JCLEC-MO

JCLEC for multi-objective and many-objective optimisation

User guide and tutorial

Version 1.0

Aurora Ramírez, José Raúl Romero, Sebastián Ventura
KDIS Research Group. University of Córdoba.
http://www.uco.es/grupos/kdis

March 9, 2018

Citing JCLEC-MO

Please, use the following citation to refer to JCLEC-MO:

A. Ramírez, J.R. Romero, S. Ventura. An Extensible JCLEC-based Solution for the Implementation of Multi-Objective Evolutionary Algorithms. *In Proceedings of the Companion Publication of the 17th Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*, pp. 1085-1092, ACM, 2015.

If you want to cite this document, you can also use the following citation:

A. Ramírez, J.R. Romero, S. Ventura (2018) JCLEC-MO: User guide and tutorial. Version 1.0. Available for download from *http://www.uco.es/grupos/kdis/jclec-mo*

CONTENTS

LIST OF TABLES

1. INTRODUCTION

JCLEC-MO [1] is an extensible Java (JRE 8+) library aimed at providing specific algorithms and functionalities for both multi-objective and many-objective optimisation. Reusing general functionalities provided by JCLEC *core* (v4.0) [2], the framework includes implementations of well-known approaches like SPEA2 and NSGA-II, as well as other recently proposed methods based on decomposition techniques or landscape partition. In addition, JCLEC-MO provides a set of utilities to create experiments, report outcomes, evaluate the performance of algorithms in terms of quality indicators, as well as the necessary support to solve real-world optimisation problems. JCLEC-MO is compatible with other Java applications and analytical tools like datapro4j [3] and R[1].

The main reason behind the creation of this software was the growing interest in the resolution of multi-objective problems (MOPs) and many-objective problems (MaOPs). Separating the required functionalities to address multi-objective optimisation (MOO) from JCLEC *core* would help controlling their development in a more independent way and, at the same time, would allow an independent adaptation of those previously available elements of the library. As a result, JCLEC-MO better covers the specific necessities of both MOO practitioners and researchers.

1.1 Main functionalities

The key features of JCLEC-MO are listed below:

Built on top of a multi-purpose framework. JCLEC-MO can use all the functionalities of the generic library for evolutionary computation (JCLEC). It provides a wide set of genetic operators, e.g. selectors, crossovers and mutators. Additionally, different evolutionary paradigms can be applied to solve MOPs, including genetic programming (GP).

Ease of extension. Due to the continuous advances within the MOO field, JCLEC-MO has been designed considering extensibility as a key feature. This framework provides a number of extension points that properly allow integrating new algorithms, indicators and benchmarks. To do this, great efforts have been devoted to provide a precise specification of the architecture and a modular design of the algorithms.

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

Availability of generic algorithms. JCLEC-MO looks for the independence of the evolution steps that are really proposed by the multi-objective algorithm, meaning that it would be possible to combine each of them with any available, current or future, metaheuristic paradigm.

Flexibility of the MOP definition. Although JCLEC allows the definition of both minimisation and maximisation problems, additional characteristics of the MOP definition can be considered here. JCLEC-MO incorporates a new way for handling constraints, which lied on the user's responsibility in JCLEC, and the possibility of solving MOPs where some objectives should be maximised and others should be minimised.

Recently proposed algorithms. The rapid advance of MOO techniques makes necessary to count on the most recent algorithm variants in order to provide developers and researchers with competitive tools. In this sense, JCLEC-MO provides a wide variety of algorithms, including the most representative ones of different families of approaches, and some recently proposed many-objective algorithms.

Experimental support. Utilities for MOO research include well-established benchmarks and quality indicators in order to validate the performance of algorithms, and the generation of reports containing experimental outcomes. Post-processing steps are provided by using the datapro4j library, including access to R functionalities like generating graphics and applying statistical tests.

1.2 Licence

This software was developed by members of the Knowledge Discovery and Intelligent Systems (KDIS) Research Group at the University of Córdoba, Spain. For further information on the library and modifications, please visit the URL `http://www.uco.es/grupos/kdis/jclec-mo`.

1.3 Getting started

This section helps users to take their first steps to discover JCLEC-MO, including the explanation of the installation process and some details about the structure of the project.

1.3.1 How to obtain the software

Download

JCLEC-MO is provided for download in different ways:

- The runnable *jar* file, *jclec-mo.jar*.
- The binary *jar* file, *jclec-mo-1.0.jar*.

They can be obtained from the website: `http://www.uco.es/grupos/kdis/jclec-mo`

External dependencies

JCLEC-MO requires the following external libraries:

- JCLEC *base* (v4.0+), either in binary form or as source files, which can be obtained from `http://jclec.sourceforge.net`.
- datapro4j library core (v1.0+), and the additional module to access to R[2]. They are available for download from `http://www.uco.es/grupos/kdis/jclec-mo`.
- Apache Commons libraries, which can be obtained from `https://commons.apache.org`. The following libraries are required: *collections* (v3.2.2), *configuration* (v1.10), *lang* (v.2.6) and *logging* (1.2+)[3].
- JUnit (v4.12+), as well as harmcrest-core (v1.3+), which are available for download from `http://junit.org`.

[2]R *core* and *rJava* package should be installed. They can be obtained from `https://www.r-project.org` and `http://www.rforge.net/rJava`, respectively

[3]Due to compatibility issues between JCLEC *base* and Apache Commons latest versions, JCLEC-MO is required to include *collections* v.3.2.2, *configuration* v1.10 and *lang* v2.6.

Users can download those external libraries that they really need. For instance, datapro4j is only required by some *reporters* (see Section 3.3) and *handlers* (see Section 3.4), whereas *JUnit* has been included for testing purposes. All these dependencies can be directly obtained from the website: `http://www.uco.es/grupos/kdis/jclec-mo`.

Installation

If JCLEC-MO is downloaded as a runnable *jar* file, the user can directly start working with it. External dependencies do not require installation, although they should be accessible to JCLEC-MO. According to the MANIFEST.MD file included within the executable file, external dependencies are expected to be located in a subfolder named *jclec-mo-1.0_lib* inside the directory where *jclec-mo-1.0.jar* has been downloaded.

Available documentation

JCLEC-MO source code has been carefully documented using *Javadoc* tags. The corresponding API in HTML is available from `http://www.uco.es/grupos/kdis/jclec-mo/v1/api`. A PDF version can be also downloaded from the website.

In addition, several documents can be consulted in order to know about the design of JCLEC-MO, as well to learn how to use both basic and advanced functionalities:

- An overall vision of the preliminary architecture of JCLEC-MO was presented at EvoSoft workshop during GECCO'15 conference:

A. Ramírez, J.R. Romero, S. Ventura. *An Extensible JCLEC-based solution for the Implementation of Multi-Objective Evolutionary Algorithms.* GECCO Companion'15. pp. 1085-1092. 2015. `http://dx.doi.org/10.1145/2739482.2768461`

- A complete guide to end-users (this document) can be downloaded from `http://www.uco.es/grupos/kdis/jclec-mo`.

- The design specification, that might be of interest to advanced users and programmers, is also available from `http://www.uco.es/grupos/kdis/jclec-mo`.

1.3.2 How to execute experiments

Running an experiment on console

Experiments in JCLEC-MO are deployed from configuration files. To execute an experiment, the configuration file should be specified as the parameter of the executable program:

```
java -jar jclec-mo-1.0.jar experiment.xml
```

Running a sequence of experiments

JCLEC core provides batch processing functionalities, thus allowing the execution of a set of experiments in the same run. More specifically, JCLEC can retrieve multiple values for a given component of the configuration, or even several ones, just by including the *multi* flag. JCLEC will automatically execute as many experiments as combinations of multiple parameters can be established. The rest of parameters will be applied to all the experiments. The invocation process remains the same to that explained above. Section 4 includes some examples on the use of this feature to create advanced experiments.

Running experiments by using external Java applications

Programmers can integrate experiments in their own Java processes since JCLEC-MO defines two classes, `MOExperiment` and `MOExperimentRunner`, aimed at creating and running MOO experiments, respectively. As can been seen in the following code snapshot, the former class can be used to associate one or more configuration files with an experiment, using the path to either a file or a directory (lines 2-3). The latter class will receive the created collection of experiments and then will execute them sequentially (lines 5-8).

```
1 // MO experiment
2 MOExperiment myExperiment = new MOExperiment();
3 myExperiment.addConfigurationsFromDirectory("./configuration-files/");
4 // JCLEC-MO runner
5 MOExperimentRunner runner = new MOExperimentRunner();
6 for(int i=0; i<myExperiment.getNumberOfConfigurations(); i++){
7     runner.executeSequentially(myExperiment.getConfiguration(i));
8 }
```

2. EXECUTING AN ALGORITHM

JCLEC-MO provides a number of native algorithms that can be directly executed to solve different MOPs. Similarly to JCLEC, the elements constituting an experiment should be defined in a configuration file in XML format [2]. This chapter details the set of available multi-objective algorithms and how they should be specified in the configuration file. Then, two examples are explained in order to show the complete configuration of an experiment for solving well-known MOO test problems.

2.1 Catalogue of algorithms

JCLEC-MO differs from JCLEC in the way in which an algorithm is configured. In JCLEC, all the evolutionary steps are defined by the `PopulationAlgorithm` abstract class, so any concrete class extending from it should implement the entire search process. Frequently, multi-objective approaches are mainly focused on steps that are independent from the problem definition. On the basis of this idea, JCLEC-MO performs the execution of the search process by interacting the following two elements:

- The metaheuristic algorithm: A class that indicates the metaheuristic paradigm to be used, which determines the variation mechanism.
- The multi-objective approach: A class that implements those steps of the algorithm that are really related to a multi-objective proposal. Here, it will be referred to as the *strategy*.

Further details about the design of these elements and the execution workflow between them can be consulted in the advanced documentation (see Section 1.3.1).

2.1.1 Metaheuristic paradigms

Table 2.1 shows the list of metaheuristic paradigms currently supported. These algorithms define their own mechanism to generate offspring from the set of selected solutions. The rest of the search process, i.e. initialisation, selection and replacement, is controlled by

Table 2.1: List of available algorithms and their variation operators

Genetic Algorithm (GA)	
class	`net.sf.jclec.mo.algorithm.MOGeneticAlgorithm`
recombinator	A recombinator with certain probability (`FilteredRecombinator`).
mutator	A mutator with certain probability (`FilteredMutator`).
Evolution Strategy (ES)	
class	`net.sf.jclec.mo.algorithm.MOESAlgorithm`
recombinator	A recombinator extending `AbstractRecombinator`, which is optional.
mutator	A mutator extending `AbstractMutator`.
lambda	The number of offspring to be generated, expressed as an integer value.
Genetic Programming (GP)	
class	`net.sf.jclec.mo.algorithm.MOGPAlgorithm`
recombinator	A `FilteredRecombinator` for tree encoding.
mutator	A `FilteredMutator` for tree encoding.
Evolutionary Programming (EP)	
class	`net.sf.jclec.mo.algorithm.MOEPAlgorithm`
mutator	An `AbstractMutator` for tree encoding.
GA without mutation	
class	`net.sf.jclec.mo.algorithm.MOOnlyRecombinatorAlgorithm`
recombinator	A recombinator with certain probability (`FilteredRecombinator`).
Particle Swarm optimisation (PSO)	
class	`net.sf.jclec.mo.algorithm.MOPSOAlgorithm`

an abstract algorithm, either `MOECAlgorithm` or `MOPSOAlgorithm`, that coordinates the interaction of all the components taking part in the experiment.

