scheme of the implementation of class `NSGAII`.

```
 36   public SolutionSet execute() throws JMException {
 37     int populationSize ;
 38     int maxEvaluations ;
 39     int evaluations    ;
 40
 41     QualityIndicator indicators          ; // QualityIndicator object
 42     int              requiredEvaluations ; // Use in the example of use of the
 43                                            // indicators object (see below)
 44
 45     SolutionSet population          ;
 46     SolutionSet offspringPopulation ;
 47     SolutionSet union               ;
 48
 49     Operator mutationOperator  ;
 50     Operator crossoverOperator ;
 51     Operator selectionOperator ;
 52
 53     Distance distance = new Distance()  ;
        ...
168   } // execute
```

Figure 3.10: `execute()` method: declaring objects.

```
 36   public SolutionSet execute() throws JMException {
        ...
 55     //Read the parameters
 56     populationSize    = ((Integer)getInputParameter("populationSize")).intValue();
 57     maxEvaluations    = ((Integer)getInputParameter("maxEvaluations")).intValue();
 58     indicators        = (QualityIndicator)getInputParameter("indicators") ;
 59
 60     //Initialize the variables
 61     population  = new SolutionSet(populationSize);
 62     evaluations = 0;
 63
 64     requiredEvaluations = 0 ;
 65
 66     //Read the operators
 67     mutationOperator  = operators_.get("mutation");
 68     crossoverOperator = operators_.get("crossover");
 69     selectionOperator = operators_.get("selection");
        ...
168   } // execute
```

Figure 3.11: `execute()` method: initializing objects.

```
36  public SolutionSet execute() throws JMException {
    ...
71    // Create the initial solutionSet
72    Solution newSolution;
73    for (int i = 0; i < populationSize; i++) {
74      newSolution = new Solution(problem_);
75      problem_.evaluate(newSolution);
76      problem_.evaluateConstraints(newSolution);
77      evaluations++;
78      population.add(newSolution);
79    } //for
    ...
```

Figure 3.12: `execute()` method: initalizing the population.

```
36  public SolutionSet execute() throws JMException {
      ...
81    // Generations ...
82    while (evaluations < maxEvaluations) {
83
84      // Create the offSpring solutionSet
85      offspringPopulation  = new SolutionSet(populationSize);
86      Solution [] parents = new Solution[2];
87      for (int i = 0; i < (populationSize/2); i++){
88        //obtain parents
89        if (evaluations < maxEvaluations) {
90        parents[0] = (Solution)selectionOperator.execute(population);
91        parents[1] = (Solution)selectionOperator.execute(population);
92        Solution [] offSpring = (Solution []) crossoverOperator.execute(parents);
93        mutationOperator.execute(offSpring[0]);
94        mutationOperator.execute(offSpring[1]);
95        problem_.evaluate(offSpring[0]);
96        problem_.evaluateConstraints(offSpring[0]);
97        problem_.evaluate(offSpring[1]);
98        problem_.evaluateConstraints(offSpring[1]);
99        offspringPopulation.add(offSpring[0]);
100       offspringPopulation.add(offSpring[1]);
101       evaluations += 2;
102     } if
103   } // for
        ...
168 } // execute
```

Figure 3.13: `execute()` method: main loop.

```
 36  public SolutionSet execute() throws JMException {
       ...
 81    // Generations ...
 82    while (evaluations < maxEvaluations) {
         ...
108
109      // Create the solutionSet union of solutionSet and offSpring
107      union = ((SolutionSet)population).union(offspringPopulation);
108
109      // Ranking the union
110      Ranking ranking = new Ranking(union);
111
112      int remain = populationSize;
113      int index  = 0;
114      SolutionSet front = null;
115      population.clear();
116
117      // Obtain the next front
118      front = ranking.getSubfront(index);
119
120      while ((remain > 0) && (remain >= front.size())){
121        //Assign crowding distance to individuals
122        distance.crowdingDistanceAssignment(front, problem_.getNumberOfObjectives());
123        //Add the individuals of this front
124        for (int k = 0; k < front.size(); k++ ) {
125          population.add(front.get(k));
126        } // for
127
128        //Decrement remain
129        remain = remain - front.size();
130
131        //Obtain the next front
132        index++;
133        if (remain > 0) {
134          front = ranking.getSubfront(index);
135        } // if
136      } // while
137
138      // Remain is less than front(index).size, insert only the best one
139      if (remain > 0) {  // front contains individuals to insert
140        distance.crowdingDistanceAssignment(front,problem_.getNumberOfObjectives());
141        front.sort(new jmetal.base.operator.comparator.CrowdingComparator());
142        for (int k = 0; k < remain; k++) {
143          population.add(front.get(k));
144        } // for
145
146        remain = 0;
147      } // if
         ...
160    } // while
       ...
171  } // execute
```

Figure 3.14: `execute()` method: ranking and crowding.

```
 36   public SolutionSet execute() throws JMException {
        ...
 81     // Generations ...
 82     while (evaluations < maxEvaluations) {
          ...
149       // This piece of code shows how to use the indicator object into the code
150       // of NSGA-II. In particular, it finds the number of evaluations required
151       // by the algorithm to obtain a Pareto front with a hypervolume higher
152       // than the hypervolume of the true Pareto front.
153       if ((indicators != null) &&
154           (requiredEvaluations == 0)) {
155         double HV = indicators.getHypervolume(population) ;
156         if (HV >= (0.98 * indicators.getTrueParetoFrontHypervolume())) {
157           requiredEvaluations = evaluations ;
158         } // if
159       } // if
100    } // while
       ...
168  } // execute
```

Figure 3.15: `execute()` method: using the hypervolume quality indicator.

```
 36   public SolutionSet execute() throws JMException {
        ...
162     // Return as output parameter the required evaluations
163     setOutputParameter("evaluations", requiredEvaluations);
164
165     // Return the first non-dominated front
166     Ranking ranking = new Ranking(population);
167     return ranking.getSubfront(0);
168  } // execute
```

Figure 3.16: `execute()` method: end of the method.

In particular, it shows the code we used in [23] 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 153), 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 [23] for further details). In case of sucess, the variable `requiredEvaluations` is assigned the current number of function evaluations (line 157). 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 154.

The last sentences of the `execute()` method are included in Figure 3.16. In line 163 we can observe that the variable requiredEvaluations is returned as output parameter. Finally, we apply ranking to the resulting population to return only non-dominated solutions (lines 166-167).

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 Figure 3.17, which contains the import section of the program. The logging classes (lines 38-39) are needed to use a logger object, which allows us to log the messages of the program.

The code in Figure 3.18 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.).

```
...
22 package jmetal.metaheuristics.nsgaII;
23
24 import jmetal.base.*;
25 import jmetal.base.operator.crossover.*   ;
26 import jmetal.base.operator.mutation.*    ;
27 import jmetal.base.operator.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
38 import java.util.logging.FileHandler;
39 import java.util.logging.Logger;
40
41 import jmetal.qualityIndicator.QualityIndicator;
...
```

Figure 3.17: `NSGAII_main`: importing packages.

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

Figure 3.18: `NSGAII_main`: main method.

- `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, 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.

Figure 3.19 contains the code used to declare the objects required to execute the algorithm (lines 62-68). The logger object is initialized in lines 71-73, and the log messages will be written in a file named "NSGAII_main.log". The sentences included between lines 76 and 93 process the arguments of the `main()` method. The default problem is indicated after line 75. 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 95. Next line contains the sentence that should be used if the steady-state version of NSGA-II is intended to be executed.

Once an object representing the algorithm to run has been created, it must be configured. In the code included in Figure 3.20, the input parameters are set in lines 99-100, the crossover and mutation operators are specified in lines 103-109, and the selection operator is chosen in line 112. Once the operators have been specified, they are added to the algorithm object in lines 115-117. The sentence in line 120 sets the indicator object as input parameter.

When the algorithm has been configured, it is executed by invoking its `execute()` method (line 124 in Figure 3.21). When it has finished, the running time is reported, and the obtained solutions and

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

Figure 3.19: `NSGAII_main`: declaring objects, processing the arguments of `main()`, and creating the algorithm.

```
     ...
 57  public static void main(String [] args) throws
       ...
 98    // Algorithm parameters
 99    algorithm.setInputParameter("populationSize",100);
100    algorithm.setInputParameter("maxEvaluations",25000);
101
102    // Mutation and Crossover for Real codification
103    crossover = CrossoverFactory.getCrossoverOperator("SBXCrossover");
104    crossover.setParameter("probability",0.9);
105    crossover.setParameter("distributionIndex",20.0);
106
107    mutation = MutationFactory.getMutationOperator("PolynomialMutation");
108    mutation.setParameter("probability",1.0/problem.getNumberOfVariables());
109    mutation.setParameter("distributionIndex",20.0);
110
111    // Selection Operator
112    selection = SelectionFactory.getSelectionOperator("BinaryTournament2") ;
113
114    // Add the operators to the algorithm
115    algorithm.addOperator("crossover",crossover);
116    algorithm.addOperator("mutation",mutation);
117    algorithm.addOperator("selection",selection);
118
119    // Add the indicator object to the algorithm
120    algorithm.setInputParameter("indicators", indicators) ;
       ...
146  } //main
142 } // NSGAII_main
```

Figure 3.20: `NSGAII_main`: configuring the algorithm to execute.

their objectives values are stored in two files (lines 130 and 132). Finally, if the indicator object is not null, a number of quality indicators are calculated (lines 135-143) and printed, as well as the number of evaluations returned by the algorithm as an output parameter.

