jMetal: a Java Framework for Developing Multi-Objective Optimization Metaheuristics*

Juan J. Durillo, Antonio J. Nebro, Francisco Luna, Bernabé Dorronsoro, Enrique Alba

Departamento de Lenguajes y Ciencias de la Computación
E.T.S. Ingeniería Informática
Campus de Teatinos, 29071 Málaga (SPAIN)

email{durillo,antonio,flv,bernabe,eat}@lcc.uma.es

TECH-REPORT: ITI-2006-10

Abstract

This paper introduces jMetal, an object-oriented Java-based framework aimed at facilitating the development of metaheuristics for solving multi-objective optimization problems (MOPs). jMetal provides a rich set of classes which can be used as the building blocks of multi-objective metaheuristics; thus, taking advantage of code-reusing, the algorithms share the same base components, such as implementations of genetic operators and density estimators, so making the fair comparison of different metaheuristics for MOPs possible. The framework also incorporates a significant set of problems used as a benchmark in many comparative studies. We have implemented a number of multi-objective metaheuristics using jMetal, and an evaluation of these techniques using our framework is presented.

1 Introduction

Most optimization problems in the real world are complex, and one aspect that contributes to that complexity is their multi-objective nature. Other reasons have to do with the fact that function evaluations can be computationally expensive and the search spaces tend to be very large. As a consequence, exact techniques are often not applicable, so stochastic methods are mandatory; in particular, metaheuristics are popular algorithms to solve multi-objective optimization problems (MOPs). Among them, evolutionary algorithms have been investigated by many authors, and some of the most well-known algorithms for solving MOPs belong to this class (e.g. NSGA-II [6], PAES [7], SPEA2 [14]). Nevertheless, in recent years there is a trend to adapt other kinds of metaheuristics (sometimes called "alternative methods", with reference to evolutionary algorithms), such particle swarm optimization [10], or scatter search [8].

Currently, the rigorous comparison of different metaheuristics for solving multi-objective optimizations problems (MOPs) is an open unsolved issue, which is due for four reasons at least. First, there is not a unique benchmark of MOPs commonly accepted by the multi-objective research community; second, there is no agreement in which metrics should be used to assess the performance of the algorithms; third, the parameter settings may vary among the experiments carried out in different studies. Finally, the same metaheuristic can be implemented in several ways, what can affect even the numerical the performance and the working of the technique (e.g., the programming language used in the implementation, the sorting algorithm applied to rank populations, or the use of binary-coded real genes with different number of bits). Our work focuses in this last issue, the implementation of the techniques.

*This work has been partially funded by the Ministry of Science and Technology and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project).

In this paper, we propose a software framework called jMetal (from *metaheuristic algorithms in Java*) aimed at making easy to develop metaheuristic algorithms for solving MOPs. By using the object-oriented facilities provides by the Java language, the framework provides a rich set of classes that can be used as the basic building blocks of multi-objective metaheuristics. This way, by sharing the same base components, jMetal can be used to carry out fair comparisons of different techniques. To illustrate this idea, we use jMetal to compare a number of metaheuristics which are representative of the state-of-the-art, including three evolutionary algorithms (SPEA2 [14], NSGA-II [6], and PAES [7]), a particle swarm optimization algorithm (OMOPSO [10]), and a scatter search (AbYSS [9]).

The contributions of the paper can be summarized in the following:

- We describe the framework jMetal for developing metaheuristics for solving MOPs. We discuss the main design goals driving the framework, and give some details about its architecture and implementation in Java.
- To test jMetal, the implementation of the algorithms NSGA-II and SPEA2, developed on top of it, are compared against their reference implementations.
- We use jMetal to compare five metaheuristic algorithms for solving MOPs.

The rest of the paper is organized as follows. In Section 2, we discuss the motivation of our work. The design goals driving us when developing jMetal are presented in Section 3. The architecture and some implementations details of our framework are detailed in the next section. Section 5 includes an analysis of the jMetal versions of NSGA-II and SPEA2. In Section 6, we use jMetal to make a comparative study of five metaheuristics for solving MOPs. Finally, Section 7 contains the conclusions and some lines of future work.

2 Motivation and Related Work

In this section, we focus on the motivation driving the development of jMetal, which is closely related to other works concerning the implementation of multi-objective metaheuristics.