2.1.2 Multi- and many-objective algorithms

Table 2.2 shows the list of multi- and many-objective algorithms currently implemented, classified according to their families. JCLEC-MO also provides variants to deal with constrained problems, which can be found in the package `net.sf.jclec.mo.strategy.constrained`.

Regarding the type of MOP to be solved, it should be noted that JCLEC-MO can deal with both maximisation and minimization problems. Due to the algorithms' definition, combining both types of objective functions, e.g. one objective that should be maximised and other objective that should be minimised, is only supported by the following algorithms: ϵ-MOEA, MOCHC, NSGA-II, NSGA-III, OMOPSO, PAES, SPEA2 and SMPSO. Next, a brief description of each algorithm is provided.

SPEA2. The Strength Pareto Evolutionary Algorithm 2 (SPEA2) was proposed by E. Zitzler, M. Laumanns and L. Thiele in 2001 [4], as an improved version of SPEA. It

Table 2.2: List of multi-objective algorithm and their source class

Algorithms based on Pareto dominance	
SPEA2	`net.sf.jclec.mo.strategy.SPEA2`
NSGA-II	`net.sf.jclec.mo.strategy.NSGA2`
MOCHC	`net.sf.jclec.mo.strategy.MOCHC`
Multi-objective Evolution Strategies	
(1+1)-PAES	`net.sf.jclec.mo.strategy.PAES`
(1+λ)-PAES / (μ+λ)-PAES	`net.sf.jclec.mo.strategy.PAESlambda`
Algorithms based on indicators	
IBEA (using hypervolume)	`net.sf.jclec.mo.strategy.IBEAhv`
IBEA (using ϵ-indicator)	`net.sf.jclec.mo.strategy.IBEAe`
SMS-EMOA	`net.sf.jclec.mo.strategy.SMSEMOA`
HypE	`net.sf.jclec.mo.strategy.HypE`
Algorithms based on decomposition	
MOEA/D (Weighted Sum approach)	`net.sf.jclec.mo.strategy.MOEADws`
MOEA/D (Tchebycheff approach)	`net.sf.jclec.mo.strategy.MOEADte`
Algorithms based on landscape partition	
ϵ-MOEA	`net.sf.jclec.mo.strategy.SSeMOEA`
GrEA	`net.sf.jclec.mo.strategy.GrEA`
Algorithms based on reference points	
NSGA-III	`net.sf.jclec.mo.strategy.NSGA3`
RVEA	`net.sf.jclec.mo.strategy.RVEA`
Algorithms based on preferences	
PAR	`net.sf.jclec.mo.strategy.PAR`
PSO algorithms	
OMOPSO	`net.sf.jclec.mo.strategy.OMOPSO`
SMPSO	`net.sf.jclec.mo.strategy.SMPSO`

incorporates a fine-grained fitness assignment strategy, a density estimation technique, and an enhanced archive truncation method.

NSGA2. The Non Dominating Sorting Genetic Algorithm II (NSGA-II) was proposed by K. Deb *et al.* in 2002 [5] with the aim of improving the performance of its predecessor, NSGA. It is a multi-objective algorithm based on the Pareto dominance principle having low computational requirements. NSGA-II performs an elitism-preservation optimisation process that uses a dominance ranking to classify the population into fronts and a truncation operator based on a crowding distance to guide the search.

MOCHC. This algorithm is an adapted version of the CHC algorithm [6]. Proposed by A.J. Nebro *et al.* in 2007, MOCHC applies the same restart procedure that CHC, but it includes a new environmental selection based on the mechanism defined by NSGA-II.

PAES. The Pareto Archived Evolution Strategy (PAES) is an algorithm proposed by

J.D. Knowles and D.W. Corne in 2000 to solve MOPs under the evolution strategy (ES) paradigm [7]. PAES is a (1+1) ES that uses a reference archive to compare the mutated individual with those non-dominated solutions previously found in order to decide whether the mutant should replace the current individual. The same authors proposed the adaptation of the approach to (1+λ) and (μ+λ) ES. In these variants, a fitness value is assigned to each individual, which is then used to decided whether they will be included in the archive.

IBEA. The Indicator Based Evolutionary Algorithm (IBEA) is a general multi-objective algorithm proposed by E. Ziztler and S. Kunzli in 2004 [8]. It allows the user to set the desired quality indicator that will guide the search. JCLEC-MO provides two implementations of this general approach using two well-known indicators, hypervolume (HV) and ϵ-indicator.

SMS-EMOA. The S Metric Selection Evolutionary Multi-Objective Algorithm (SMS-EMOA) is a steady-state algorithm proposed by M. Beume *et al.* in 2007 [9]. It uses the S metric (hypervolume) to select the individuals that contribute the most to that metric. More specifically, a non-dominated sorting method is used to select the survivors. Then, SMS-EMOA uses the S contribution to discard solutions belonging to the worst-ranked front.

HypE. The Hypervolume Estimation Algorithm (HypE) was proposed by J.Bader and E. Ziztler in 2011 and it is characterised by the use of the hypervolume indicator to guide the search toward the Pareto front (PF) [10]. The fitness assignment method, used in both the mating and the environmental selection, is based on the contribution of each individual to the indicator. A Monte Carlo simulation can be used to sample the population in order to compute estimated values, allowing a fast procedure when the number of objectives is high.

MOEA/D. The Multi-Objective Evolutionary Algorithm based on Decomposition simultaneously performs an optimisation of diverse scalar subproblems [11]. Proposed by Q. Zhang and H. Li in 2007 as a generic framework, MOEA/D assigns a weight vector to every individual in the population, each one being focused on the resolution of the subproblem represented by its weight vector. Neighborhood information between subproblems is required for mating and environmental selection, whilst Pareto dominance is used to update the archive. Different approaches can be used to assign the fitness value, i.e. how well each individual solve its corresponding subproblem, so this algorithm is implemented as an abstract *strategy*. Two approaches have been implemented in JCLEC-MO: the weighted sum approach and the *Tchebycheff* approach.

ϵ-MOEA. It is a steady state algorithm proposed by M. Laummans *et al.* in 2002 that adapts the concept of Pareto dominance [12]. It divides the objective space into hypercubes, so individuals are compared by means of the ϵ-dominance relation over the resulting landscape partition.

GrEA. The Grid-based Evolutionary Algorithm (GrEA) was proposed by S. Yang *et al.* in 2013 as a landscape partition method for many-objective optimisation [13]. It proposes the partition of the objective space into grids, which are adaptively constructed considering the objectives values in the current population. Inspired by NSGA-II, GrEA also ranks the population into fronts during the environmental selection phase. Crowding distance and metrics that measure the spread of solutions are based on grid properties.

NSGA-III The Non-Dominating Sorting Genetic Algorithm III (NSGA-III) is a many-objective algorithm proposed by K. Deb and H. Jain in 2014 [14]. NSGA-III replaces the crowding distance used in NSGA-II by a survival mechanism that considers the distance of equivalent solutions, i.e. those belonging to the same front, to a set of reference points. The set of reference points can be provided by the used, simulating the most interesting search directions, or be automatically generated following the Das and Denni's approach.

RVEA The Reference Vector-gruided Evolutionary Algorithm (RVEA) is a many-objective algorithm proposed by R. Cheng *et al.* in 2016 [15]. Similar to NSGA-III, RVEA maintains a set of reference points that are use during environmental selection to choose the most promising solutions. A scalarisation approach, named angle-penalised distance, is used to decide the best solution for each reference vector. Instead of distances, RVEA is based on the angle between the solution in the objective space and the reference vector.

PAR The Preference-based Adaptive Region of interest (PAR) strategy is a technique that can be included in any MOEA to create preference-based algorithms. It was proposed by F. Goulart and F. Campelo in 2016 [16]. PAR defines preferences as a reference point that will be used to adaptively delimit the region of interest (ROI) and bias the search towards it.

OMOPSO. The OMOPSO algorithm was proposed by M. Reyes-Sierra and C.A. Coello Coello in 2005 [17]. This algorithm considers a fixed-size archive of external solutions and a turbulence mechanism based on two types of mutations. The ϵ-dominance principle and the crowding distance proposed by NSGA-II are used to discard solutions when the number of non-dominated solutions exceeds the archive size.

SMPSO. The Speed-constrained Multi-objective PSO (SMPSO) algorithm was proposed by A.J. Nebro *et al.* in 2009 [18]. Inspired by OMOPSO, this algorithm modifies the turbulence mechanism, which is now based on a unique mutator, and the velocity update mechanism.

2.2 Configuration file

Experiments in JCLEC-MO are created using a configuration file in XML format. The general structure of the configuration file is quite similar to that used in JCLEC, although

several new tags have been created to specify the new elements defined in JCLEC-MO. Firstly, this section explains the general structure of an experiment, including the required tags and their parameters. Next, the new elements defined for solving MOPs and their parametrisation are detailed.

2.2.1 General structure

The main elements comprising an experiment in JCLEC-MO are listed below:

Process. The metaheuristic algorithm to be executed. The configuration file should specify the path to the class implementing the algorithm, which will be commonly located in the package `net.sf.jclec.mo.algorithm`.

```
<process algorithm-type="net.sf.jclec.mo.algorithm.x">
```

Strategy. The multi-objective algorithm to be used. The path to the class should be set (native *strategies* can be found in the package `net.sf.jclec.mo.strategy`). Some multi-objective *strategies* define and manage specific properties of the solutions, e.g. ranking front and crowding distance in NSGA-II. In such a case, the fitness class used to encapsulate these properties need to be specified (some implementations are available from the package `net.sf.jclec.mo.evaluation.fitness`). If the *strategy* does not require a specific fitness object, the `MOFitness` class will be used by default. Additional parameters could be defined by a specific *strategy*, as will be shown later.

```
<mo-strategy type="net.sf.jclec.mo.strategy.x">
   <fitness type="net.sf.jclec.mo.evaluation.fitness.x" />
   ...
</mo-strategy>
```

General parameters. As any JCLEC experiment, the random number generator, the population size and the stopping criterion, e.g. a maximum of generations or evaluations, should be specified.

```
<rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789"/>
<population-size>100</population-size>
<max-of-generations>100</max-of-generations>
```

Encoding and provider. The type of encoding and the mechanism to create the initial population require the configuration of a pair of classes, named the *species* and the *provider*, respectively. The definition of these elements is inherited from JCLEC, so the classes currently available to manage binary, integer, permutation, real and

tree genotypes are located in several packages belonging to the library core, e.g. `net.sf.jclec.binarray`.

```
<provider type="net.sf.jclec.x"/>
<species type="net.sf.jclec.x">
    ...
</species>
```

Evaluator and objectives. The *evaluator* represents the class in charge of the evaluation of solutions. In JCLEC-MO, the list of objective functions should be specified, as well as their parameters. More specifically, each objective function is defined in terms of the class implementing it, the upper and lower bounds and whether it should be maximised or minimised. Some examples can be found in the `net.sf.jclec.mo.problem` package.

```
<evaluator type="net.sf.jclec.mo.evaluation.x">
  <objectives>
    <objective type="net.sf.jclec.mo.problem.x" min="0" max="1" maximize="false" />
    <objective type="net.sf.jclec.mo.problem.x" min="0" max="1" maximize="false" />
  </objectives>
</evaluator>
```