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

Figure 3.21: `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, 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 (modifying the corresponding `algorithm_main.java` files or changing parameter values in an associated `Settings` object).
2. Optionally, configure the problems to solve. For example, the DTLZ problems are configured by default with three objectives, while the WFG 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. Run the experiments.
4. Analyze the results. We use shell and Matlab scripts to obtain Latex tables, as the one included in Table 4.1 (taken from [8]). Additionally, we usually generate boxplots with the R package to obtain more information from the results.

Given that many researchers are not familiarized with bash or Matlab script programming nor R, we started to work in translating these scripts to Java and to incorporating them to jMetal. The result is the `jmetal.experiments` package, first available in jMetal 2.0.

This chapter is devoted mainly to explaining the use of this package. First, we describe the structure of the `jmetal.experiments.Settings` class and how it can be used to configure NSGA-II; then, we

Table 4.1: Median and interquartile range of the (additive) Epsilon (I_ϵ) indicator

Problem	NSGA-II $\tilde{x}_{IQR}$	SPEA2 $\tilde{x}_{IQR}$	GDE3 $\tilde{x}_{IQR}$	MOCell $\tilde{x}_{IQR}$	CellDE $\tilde{x}_{IQR}$	
DTLZ1	7.62e-2 $_{7.2e-2}$	4.16e-2 $_{8.4e-3}$	4.80e-2 $_{6.6e-3}$	5.35e-1 $_{5.1e-1}$	3.34e-2 $_{3.3e-3}$	+
DTLZ2	1.24e-1 $_{2.0e-2}$	8.20e-2 $_{9.5e-3}$	1.17e-1 $_{1.7e-2}$	7.99e-2 $_{8.5e-3}$	7.62e-2 $_{8.8e-3}$	+
DTLZ3	4.51e+0 $_{2.7e+0}$	4.73e+0 $_{3.0e+0}$	1.36e+1 $_{5.2e+0}$	1.67e+1 $_{7.8e+0}$	3.55e+0 $_{3.3e+0}$	+
DTLZ4	1.12e-1 $_{2.4e-2}$	7.93e-2 $_{5.6e-1}$	1.08e-1 $_{1.9e-2}$	6.92e-2 $_{1.0e-2}$	6.77e-2 $_{8.6e-3}$	+
DTLZ5	1.07e-2 $_{2.6e-3}$	7.74e-3 $_{1.5e-3}$	5.58e-3 $_{4.8e-4}$	8.08e-3 $_{1.6e-3}$	6.55e-3 $_{1.1e-3}$	+
DTLZ6	8.57e-1 $_{1.3e-1}$	7.82e-1 $_{6.3e-2}$	5.10e-3 $_{5.5e-4}$	1.72e+0 $_{1.5e-1}$	6.00e-3 $_{7.9e-4}$	+
DTLZ7	1.27e-1 $_{4.5e-2}$	9.82e-2 $_{1.2e-2}$	1.20e-1 $_{3.6e-2}$	1.15e-1 $_{3.0e-2}$	8.42e-2 $_{1.6e-2}$	+
WFG1	5.66e-1 $_{6.8e-2}$	6.56e-1 $_{1.1e-1}$	7.76e-1 $_{1.1e-1}$	6.30e-1 $_{1.8e-1}$	1.03e+0 $_{1.5e-1}$	+
WFG2	3.23e-1 $_{6.4e-2}$	2.37e-1 $_{3.4e-2}$	3.02e-1 $_{4.5e-2}$	2.56e-1 $_{3.8e-2}$	2.52e-1 $_{3.9e-2}$	+
WFG3	1.24e-1 $_{3.5e-2}$	9.22e-2 $_{1.7e-2}$	1.08e-1 $_{3.6e-2}$	8.57e-2 $_{1.8e-2}$	1.04e-1 $_{3.0e-2}$	+
WFG4	4.32e-1 $_{7.8e-2}$	3.26e-1 $_{3.8e-2}$	4.21e-1 $_{1.0e-1}$	2.95e-1 $_{4.3e-2}$	3.10e-1 $_{4.1e-2}$	+
WFG5	4.71e-1 $_{7.8e-2}$	3.52e-1 $_{4.6e-2}$	4.34e-1 $_{6.4e-2}$	3.44e-1 $_{4.2e-2}$	3.30e-1 $_{4.7e-2}$	+
WFG6	4.31e-1 $_{6.7e-2}$	3.30e-1 $_{4.9e-2}$	3.94e-1 $_{6.2e-2}$	3.13e-1 $_{4.4e-2}$	2.81e-1 $_{3.6e-2}$	+
WFG7	4.65e-1 $_{8.7e-2}$	3.37e-1 $_{3.9e-2}$	4.57e-1 $_{1.1e-1}$	3.07e-1 $_{3.8e-2}$	2.95e-1 $_{3.7e-2}$	+
WFG8	7.51e-1 $_{9.2e-2}$	6.22e-1 $_{1.4e-1}$	7.56e-1 $_{5.4e-2}$	6.26e-1 $_{1.6e-1}$	6.38e-1 $_{3.3e-2}$	+
WFG9	4.39e-1 $_{7.2e-2}$	3.28e-1 $_{4.2e-2}$	4.25e-1 $_{5.8e-2}$	3.13e-1 $_{4.5e-2}$	3.14e-1 $_{3.7e-2}$	+

analyze the `jmetal.experiments.Main` class. Finally, we illustrate with two examples the use of the `jmetal.experiments.Experiment` class. The functionaly of the package in its current state is limited, but it is usable and useful. More features will be incorporated in the near future.

Let us note that since jMetal 3.0, a graphical tool is available for performing most of the tasks described in this chapter. By using the tool, which will described in Section 4.8, there is no need of knowing about underlaying implementation details. However, a finer control can be achieved by manipulating the source codes.

4.1 The `jmetal.experiments.Settings` Class

The motivation of designing this class has to do with the fact that in the tradicional 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, the alternative approach is to define the configuration of a metaheuristic in an object which will contain the default settings and will allow to modify them. Figure 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 24) and a string to store the file containing the true Pareto front of the problem if quality indicators are to be applied (line 25).
- The problem can be set either when creating the object (lines 36-37), either by using the method `setProblem()` (lines 162-164).
- The default settings are stablished in the `configure()` method (line 35). This method must be defined in the corresponding subclasses of `Settings`.
- The values of the parameters can be modified by using a Java `Properties` object, passing it as an argument to second definition of the `configure()` method (line 54).

```
 1 /**
 2  * Settings.java
 3  *
 4  * @author Antonio J. Nebro
 5  * @author Juan J. Durillo
 6  * @version 1.0
 7  *
 8  * Abstract Settings class
 9  */
10
10 package jmetal.experiments;
   ...
23 public abstract class Settings {
24   protected Problem problem_ ;
25   public String paretoFrontFile_ ;
26
27   /**
28    * Constructor
29    */
30   public Settings() {
31   } // Constructor
32
33   /**
34    * Constructor
35    */
36   public Settings(Problem problem) {
37     problem_ = problem ;
38   } // Constructor
39
40   /**
41    * Default configure method
42    * @return A problem with the default configuration
43    * @throws jmetal.util.JMException
44    */
45   abstract public Algorithm configure() throws JMException ;
46
47   /**
48    * Configure method. Change the default configuration
49    * @param settings
50    * @return A problem with the settings indicated as argument
51    * @throws jmetal.util.JMException
52    * @throws ClassNotFoundException
53    */
54   abstract public Algorithm configure(Properties settings) throws JMException ;
   ...
158   /**
159    * Change the problem to solve
160    * @param problem
161    */
162   void setProblem(Problem problem) {
163     problem_ = problem ;
164   } // setProblem
165
166 } // Settings
```

Figure 4.1: The `jmetal.experiments.Settings` class.

```
   ...
26 public class NSGAII_Settings extends Settings {
27   public int populationSize_                 = 100   ;
28   public int maxEvaluations_                 = 25000 ;
29   public double mutationProbability_         = 1.0/problem_.getNumberOfVariables() ;
30   public double crossoverProbability_        = 0.9   ;
31   public double mutationDistributionIndex_   = 20.0  ;
32   public double crossoverDistributionIndex_  = 20.0  ;
33
34  /**
35   * Constructor
36   * @throws JMException
37   */
38  public NSGAII_Settings(Problem problem) throws JMException {
39    super(problem) ;
40  } // NSGAII_Settings
     ...
90 } // NSGAII_Settings
```

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

4.2 An example: 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 Figure 4.2 (lines 27-32), which includes also the constructor of the class (lines 38-40), and in Figure 4.3, which contains the implementation of the `configure()` method.