Currently multi-objective optimization is a high topic and, if we focus on metaheuristics, new algorithms appear continuously. Whenever a new technique is proposed, its authors try to demonstrate that it outperforms in some aspect those metaheuristics considered as the best ones, namely NSGA-II and SPEA2. However, although these algorithms are explained in detail in reports, papers, and books, there exist many different implementations of them. Let us consider NSGA-II. It is fully described in [6], and an implementation in C was provided by Deb[1]. That implementation suffered two major revisions in 2005, and it can be considered as the "official" NSGA-II algorithm. However, many researches have implemented NSGA-II on their own or they have used other available codes in the Web. Frequently, those "alternative" implementations of NSGA-II have not been compared against Deb's version, and when the presented results are shown, sometimes it is easy to verify that they are significantly worse than the ones obtained with Deb's code. An example is the NSGA-II version developed using the PISA framework [1].

Concerning PISA, this software system is mainly known by offering the"official" implementation of SPEA2[2]. The principles underlying PISA is to separate the algorithm-specific part of an optimizer from the application-specific part. This is carried out using a shared-file mechanism to communicate the module executing the application with the module running the metaheuristic. Communicating processes through shared files has the inconvenience of being expensive in time, so the algorithms developed using PISA have an execution time penalty. As a matter of fact, some studies argue that SPEA2 is slow, but if the performance of the algorithm is measured using PISA's implementation, it should be noted that a significant part of the inefficiency of SPEA2 can be due to the way it is implemented, nor to the algorithm itself.

If we intend to use the reference implementations of NSGA-II and SPEA2 we also have to face another issue. We will probably use the simulated binary crossover and polynomial mutation operators included in these implementations, but if we examine the codes of these operators they are not exactly the same each other. Thus, it is difficult to explain whether the dissimilarities obtained by the algorithms are due to their working principles or to the use of different implementations of the operators.

[1]The implementation of NSGA-II is available for downloading at: `http://www.iitk.ac.in/kangal/codes.shtml`

[2]The implementation of SPEA2 is available at: `http://www.tik.ee.ethz.ch/pisa/selectors/spea2/spea2.html`

In this context, two facts emerge. First, many researches develop their metaheuristics on their own, without taking benefits of reusing in a clean way the existing code. Second, it is not trivial to compare a new metaheuristic for MOPs against NSGA-II and SPEA2 in a fair way. A solution could be to take as a reference, for example, Deb's implementation of NSGA-II. However, this code is written in C, a non-object oriented language, what often hinders its utilization to develop new algorithms. The same can be argued if we intend to re-use PISA's implementation of SPEA2. Another choice is to use one of the object-oriented systems developed to implement evolutionary algorithms, such as Open BEAGLE [2], JavaEva [12], or ParadisEO [3]. Although they provide some support for developing multi-objective metaheuristics, the first two systems are mainly designed to single-objective optimization algorithms, and ParadisEO is aimed at parallel and distributed evolutionary algorithms. As a consequence, the designer of a new technique must cope with many issues in the codes which probably are out of his/her interest.

With these considerations in mind, we took the approach of designing a framework specifically oriented to multi-objective metaheuristic algorithms from scratch. The framework must allow to solve the aforementioned drawbacks to some extent. By using Java as programming language, we guarantee that the resulting programs are portable, and the object-oriented features of Java will allow us to facilitate the code-reuse in the algorithms.

3 Design Goals

Our purpose is that jMetal can be used by many researchers to develop their own algorithms and also to allow them to compare their new techniques against other metaheuristics. With this idea in mind, the main design goals driving jMetal are the following:

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

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

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

- **Extensibility**. New algorithms, operators, and MOPs should be easily added to the framework. This goal is achieved by using some mechanisms of Java, such as inheritance and late binding. For example, all the MOPs inherits from the class `Problem` (see Fig. 1), so a new problem can be created just by writing the methods specified by that class; once the class defining the new problem is compiled, nothing more has to be done: the late binding mechanism allows to load the code of the MOP only when this is requested by an algorithm. This way, jMetal allows to separate the algorithm-specific part from the application-specific part, as it is done in PISA.

In the following, we discuss other features of jMetal that, all together, makes our framework a unique system compared to existing proposals.