Variation operators. Most of the evolutionary algorithms require the configuration of the genetic operators that define the variation mechanism. Depending on the selected algorithm, these elements are mandatory or not (see Section 2.1.1). Frequently, they might define some parameters, e.g. the mutation probability. JCLEC-MO can use all the operators defined in JCLEC, which are stored in several packages organised by the type of encoding used. For instance, recombinators for binary individuals are available from the package `net.sf.jclec.binarray.rec`, whilst mutators can be found in the package `net.sf.jclec.binarray.mut`. The optional turbulence operation in MOPSO is done by using mutators compatible to real encoding (see the package `net.sf.jclec.realarray.mut`).

```
<recombinator type="net.sf.jclec.x" rec-prob="0.9" />
<mutator type="net.sf.jclec.x" mut-prob="0.15" />
```

Reporters. Outcomes of an experiment can be reported in several different ways using *listeners*. JCLEC-MO provides a collection of highly configurable classes specifically designed to retrieve results and statistics about the search process, though new ones can be added if required. Currently available *listeners* are located in the package `net.sf.jclec.mo.listener`.

```
<listener type="net.sf.jclec.mo.listener.x">
    ...
</listener>
```

An overview of the overall structure of the configuration file is shown below:

```
<experiment>
  <!-- The type of algorithm (GA, GP, PSO...) -->
  <process algorithm-type="net.sf.jclec.mo.algorithm.x">

    <!-- The multi-objective algorithm -->
    <mo-strategy type="net.sf.jclec.mo.strategy.x">
      <!-- Fitness class to be used -->
      <fitness type="net.sf.jclec.mo.evaluation.fitness.x" />
      ...
    </mo-strategy>

    <!-- General parameters -->
    <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789"/>
    <population-size>100</population-size>
    <max-of-generations>100</max-of-generations>

    <!-- Encoding and initialisation mechanism -->
    <provider type="net.sf.jclec.x"/>
    <species type="net.sf.jclec.x">
      ...
    </species>

    <!-- Evaluator -->
    <evaluator type="net.sf.jclec.mo.problems.x">
      <!-- List of objective functions -->
      <objectives>
        <objective type="net.sf.jclec.mo.problem.x" min="0" max="1" maximize="false" />
        <objective type="net.sf.jclec.mo.problem.x" min="0" max="1" maximize="false" />
      </objectives>
    </evaluator>

    <!-- Genetic operators -->
    <recombinator type="net.sf.jclec.x" rec-prob="0.9" />
    <mutator type="net.sf.jclec.x" mut-prob="0.15" />

    <!-- Listeners -->
    <listener type="net.sf.jclec.mo.listener.x">
      ...
    </listener>
    ...
  </process>
</experiment>
```

2.2.2 Specific algorithm parameters

Each multi-objective algorithm might require its own parameters, which should be also included in the configuration file. This section details the setting of each algorithm, speci-

fying the XML tags used to retrieve their parameters. The following general considerations must be taken into account:

- An optional parameter indicates whether a parameter is mandatory.
- A default value specifies whether or not JCLEC-MO can assume a default value for that parameter.

SPEA2 parameters

Table 2.3 shows the list of parameters of SPEA2.

Table 2.3: SPEA2 parameters

Parameter	XML tag	Optional	Default value
Size of the archive (S_A)	`archive-size`	No	Population size (S_P)
k-value (k-nearest neighbour)	`k-value`	No	$\sqrt{S_P + S_A}$

An example of the configuration of SPEA2 can be found below:

```
<mo-strategy type="net.sf.jclec.mo.strategy.SPEA2">
  <archive-size>100</archive-size>
  <k-value>5</k-value>
</mo-strategy>
```

NSGA-II parameters

Although NSGA-II does not define any parameter, the JCLEC-MO implementation requires the specification of a fitness object to store the ranking front and the crowding distance (see Table 2.4).

Table 2.4: NSGA-II parameters

Parameter	XML tag	Optional	Default value
Fitness class	`fitness`	No	No

JCLEC-MO provides an implementation of a specific fitness, named `NSGA2MOFitness`. Thus, the configuration file should include the following information:

```
<mo-strategy type="net.sf.jclec.mo.strategy.NSGA2">
  <fitness type="net.sf.jclec.mo.evaluation.fitness.NSGA2MOFitness" />
</mo-strategy>
```

MOCHC parameters

Table 2.5 shows the parameters of MOCHC.

Table 2.5: MOCHC parameters

Parameter	XML tag	Optional	Default value
Fitness	`fitness`	No	No
Cataclysmic mutator	`mutator`	No	No
Distance	`distance`	No	No
Initial distance threshold	`initial-d`	No	No
Threshold after restarting	`restart-d`	No	No
Number of survivors	`number-of-survivors`	No	5% of S_P
Converge threshold	`convergence-value`	No	1

As MOCHC is based on the properties defined by NSGA-II, the same fitness object can be used here. In addition, JCLEC-MO allows executing MOCHC regardless of the problem encoding, provided that the user specifies the distance measure that should be considered to evaluate the convergence of the search process. Since MOCHC does not use mutation to generate offspring, this *strategy* should be combined with `MOOnlyRecombinatorAlgorithm`. An example of the MOCHC configuration is shown below:

```
<process algorithm-type="net.sf.jclec.mo.algorithm.MOOnlyRecombinatorAlgorithm">
  <mo-strategy type="net.sf.jclec.mo.strategy.MOCHC">
    <!-- Fitness class to be used -->
    <fitness type="net.sf.jclec.mo.evaluation.fitness.NSGA2MOFitness" />
    <!-- Cataclysmic mutator -->
    <mutator type="net.sf.jclec.binarray.mut.OneLocusMutator" mut-prob="0.35" />
    <!-- Distance -->
    <distance type="net.sf.jclec.binarray.HammingDistance" />
    <!-- Rest of parameters -->
    <initial-d>3</initial-d>
    <restart-d>2</restart-d>
    <convergence-value>1</convergence-value>
    <number-of-survivors>5</number-of-survivors>
  </mo-strategy>
</process>
```

PAES parameters

Table 2.6 shows the parameters required in order to properly configure PAES.

The implementation of PAES makes use of a special fitness object to store additional properties of the individuals being evolved. JCLEC-MO provides a compatible implemen-

Table 2.6: (1+1)-PAES parameters

Parameter	XML tag	Optional	Default value
Fitness class	`fitness`	No	No
Number of bisections	`number-of-bisections`	No	No
Size of the archive	`archive-size`	No	No

tation, the `PAESMOFitness`, that should be specified in the configuration file. In addition, `MOESAlgorithm` should be selected as the type of algorithm, whereas the population size, which is a mandatory parameter of JCLEC, should be configured accordingly. λ parameter can be omitted, since its default value is equal to 1. To sum up, the configuration of PAES should be as follows:

```
<process algorithm-type="net.sf.jclec.mo.algorithm.MOESAlgorithm">
  <population-size>1</population-size>
  <mo-strategy type="net.sf.jclec.mo.strategy.PAES">
    <fitness type="net.sf.jclec.mo.evaluation.fitness.PAESMOFitness" />
    <number-of-bisections>10</number-of-bisections>
    <archive-size>20</archive-size>
  </mo-strategy>
  ...
</process>
```

Table 2.7 shows the set of parameters needed to configure the (1+λ) and (μ+λ) PAES variants. Notice that μ and λ should be both specified in the algorithm (see Section 2.1.1) and the *strategy*. More specifically, the μ value should be equal to the value configured using the `population-size` tag, whilst the λ value configured in the *strategy* should be equal to the value specified in the algorithm using the same tag.

Table 2.7: (1+λ)-PAES / (μ+λ)-PAES parameters

Parameter	XML tag	Optional	Default value
Fitness class	`fitness`	No	No
Number of bisections	`number-of-bisections`	No	No
Size of the archive	`archive-size`	No	No
Number of parents (μ)	`mu`	No	No
Number of offspring (λ)	`lambda`	No	No

Considering all the needed parameters, the configuration of (1+λ) and (μ+λ) variants should include the following elements:

```
<process algorithm-type="net.sf.jclec.mo.algorithm.MOESAlgorithm">
  <population-size>1</population-size> <!-- (mu) -->
  <lambda>4</lambda> <!-- (lambda) -->
  <mo-strategy type="net.sf.jclec.mo.strategy.PAESlambda">
    <fitness type="net.sf.jclec.mo.evaluation.fitness.PAESMOFitness" />
    <number-of-bisections>10</number-of-bisections>
```

```
        <archive-size>20</archive-size>
        <mu>1</mu>
        <lambda>4</lambda>
    </mo-strategy>
    ...
</process>
```

IBEA parameters

Table 2.8: IBEA parameters

Parameter	XML tag	Optional	Default value
Scaling factor (k)	`k`	No	No
Reference point for *HV* computation	`rho`	No	0.0 (for all objectives)

The required parameters of IBEA might depend on the selected indicator. Table 2.8 shows the common parameter (scaling factor) and an additional parameter that should be included if the algorithm uses the hypervolume as indicator. The configuration files for the two implemented variants are shown below:

```
<mo-strategy type="net.sf.jclec.mo.strategy.IBEAe"> <!-- IBEA using epsilon-indicator -->
    <k>0.05</k>
</mo-strategy>
```

```
<mo-strategy type="net.sf.jclec.mo.strategy.IBEAhv"> <!-- IBEA using hypervolume -->
    <k>0.05</k>
    <rho>0.0</rho>
</mo-strategy>
```

SMS-EMOA parameters

Table 2.9 shows the parameters required by SMS-EMOA.

Table 2.9: SMS-EMOA parameters

Parameter	XML tag	Optional	Default value
Parent selector	`parents-selectors`	No	No

As a specific mating selection process was not designed by the authors, this *strategy* allows configuring how to create the parents pool by setting a *selector*. Thus, any JCLEC *selector* can be used, taking into account its parameters and restrictions. For instance, a tournament selection can be configured as follows:

```
<mo-strategy type="net.sf.jclec.mo.strategy.SMSEMOA">
  <!-- Selector -->
  <parents-selector type="net.sf.jclec.selector.TournamentSelector">
    <tournament-size>2</tournament-size>
  </parents-selector>
</mo-strategy>
```

HypE parameters

In HypE, the sampling size should be specified (see Table 2.10). Nevertheless, this implementation can also be executed considering the exact HV contribution.

Table 2.10: HypE parameters

Parameter	XML tag	Optional	Default value
Sampling size	`sampling-size`	No	10000

In order to execute HypE without estimated HV values, the sampling size parameter should be equal to -1. The resulting configuration of HypE algorithm looks as follows:

```
<mo-strategy type="net.sf.jclec.mo.strategy.HypE" >
  <sampling-size>5000</sampling-size>
</mo-strategy>
```

MOEA/D parameters

Table 2.11 shows the parameters required by MOEA/D. Notice that the implementations uses the control parameter H to automatically generate the set of uniformly spread vectors. That value, in combination with the number of objectives, will determine the population size, since it should be equal to the number of generated vectors [11].

Table 2.11: MOEA/D parameters

Parameter	XML tag	Optional	Default value
neighborhood size (τ)	`t`	No	10
Maximum number of replacements	`nr`	No	2
Number of weight vectors (H)	`h`	No	No
Use external population	`external-pop`	Yes	True

Since the specific approaches do not define additional parameters, their configuration is quite similar:

```
<mo-strategy type="net.sf.jclec.mo.strategy.MOEADws"> <!-- Weighted sum -->
<!-- <mo-strategy type="net.sf.jclec.mo.strategy.MOEADte"> --> <!-- Tchebycheff -->
   <t>2</t>
   <nr>1</nr>
   <h>99</h>
</mo-strategy>
```

ϵ-MOEA parameters

In this algorithm, the length of each hypercube has to be configured. However, JCLEC-MO also provides an optional automatic configuration of the lengths considering the desired number of hypercubes and the bounds of each objective function. Table 2.12 shows the parameters of ϵ-MOEA.