To modify specific parameters, we make use of Java properties. A `Property` object is map of pairs (key, value), where the key and the value are strings. Thus, we have to define the keys in order to specify the parameters we want to change. The way to do this is very simple: the state variables defined in the subclass of `Settings` are used as keys in the properties object. There are two requirements: 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`:

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

- Let us modify the crossover probability, which is set in the `crossoverProbability_` (Figure 4.2, line 30) to 1.0 (the default value is 0.9):

  ```
  Properties parameters = new Properties ;
  parameters.setProperty("crossoverProbability_", "1.0") ;
  Algorithm nsgaII = new NSGAII_Settings(problem).configure(parameters) ;
  ```

- The algorithm can be executed now:

  ```
  SolutionSet resultaPopulation = nsgaII.execute() ;
  ```

In jMetal 3.0, 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.

```
    ...
43  /**
44   * Configure NSGAII with user-defined parameter settings
45   * @return A NSGAII algorithm object
46   * @throws jmetal.util.JMException
47   */
48  public Algorithm configure() throws JMException {
49    Algorithm algorithm ;
50    Operator  selection ;
51    Operator  crossover ;
52    Operator  mutation  ;
53
54    QualityIndicator indicators ;
55
56    // Creating the algorithm. There are two choices: NSGAII and its steady-
57    // state variant ssNSGAII
58    algorithm = new NSGAII(problem_) ;
59    //algorithm = new ssNSGAII(problem_) ;
60
61    // Algorithm parameters
62    algorithm.setInputParameter("populationSize", populationSize_);
63    algorithm.setInputParameter("maxEvaluations", maxEvaluations_);
64
65    // Mutation and Crossover for Real codification
66    crossover = CrossoverFactory.getCrossoverOperator("SBXCrossover");
67    crossover.setParameter("probability", crossoverProbability_);
68    crossover.setParameter("distributionIndex",distributionIndexForCrossover_);
69
70    mutation = MutationFactory.getMutationOperator("PolynomialMutation");
71    mutation.setParameter("probability", mutationProbability_);
72    mutation.setParameter("distributionIndex",distributionIndexForMutation_);
73
74    // Selection Operator
75    selection = SelectionFactory.getSelectionOperator("BinaryTournament2") ;
76
77    // Add the operators to the algorithm
78    algorithm.addOperator("crossover",crossover);
79    algorithm.addOperator("mutation",mutation);
80    algorithm.addOperator("selection",selection);
81
82    // Creating the indicator object
83    if (! paretoFrontFile_.equals("")) {
84      indicators = new QualityIndicator(problem_, paretoFrontFile_);
85      algorithm.setInputParameter("indicators", indicators) ;
86    } // if
87
88    return algorithm ;
89  }// configure
90 }// NSGAII_Settings
```

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

```
   ...
21 public class Main {
22   public static Logger      logger_ ;      // Logger object
23   public static FileHandler fileHandler_ ; // FileHandler object
24
25   /**
26    * @param args Command line arguments.
27    * @throws JMException
28    * @throws IOException
29    * @throws SecurityException
30    * Usage: three options
31    *      - jmetal.experiments.Main algorithmName
32    *      - jmetal.experiments.Main algorithmName problemName
33    *      - jmetal.experiments.Main algorithmName problemName paretoFrontFile
34    */
35   public static void main(String [] args) throws
36                                     JMException, SecurityException, IOException {
37     Problem   problem   ;         // The problem to solve
38     Algorithm algorithm ;         // The algorithm to use
39
40     QualityIndicator indicators ; // Object to get quality indicators
41
42     Properties properties;
43     Settings settings = null;
44
45     String algorithmName = "";
46     String problemName = "Kursawe" ; // Default problem
47     String paretoFrontFile;
48
49     properties   = new Properties() ;
50     indicators = null ;
       ...
```

Figure 4.4: `jmetal.experiments.Main`: main method.

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`. We see in Figure 4.4 the three ways to run the program (lines 31-33), 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.).

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
```

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, the problems, the quality indicators, and the number of independent runs. 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` intended to study 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 enumarate the steps to follow in order to define our own `Experiment` subclass:

1. `NSGAIIStudy` must inherits from `Experiment`:

```
   ...
20 /**
21  * @author Antonio J. Nebro
22  */
23 public class NSGAIIStudy extends Experiment {
   ...
```

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

```
   ...
20 /**
21  * @author Antonio J. Nebro
22  */
23 public class NSGAIIStudy extends Experiment {
24
25  /**
26   * Configures the algorithms in each independent run
27   * @param problem The problem to solve
28   * @param problemIndex
29   * @param algorithm Array containing the algorithms to run
30   * @throws ClassNotFoundException
31   */
32  public synchronized void algorithmSettings(Problem problem,
33                                             int problemIndex,
34                                                 Algorithm[] algorithm)
35    throws ClassNotFoundException {
...
```

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:

```
   ...
32  public synchronized void algorithmSettings(Problem problem,
33                                      int problemIndex,
34                                          Algorithm[] algorithm)
35    throws ClassNotFoundException {
36    try {
37       int numberOfAlgorithms = algorithmNameList_.length;
38
39       Properties[] parameters = new Properties[numberOfAlgorithms];
40
41       for (int i = 0; i < numberOfAlgorithms; i++) {
42         parameters[i] = new Properties();
43       } // for
44
45       parameters[0].setProperty("crossoverProbability_", "1.0");
46       parameters[1].setProperty("crossoverProbability_", "0.9");
47       parameters[2].setProperty("crossoverProbability_", "0.8");
48       parameters[3].setProperty("crossoverProbability_", "0.7");
49
50       if ((!paretoFrontFile_[problemIndex].equals("")) ||
51         (paretoFrontFile_[problemIndex] == null)) {
52         for (int i = 0; i < numberOfAlgorithms; i++)
53          parameters[i].setProperty("paretoFrontFile_",
54                                    paretoFrontFile_[problemIndex]);
55       } // if
56
57       for (int i = 0; i < numberOfAlgorithms; i++)
58         algorithm[i] = new NSGAII_Settings(problem).configure(parameters[i]);
59
   ...
67 } // algorithmSettings
```

In this example, as we are interested in four configurations of NSGA-II, with four different crossover probabilities, we define a Java `Property` object per algorithm to indicate the desired values (lines 45-48). The code between lines 50-55 is used to incorporate the names of the Pareto front files if they are specified. Finally, the `Algorithm` objects are created and they are ready to be executed (lines 57-58).

3. 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:

```
   ...
57   public static void main(String[] args) throws JMException, IOException {
58     NSGAIIStudy exp = new NSGAIIStudy() ;
   ...
```

4. 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 '_').

```
   ...
72  exp.experimentName_  = "NSGAIIStudy" ;
   ...
```

5. 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:

```
   ...
73 exp.algorithmNameList_   = new String[] {
74   "NSGAIIa", "NSGAIIb", "NSGAIIc", "NSGAIId"} ;
75 exp.problemList_     = new String[] {
76   "ZDT1", "ZDT2", "ZDT3", "ZDT4", "DTLZ1", "WFG2"} ;
77 exp.paretoFrontFile_ = new String[] {
78    "ZDT1.pf", "ZDT2.pf", "ZDT3.pf", "ZDT4.pf", "DTLZ1.2D", "WFG2.2D"} ;
79 exp.indicatorList_   = new String[] {"HV", "SPREAD", "IGD", "EPSILON"} ;
   ...
```

   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:

```
     ...
24 public DTLZ1(String solutionType){
25   this(7,2,solutionType);
26 } // DTLZ1
```

   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.

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

```
   ...
83 exp.experimentBaseDirectory_ = "/home/antonio/Softw/pruebas/pruebas/" +
84                                exp.experimentName_;
85 exp.paretoFrontDirectory_ = "/home/antonio/Softw/pruebas/paretoFronts"
   ...
```

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

```
   ...
87 exp.algorithmSettings_ = new Settings[numberOfAlgorithms] ;
   ...
```

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

```
   ...
89 exp.independentRuns_ = 30 ;
   ...
```

9. 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 4.7 for further details):

```
   ...
91 // Run the experiments
92 int numberOfThreads ;
93 exp.runExperiment(numberOfThreads = 6) ;
...
```

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

```
...
95 // Generate latex tables (comment this sentence is not desired)
96 exp.generateLatexTables() ;
...
```

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:

```
98  // Configure the R scripts to be generated
99  int rows  ;
100 int columns  ;
101 String prefix ;
102 String [] problems ;
103
104 rows = 2 ;
105 columns = 2 ;
106 prefix = new String("ZDT");
107 problems = new String[]{"ZDT1", "ZDT2","ZDT3", "ZDT4"} ;
108
109 boolean notch ;
110 exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch = true) ;
```

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:

```
...
111 exp.generateRWilcoxonScripts(problems, prefix) ;
...
```

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
```