To design jMetal we have employed UML (Unified Modeling Language) [11], a tool used by software engineers in software projects. Using UML, it is possible to design and document a software system before it has to be written. This way, we can find bugs in the design phases and modifications in the system can be carried out on UML documentation, which will result in more quality programs. The people intending

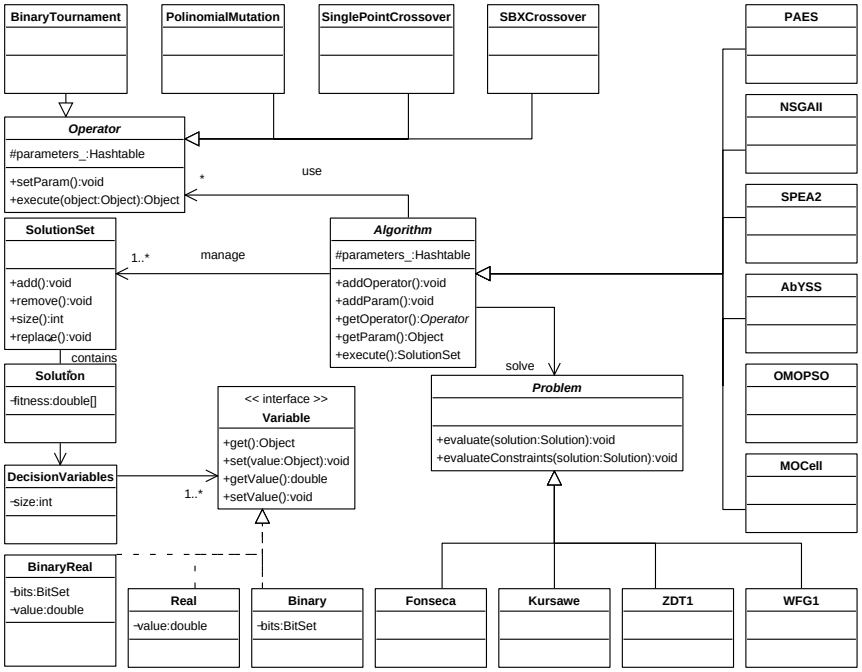

Figure 1: UML class diagram of jMetal

to use the programs can learn about them without having to dive into the source codes, because the UML diagrams ease to understand the software.

jMetal is an open source project, and the software can be obtained freely from the Web in the following URL: `http://neo.lcc.uma.es/metal/index.html`. The codes are maintained and updated by the developers working on it. Currently, the framework is fully functional, and it includes many metaheuristics (NSGA-II, SPEA2, PAES, PESA-II, OMOPSO, AbYSS, MOCell) and most of the MOPs used in comparative studies (the families ZDT and DTLZ, the problems developed using WFG, as well as classical MOPs such as the studies of Schaffer, Kursawe, Fonseca, etc.). It is worth mentioning that the accuracy of the metaheuristics developed under jMetal has been validated by comparing the implementation of some state-of-the-arts algorithms in jMetal against the original implementations of them (see Section 5).

Finally, to fulfill the goal of facilitating the comparison among metaheuristics for solving MOPs, jMetal is complemented with a set of tools which will be available in the jMetal Web site. This tools include utilities for making series of experiments, for applying performance metrics, and for carrying out statistical analyses. The Web site also includes the true Pareto fronts of wide set of MOPs used to assess the performance of the algorithms.

4 Architecture and Implementation

As commented in the previous section, to describe the architecture of jMetal we use UML. A UML class diagram representing the main components and their relationships is depicted in Fig. 1. The diagram is a simplified version in order to make it understandable. The basic architecture of jMetal relies in that an `Algorithm` solves a `Problem` using one (and possibly more) `SolutionSet` and a set of `Operator` objects.

4.1 Description of the Base Classes

Let us comment first the class `Algorithm`. This is the main class of jMetal, and all the metaheuristics inherit from it. An instance object from `Algorithm` will probably require some application-specific parameters, that can be added and accessed using the methods `addParameter()` and `getParameter()`, respectively. Similarly, an algorithm will also use some operators, so there are methods for incorporating them (`addOperator()`)

and to get them (`getOperator()`). The most important method in `Algorithm` is `execute()`, which starts the execution of the algorithm. An example of how these methods are used is given in the next section.