Table 2.12: ϵ-MOEA parameters

Parameter	XML tag	Optional	Default value
Fitness class	`fitness`	No	No
Length of an hypercube	`epsilon-value`	Yes	No
Number of hypercubes	`number-of-hypercubes`	Yes	10

The current implementation of ϵ-MOEA will use the list of ϵ values provided by the user, which should contain as many values as the number of objectives of the problem to be solved. If only one value has been configured, it will be assigned as the length of the hypercubes in all the directions of the objective space. Only in case that ϵ values cannot be found in the configuration file, ϵ-MOEA will consider the second criterion, i.e. the number of hypercubes. Finally, the use of hypercubes requires storing the position of each individual in the resulting landscape partition. To do this, ϵ-MOEA needs a proper fitness object, like `HypercubeMOFitness`. A possible configuration of this *strategy* can be found below:

```
<mo-strategy type="net.sf.jclec.mo.strategy.SSeMOEA">
   <fitness type="net.sf.jclec.mo.fitness.strategy.HypercubeMOFitness" />
   <!-- Epsilon values -->
   <epsilon-values>
      <!-- Epsilon values (lengths of hypercube for each objective). If only one value
      is set, it will be applied to all the objectives -->
      <epsilon-value>0.01</epsilon-value>
      <epsilon-value>0.02</epsilon-value>
      <!-- Alternatively, you can set the number of hypercubes instead of their lengths -->
      <!--<number-of-hypercubes>5</number-of-hypercubes>-->
   </epsilon-values>
</mo-strategy>
...
<evaluator type="net.sf.jclec.mo.evaluation.x">
      <objectives>
```

```
        <objective type="net.sf.jclec.mo.problem.x" ... />
        <objective type="net.sf.jclec.mo.problem.x" ... />
      </objectives>
</evaluator>
```

GrEA parameters

Table 2.13 shows the parameters required by GrEA.

Table 2.13: GrEA parameters

Parameter	XML tag	Optional	Default value
Fitness class	`fitness`	No	No
Number of grids	`div`	No	No

Similarly to NSGA-II, GrEA requires a specific fitness object to store several grid measures. A compatible implementation is provided by JCLEC-MO, represented by the class `GrEAMOFitness`. Therefore, the configuration file should also indicate the path to this class:

```
<mo-strategy type="net.sf.jclec.mo.strategy.GrEA">
  <fitness type="net.sf.jclec.mo.fitness.strategy.GrEAMOFitness" />
  <div>5</div>
</mo-strategy>
```

NSGA-III parameters

Table 2.14 shows the parameters of NSGA-III.

Table 2.14: NSGA-III parameters

Parameter	XML tag	Optional	Default value
Fitness class	`fitness`	No	No
Does user provide reference points?	`user-points`	No	False
File that contains the points	`path`	Yes	No
No. of divisions in the boundary layer	`p1`	Yes	No
No. of divisions in the inner layer	`p2`	Yes	-1

The implementation of NSGA-III works as follows. If the value of `user-points` is *true*, the algorithm will load the reference points from the file specified in the `path` parameter. Otherwise, it will consider `p1` and `p2` values in order to automatically generate a set of well-spread points. If only one layer should be considered, `p2` should be equal to -1.

In addition, NSGA-III requires a fitness object to store the ranking front, among other properties. JCLEC-MO provides an example class with the required functionality, named `NSGA3MOFitness`, which can be used to complete the required configuration file:

```
<mo-strategy type="net.sf.jclec.mo.strategy.NSGA3">
   <fitness type="net.sf.jclec.mo.evaluation.fitness.NSGA3MOFitness" />
   <user-points>false</user-points>
   <!--<path>files/refPoints.txt</path>-->
   <p1>2</p1>
   <p2>1</p2>
</mo-strategy>
```

RVEA parameters

Table 2.15 shows the parameters of RVEA.

Table 2.15: RVEA parameters

Parameter	XML tag	Optional	Default value
No. of divisions in the boundary layer	`p1`	Yes	No
No. of divisions in the inner layer	`p2`	Yes	-1
Update frequency	`fr`	No	0.1
Alpha	`alpha`	No	2

RVEA considers `p1` and `p2` values in order to automatically generate a set of well-spread reference points. If `p2` is equal to -1, only one layer is considered. Parameter `fr` establishes the frequency to update reference points, and should be a value between 0 and 1. For instance, if the maximum number of generations is 100 and `fr` is set to 0.1, the update mechanism will be executed every 10 generations. `Alpha` is the exponent used to compute the angle-penalized distance. A typical configuration file for RVEA could be as follows:

```
<mo-strategy type="net.sf.jclec.mo.strategy.RVEA">
   <p1>2</p1>
   <p2>1</p2>
   <fr>0.2</fr>
   <alpha>3</alpha>
</mo-strategy>
```

PAR parameters

Table 2.16 shows the parameters of PAR.

Table 2.16: PAR parameters

Parameter	XML tag	Optional	Default value
Rho	`rho`	No	10E-6
Reference point	`refPoint`	No	No
Parent selector	`parents-selectors`	No	No

As PAR does not propose any specific mating selection, a parent selector from the JCLEC catalog should be configured. The user's preferences are expressed by means of a reference point, which indicates the desired objective value for each objective. The `rho` parameter is used to compute the achievement scalarization function and should be a small number greater than 0. The resulting configuration file could look as follows:

```
<mo-strategy type="net.sf.jclec.mo.strategy.PAR">
        <rho>0.0001</rho>
        <!-- Reference point -->
        <refPoint>
            <obj1>0.5</obj1>
            <obj2>0.0</obj2>
        </refPoint>
        <!-- Selector -->
        <parents-selector type="net.sf.jclec.selector.TournamentSelector">
            <tournament-size>2</tournament-size>
        </parents-selector>
    </mo-strategy>
```

OMOPSO parameters

The parameters of this algorithm are shown in Table 2.17.

Table 2.17: OMOPSO parameters

Parameter	XML tag	Optional	Default value
Fitness	`fitness`	No	No
Archive maximum size	`archive-size`	Population size	
Length of an hypercube	`epsilon-value`	Yes	No
Number of hypercubes	`number-of-hypercubes`	Yes	10

OMOPSO should be combined with a PSO algorithm (`MOPSOAlgorithm`) and an appropriate fitness object, e.g. `OMOPSOFitness`. The configuration with respect to the hypercubes is the same than that explained for ϵ-MOEA:

```
<process algorithm-type="net.sf.jclec.mo.algorithm.MOPSOAlgorithm">
  <mo-strategy type="net.sf.jclec.mo.strategy.OMOPSO">
    <fitness type="net.sf.jclec.mo.evaluation.fitness.OMOPSOMOFitness" />
```

```
      <archive-size>100</archive-size>
      <!-- Epsilon values -->
      <epsilon-values>
      <!-- Epsilon values (lengths of hypercube for each objective). If only one value
      is set, it will be applied to all the objectives -->
      <epsilon-value>0.01</epsilon-value>
      <epsilon-value>0.02</epsilon-value>
      <!-- Alternatively, you can set the number of hypercubes instead of their lengths -->
      <!--<number-of-hypercubes>5</number-of-hypercubes>-->
   </mo-strategy>
   ...
   <evaluator type="net.sf.jclec.mo.evaluation.x">
      <objectives>
         <objective type="net.sf.jclec.mo.problem.x" ... />
         <objective type="net.sf.jclec.mo.problem.x" ... />
      </objectives>
</evaluator>
</process>
```

SMPSO parameters

As SMPSO is based on OMOPSO, the list of parameters is the same (see Table 2.17), and only the path to the *strategy* should be modified:

```
<process algorithm-type="net.sf.jclec.mo.algorithm.PSOAlgorithm">
   <mo-strategy type="net.sf.jclec.mo.strategy.SMOPSO">
   ...
   </mo-strategy>
</process>
```

2.3 Example 1: ZDT test problem with NSGA-II

This section includes a complete example on how to define an experiment aimed at solving an already implemented problem. JCLEC-MO provides the implementation of a set of benchmarks like the ZDT test suite [19] in the package `net.sf.jclec.mo.problem`. Here, the ZDT4 problem will be used.

In short, ZDT4[1] is defined as a bi-objective MOP with 10 decision variables (m) in a continuous search space, i.e. using a real encoding. In order to solve this problem using NSGA-II, the following components should be configured:

[1] A more detailed description can be found at: `http://people.ee.ethz.ch/~sop/download/supplementary/testproblems/zdt4/`

1. Algorithm and *strategy*. This example will solve the problem using a genetic algorithm (`MOGeneticAlgorithm`) and the NSGA-II multi-objective *strategy* (`NSGA2`).

```
<process algorithm-type="net.sf.jclec.mo.algorithm.MOGeneticAlgorithm">
  <mo-strategy type="net.sf.jclec.mo.strategy.NSGA2">
    <fitness type="net.sf.jclec.mo.evaluation.fitness.NSGA2MOFitness" />
  </mo-strategy>
</process>
```

2. Encoding and initialisation mechanism. As this problem requires a real encoding, several classes of the `net.sf.jclec.realarray` package will be selected. The *species* element determines the type of individual to be used and the constraints with respect to the decision variables. Here, the genotype is composed of 10 genes, the first varying between 0 and 1 and the rest lying in the range [-5,5], as defined by the problem statement. A compatible *provider* (`RealArrayCreator`) will be in charge of creating the initial population completely at random.

```
<provider type="net.sf.jclec.realarray.RealArrayCreator"/>
<species type="net.sf.jclec.realarray.RealArrayIndividualSpecies">
  <genotype-schema>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
    <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
           closure="closed-closed" />
  </genotype-schema>
</species>
```

3. Evaluator. Since this problem does not require additional information, the default evaluator (`MOEvaluator`) has been selected. Notice that the list of objective functions that implements the ZDT4 minimization problem should be also specified.

```
<evaluator type="net.sf.jclec.mo.evaluation.MOEvaluator">
  <objectives>
    <objective type="net.sf.jclec.mo.problem.zdt.ZDT4F1Objective" maximize="false" />
```

```
        <objective type="net.sf.jclec.mo.problem.zdt.ZDT4F2Objective" maximize="false" />
    </objectives>
</evaluator>
```

4. Genetic operators. Both a *recombinator* and a *mutator* for real encoding should be selected among those provided by the library core. For instance, this experiment uses the `ArithmeticCrossover` and the `NonUniformMutator`. The recombination and mutation probabilities should be also determined.

```
<recombinator type="net.sf.jclec.realarray.rec.ArithmeticCrossover" rec-prob="0.8" />
<mutator type="net.sf.jclec.realarray.mut.NonUniformMutator" mut-prob="0.01" />
```

5. General parameters. For this problem, 100 individuals and 250 generations will be used. The random seed should be also specified, e.g. 123456789.

```
<rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789"/>
<population-size>100</population-size>
<max-of-generations>250</max-of-generations>
```