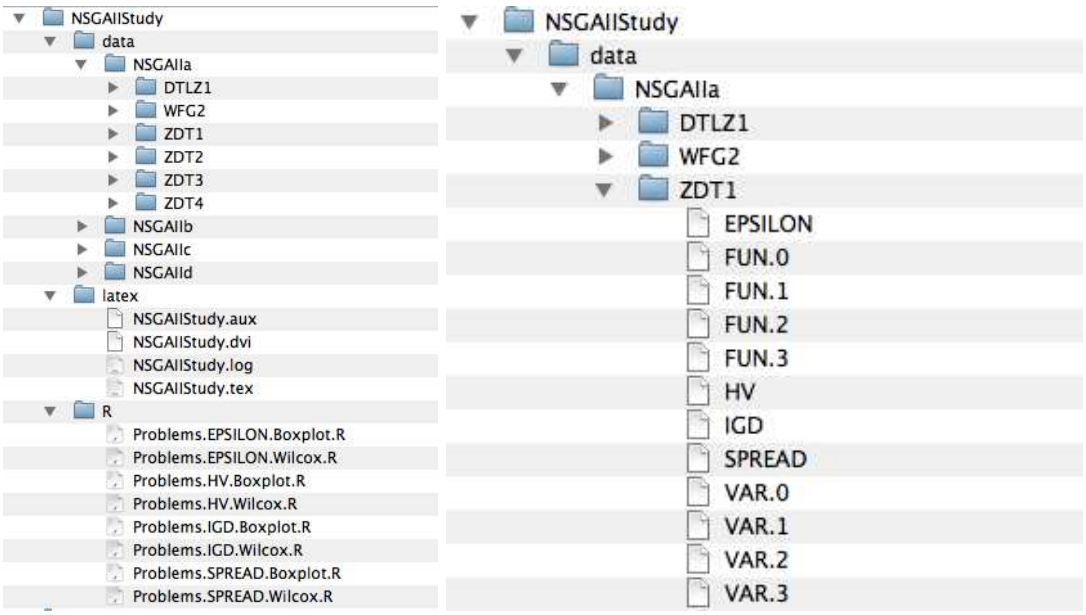

Figure 4.5: Output directories and files after running the experiment.

After the execution of the algorithms, we obtain the directory tree depicted in Figure 4.5. 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 4.5-left, the directory named `NSGAIIStudy` has three directories: `data`, `R`, and `latex`. The `data` directory contains (see Figure 4.5-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. This file contains tables with statistical results per quality indicator. You just need to compile the file with your favorite Latex tool. For example, you could simply type:

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

Table 4.2: HV. Mean and standard deviation

	NSGAIIa	NSGAIIb	NSGAIIc	NSGAIId
ZDT1	$6.60e-01_{3.0e-04}$	$6.59e-01_{2.9e-04}$	$6.59e-01_{2.5e-04}$	$6.59e-01_{3.5e-04}$
ZDT2	$3.26e-01_{2.8e-04}$	$3.26e-01_{3.2e-04}$	$3.26e-01_{3.4e-04}$	$3.25e-01_{3.4e-04}$
ZDT3	$5.15e-01_{1.5e-04}$	$5.15e-01_{6.2e-04}$	$5.15e-01_{1.4e-04}$	$5.14e-01_{1.7e-04}$
ZDT4	$6.56e-01_{4.5e-03}$	$6.56e-01_{2.9e-03}$	$6.53e-01_{4.2e-03}$	$6.52e-01_{4.1e-03}$
DTLZ1	$4.87e-01_{4.2e-03}$	$4.71e-01_{8.7e-02}$	$4.87e-01_{3.5e-03}$	$4.65e-01_{8.7e-02}$
WFG2	$5.62e-01_{1.3e-03}$	$5.62e-01_{1.3e-03}$	$5.62e-01_{1.3e-03}$	$5.62e-01_{1.4e-03}$

Table 4.3: HV. Median and IQR

	NSGAIIa	NSGAIIb	NSGAIIc	NSGAIId
ZDT1	$6.60e-01_{4.8e-04}$	$6.59e-01_{3.5e-04}$	$6.59e-01_{3.8e-04}$	$6.59e-01_{4.7e-04}$
ZDT2	$3.26e-01_{3.7e-04}$	$3.26e-01_{4.1e-04}$	$3.26e-01_{3.9e-04}$	$3.25e-01_{5.3e-04}$
ZDT3	$5.15e-01_{1.8e-04}$	$5.15e-01_{1.7e-04}$	$5.15e-01_{2.3e-04}$	$5.14e-01_{2.7e-04}$
ZDT4	$6.57e-01_{3.6e-03}$	$6.56e-01_{4.5e-03}$	$6.53e-01_{6.2e-03}$	$6.53e-01_{4.4e-03}$
DTLZ1	$4.88e-01_{5.6e-03}$	$4.88e-01_{5.0e-03}$	$4.88e-01_{6.2e-03}$	$4.84e-01_{8.2e-03}$
WFG2	$5.63e-01_{2.6e-03}$	$5.61e-01_{2.2e-03}$	$5.61e-01_{2.6e-03}$	$5.62e-01_{2.5e-03}$

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

```
pdflatex NSGAIIStudy.tex
```

As an example of the obtained output, Table 4.2 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 4.3. 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 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 4.6 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 4.7

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 4.4 to 4.9. 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 4.10 groups the other tables into one.

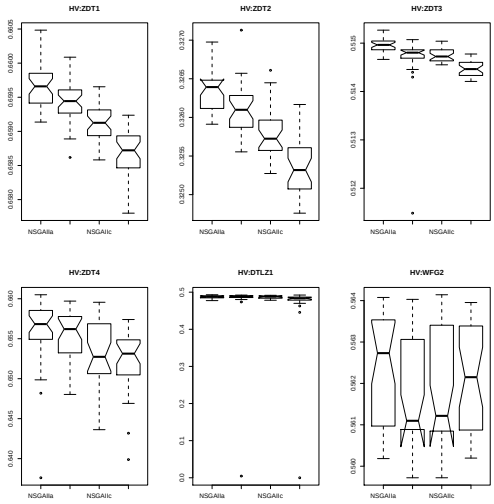

Figure 4.6: Boxplots of the values obtained after applying the hypervolume quality indicator (notch = true).

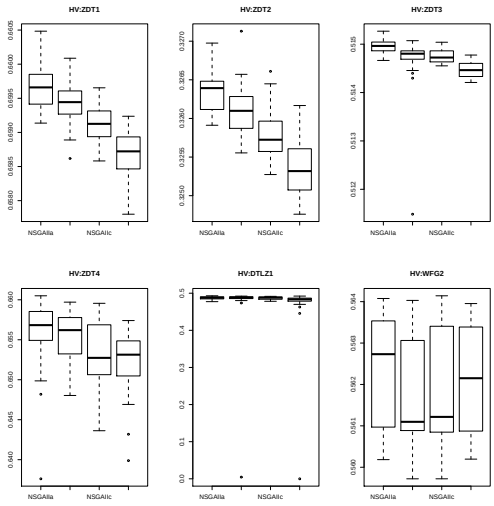

Figure 4.7: Boxplots of the values obtained after applying the hypervolume quality indicator (notch = false).

Table 4.4: ZDT1.HV.

	NSGAIIb	NSGAIIc	NSGAIId
NSGAIIa	▲	▲	▲
NSGAIIb		▲	▲
NSGAIIc			▲

Table 4.5: ZDT2 .HV.

	NSGAIIb	NSGAIIc	NSGAIId
NSGAIIa	▲	▲	▲
NSGAIIb		▲	▲
NSGAIIc			▲

Table 4.6: ZDT3.HV.

	NSGAIIb	NSGAIIc	NSGAIId
NSGAIIa	▲	▲	▲
NSGAIIb		–	▲
NSGAIIc			▲

Table 4.7: ZDT4 .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 `algorithmSettigns` method (file: `jmetal.experiments.StandardStudy`) is the following:

```
29  /**
30   * Configures the algorithms in each independent run
31   * @param problem The problem to solve
32   * @param problemIndex
33   * @throws ClassNotFoundException
34   */
35  public void algorithmSettings(Problem problem,
36                                int problemIndex,
37                                Algorithm[] algorithm) throws ClassNotFoundException {
38    try {
39      int numberOfAlgorithms = algorithmNameList_.length;
40
41      Properties[] parameters = new Properties[numberOfAlgorithms];
42
43      for (int i = 0; i < numberOfAlgorithms; i++) {
44        parameters[i] = new Properties();
45      } // for
46
47      if (!paretoFrontFile_[problemIndex].equals("")) {
48        for (int i = 0; i < numberOfAlgorithms; i++)
49          parameters[i].setProperty("paretoFrontFile_", paretoFrontFile_[problemIndex]);
50        } // if
51
52        algorithm[0] = new NSGAII_Settings(problem).configure(parameters[0]);
53        algorithm[1] = new SPEA2_Settings(problem).configure(parameters[1]);
54        algorithm[2] = new MOCell_Settings(problem).configure(parameters[2]);
55        algorithm[3] = new SMPSO_Settings(problem).configure(parameters[3]);
56        algorithm[4] = new GDE3_Settings(problem).configure(parameters[4]);
57      } catch (IllegalArgumentException ex) {
```

Table 4.8: DTLZ1 .HV.