As its name suggest, the class `SolutionSet` represents a set of `Solution` objects, which are composed of `DecisionVariables` object, which contains an array of `Variable` objects. `Variable` is an abstract class (or interface) aimed at defining and using variables implemented having different representations (e.g., real, binary, or binary-coded real, as it is shown in Fig. 1). If we are familiar with evolutionary algorithms, these classes are equivalent to the population, individual, chromosome, and gene components, commonly used in the implementation of these metaheuristics. The proposed scheme is very flexible, because a `DecisionVariables` object is not restricted to contain variables of the same representation; instead, it can be composed of an array of mixed variables. It is also extensible, because new representations can be incorporated easily into the framework, just inheriting from `Variable`.

The class `Problem` contains two basic methods, which are called `evaluate()` and `evaluateConstraints()`. Both methods receive a `Solution` object; the first one evaluates it, while the second one determines the overall constraint violation of the solution. All the problems have to define the `evaluate()` method, while only problems having side constraints have to define `evaluateConstraints()`.

The last base class of jMetal is `Operator`. This class represents a generic operator to be used in the algorithm (e.g., crossover, mutation, or selection). As `Algorithm`, it contains a `parameter_` field, which is used to record the parameters of the operator. As an example, a SBX crossover requires a crossover probability (as in most crossover operators) plus a value for the distribution index (specific of SBX), it receives as parameter two parent individuals and, as a result, it returns the offspring. The use of the generic Java class `Object` allows a high degree of flexibility. Thus, a `BinaryTournament` operator will receive as parameter a `SolutionSet` object and it will return a `Solution` object, while a `PolynomialMutation` operator will receive a `Solution` object and it will return a null value.

Besides the base classes, jMetal provides an utility package (`jmetal.util`) which contains a set of classes providing features useful in many metaheuristics, such as population ranking, the hypercube-based density estimator used in PAES, calculating the strength of the individuals in a solution set, etc. An example of the use of classes within this package is included in next section.

4.2 Case Study: NSGA-II

To illustrate the use of jMetal in a real example, we describe here some details of the implementation of NSGA-II, and how it is configured and used.

Under jMetal, a metaheuristic is composed of a class defining the algorithm itself and another class to execute it. This second class is also used to specify the problem to solve, the operators to use, the parameters of the algorithm, and whatever parameters need to be set. Let us call this two classes `NsgaII` and `NsgaII_Main`, respectively. In Fig. 2 we include an extract of the program `NsgaII_Main.java` (in Java, each class is defined in a file with the same name of the class and the extension `.java`). We have omitted some sentences for error control and package import. We comment the main sentences of the program next.

A MOP is represented by an object of the proper class; in the example, a problem Kursawe is created (line 12). This problem has two parameters, indicating that it has three decision variables which are real-coded (an alternative would be to use binary-coded variables). The metaheuristic is created in line 13, where an instance of NSGA-II is requested. The next two sentences (lines 15 and 16) are used to set parameter values, such as the population and the maximum number of evaluations to compute; we can use similar sentences to set as many parameters as the algorithm needs. Next (lines 18 to 26), three groups of sentences are used to specify the crossover, mutation, and selection operators. An operator can require zero (e.g., selection) or more parameters (e.g., crossover and mutation). Once the operators have been created and configured, they are incorporated to the algorithm (lines 28-30). The sentence (line 32) just starts the execution of the algorithm, which will produce as a result a set of Pareto optimal solutions. The last two lines are used to write into two files the values of the variables and the objectives.

An extract of the implementation of the class `NsgaII` is included in Fig. 3. We concentrate in method `execute`. The first step is to obtain the parameter values and operators (lines 10-15) indicated in the `NsgaII_Main` program. Then, the two populations required by the algorithm are created (lines 17-18), and the initial population is initialized. The main loop of the algorithm starts in line 25. It contains the breeding loop, typical of genetic algorithms: two parents are selected (lines 27-28), a pair of children are obtained after

```
 1. import jmetal.base.*          ;
 2. import jmetal.problems.Kursawe ;
 3.
 4. public class NsgaII_Main {
 5.   public static void main(String[] args) {
 6.     Problem   problem   ;  // The problem to solve
 7.     Algorithm algorithm ;  // The algorithm to use
 8.     Operator  crossover ;  // Crossover operator
 9.     Operator  mutation  ;  // Mutation operator
10.     Operator  selection ;  // Selection operator
11.
12.     problem   = new Kursawe(3, "Real") ; // 3 variables, real representation
13.     algorithm = new NsgaII(problem)     ;
14.
15.     algorithm.setInputParameter("populationSize",100);
16.     algorithm.setInputParameter("maxEvaluations",25000);
17.
18.     crossover = CrossoverFactory.getCrossoverOperator("SBXCrossover");
19.     crossover.setParameter("probability",0.9);
20.     crossover.setParameter("distributionIndex",20.0);
21.
22.     mutation = MutationFactory.getCrossoverOperator("PolinomialMutation");
23.     mutation.setParameter("probability",1.0/problem.getNumberOfVariables());
24.     mutation.setParameter("distributionIndex",20.0);
25.
26.     selection = new BinaryTournament();
27.
28.     algorithm.addOperator("crossover",crossover);
29.     algorithm.addOperator("mutation" ,mutation);
30.     algorithm.addOperator("selection",selection);
31.
32.     SolutionSet paretoFront = algorithm.execute() ;
33.
34.     paretoFront.printVariablesToFile("VAR");
35.     paretoFront.printObjectivesToFile("FUN");
36.   } // main
37. } // NsgaII_Main
```