The complete configuration file in XML format is detailed below:

```
<experiment>
    <!-- The type of algorithm -->
    <process algorithm-type="net.sf.jclec.mo.algorithm.MOGeneticAlgorithm">

        <!-- The multi-objective approach -->
        <mo-strategy type="net.sf.jclec.mo.strategy.NSGA2">
            <fitness type="net.sf.jclec.mo.evaluation.fitness.NSGA2MOFitness" />
        </mo-strategy>

        <!-- General parameters -->
        <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789"/>
        <population-size>100</population-size>
        <max-of-generations>250</max-of-generations>

        <!-- Encoding and initialisation mechanism -->
        <provider type="net.sf.jclec.realarray.RealArrayCreator"/>
        <species type="net.sf.jclec.realarray.RealArrayIndividualSpecies">
            <!-- The number of genes depends on the ZDT problem -->
            <!-- ZDT4 with m=10 -->
            <genotype-schema>
                <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
                       closure="closed-closed" />
                <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                       closure="closed-closed" />
                <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                       closure="closed-closed" />
                <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                       closure="closed-closed" />
```

```
            <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                   closure="closed-closed" />
            <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                   closure="closed-closed" />
            <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                   closure="closed-closed" />
            <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                   closure="closed-closed" />
            <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                   closure="closed-closed" />
            <locus type="net.sf.jclec.util.range.Interval" left="-5" right="5"
                   closure="closed-closed" />
        </genotype-schema>
    </species>

    <!-- Evaluator for ZDT problems -->
    <evaluator type="net.sf.jclec.mo.evaluation.MOEvaluator">
        <objectives>
            <objective type="net.sf.jclec.mo.problem.zdt.ZDT4F1Objective" maximize="false" />
            <objective type="net.sf.jclec.mo.problem.zdt.ZDT4F2Objective" maximize="false" />
        </objectives>
    </evaluator>

    <!-- Genetic operators -->
    <recombinator type="net.sf.jclec.realarray.rec.ArithmeticCrossover" rec-prob="0.8" />
    <mutator type="net.sf.jclec.realarray.mut.NonUniformMutator" mut-prob="0.01" />
  </process>
</experiment>
```

2.4 Example 2: DLTZ test problem with SPEA2

This section shows more advances functionalities such as parallel evaluation and reporting outcomes. Here, SPEA2 is the selected multi-objective evolutionary algorithm to solve DTLZ1 optimisation problem[2]. DTLZ is another well-known test suite for MOO research [20], whose problems can be configured in terms of the number of objectives and decision variables to be used.

In DTLZ1, according to the problem parameters suggested by the authors, if $M = 2$ and $k = 5$, then the genotype length should be equal to 6, each gen varying in the range [0,1]. As for the elements taking part in the definition of the experiment, all of them should be compatible with a real encoding, i.e. only the classes located in the package `net.sf.jclec.realarray` can be selected. As a result, the general configuration of these elements is the same to that shown in Section 2.3.

[2]A more detailed description can be found at: `http://people.ee.ethz.ch/~sop/download/supplementary/testproblems/dtlz1/`

Other additional features considered in this example are the following:

1. A different evolutionary paradigm. SPEA2 will be applied in combination with the ES paradigm.

```
<process algorithm-type="net.sf.jclec.mo.algorithm.MOESAlgorithm">
  <mo-strategy type="net.sf.jclec.mo.strategy.SPEA2">
    <archive-size>10</archive-size>
    <k-value>5</k-value>
  </mo-strategy>
</process>
```

2. A different stopping criterion. Instead of defining a maximum number of generations, this experiment uses a maximum number of evaluations as stopping criterion. To do this, the `max-of-evaluations` parameter should be specified in the configuration file.

```
<max-of-evaluations>10000</max-of-evaluations>
```

3. Parallel evaluation of individuals. JCLEC-MO includes a parallel evaluator for the DTLZ1 problem. Notice that this evaluator also requires the list of objective functions. In this case, only one class is used to implement the two objective functions (`DTLZ1Objective`), whose behavior is adapted considering the number of objectives configured.

```
<evaluator type="net.sf.jclec.mo.problem.dtlz.DTLZ1ParallelEvaluator">
  <objectives>
    <objective type="net.sf.jclec.mo.problem.dtlz.DTLZ1Objective" maximize="false" />
    <objective type="net.sf.jclec.mo.problem.dtlz.DTLZ1Objective" maximize="false" />
  </objectives>
</evaluator>
```

4. Generation of reports. In order to know how the evolution is being performed, a simple reporter (`MOPopulationReporter`) will be used here. More specifically, the current population and the archive will be printed on console every 50 generations.

```
<listener type="net.sf.jclec.mo.listener.MOPopulationReporter">
  <report-frequency>50</report-frequency>
  <report-title>report-spea2-dtlz</report-title>
</listener>
```

The complete configuration file in XML format is detailed below:

```
<experiment>
  <!-- The type of algorithm -->
  <process algorithm-type="net.sf.jclec.mo.algorithm.MOESAlgorithm">
    <!-- The multi-objective approach -->
    <mo-strategy type="net.sf.jclec.mo.strategy.SPEA2">
      <archive-size>10</archive-size>
      <k-value>5</k-value>
    </mo-strategy>
    <!-- General parameters -->
    <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789"/>
    <population-size>100</population-size>
    <max-of-evaluations>10000</max-of-evaluations>
    <!-- Encoding and initialisation mechanism -->
    <provider type="net.sf.jclec.realarray.RealArrayCreator"/>
    <species type="net.sf.jclec.realarray.RealArrayIndividualSpecies">
      <!-- The number of genes depends on the problem definition (M=2, k=5, n=6) -->
      <genotype-schema>
        <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
               closure="closed-closed" />
        <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
               closure="closed-closed" />
        <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
               closure="closed-closed" />
        <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
               closure="closed-closed" />
        <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
               closure="closed-closed" />
        <locus type="net.sf.jclec.util.range.Interval" left="0" right="1"
               closure="closed-closed" />
      </genotype-schema>
    </species>
    <!-- Evaluator -->
    <evaluator type="net.sf.jclec.mo.problem.dtlz.DTLZ1ParallelEvaluator">
      <objectives>
        <objective type="net.sf.jclec.mo.problem.dtlz.DTLZ1Objective" maximize="false" />
        <objective type="net.sf.jclec.mo.problem.dtlz.DTLZ1Objective" maximize="false" />
      </objectives>
    </evaluator>
    <!-- Genetic operators -->
    <mutator type="net.sf.jclec.realarray.mut.NonUniformMutator" mut-prob="0.1" />
    <!-- Reporter -->
    <listener type="net.sf.jclec.mo.listener.MOPopulationReporter">
      <report-frequency>50</report-frequency>
      <report-title>report-spea2-dtlz</report-title>
    </listener>
  </process>
</experiment>
```

3. REPORTING OUTCOMES

JCLEC-MO provides a number of facilities aimed at visualising outcomes, computing quality indicators and generating reports. This chapter details the configuration of the four types of elements involved in these processes, i.e. *commands*, *indicators*, *reporters* and *handlers*, which might be useful to create more complex experiments. Further details about their design and how they could be extended can be found in the design specification document (see Section 1.3.1).

3.1 Commands

Commands implement commonly used operations whose execution might be necessary during the generation of reports. The set of available *commands*, stored in the package `net.sf.jclec.mo.command`, can be divided into two categories:

- Operations over solutions. JCLEC-MO defines *commands* to perform some operations over a set of solutions, e.g. sorting them, extracting the Pareto set (PS), counting feasible and infeasible solutions, or splitting a population according to a comparison criteria.
- Objective transformations. Inverting and scaling objective values are operations commonly performed by MOEAs. In addition, these *commands* might be used to guarantee that the PF satisfies the conditions required by some *indicators*.
- Vector generation. Some strategies require the generation of weight vectors, e.g. MOEA/D, or reference points, e.g. NSGA-III, to conduct the search.

Commands can appear as parameters of some *reporters*, as detailed later. For these elements, the path to the class implementing the operation has to be specified using the `type` tag. If a *command* defines its own parameters, they should be also included:

```
<listener type="net.sf.jclec.mo.listener.x">
    ...
    <!-- Invert objective values -->
    <command-invert type="net.sf.jclec.mo.command.ObjectiveInverter" />

    <!-- Scale objective values -->
```

```
    <command-scale type="net.sf.jclec.mo.command.ObjectiveScaler">
        <bounds>
            <bound min="0.0" max="20.0"/>
            <bound min="4.0" max="15.0"/>
        </bounds>
    </command-scale>

    <!-- Extract non-dominated solutions -->
    <command-extract type="net.sf.jclec.mo.command.NonDominatedSolutionsExtractor">
        <comparator type="net.sf.jclec.mo.comparator.fcomparator.ParetoComparator" />
    </command-extract>
    ...
</listener>
```

Table 3.1: List of available indicators and their prerequisites

Indicator	**In [0,1]**	**Min./Max.**
Unary indicators		
Hypervolume (HV)	Yes	Max.
Overall Nondominated Vector Generation (ONVG)	-	-
Spacing	Recommended	-
Binary indicators		
I_ϵ	Recommended	-
$I_{\epsilon+}$	Recommended	-
Error Ratio (ER)	-	-
Spread	Recommended	Max.
Generalised Spread	Recommended	Max.
Generational Distance (GD)	Recommended	-
Inverted Generational Distance (IGD)	Recommended	-
Hyperarea Ratio (HR)	Yes	Max.
Maximum PF Error	Recommended	-
Nondominated Vector Addition (NVA)	-	-
ONVG Ratio (ONVGR)	-	-
R2 indicator	Recommended	Max.
R3 indicator	Recommended	Max.
Two Set Coverage (Coverage)	-	-
Ternary indicators		
Relative Progress	Recommended	-

3.2 Quality indicators

Indicators are performance measures defined to evaluate the quality of a PF or to compare two PFs according to criteria like distances or covered area. The set of available *indicators*, stored in the package `net.sf.jclec.mo.indicator`, can be classified as follows:

- *Unary indicators*. This type of measure only requires one PF.
- *Binary indicators*. These *indicators* compute metrics that evaluated the performance on the basis of two PFs, e.g. a PF approximation and a reference PF.
- *Ternary indicators*. This type of *indicator* requires the configuration of three PFs.

Table 3.1 shows the complete list of currently available *indicators*, whose definitions can be found in [21]. This table also summarises some pre-conditions, i.e. whether the *indicator* requires an objective space in the range [0,1] (*In [0,1]*) and a specific type of optimisation problem (*Min./Max.*). The symbol − is used to represent that the corresponding *indicator* has not pre-conditions.

These measures can be automatically computed by two *reporters*, `MOIndicatorReporter` and `MOComparisonReporter`. The list of desired *indicators* should be specified in the configuration file using the `indicators` tag. Additional parameters, e.g. a file containing the second PF required by binary *indicators*, can be also included:

```
<listener type=type="net.sf.jclec.mo.listener.x">
    ...
    <!-- The list of quality indicators (example) -->
    <indicators>
        <indicator type="net.sf.jclec.mo.indicator.GenerationalDistance"/>
        <indicator type="net.sf.jclec.mo.indicator.Hypervolume"/>
        <indicator type="net.sf.jclec.mo.indicator.Spread"/>
        ...
        <!-- The file containing the second/true pareto front to compare with -->
        <second-pareto-front>truePF.txt</second-pareto-front>
        </indicators>
</listener>
```

3.3 Reporters

In JCLEC-MO, different kinds of reports can be generated, while the outcomes can be shown on console and/or saved in files. Table 3.2 shows the reporting parameters that can be configured for an experiment. If the user wants to save the generated reports, the selected *listener* will automatically create its own subfolder inside the reporting directory. In addition, if the corresponding subfolder contain previous results, the reports will be automatically numbered, allowing the storage of multiple executions of the same experiment.