	NSGAIIb	NSGAIIc	NSGAIId
NSGAIIa	–	–	▲
NSGAIIb		–	▲
NSGAIIc			▲

Table 4.9: WFG2.HV.

	NSGAIIb	NSGAIIc	NSGAIId
NSGAIIa	–	–	–
NSGAIIb		–	–
NSGAIIc			–

Table 4.10: ZDT1 ZDT2 ZDT3 ZDT4 DTLZ1 WFG2 .HV.

	NSGAIIb						NSGAIIc						NSGAIId					
NSGAIIa	▲	▲	▲	–	–	–	▲	▲	▲	▲	–	–	▲	▲	▲	▲	▲	–
NSGAIIb							▲	▲	–	▲	–	–	▲	▲	▲	▲	▲	–
NSGAIIc													▲	▲	▲	–	▲	–

```
58      Logger.getLogger(StandardStudy.class.getName()).log(Level.SEVERE, null, ex);
59    } catch (IllegalAccessException ex) {
60      Logger.getLogger(StandardStudy.class.getName()).log(Level.SEVERE, null, ex);
61    } catch  (JMException ex) {
62      Logger.getLogger(StandardStudy.class.getName()).log(Level.SEVERE, null, ex);
63    }
64  } // 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 47-51).

The `main` method is included below, where we can observe the algorithm name list (lines 61-62), the problem list (lines 63-67), and the list of the names of the files containing the Pareto fronts (lines 68-75). The rest of the code is similar to `NSGAIIStudy`: the list of indicators is included in line 77, the directory to write the results and the one containing the Pareto fronts are specified next (lines 81-83), the number of independent runs is indicated in line 88, and finally the methods to run the algorithms and to generate the latex tables are invoked (lines 91-94).

```
72   public static void main(String[] args) throws JMException, IOException {
73     StandardStudy exp = new StandardStudy();
74
75     exp.experimentName_ = "StandardStudy";
76     exp.algorithmNameList_ = new String[]{
77                                 "NSGAII", "SPEA2", "MOCell", "SMPSO", "GDE3"};
78     exp.problemList_ = new String[]{"ZDT1", "ZDT2","ZDT3", "ZDT4","ZDT6",
79                                     "DTLZ1","DTLZ2","DTLZ3","DTLZ4","DTLZ5",
80                                     "DTLZ6","DTLZ7",
81                                     "WFG1","WFG2","WFG3","WFG4","WFG5","WFG6",
82                                     "WFG7","WFG8","WFG9"};
83     exp.paretoFrontFile_ = new String[]{"ZDT1.pf", "ZDT2.pf","ZDT3.pf",
84                                     "ZDT4.pf","ZDT6.pf",
85                                     "DTLZ1.2D.pf","DTLZ2.2D.pf","DTLZ3.2D.pf",
86                                     "DTLZ4.2D.pf","DTLZ5.2D.pf","DTLZ6.2D.pf",
87                                     "DTLZ7.2D.pf",
88                                     "WFG1.2D.pf","WFG2.2D.pf","WFG3.2D.pf",
89                                     "WFG4.2D.pf","WFG5.2D.pf","WFG6.2D.pf",
90                                     "WFG7.2D.pf","WFG8.2D.pf","WFG9.2D.pf"};
91
92     exp.indicatorList_ = new String[]{"HV", "SPREAD", "EPSILON"};
93
94     int numberOfAlgorithms = exp.algorithmNameList_.length;
95
```