Figure 2: Main components of program `NsgaII_Main.java`

crossover (line 30), the children are mutated (lines 32-33) and evaluated (lines 35-36), and finally they are inserted into the offspring population (lines 38-39). Next follows the rest of NSGA-II (see Figure 4): the two populations are joined and ranked, and a new population is obtained selecting the best ranked individuals, applying the crowding distance to choose the best ones in the last selected rank.

In Figure 4 we detail the use of two objects belonging to the utility package of jMetal (`jmetal.util`). The first object is an instance of the class `jmetal.util.Distance` (line 2), which includes methods for computing different distance values related to a solution set; we use this object to obtain the crowding distance of the solutions in the solution set representing the ranks of the population (lines 19 and 35). The second object is an instance of the class `jmetal.util.Ranking`, which allows to rank a solution set according to the ranking method of NSGA-II (note that we assign to the first rank a value of 0, while in NSGA-II the first rank is 1). The ranking is performed in line 7, and the subfronts are obtained from the ranking object (lines 14 and 31).

The example of NSGA-II shows that the code of the algorithms in jMetal is very readable and easy to understand, which satisfies the first goal of jMetal (see Section 3). The solution we provide is also flexible. For example, if we want to use a binary representation of the variables, only the `NSGAII_Main` class should be modified, as indicated in Fig. 5. Once the classes have been compiled, the resulting program can be run in any machine having the Java runtime system installed (*portability*). It is worth mentioning here the benefits of code-reuse provided by the object-oriented design of jMetal. Thus, classes such as `jmetal.util.Ranking` can be applied in any other metaheuristic simply using it as it is shown in Figure 4.

5 Analysis of NSGA-II and SPEA2

In this section, we include a comparative study of the metaheuristics NSGA-II and SPEA2, implemented with jMetal, against the reference implementations of them. Our goal is to assess whether the Java versions of these algorithms are competitive, and allowing further comparative studies with jMetal. Because of space

```
 1. package jmetal.metaheuristic.nsgaII ;
 2. import jmetal.base.*           ;
 3. import jmetal.util.* ;
 4.
 5. public class NsgaII extends Algorithm {
 6.   ... // private variables declaration
 7.   public NsgaII(Problem problem) { this.problem = problem ; }
 8.
 9.   public SolutionSet execute() {
10.     populationSize    = ((Integer)getInputParameter("populationSize")).intValue() ;
12.     maxEvaluations    = ((Integer)getInputParameter("maxEvaluations")).intValue() ;
13.     mutation          = operators.get("mutation")  ;
14.     crossover         = operators.get("crossover") ;
15.     selection         = operators.get("selection") ;
16.
17.     population          = new SolutionSet(populationSize) ;
18.     offspringPopulation = new SolutionSet(populationSize) ;
19.     ... // Create and initilize the initial population
20.
21.     Solution[] offSpring ;
22.     Solution[] parent = new Solution[2] ;
23.     evaluations = 0;
24.
25.     while (evaluations < maxEvaluations) { // Main loop
26.       for (int i = 0; i < populationSize/2; i++) {
27.         parent[0] = (Solution)selection.execute(population);
28.         parent[1] = (Solution)selection.execute(population);
29.
30.         offSpring = (Solution []) crossover.execute(parent);
31.
32.         mutation.execute(offSpring[0]);
33.         mutation.execute(offSpring[1]);
34.
35.         problem.evaluate(offSpring[0]);
36.         problem.evaluate(offSpring[1]);
37.
38.         offspringPopulation.addIndividual(offSpring[0]);
39.         offspringPopulation.addIndividual(offSpring[1]);
40.         evaluations += 2 ;
41.       } // for
42.       ... // ranking and crowding phase (see Figure 4)
43.     } // while
44.   } // execute
45. } // NsgaII
```

Figure 3: Implementation of class `NsgaII`

limitations, we do not describe here NSGA-II and SPEA2; they are widely known, and they are detailed in [6] and [14], respectively.