Some *reporters* do not use the output flags, i.e. console and/or file, as they are aimed at generating reports with a specific format. The `filter-from-archive` parameter is a

Table 3.2: General parameters of a reporter

Parameter	XML tag	Optional	Default value
Title of the report	`report-title`	No	*untitled*
Report frequency	`report-frequency`	No	10 generations
Show report on console?	`report-on-console`	No	*true*
Save report in a file?	`report-on-file`	No	*false*
Filter archive?	`filter-from-archive`	No	*false*
Solution extractor	`command-extract`	No	Pareto dominance

boolean flag that should be activated if the archive of the selected multi-objective *strategy* can include both dominated and non-dominated solutions. In this case, the archive should be processed to remove dominated individuals before reporting it. To do this, an *extractor command* will be executed, considering the Pareto dominance principle by default.

The list of currently available reporters, including their possible configuration, is provided next:

- `MOPopulationReporter`. It reports the set of solutions belonging to both the current population and the archive at the given frequency. Solutions can be shown on console and/or saved in files:

```
<listener type="net.sf.jclec.mo.listener.MOPopulationReporter">
    <report-frequency>20</report-frequency>
    <report-on-file>true</report-on-file>
    <report-on-console>false</report-on-console>
    <report-title>my-report</report-title>
</listener>
```

- `MOParetoSetReporter`. It extracts the non-dominated solutions from both the population and the archive. Apart from the general parameters, the user should specify whether the archive has to be filtered and the *command* to do it.

```
<listener type="net.sf.jclec.mo.listener.MOParetoSetReporter">
    <report-frequency>25</report-frequency>
    <report-on-file>true</report-on-file>
    <report-on-console>true</report-on-console>
    <report-title>my-report</report-title>
    <filter-from-archive>true</filter-from-archive>
    <command-extract type="net.sf.jclec.mo.command.NonDominatedSolutionsExtractor">
        <comparator type="net.sf.jclec.mo.comparator.fcomparator.ParetoComparator" />
    </command-extract>
</listener>
```

- `MOParetoFrontRepoter`. It creates a CSV file containing the obtained PF at the given frequency. If the algorithm uses an archive, the non-dominated solutions will

be extracted from it, considering the `filter-from-archive` flag and the extractor *command*. Otherwise, they are obtained from the current population.

```
<listener type="net.sf.jclec.mo.listener.MOParetoFrontReporter">
  <report-frequency>25</report-frequency>
  <report-title>my-report</report-title>
  <filter-from-archive>true</filter-from-archive>
  <command-extract type="net.sf.jclec.mo.command.NonDominatedSolutionsExtractor">
    <comparator type="net.sf.jclec.mo.comparator.fcomparator.ParetoComparator" />
  </command-extract>
</listener>
```

- `MOIndicatorReporter`. This reporter computes quality indicators, which are specified using a customisable list. Depending on the selected measures, the execution of *commands* might be necessary. If the user does not specify them, the following default implementations are considered: objective values are scaled taking the bounds from the current population, inversion is performed when required, and the PS is extracted evaluating the standard Pareto dominance.

```
<listener type="net.sf.jclec.mo.listener.MOIndicatorReporter">
  <!-- General parameters -->
  ...
  <!-- The list of quality indicators (example) -->
  <indicators>
    <indicator type="net.sf.jclec.mo.indicator.Hypervolume"/>
    ...
  </indicators>
  <!-- Commands -->
  <command-invert type="net.sf.jclec.mo.command.ObjectiveInverter" />
  ...
</listener>
```

- `MOComparisonReporter`. This reporter creates a CSV file containing the value of a quality indicator for several algorithms. Its configuration process is similar to that explained for `MOIndicatorReporter`, only requiring two additional parameters to specify the number of algorithms to be compared and the number of executions that will be performed for each one:

```
<listener type="net.sf.jclec.mo.listener.MOComparisonReporter">
  <!-- General parameters -->
  ...
  <!-- Comparison parameters -->
  <number-of-algorithms>2</number-of-algorithms>
  <number-of-executions>3</number-of-executions>
  <!-- The list of quality indicators (example) -->
  <indicators>
    <indicator type="net.sf.jclec.mo.indicator.Hypervolume"/>
    ...
  </indicators>
  <!-- Commands -->
```

```
    <command-invert type="net.sf.jclec.mo.command.ObjectiveInverter" />
    ...
</listener>
```

`MOParetoFrontReporter` and `MOComparisonReporter` make use of the datapro4j library [3] to generate the reporting files. If the user wants to use these *reporters*, the library should be included in the classpath.

Finally, it should be noted that more than one reporter can be configured for a given experiment, as shown below:

```
<experiment>
  <process algorithm-type="net.sf.jclec.mo.algorithm.x">
    ...
    <!-- Listeners -->
    <!-- Reporting the whole population -->
    <listener type="net.sf.jclec.mo.listener.MOPopulationReporter">
      ...
    </listener>
    <!-- Reporting the PF -->
    <listener type="net.sf.jclec.mo.listener.MOParetoFrontReporter">
      ...
    </listener>
  </process>
</experiment>
```

3.4 Handlers

Users can implement their own post-processing analysis as chains of specific operations, here named *handlers*. After the execution of one or more experiments, *handlers* can load the outcomes generated by *reporters* and then generate summary reports and grapchics, or apply statistical tests. All these operations make use of datapro4j, including its module to invoke R commands, so the user's code should be properly configured. Currently available *handlers* (see the package `net.sf.jclec.mo.experiment.handler`), are:

- `ApplyStatisticalTest`. If the `MOComparisonReporter` has been used in several experiments, this *handler* retrieves the results for all of them and applies a statistical test for each indicator found. If the number of algorithms is equal to 2, the Wilcoxon test is performed. For a greater number of samples, the Friedman test is considered.
- `ComputeIndicators`. Quality indicators can be computed after the experiment instead of during its execution, which might be useful if the true PF is unknown a priori.

- `GenerateAlgorithmPF`. Taking all the PFs returned by the same algorithm but in different executions, this *handler* generates an unique PF representing the non-dominated solutions that the algorithm has found so far.
- `GenerateIndicatorBoxPlot`. If the `MOComparisonReporter` has been selected for an experiment, a boxplot for each quality indicator is generated with its results.
- `GenerateParallelPlots`. Considering a PF in CSV format, this *handler* creates a parallel coordinates plot to visualise all the solutions.
- `GenerateReferencePF`. If multiple algorithms have been executed in an experiment, a reference PF can be created extracting the non-dominated solutions found by all of them.
- `ScaleAlgorithmPF`. This *handler* scales the objective values of the PFs generated by `MOParetoFrontReporter`.

The following code fragment shows an example of how a post-processing chain can be defined. After the creation (lines 3-4) and execution of a set of experiments (lines 6-9), the *handlers* can be configured (lines 14-15). The name of the experiment should be equal to that configured in the *reporters* (line 11), thus *handlers* can automatically retrieve the outcomes. In addition, a boolean array indicating whether each objective should be maximised (*true*) or minimised (*false*) is required to properly evaluate the Pareto dominance (line 12). `GenerateParallelPlots` has two additional parameters, one that specifies whether the objective values have been previously scaled (*true*) or not (*false*), and another that indicates the format of the resulting plot file (line 15). Once *handlers* have been configured, the chain is constructed (line 17) and then processed (line 19). The resulting plots will be saved in a subfolder named *"plots"*, inside the same reporting directory. Each plot will represent the PF found by an algorithm when all its executions are considered.

```
1  public static void main(String [] args){
2      // MO experiment
3      MOExperiment myExperiment = new MOExperiment();
4      myExperiment.addConfigurationsFromDirectory("./configuration-files/");
5      MOExperimentRunner runner = new MOExperimentRunner();
6      for(int i=0; i<myExperiment.getNumberOfConfigurations(); i++){
7          runner.executeSequentially(myExperiment.getConfiguration(i));
8      }
9      // Post-processing
10     String dir = "./reports/myExperiment";
11     boolean [] mop = new boolean[]{false,false, false };
12     // Create the handlers
13     MOExperimentHandler handler1 = new GenerateAlgorithmPF(dir,mop);
14     MOExperimentHandler handler2 = new GenerateParallelPlots(dir, false, mop.length, 1); // png file
15     handler1.setSuccessor(handler2);
16     // Execute the chain
17     handler1.process();
18 }
```

4. CODING NEW PROBLEMS

JCLEC-MO can solve user-defined optimisation problems, which requires coding some domain-specific elements. This chapter is focused on how to develop and integrate these elements as part of a new experiment, taking JCLEC-MO classes as a basis.

4.1 Required component definitions

In order to implement a new MOP, at least the following domain-specific elements need to be defined:

1. Encoding of solutions. Three elements are usually involved in the encoding of solutions: *individual*, *species* and *provider*. If the MOP to be solved uses an encoding already available at JCLEC, the user only needs to configure it as explained in Section 2.2.1. Further information about the behavior and required parameters of these elements can be found in the JCLEC documentation [2]. Nevertheless, the example given in Section 4.3 covers the implementation of a new type of encoding extending the core classes.

2. Evaluation of solutions. In JCLEC-MO, the evaluation mechanism requires the implementation of the objective functions. If the MOP uses further information to evaluate solutions than that provided by the solutions themselves, the implementation of a specific *evaluator* will be also necessary.

3. Genetic operators. The library core includes a wide set of genetic operators, although new operators can be included, if desired. Interested readers are referred to the JCLEC documentation [2].

4. Constraint handling. JCLEC-MO provides a new mechanism to deal with constraints, so some considerations should be taken into account when coding constrained MOPs. Section 4.3 explains how to code and solve a problem with constraints.

Next, two well-known optimisation problems, i.e. the travelling salesman problem (TSP) and the knapsack problem, have been selected as two representative examples of how to integrate user's classes in JCLEC-MO.

4.1.1 Evaluation of solutions

When coding a new problem, the user might need to include domain information in form of parameters or problem instances. In such a case, a specific *evaluator* should be created either by extending from `MOEvaluator` or from `MOParallelEvaluator`. Next, the configuration method can be overridden in order to retrieve new parameters or load files. The *evaluator* is in charge of making these data available to the objectives, if required. A template for the configuration method is shown below:

```
public class MyMOEvaluator extends MOEvaluator {
    ...
    public void configure(Configuration settings) {
        // Call super method
        super.configuration();
        // Retrieve specific parameters
        ...
    }
}
```

Once the evaluator has been created, the objective functions should be coded. The `Objective` abstract class defines the general functionality that every objective function should provide. It also defines a configuration method that can be used to retrieve parameters. The evaluation method should be implemented according to the given MOP, but always returning a fitness object that quantifies the quality of the received solution with respect to the objective. A general scheme of this process is shown below:

```
public class myObjectiveFunction extends Objective {
    public IFitness evaluate(IIndividual solution) {
        // Evaluate this objective for the given solution
        double objectiveValue = ...
        IFitness fitness = new SimpleValueFitness(objectiveValue);
        return fitness;
    }
}
```

4.1.2 Constraint handling

Since some multi-objective approaches define their own mechanisms to deal with infeasible solutions, JCLEC-MO defines the interface `IConstrained`, which declares methods serving to provide information about the sort of solution encoded. This interface should be only implemented by those individuals or particles representing solutions of a constrained MOP. Thus, other elements of the evolutionary process, e.g. *algorithms* and *evaluators*, will be able to distinguish between infeasible and feasible solutions. The interface `IConstrained` is specified as follows:

```
1  public interface IConstrained {
2      /**
3       * Get whether the solution is feasible.
4       * @return True if the solution is feasible, false otherwise.
5       * */
6      public boolean isFeasible();
7
8      /**
9       * Set whether the solution is feasible.
10      * @param feasible The boolean value that has to be set.
11      * */
12     public void setFeasible(boolean feasible);
13
14     /**
15      * Get the degree of constraint violation.
16      * @return A double value representing the degree of infeasibility.
17      * */
18     public double degreeOfInfeasibility();
19
20     /**
21      * Set the degree of constraint violation.
22      * @param degree The degree of infeasibility that has to be set.
23      * */
24     public void setDegreeOfInfeasibility(double degree);
25 }
```

4.2 Example 3: The Travelling Salesman Problem

This optimisation problem [22] was originally defined as a single-objective problem, being later extended to a multi-objective formulation. In this variant, each objective considers a different matrix of distances (or other measure) for the nodes making up the graph. Given that a permutation encoding is already available in JCLEC, the user only needs to code the evaluation phase. Then, a configuration file including the all the needed classes will be created.