```
96     exp.experimentBaseDirectory_ = "/Users/antonio/Softw/pruebas/jmetal/" +
97                                    exp.experimentName_;
98     exp.paretoFrontDirectory_ = "/Users/antonio/Softw/pruebas/paretoFronts";
99
100    exp.algorithmSettings_ = new Settings[numberOfAlgorithms];
101
102    exp.independentRuns_ = 100;
103
104    // Run the experiments
105    int numberOfThreads ;
106    exp.runExperiment(numberOfThreads = 4) ;
107
108    // Generate latex tables
109    exp.generateLatexTables() ;
110
111     // Configure the R scripts to be generated
112     int rows  ;
113     int columns  ;
114     String prefix ;
115     String [] problems ;
116     boolean notch ;
117
118     // Configuring scripts for ZDT
119     rows = 3 ;
120     columns = 2 ;
121     prefix = new String("ZDT");
122     problems = new String[]{"ZDT1", "ZDT2","ZDT3", "ZDT4","ZDT6"} ;
123
124     exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch = true) ;
125     exp.generateRWilcoxonScripts(problems, prefix) ;
126
127     // Configure scripts for DTLZ
128     rows = 3 ;
129     columns = 3 ;
130     prefix = new String("DTLZ");
131     problems = new String[]{"DTLZ1","DTLZ2","DTLZ3","DTLZ4","DTLZ5",
132                                    "DTLZ6","DTLZ7"} ;
133
134     exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch = true) ;
135     exp.generateRWilcoxonScripts(problems, prefix) ;
136
137     // Configure scripts for WFG
138     rows = 3 ;
139     columns = 3 ;
140     prefix = new String("WFG");
141     problems = new String[]{"WFG1","WFG2","WFG3","WFG4","WFG5","WFG6",
142                                    "WFG7","WFG8","WFG9"} ;
143
144     exp.generateRBoxplotScripts(rows, columns, problems, prefix, notch = true) ;
145     exp.generateRWilcoxonScripts(problems, prefix) ;
146   } // main
```

We can observe that we invoke three times the methods `generateRBoxplotsScript()` and `generateRWilcoxonScr`

one per problem family.

4.6 Using quality indicators

When any of the algorithms provided by jMetal are executed to solve a problem, two files containing the approximation 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. To illustrate the use of this class we provide two examples.

The first example is the `jmetal/metaheuristics/moead/MOEAD_main.java` file, which executes the algorithm MOEA/D-DE. Let us examine 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.

The second example is comented in Section 3.3.1, where the use of the hypervolume inside NSGA-II to measure the convergence speed of the algorithm was detailed.

4.7 Running experiments in parallel

In jMetal 2.2 we introduce a first approximation to make use of current multi-core processors to speed-up the execution of experiments. We use Java threads for that purpose.

In the current implementation, the `runExperiment()` method, defined in `jmetal.experiments.Experiment`, has an integer parameter indicating the number T of threads to used. Each thread will be assigned a

subset of the P problems to be solved of size P/T; in case of declaring more threads that problems, i.e., $T > P$, only P threads will be created. If (P MOD T) $\neq 0$, the last subset of problems will be solved by the last thread.

We have tested this new feature by running the `NSGAIIStudy` experiment (see Section 4.4). The computer is a MacBook with a Core 2 Duo 2GHz processor and 2 GB of RAM, running Mac OS X 10.5.7 (9j61); the Java version is "Java(TM) SE Runtime Environment (build 1.6.0_07-b06-153)". The computing time using one thread is roughly 12 minutes, while using six threads (one per problem), it is reduced to 10 minutes. The speed-up is far from lineal, but we have to take into account that each individual run of the algorithms is around 1-2 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.

4.8 jMetal Graphical User Interface

In this section, we describe the two graphical tools available in jMetal. Both tools are included in the `jmetal.gui` package, and they are called Simple Execution Support GUI and Experiment Support GUI. We will refer them to as SES_GUI and ES_GUI, respectively.

Please, have into account that this is the first version of these tools. There are still bugs to be found, and many changes are in progress. Consider them as beta versions.

To run the graphical tool there is a requirement: the file name `gui.data`, that is provided in the jmetal distribution tarball, must be in the current directory where the tools are launched.

4.8.1 The Simple Execution Support GUI (SES_GUI)

The SES_GUI is intended to configure and execute an algorithm to solve a problem and plotting the computed Pareto front approximation. It is currently limited to bi-objective problems. The program can be launched by executing `jmetal.gui.SingleExecutionSupportGUI.java`. Figure 4.8 show the window which appears. By default, the algorithm AbYSS and the `ContrEx` problem are selected.

Let us suppose that we are interested in using NSGA_II to solve the ZDT1 problem. The steps to follow are:

1. Select the NSGA_II algorithm (see Figure 4.9).
2. Set the algorithm parameters and choose the ZDT1 problem (see Figure 4.10).
3. Execute the algorithm. The result is depicted in Figure 4.11.

4.8.2 The Execution Support GUI (ES_GUI)

While the SES_GUI is useful to test algorithms over problems, allowing to play with different parameter settings to observe the result fronts, the ES_GUI is a tool to carry out experimental studies. It can be launched by running the `jmetal.gui.ExperimentsSupportGUI.java` program.

Figure 4.12 show the ES_GUI window after selecting the algorithms to run (left column), the problems to solve (center column), and the quality indicators to apply (right column). By the default, the algorithms are executed by using pre-defined values, but these can be modified easily.

The name of the experiment is given in the top part of the ES_GUI window, as can be seen in Figure 4.12. The bottom part contains the experiment settings, which corresponds to the parameters explained previously in section 4.4.

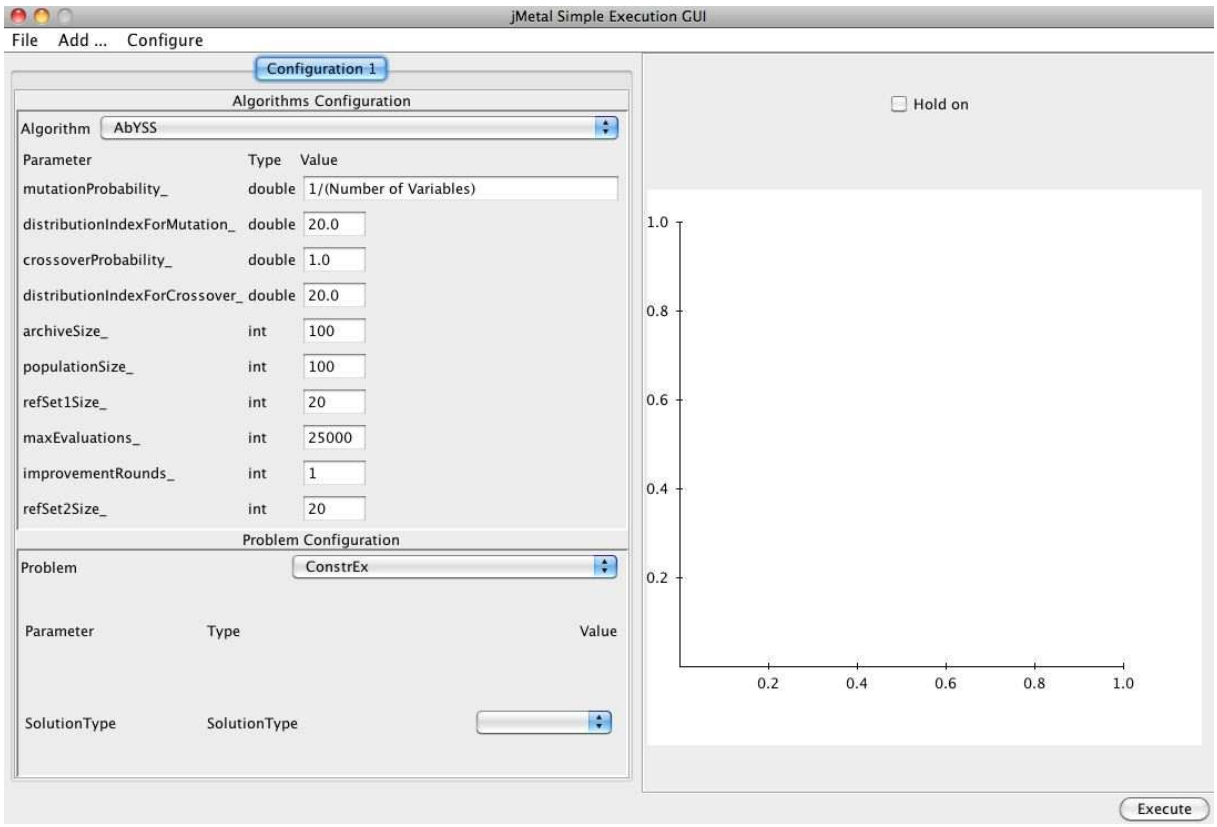

Figure 4.8: Simple Execution Support GUI

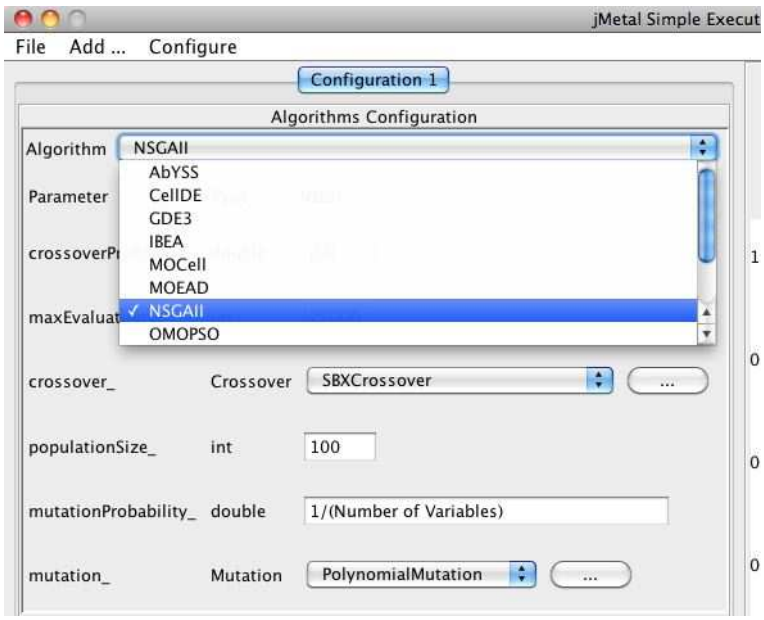

Figure 4.9: Simple Execution Support GUI. Selecting the NSGA_II algorithm.

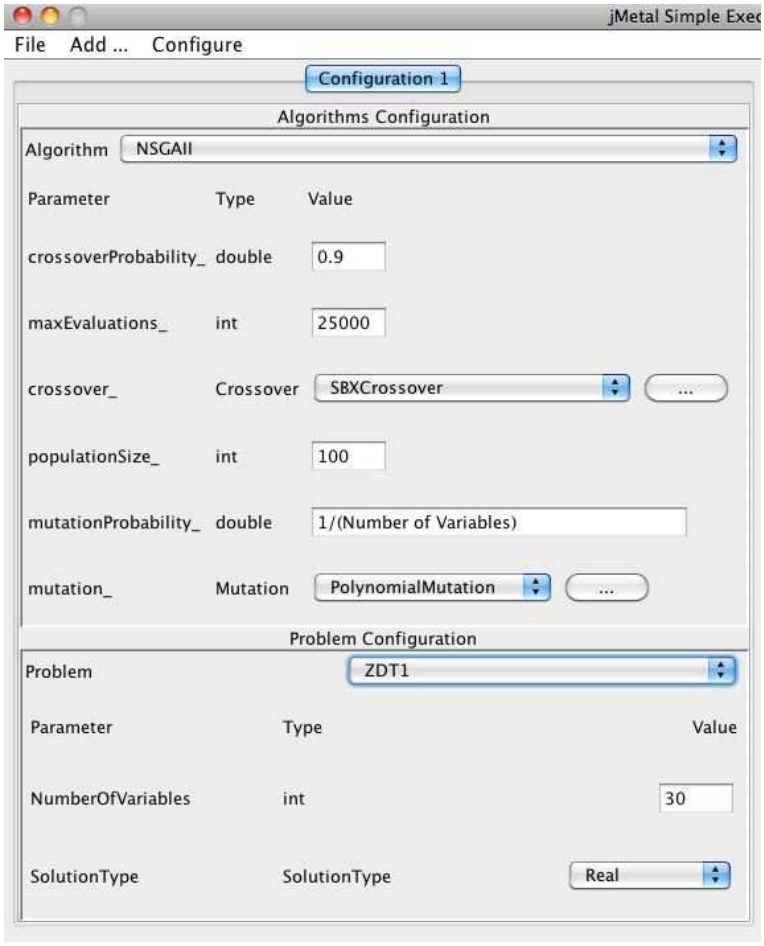

Figure 4.10: Simple Execution Support GUI. Setting the parameters and chosing the problem to solve.

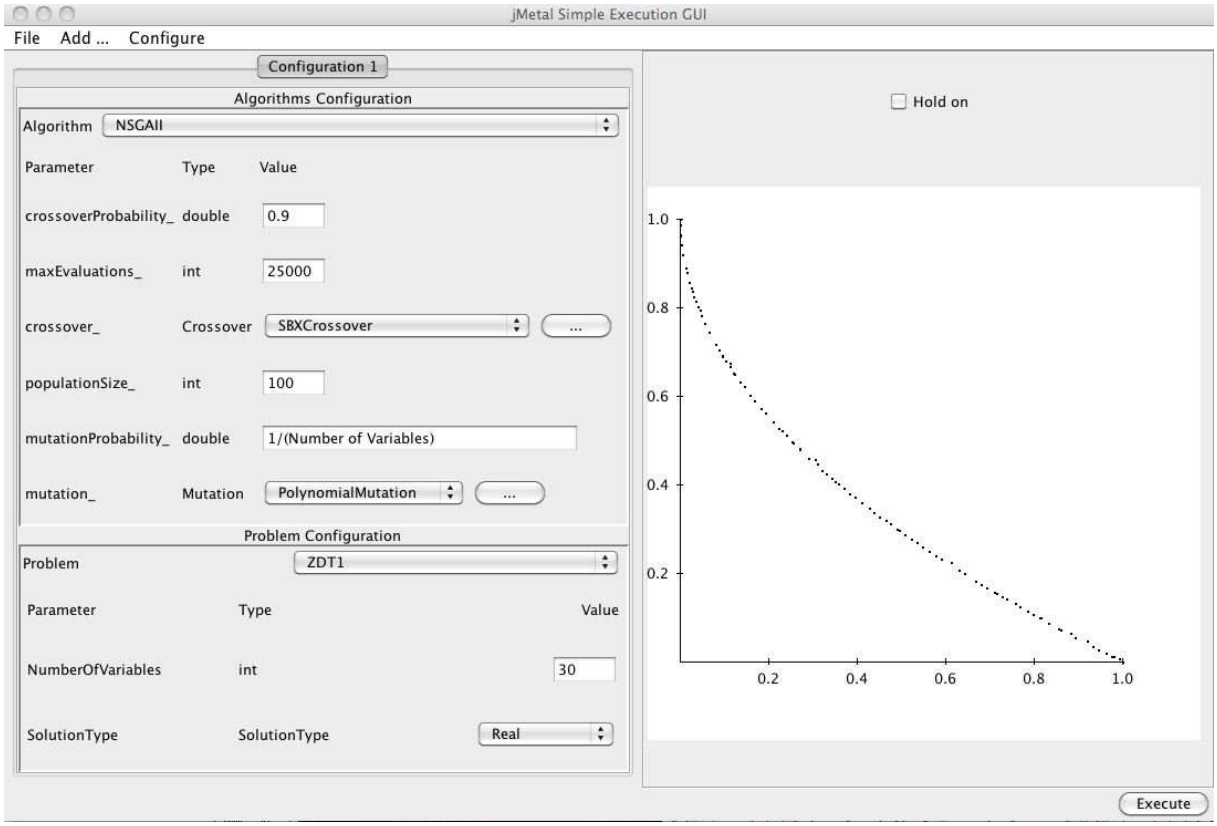

Figure 4.11: Simple Execution Support GUI. Pareto front approximation of problem ZDT1.

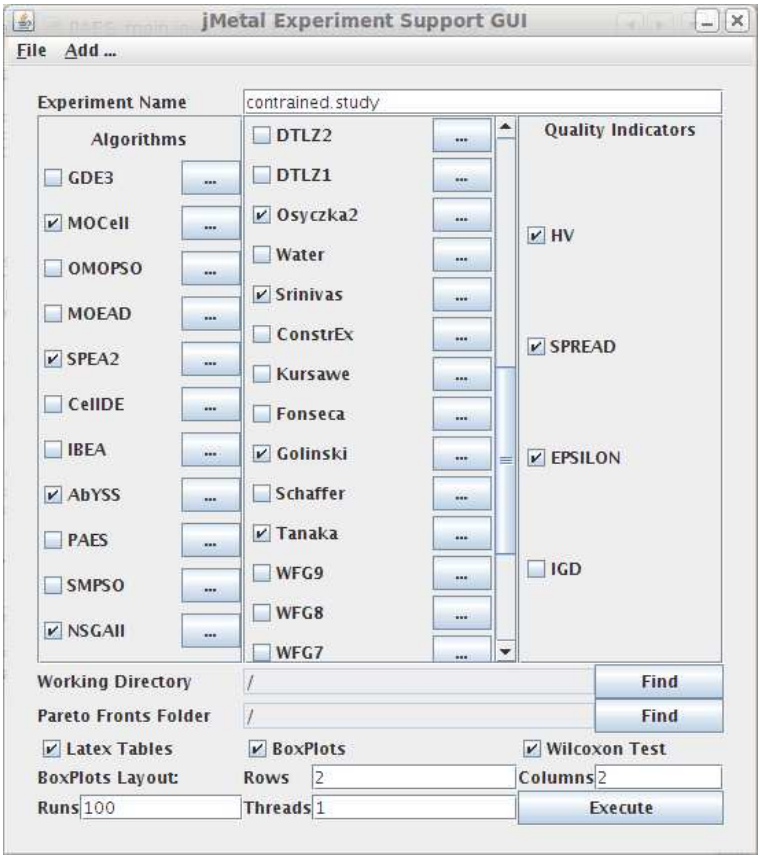

Figure 4.12: Experiment Support GUI. Configuring an experiment.

Chapter 5

How-to's

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

5.1 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 default settings, included in the piece of code below. The code is very simple, and we only comment here that if we want to use a default mutation probability $1/L$, where L is the length of the chromosome binary string, we need to create a solution to apply it the `getNumberOfBits()` method.