We describe next the methodology we have used to evaluate the algorithms, including the benchmark problems and performance metrics.

5.1 Methodology of the Experiments

Each algorithm is run until 25,000 function evaluations have been computed. We have made 100 independent runs of each experiment, and the results includes the median, $\tilde{x}$, and interquartile range, IQR, as measures of location (or central tendency) and statistical dispersion, respectively. Since we are dealing with stochastic algorithms and we want to provide the results with confidence, the following statistical analysis has been performed in all this work. First, a Kolmogorov-Smirnov test is performed in order to check whether the values of the results follow a normal (gaussian) distribution or not. If so, an ANOVA test is done, otherwise we perform a Kruskal-Wallis test. We always consider in this work a confidence level of 95% (i.e., significance level of 5% or p-value under 0.05) in the statistical tests, which means that the differences are unlikely to have occurred by chance with a probability of 95%. Successful tests are marked with "+" symbols in the last column in all the tables; conversely, "–" means that no statistical confidence was found (p-value > 0.05). The best result for each problem has a grey colored background.

As benchmark, we have chosen the classical bi-objective unconstrained problems problems Schaffer, Fonseca, and Kursawe, as well as the problems ZDT1, ZDT2, ZDT3, ZDT4, and ZDT6. Furthermore, the

```
 1. ...
 2. jmetal.util.Distance distance = new Distance() ; // Creating a distance object
 3.
 4. // The parent and the offspring are joined into a unique population
 5. union = ((SolutionSet)population).union(offspringPopulation) ;
 6.
 7. jmetal.util.Ranking ranking = new Ranking(union) // Creating a Ranking object
 8. population.clear() ; // The parent population is cleared
 9.
10. int index = 0 // index is used to specify a subfront
11. int remainingIndividuals = populationSize ;
12.
13. // The first subfront is obtained
14. SolutionSet front = ranking.getSubfront(index) ;
15.
16. // Copy individuals from the subfronts to the new parent population
17. while ((remainIndividuals > 0) && (remainIndividuals >= front.size())) {
18.   // Assign crowding distance to the individuals
19.   distance.crowdingDistanceAsignament(front, problem_.getNumberOfObjectives());
20.
21.   // Add the individuals of the current front to the next population
22.   for (int k = 0; k < front.size(); k++)
23.     population.add(front.get(k)) ;
24.
25.   // Update remainingIndividuals
26.   remainingIndividuals = remainingIndividuals - front.size() ;
27.
28.   // Obtain the next front
29.   index ++ ;
30.   if (remainingIndividuals > 0)
31.     front = ranking.getSubfront(index) ;
32. } //while
33.
34. if (remainingIndividuals > 0) { // The crowding distance is used to choose the remaining individuals
35.   distance.crowdingDistanceAssignament(front, problem_.getNumberOfObjectives()) ;
36.   front.sort(new jmetal.base.operator.comparator.CrowdingComparator()); // Sorting the subfront
37.   for (int k = 0 ; k < remainingIndividuals; k++) // Adding the remaining individuals
38.     population.add(front.get(k);
39. } // if
40. ...
```

Figure 4: Applying ranking and crowding in the class `NsgaII`

```
        ...
12.     problem   = new Kursawe(3, "BINARY_REAL", bitsPerVariable) ;
        ...
18.     crossover = CrossoverFactory.getCrossoverOperator("SinglePointCrossover");
19.     crossover.setParam("probability",0.9);
        ...
22.     mutation = MutationFactory.getCrossoverOperator("BitFlipMutation");
24.     mutation.setParam("probability",1.0/(3*bitsPerVariable));
        ...
```

Figure 5: Modifications in program `NsgaII_Main.java` to change the variable representation

following constrained bi-objective MOPs have been used: Osyczka2, Tanaka, Srinivas, and Constr_Ex[3]. We do not include their formulation because they are well-known MOPs, which are frequently used in similar studies (see [4] or [5] for details about them). For assessing the performance of the algorithms, we have used two metrics, the Generational Distance metric [13] and the Hypervolume metric [15].