4.2.1 Evaluator and objectives

The TSP requires a matrix of values representing distances between cities or costs from travelling to one city to another, so the goal is to minimise the overall distance (or cost). In the multi-objective version here addressed, each objective has its own matrix of values, which can be loaded from a file representing the problem instance. The next code fragment shows how the list of problem instances can be retrieved from the configuration file using Apache facilities. After the configuration of the general parameters (line 5), the *evaluator*

should load as many files as objectives have been defined (lines 7-8), and then assign the matrices to each objective (lines 13-19). In this example, problems instances follows the TSPLIB format[1]. Nevertheless, the *evaluator* can be extended or modified to work with other kind of data format.

```
1  public class MOTSPEvaluator extends MOEvaluator {
2      ...
3      public void configure(Configuration settings) {
4          // Call super method
5          super.configure(settings);
6          // Get the list of problem instances
7          Configuration objSettings = settings.subset("problem-instances");
8          String filenames [] = objSettings.getStringArray("problem-instance");
9          int size = filenames.length;
10         // Load each instance and copy it to the corresponding objective
11         double [][] maxtrix;
12         Objective obj;
13         for(int i=0; i<size; i++){
14             matrix = loadProblemInstance(filenames[i]);
15             obj = this.getObjectives().get(i);
16             if(obj instanceof MOTSPObjective){
17                 ((MOTSPObjective)obj).setMatrixofValues(matrix);
18             }
19         }
20     }
21 }
```

After the implementation of the *evaluator*, the objective functions should be coded. As the only difference between one objective and others is the specific matrix of values being evaluated over the graph, only one class needs to be considered here (see the code snapshot below). After checking the type of encoding (line 11), the genotype of the solution can be extracted (line 13) and the corresponding graph is evaluated (lines 14-20).

```
1  public class MOTSPObjective extends Objective {
2      /* Matrix containing the data of a specific measure between
3        two nodes in the TSP problem (distances, cost...) */
4      private double [][] measure;
5      /* Evaluation method */
6      public IFitness evaluate(IIndividual solution) {
7          IFitness fitness = null;
8          int [] genotype;
9          int length;
10         // Check the genotype encoding
11         if(solution instanceof OrderArrayIndividual){
12             // Get the genotype
13             genotype = ((OrderArrayIndividual)solution).getGenotype();
14             double totalValue = 0.0;
15             // Calculate total value of the measure
16             length = genotype.length-1;
```

[1]Problem instances are publicly available at: `http://eden.dei.uc.pt/~paquete/tsp/`

```
17            for (int i=0; i<length; i++)
18                totalValue += this.measure[genotype[i]][genotype[i+1]];
19            totalValue += this.measure[genotype[length]][genotype[0]];
20            fitness = new SimpleValueFitness(totalValue);
21        }
22        return fitness;
23    }
24 }
```

The complete classes can be found in the package `net.sf.jclec.mo.problem.tsp`.

4.2.2 Configuration file

Having coded the evaluation mechanism, an experiment to solve the TSP can be easily run. Here, a unique XML file will be used to execute the same algorithm but considering five different random seeds as a way to illustrate the batch processing functionality. Two additional considerations should be taken into account. On the one hand, the use of individuals having a permutation as genotype can be specified by means of two classes: `OrderArrayIndividualSpecies` and `OrderArrayCreator`. On the other hand, JCLEC provides two *recombinators* and two *mutators* that are compatible to the required encoding. In this example, the `OrderPMXCrossover` and the `Order2OptMutator` operators will be used. All these classes can be found in the package `net.sf.jclec.orderarray`. Considering MOEA/D with the *Tchebycheff approach* as the multi-objective *strategy* and a genetic algorithm as the evolutionary paradigm, the resulting configuration file will contain the following elements:

```
<experiment>
  <!-- The type of algorithm -->
  <process algorithm-type="net.sf.jclec.mo.algorithm.MOGeneticAlgorithm">

    <!-- The multi-objective approach -->
    <mo-strategy type="net.sf.jclec.mo.strategy.MOEADte">
      <t>5</t>
      <nr>5</nr>
      <h>99</h>
    </mo-strategy>

    <!-- General parameters -->
    <rand-gen-factory multi="true">
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789" />
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="234567891" />
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="345678912" />
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="456789123" />
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="567891234" />
    </rand-gen-factory>
    <population-size>100</population-size>
    <max-of-generations>300</max-of-generations>
```

```
    <!-- Encoding and initialisation mechanism -->
    <species type="net.sf.jclec.orderarray.OrderArrayIndividualSpecies"
                genotype-length="100"/>
    <provider type="net.sf.jclec.orderarray.OrderArrayCreator"/>

    <!-- Evaluator for the multi-objective TSP -->
    <evaluator type="net.sf.jclec.mo.problem.tsp.MOTSPEvaluator">
      <!-- Problem instances -->
      <problem-instances>
        <problem-instance>problems/tsp/kroA100.tsp</problem-instance>
        <problem-instance>problems/tsp/kroB100.tsp</problem-instance>
      </problem-instances>
      <!-- List of objectives -->
      <objectives>
        <objective type="net.sf.jclec.mo.problem.tsp.MOTSPObjective" maximize="false" />
        <objective type="net.sf.jclec.mo.problem.tsp.MOTSPObjective" maximize="false" />
      </objectives>
    </evaluator>

    <!-- Genetic operators -->
    <recombinator type="net.sf.jclec.orderarray.rec.OrderPMXCrossover" rec-prob="0.9" />
    <mutator type="net.sf.jclec.orderarray.mut.Order2OptMutator" mut-prob="0.1" />

    <!-- Reporters -->
    <listener type="net.sf.jclec.mo.listener.MOParetoFrontReporter">
      <report-frequency>100</report-frequency>
      <report-title>report-moead-tsp</report-title>
    </listener>

    <listener type="net.sf.jclec.mo.listener.MOParetoSetReporter">
      <report-frequency>100</report-frequency>
      <report-title>report-moead-tsp</report-title>
    </listener>
  </process>
</experiment>
```

4.3 Example 4: The Knapsack Problem

Similarly to the TSP, the knapsack problem [23] was adapted to multi-objective optimisation. The multi-objective variant is a generalisation of its single-objective version, where each objective looks for the optimisation of the total profits of a different knapsack. Additionally, the weight constraint should be satisfied for every knapsack, which allows showing the capabilities of JCLEC-MO for dealing with constrained MOPs. A binary encoding will be used in this example.

4.3.1 Encoding and species for a constrained problem

A new type of individual should be coded not only by extending `BinArrayIndividual`, but also by implementing the interface `IConstrained`. Two properties have been defined to provide the required functionality: a boolean property to specify whether a solution is feasible, and a double value to express to what extend an individual is infeasible. The latter will be used to penalise those solutions exceeding the weight threshold the most, as will be shown later. The next code fragment shows the implementation of the methods specified by the aforementioned interface:

```
1  public class BinArrayConstrainedIndividual extends BinArrayIndividual implements
2  IConstrained {
3      /** This property indicates whether the individual is feasible */
4      private boolean isFeasible;
5      /** The degree of infeasibility */
6      private double degreeOfInfeasibility;
7      /** Empty constructor. */
8      public BinArrayConstrainedIndividual() {
9          super();
10     }
11     /** Parameterized constructor.
12      * @param genotype The genotype of the individual.
13      * */
14     public BinArrayConstrainedIndividual(byte[] genotype) {
15         super(genotype);
16     }
17     /** Parameterized constructor.
18      * @param genotype The genotype of the individual.
19      * @param fitness The fitness of the individual.
20      * */
21     public BinArrayConstrainedIndividual(byte[] genotype, IFitness fitness) {
22         super(genotype, fitness);
23     }
24     /* IConstrained interface methods */
25     public boolean isFeasible() {
26         return this.isFeasible;
27     }
28     public void setFeasible(boolean feasible) {
29         this.isFeasible = feasible;
30     }
31     public double degreeOfInfeasibility() {
32         return this.degreeOfInfeasibility;
33     }
34     public void setDegreeOfInfeasibility(double degree) {
35         this.degreeOfInfeasibility = degree;
36     }
37     ...
38 }
```

Having defined the type of individuals to be used, a new *species* should be coded as it is the responsible of creating the individuals from a given genotype. This process is performed by the method `createIndividual`, which can be implemented as follows:

```
1 public class BinArrayConstrainedIndividualSpecies extends BinArraySpecies implements IConfigure {
2     ...
3     public BinArrayIndividual createIndividual(byte[] genotype) {
4         return new BinArrayConstrainedIndividual(genotype);
5     }
6 }
```

4.3.2 Evaluator and objectives

The knapsack problem requires the definition of the set of weights and profits of the items than can be included in each knapsack. This information can be loaded from a file by an *evaluator* using the configuration method (see the code snapshot below). In this example, the problem instance will include the data for all the knapsacks using the format defined in the test suite maintained by E. Ziztler and M. Laumanns[2].

```
1  public class MOKnapsackEvaluator extends MOEvaluator {
2      ...
3      public void configure(Configuration settings) {
4          super.configure(settings);
5          // Load the problem instance
6          try{
7              String filename = settings.getString("problem-instance");
8              loadProblemInstance(filename);
9          }catch(IllegalArgumentException e){
10             System.err.println("A problem instance is required");
11             e.printStackTrace();
12         }
13          ...
14     }
15 }
```

Each objective function will perform the same evaluation process but considering a different set of weights and profits. The implementation of the evaluation method of an objective function should be similar to that shown in the following code example. Firstly, this method calculates the total profit of the currently stored items (lines 22-25), assigning that quantity as the objective value (line 27). Then, the method should check the weight constraint. In this example, if the capacity of the corresponding knapsack is exceeded (line 29), the solution will be marked as infeasible (line 34). As the degree of infeasibility

[2]Problem instances, optimal Pareto fronts and experimental results to compare with can be downloaded from: `http://www.tik.ee.ethz.ch/sop/download/supplementary/testProblemSuite`

need to be determined, the maximum excess considering all the knapsacks is used here to establish differences among all the infeasible solutions (lines 37-40).

```
1  public class MOKnapsackObjective extends Objective {
2    /* Profits of the items in the knapsack */
3     private double [] profits;
4     /* Weights of the items in the knapsack */
5     private double [] weights;
6     /* Capacity of the knapsack */
7     private double capacity;
8      ...
9     /* Configuration method */
10    public IFitness evaluate(IIndividual solution) {
11       IFitness fitness = null;
12       byte [] genotype;
13       int length;
14       double totalProfit = 0.0, totalWeight = 0.0;
15       double currentExcess, previousExcess;
16       // Check if the solution has a valid encoding
17       if(solution instanceof BinArrayIndividual){
18          // Get the binary genotype
19          genotype = ((BinArrayIndividual) solution).getGenotype();
20          length = genotype.length;
21          // Calculate the profit for this knapsack
22          for(int i=0; i<length; i++){
23             totalProfit += genotype[i]*this.profits[i];
24             totalWeight += genotype[i]*this.weights[i];
25          }
26          // Set the fitness value
27          fitness = new SimpleValueFitness(totalProfit);
28          // Check the weight constraint
29          if(totalWeight <= this.capacity){
30             ((IConstrained)solution).setFeasible(true);
31             ((IConstrained)solution).setDegreeOfInfeasibility(0.0);
32          }
33          else{
34             ((IConstrained)solution).setFeasible(false);
35             // The maximum excess (considering all the objectives)
36             // will be set as the degree of infeasibility
37             previousExcess = ((IConstrained)solution).degreeOfInfeasibility();
38             currentExcess = totalWeight-this.capacity;
39             if(currentExcess > previousExcess)
40                ((IConstrained)solution).setDegreeOfInfeasibility(currentExcess);
41          }
42       }
43       return fitness;
44    }
45 }
```