```
29 public class NSGAIIBinary_Settings extends Settings{
30
31   // Default settings
32   int populationSize_ = 100   ;
33   int maxEvaluations_ = 25000 ;
34
35   // To obtain the default mutation probability, we need to know the lenght of
36   // the binary string. One way to do it is by using the getNumberOfBits() method
37   // applied to a solution
38   Solution dummy = new Solution(problem_) ;
39
40   double mutationProbability_  = 1.0/dummy.getNumberOfBits() ;
41   double crossoverProbability_ = 0.9 ;
42
43   String paretoFrontFile_ = "" ;
44
45   /**
46    * Constructor
47    */
48   public NSGAIIBinary_Settings(Problem problem) {
49     super(problem) ;
50   } // NSGAIIBinary_Settings
```

In the `configure()` method, we specify the single point and bit-flip crossover and mutation operators, respectively, in lines 73 and 77:

```
52   /**
53    * Configure NSGAII with user-defined parameter settings
54    * @return A NSGAII algorithm object
55    * @throws jmetal.util.JMException
56    */
57   public Algorithm configure() throws JMException {
58     Algorithm algorithm ;
59     Operator  selection ;
60     Operator  crossover ;
61     Operator  mutation  ;
62
63     QualityIndicator indicators ;
64
65     // Creating the problem
66     algorithm = new NSGAII(problem_) ;
67
68     // Algorithm parameters
69     algorithm.setInputParameter("populationSize", populationSize_);
70     algorithm.setInputParameter("maxEvaluations", maxEvaluations_);
71
72     // Mutation and Crossover for Real codification
73     crossover = CrossoverFactory.getCrossoverOperator("SinglePointCrossover");
74     crossover.setParameter("probability", crossoverProbability_);
75
76     mutation = MutationFactory.getMutationOperator("BitFlipMutation");
77     mutation.setParameter("probability", mutationProbability_);
78
79     // Selection Operator
80     selection = SelectionFactory.getSelectionOperator("BinaryTournament2") ;
81
82     // Add the operators to the algorithm
83     algorithm.addOperator("crossover",crossover);
84     algorithm.addOperator("mutation",mutation);
85     algorithm.addOperator("selection",selection);
86
87    // Creating the indicator object
88    if (! paretoFrontFile_.equals("")) {
89       indicators = new QualityIndicator(problem_, paretoFrontFile_);
90       algorithm.setInputParameter("indicators", indicators) ;
91    } // if
92     return algorithm ;
93   } // configure
```

Finally, to use this settings class, we need to indicate in `jmetal.experiments.Main` that the encoding of the solutions is `BinaryReal` instead of `Real`, in lines 61, 69, and 78; that is:

`Object [] problemParams = {"BinaryReal"};`

instead of

Object [] problemParams = {"Real"};.

If we want to use NSGA-II to solve the ZDT5 problem, which is a binary one, we can use the `jmetal.experiments.settings.NSGAIIBinary_Settigns` as is. However, in the lines 61, 69, and 79,

we must write:

`Object [] problemParams = {};`

instead of

`Object [] problemParams = {"Binary"};`

because of the way in which ZDT5 is defined in `jmetal.problems.ZDT.ZDT5`.

5.2 How to create a new solution type having integer and real variables?

jMetal provides many built-in solution types (`Real`, `Binary`, `Int`, `Permutation`, `BinaryReal` (binary-coded real values), etc.). A solution is composed of decision variables, which also have a type (again `Real`, `Binary`, `Int`, etc). In general, the variables of most of built-in solutions are of the same type. Thus, a solution of type `Real` is only composed of variables of type `Real`. The available types of solutions and variables are included, respectively, in the enumerate types `SolutionType_` and `VariableType_`, which are both included in class `jmetal.base.Configuration`.

The type of the solutions of a MOP is specified in the constructor of the class which defines the problem. Each problem contains an array of type `VariableType_` as a state variable, which is used to store the type of each decision variable. Furthermore, it also contains a state variable called `SolutionType_`, which is used to store the type of the solutions. These variables are set when the problem constructor is invoked. If we take a look to the constructor of the class for problem Schaffer (`jmetal.problems.Schaffer.java`), this is carried out in lines 36-40:

```
20 /**
21  * Constructor.
22  * Creates a default instance of problem Schaffer
23  * @param solutionType The solution type must "Real" or "BinaryReal".
24  */
25  public Schaffer(String solutionType) {
26    numberOfVariables_   = 1 ;
27    numberOfObjectives_  = 2 ;
28    numberOfConstraints_ = 0 ;
29    problemName_         = "Schaffer";
30
31    lowerLimit_ = new double[numberOfVariables_];
32    upperLimit_ = new double[numberOfVariables_];
33    lowerLimit_[0] = -100000;
34    upperLimit_[0] =  100000;
35
36    solutionType_ = Enum.valueOf(SolutionType_.class, solutionType) ;
37
38    // All the variables are of the same type, so the solutionType name is the
39    // same than the variableType name
40    variableType_ = new VariableType_[numberOfVariables_];
41    for (int var = 0; var < numberOfVariables_; var++){
42      variableType_[var] = Enum.valueOf(VariableType_.class, solutionType) ;
43    } // for
44  } //Schaffer
```

When we started designing jMetal, we considered that the framework should be flexible enough to allow user defined solution types, probably composed of mixed variable types. We explain next how a solution containing N integer variables and M real variables can be defined.

The steps to follow are:

1. Add a name to the type of solution you want to create. This requires to edit the class `jmetal.base.Configuration.jav` and add the new solution type to the enum type `SolutionType_`. An example is the `IntReal` type, which will represent a solution having N integer variables and M real variables.

2. Modify the constructor of class `jmetal.base.DecisionVariables`. This class is used to create new solutions. The next nested else-if block contains the coded required in the example of the `IntReal` solution:

```
...
95 else if (problem.solutionType_ == SolutionType_.IntReal) {
96 for (int var = 0; var < problem_.getNumberOfVariables(); var++)
97 if (problem.variableType_[var] == VariableType_.Int)
98 variables_[var] = new Int((int)problem_.getLowerLimit(var),
99 (int)problem_.getUpperLimit(var));
100 else if (problem.variableType_[var] == VariableType_.Real)
101 variables_[var] = new Real(problem_.getLowerLimit(var),
102 problem_.getUpperLimit(var));
103 else {
104 Configuration.logger_.severe("DecisionVariables.DecisionVariables: " +
105 "error creating a Solution of type IntReal") ;
106 } // else
...
```

3. Create a problem using the new type. We have included the problem `jmetal.problems.IntRealProblem.java`; its constructors contain the code needed to set the settings of the new type of solutions properly:

```
31 /**
32   * Constructor.
33   * Creates a new instance of the IntRealProblem problem.
34   * @param intVariables Number of integer variables of the problem
35   * @param realVariables Number of real variables of the problem
36   */
37  public IntRealProblem(int intVariables, int realVariables) {
38    intVariables_  = intVariables  ;
39    realVariables_ = realVariables ;
40
41    numberOfVariables_   = intVariables_ + realVariables_ ;
42    numberOfObjectives_  = 2                               ;
43    numberOfConstraints_ = 0                               ;
44    problemName_         = "IntRealProblem"                ;
45
46    upperLimit_ = new double[numberOfVariables_] ;
47    lowerLimit_ = new double[numberOfVariables_] ;
48
49    for (int i = 0; i < intVariables; i++) {
50      lowerLimit_[i] = -5 ;
51      upperLimit_[i] =  5 ;
52    } // for
53
54    for (int i = intVariables; i < (intVariables + realVariables); i++) {
55      lowerLimit_[i] = -5.0 ;
56      upperLimit_[i] =  5.0  ;
57    } // for
```

```
58
59    solutionType_ = SolutionType_.IntReal ;
60
61    // Creating the array indicating the types of the variables
62    variableType_ = new VariableType_[numberOfVariables_];
63
64    // Initializing the types of the variables
65    for (int i = 0; i < intVariables_; i++)
66      variableType_[i] = VariableType_.Int ;
67
68    for (int i = intVariables_; i < (intVariables_ + realVariables_); i++) {
69      variableType_[i] = VariableType_.Real ;
70    } // for
71  } // IntRealProblem
```

4. Define the operators required by the problem. We will suggest to have a look to some of the crossover and mutation operators defined in jMetal, and use some of them as a model. The key point is that you can ask for the type of the solution in the `execute()` method, so you can define the operator according to your needs.

Chapter 6

What about's

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

6.1 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 MO metaheuristicas in jMetal. 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.

However, as several MO algorithms are available, developing a SO version of them is not difficult: in fact, it is very simple. We started to include as an example two simple SO genetic algorithms in the `jmetal.metaheuristics.singleObjective.geneticAlgorithm` package, but since jMetal 3.0 we offer the following algorithms:

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

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

6.2 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:

```
public class Schaffer extends Problem {

 /**
  * Constructor.
  * Creates a default instance of problem Schaffer
  * @param solutionType The solution type must "Real" or "BinaryReal".s
  */
  public Schaffer(String solutionType) throws ClassNotFoundException {
    numberOfVariables_  = 1;
    numberOfObjectives_ = 2;
    numberOfConstraints_ =0;
    problemName_         = "Schaffer";

    lowerLimit_ = new double[numberOfVariables_];
    upperLimit_ = new double[numberOfVariables_];
    lowerLimit_[0] = -100000;
    upperLimit_[0] =  100000;

    if (solutionType.compareTo("BinaryReal") == 0)
     solutionType_ = new BinaryRealSolutionType(this) ;
    else if (solutionType.compareTo("Real") == 0)
     solutionType_ = new RealSolutionType(this) ;
    else {
     System.out.println("Error: solution type " + solutionType + " invalid") ;
     System.exit(-1) ;
    }
  } //Schaffer


  /**
  * Evaluates a solution
  * @param solution The solution to evaluate
   * @throws JMException
  */
  public void evaluate(Solution solution) throws JMException {
    Variable[] variable  = solution.getDecisionVariables();

    double [] f = new double[numberOfObjectives_];
    f[0] = variable[0].getValue() *
    variable[0].getValue();

    f[1] = (variable[0].getValue() - 2.0) *
           (variable[0].getValue() - 2.0);

    solution.setObjective(0,f[0]);
    solution.setObjective(1,f[1]);
  } //evaluate
} //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:

```
public Kursawe(String solutionType, Integer numberOfVariables) throws ClassNotFoundException {
  numberOfVariables_   = numberOfVariables.intValue() ;
  numberOfObjectives_  = 2                            ;
  numberOfConstraints_ = 0                            ;
  problemName_         = "Kursawe"                    ;

  upperLimit_ = new double[numberOfVariables_] ;
  lowerLimit_ = new double[numberOfVariables_] ;

  for (int i = 0; i < numberOfVariables_; i++) {
    lowerLimit_[i] = -5.0 ;
    upperLimit_[i] = 5.0  ;
  } // for

  if (solutionType.compareTo("BinaryReal") == 0)
   solutionType_ = new BinaryRealSolutionType(this) ;
  else if (solutionType.compareTo("Real") == 0)
   solutionType_ = new RealSolutionType(this) ;
  else if (solutionType.compareTo("ArrayReal") == 0)
   solutionType_ = new ArrayRealSolutionType(this) ;
  else {
   System.out.println("Error: solution type " + solutionType + " invalid") ;
   System.exit(-1) ;
  } // if
} // 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:

```
public void evaluate(Solution solution) throws JMException {
  XReal x = new XReal(solution) ;

  double aux, xi, xj           ; // auxiliar variables
  double [] fx = new double[2] ; // function values

  fx[0] = 0.0 ;
  for (int var = 0; var < numberOfVariables_ - 1; var++) {
    xi = x.getValue(var)   *
         x.getValue(var);
    xj = x.getValue(var+1) *
         x.getValue(var+1) ;
    aux = (-0.2) * Math.sqrt(xi + xj);
    fx[0] += (-10.0) * Math.exp(aux);
  } // for

  fx[1] = 0.0;
```

```
    for (int var = 0; var < numberOfVariables_ ; var++) {
      fx[1] += Math.pow(Math.abs(x.getValue(var)), 0.8) +
           5.0 * Math.sin(Math.pow(x.getValue(var), 3.0));
    } // for

    solution.setObjective(0, fx[0]);
    solution.setObjective(1, fx[1]);
  } // 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 turns 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 7

Versions and Release Notes

This manual starts with jMetal 2.0, released on December 2008. We detail next the release notes, new features and bugs fixed in the current release, jMetal 3.0, from the previous versions.

7.1 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 Section 4.8
- Added Chapter 4
- Added Chapter 5
- Added Chapter 6
- Chapter FAQ has been removed

7.2 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 [6].
- 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 4.7).

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

7.3 Version 2.1 (23^{rd} February 2009)

Release notes

This release contains as main contribution the support of automatically generating R[1] scripts, which when compiled produce figures representing boxplots of the results.

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

New features

- Class `jmetal.experiments.Experiment`: method `generateRScripts()`.
- The IBEA algorithm [38] (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.

7.4 Version 2.0 (23rd 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, 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] M. Reyes and C.A. Coello Coello. Improving PSO-based multi-objective optimization using crowding, mutation and ϵ-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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.