5.2 Algorithms Comparison

The parameter settings in these experiments have been the following: the population size is 100 individuals, the crossover and mutation operators are, respectively, simulated binary and polynomial, the crossover probability is 0.9, the mutation probability is 1/L (where L is the number of decision variables of the MOP), and the distribution indexes for crossover and mutation are 20.

We include the results obtained after applying the generational distance metric in Table 1. The comparison

[3]The SPEA2 implementation provided by PISA does not consider constrained MOPs. We modified that implementation to include the same constraint handling mechanism used by NSGA-II

Table 1: Generational distance metric. Comparison NSGA-II (jMetal) vs NSGA-II (original) – left – and SPEA2 (jMetal) vs SPEA2 (original) – right –

Problem	NSGA-II (jMetal) $\tilde{x}_{IQR}$	NSGA-II (Original) $\tilde{x}_{IQR}$		SPEA2-II (jMetal) $\tilde{x}_{IQR}$	SPEA2-II (Original) $\tilde{x}_{IQR}$	
Schaffer	2.393e-4 $_{1.8e-5}$	2.348e-4 $_{1.7e-5}$	+	2.374e-4 $_{1.8e-5}$	2.374e-4 $_{1.5e-5}$	+
Fonseca	3.481e-4 $_{4.7e-5}$	4.687e-4 $_{5.3e-5}$	+	2.181e-4 $_{2.9e-5}$	2.254e-4 $_{3.3e-5}$	+
Kursawe	2.126e-4 $_{3.4e-5}$	2.055e-4 $_{3.1e-5}$	+	1.586e-4 $_{2.0e-5}$	1.611e-4 $_{1.9e-5}$	–
ZDT1	2.174e-4 $_{4.9e-5}$	2.196e-4 $_{4.8e-5}$	–	2.211e-4 $_{2.8e-5}$	1.981e-4 $_{1.7e-5}$	+
ZDT2	1.666e-4 $_{4.3e-5}$	1.667e-4 $_{4.2e-5}$	–	1.770e-4 $_{4.8e-5}$	1.267e-4 $_{3.0e-5}$	+
ZDT3	2.107e-4 $_{2.0e-5}$	2.165e-4 $_{2.1e-5}$	–	2.320e-4 $_{2.0e-5}$	2.330e-4 $_{2.1e-5}$	–
ZDT4	5.159e-4 $_{3.5e-4}$	4.145e-4 $_{3.5e-4}$	–	5.753e-4 $_{4.4e-4}$	5.743e-2 $_{3.3e-2}$	+
ZDT6	1.034e-3 $_{1.4e-4}$	9.911e-4 $_{9.8e-5}$	+	1.750e-3 $_{2.9e-4}$	8.200e-4 $_{7.3e-5}$	+
ConstrEx	2.951e-4 $_{4.6e-5}$	2.883e-4 $_{4.2e-5}$	+	2.027e-4 $_{2.2e-5}$	2.070e-4 $_{2.3e-5}$	–
Srinivas	1.919e-4 $_{5.5e-5}$	1.883e-4 $_{4.7e-5}$	–	1.115e-4 $_{3.6e-5}$	1.123e-4 $_{2.8e-5}$	–
Osyczka2	1.044e-3 $_{9.2e-5}$	1.049e-3 $_{8.3e-5}$	–	1.472e-3 $_{2.2e-4}$	1.408e-3 $_{8.1e-5}$	–
Tanaka	7.637e-4 $_{1.2e-4}$	1.217e-3 $_{1.2e-4}$	+	6.417e-4 $_{1.2e-4}$	7.225e-4 $_{8.9e-5}$	+

Table 2: Hypervolume metric. Comparison NSGA-II (jMetal) vs NSGA-II (original) – left – and SPEA2 (jMetal) vs SPEA2 (original) – right –