4.3.3 Configuration file

In order to execute an experiment aimed at solving the knapsack problem, the path to the new classes should be included in the configuration file. The completed implementation of the previously explained classes can be found in the `net.sf.jclec.binarray` and the `net.sf.jclec.mo.problem.knapsack` packages. Notice that a constrained version of the multi-objective *strategy* should be selected. For instance, NSGA-II proposes a specific procedure to compare solutions when dealing with constrained MOPs [5], whilst other algorithms, like SPEA2, simply promote feasible solutions over infeasible ones during the fitness assignment method. Both algorithms can be included in an experiment for comparison purposes. As the true PF of some problem instances are known, several binary indicators can be obtained in order to compare the performance of the selected algorithms. Therefore, `MOIndicatorReporter` and `MOComparisonReporter` will be used here. A complete example of the configuration file required to execute the proposed experiment, considering a problem instance with 100 items and 3 knapsacks, can be found below:

```
<experiment>
  <!-- The type of EC algorithm -->
  <process algorithm-type="net.sf.jclec.mo.algorithm.MOGeneticAlgorithm">

    <!-- The multi-objective approach -->
    <mo-strategy multi="true">
      <mo-strategy type="net.sf.jclec.mo.strategy.constrained.ConstrainedNSGA2">
        <fitness type="net.sf.jclec.mo.evaluation.fitness.NSGA2MOFitness" />
      </mo-strategy>
      <mo-strategy type="net.sf.jclec.mo.strategy.constrained.ConstrainedSPEA2" />
    </mo-strategy>

    <!-- General parameters -->
    <rand-gen-factory multi="true">
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789"/>
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="234567891"/>
      <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="345678912"/>
    </rand-gen-factory>

    <population-size>100</population-size>
    <max-of-generations>300</max-of-generations>

    <!-- Encoding and initialisation mechanism -->
    <species type="net.sf.jclec.binarray.BinArrayConstrainedIndividualSpecies"
             genotype-length="100"/>
    <provider type="net.sf.jclec.binarray.BinArrayCreator"/>

    <!-- Evaluator for the multi-objective knapsack problem -->
    <evaluator type="net.sf.jclec.mo.problem.knapsack.MOKnapsackEvaluator">
      <problem-instance>problems/knapsack/knapsack.100.3</problem-instance>
      <objectives>
        <objective type="net.sf.jclec.mo.problem.knapsack.MOKnapsackObjective"
                   maximize="true" />
```

```
        <objective type="net.sf.jclec.mo.problem.knapsack.MOKnapsackObjective"
                   maximize="true" />
        <objective type="net.sf.jclec.mo.problem.knapsack.MOKnapsackObjective"
                   maximize="true" />
    </objectives>
</evaluator>

<!-- Genetic operators -->
<recombinator type="net.sf.jclec.binarray.rec.UniformCrossover" rec-prob="0.9" />
<mutator type="net.sf.jclec.binarray.mut.OneLocusMutator" mut-prob="0.1" />

<!-- Reporters -->
<listener type="net.sf.jclec.mo.listener.MOComparisonReporter">
    <!-- Generic parameters -->
    <report-frequency>150</report-frequency>
    <report-title>report-knapsack</report-title>
    <filter-from-archive>true</filter-from-archive>

    <!-- Specific parameters -->
    <number-of-algorithms>2</number-of-algorithms>
    <number-of-executions>3</number-of-executions>

    <!-- The list of quality indicators -->
    <indicators>
        <indicator type="net.sf.jclec.mo.indicator.GenerationalDistance"/>
        <indicator type="net.sf.jclec.mo.indicator.Hypervolume"/>
        <indicator type="net.sf.jclec.mo.indicator.ONVG"/>
        <!-- The true PF -->
        <second-pareto-front>problems/knapsack/knapsack100-3-pareto.txt
        </second-pareto-front>
    </indicators>

    <!-- Commands -->
    <command-invert type="net.sf.jclec.mo.command.ObjectiveInverter" />
    <command-scale type="net.sf.jclec.mo.command.ObjectiveNormalizer">
        <bounds>
            <bound min="0.0" max="5000.0"/>
            <bound min="0.0" max="5000.0"/>
            <bound min="0.0" max="5000.0"/>
        </bounds>
    </command-scale>
    <command-extract
        type="net.sf.jclec.mo.command.NonDominatedFeasibleSolutionsExtractor">
        <comparator type="net.sf.jclec.mo.comparator.fcomparator.ParetoComparator" />
    </command-extract>
</listener>

<listener type="net.sf.jclec.mo.listener.MOIndicatorReporter">
    <!-- Generic parameters -->
    <report-frequency>150</report-frequency>
    <report-on-console>true</report-on-console>
    <report-on-file>true</report-on-file>
    <report-title>report-knapsack</report-title>
```

```
        <filter-from-archive>true</filter-from-archive>

        <!-- The list of quality indicators -->
        <indicators>
          <indicator type="net.sf.jclec.mo.indicator.AdditiveEpsilon" />
          <indicator type="net.sf.jclec.mo.indicator.Epsilon" />
          <indicator type="net.sf.jclec.mo.indicator.ErrorRatio" />
          <indicator type="net.sf.jclec.mo.indicator.GeneralizedSpread" />
          <indicator type="net.sf.jclec.mo.indicator.GenerationalDistance" />
          <indicator type="net.sf.jclec.mo.indicator.HyperareaRatio" />
          <indicator type="net.sf.jclec.mo.indicator.Hypervolume" />
          <indicator type="net.sf.jclec.mo.indicator.InvertedGenerationalDistance" />
          <indicator type="net.sf.jclec.mo.indicator.MaximumError" />
          <indicator type="net.sf.jclec.mo.indicator.ONVG" />
          <indicator type="net.sf.jclec.mo.indicator.Spacing" />
          <indicator type="net.sf.jclec.mo.indicator.TwoSetCoverage" />

          <!-- True PF -->
          <second-pareto-front>problems/knapsack/knapsack100-3-pareto.txt
          </second-pareto-front>
        </indicators>

        <!-- Commands -->
        <command-invert type="net.sf.jclec.mo.command.ObjectiveInverter" />
        <command-scale type="net.sf.jclec.mo.command.ObjectiveNormalizer">
          <bounds>
            <bound min="0.0" max="5000.0" />
            <bound min="0.0" max="5000.0" />
            <bound min="0.0" max="5000.0" />
          </bounds>
        </command-scale>
        <command-extract
          type="net.sf.jclec.mo.command.NonDominatedFeasibleSolutionsExtractor">
          <comparator type="net.sf.jclec.mo.comparator.fcomparator.ParetoComparator" />
        </command-extract>
      </listener>
    </process>
</experiment>
```

REFERENCES

[1] A. Ramírez, J. R. Romero, and S. Ventura, "An Extensible JCLEC-based Solution for the Implementation of Multi-Objective Evolutionary Algorithms," in *Proceedings of the Companion Publication of the 17th Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*, pp. 1085–1092, ACM, 2015.

[2] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a java framework for evolutionary computation," *Soft Computing*, vol. 12, no. 4, pp. 381–392, 2008.

[3] J. R. Romero, J. M. Luna, and S. Ventura, *datapro4j: the data processing library for Java*. Dept. of Computer Science and Numerical Analysis, University of Córdoba (Spain). Available for download from: http://www.uco.es/grupos/kdis/datapro4j, 2012.

[4] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," in *Proceedings of the Conference on Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*, pp. 95–100, 2001.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[6] 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 *Proceedings of the 9th Conference on Genetic and Evolutionary Computation (GECCO '07)*, pp. 876–883, ACM, 2007.

[7] J. D. Knowles and D. W. Corne, "Approximating the nondominated front using the pareto archived evolution strategy," *Evolutionary Computation*, vol. 8, no. 2, pp. 149–172, 2000.

[8] E. Zitzler and S. Künzli, "Indicator-Based Selection in Multiobjective Search," in *Parallel Problem Solving from Nature (PPSN VIII)*, vol. 3242 of *Lecture Notes in Computer Science*, pp. 832–842, Springer, 2004.

[9] N. Beume, B. Naujoks, and M. Emmerich, "SMS-EMOA: Multiobjective selection based on dominated hypervolume," *European Journal of Operational Research*, vol. 181, no. 3, pp. 1653–1669, 2007.

[10] J. Bader and E. Zitzler, "HypE: An algorithm for fast hypervolume-based many-objective optimization," *Evolutionary Computation*, vol. 19, no. 1, pp. 45–76, 2011.

[11] Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, 2007.

[12] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler, "Combining Convergence and Diversity in Evolutionary Multiobjective Optimization," *Evolutionary Computation*, vol. 10, pp. 263–282, Sept. 2002.

[13] S. Yang, M. Li, X. Liu, and J. Zheng, "A Grid-Based Evolutionary Algorithm for Many-Objective Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 5, pp. 721–736, 2013.

[14] K. Deb and H. Jain, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.

[15] R. Cheng, Y. Jin, M. Olhofer, B. Sendhoff, and S. Member, "A Reference Vector Guided Evolutionary Algorithm for Many-Objective Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 5, pp. 773–791, 2016.

[16] F. Goulart and F. Campelo, "Preference-guided evolutionary algorithms for many-objective optimization," *Information Sciences*, vol. 329, pp. 236–255, 2016.

[17] M. Reyes-Sierra and C. A. Coello Coello, "Improving PSO-Based Multi-objective Optimization Using Crowding, Mutation and ϵ-Dominance," in *Evolutionary Multi-Criterion Optimization*, vol. 3410 of *Lecture Notes in Computer Science*, pp. 505–519, Springer Berlin Heidelberg, 2005.

[18] A. Nebro, J. Durillo, J. Garcia-Nieto, C. Coello Coello, F. Luna, and E. Alba, "SMPSO: A new PSO-based metaheuristic for multi-objective optimization," in *Proceedings of the IEEE Symposium on Computational Intelligence in multi-criteria decision-making (MCDM'09)*, pp. 66–73, March 2009.

[19] E. Zitzler, K. Deb, and L. Thiele, "Comparison of Multiobjective Evolutionary Algorithms: Empirical Results," *Evolutionary Computation*, vol. 8, no. 2, pp. 173–195, 2000.

[20] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable Test Problems for Evolutionary Multiobjective Optimization," in *Evolutionary Multiobjective Optimization*, Advanced Information and Knowledge Processing, pp. 105–145, Springer, 2005.

[21] C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer, 2nd ed., 2007.

[22] T. Lust and J. Teghem, "The Multiobjective Traveling Salesman Problem: A Survey and a New Approach," in *Advances in Multi-Objective Nature Inspired Computing*, vol. 272 of *Studies in Computational Intelligence*, pp. 119–141, Springer, 2010.

[23] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.