Problem	NSGA-II (jMetal) $\tilde{x}_{IQR}$	NSGA-II (Original) $\tilde{x}_{IQR}$		SPEA2-II (jMetal) $\tilde{x}_{IQR}$	SPEA2-II (Original) $\tilde{x}_{IQR}$	
Schaffer	8.294e-1 $_{2.0e-4}$	8.287e-1 $_{3.2e-4}$	+	8.294e-1 $_{1.7e-4}$	8.296e-1 $_{7.4e-5}$	+
Fonseca	3.081e-1 $_{5.1e-4}$	3.064e-1 $_{6.3e-4}$	+	3.107e-1 $_{3.1e-4}$	3.105e-1 $_{3.2e-4}$	+
Kursawe	3.996e-1 $_{2.8e-4}$	3.997e-1 $_{3.5e-4}$	+	4.009e-1 $_{2.1e-4}$	4.009e-1 $_{2.1e-4}$	–
ZDT1	6.594e-1 $_{3.5e-4}$	6.594e-1 $_{4.5e-4}$	–	6.600e-1 $_{3.5e-4}$	6.601e-1 $_{4.0e-4}$	+
ZDT2	3.261e-1 $_{4.9e-4}$	3.262e-1 $_{4.6e-4}$	–	3.263e-1 $_{7.4e-4}$	3.267e-1 $_{5.1e-4}$	+
ZDT3	5.148e-1 $_{2.1e-4}$	5.148e-1 $_{2.0e-4}$	–	5.141e-1 $_{3.4e-4}$	5.140e-1 $_{4.0e-4}$	–
ZDT4	6.542e-1 $_{4.6e-3}$	6.555e-1 $_{5.5e-3}$	–	6.518e-1 $_{1.0e-2}$	1.068e-1 $_{2.1e-1}$	+
ZDT6	3.883e-1 $_{2.4e-3}$	3.860e-1 $_{1.6e-3}$	+	3.785e-1 $_{4.3e-3}$	3.926e-1 $_{1.1e-3}$	+
ConstrEx	7.743e-1 $_{4.3e-4}$	7.746e-1 $_{3.8e-4}$	+	7.757e-1 $_{2.9e-4}$	7.751e-1 $_{4.7e-4}$	+
Srinivas	5.380e-1 $_{5.0e-4}$	5.383e-1 $_{4.7e-4}$	+	5.400e-1 $_{2.0e-4}$	5.400e-1 $_{2.1e-4}$	–
Osyczka2	7.451e-1 $_{7.8e-3}$	7.454e-1 $_{7.7e-3}$	–	7.308e-1 $_{3.4e-2}$	7.238e-1 $_{2.1e-2}$	+
Tanaka	3.76e-1 $_{4.2e-4}$	3.075e-1 $_{4.7e-4}$	–	3.085e-1 $_{7.5e-4}$	3.088e-1 $_{3.0e-4}$	+

between the two versions of NSGA-II reveals that each implementation is better in six out of the twelve chosen benchmark MOPs, so at a first glance we could conclude that none of them is superior to the other one. However, the key point here is that in six problems we obtain a symbol "–" in the last column, thus indicating that the differences in the results are non significant and they can be due to random noise. If we take a look to the significant results, we observe that the values of the two algorithms are very close, excepting the problems Fonseca and Tanaka, where the values provided by our implementation of NSGA-II are clearly better.

A similar scenario happens when comparing the two implementations of SPEA2: our version is better in seven out of the twelve problems, and in five problems the results have not statistical significance. The results of the two implementations of SPEA2 are even closer than in the case of NSGA-II. Only in three problems the differences are noticeable: the original SPEA2 algorithm is better in problems ZDT1 and ZDT6, where the jMetal version achieves a best value in problem Tanaka.

The results obtained after applying the hypervolume metric are included in Table 2. We observe again many occurrences of the symbol "–" in the table, indicating that we obtain values which are numerically similar to the original versions of the algorithms in many experiments. A pair-wise comparison of the two implementations of the algorithms shows that they provide similar values; the exception is the poor hypervolume value obtained by the original SPEA2 algorithm in problem ZDT4.

We can conclude this section stating that, in the context of the benchmark, metrics, and parameterization of the experiments carried out, our versions of both NSGA-II and SPEA2 are competitive against the reference implementations. The fact that many of the values of the two metrics are quite similar in the compared algorithms indicates that our implementations are very accurate. We want to point out that we do not intent to obtain exactly the same results with jMetal, because to do that we should copy exactly the same original code. Our purpose is to verify that our implementations are accurate and they can be used in future comparisons against other metaheuristics developed on top of jMetal; the obtained results support this claim.

Table 3: Execution times of the metaheuristics (in secs)

Problem	NSGA-II $\tilde{x}_{IQR}$	SPEA2 $\tilde{x}_{IQR}$	AbYSS $\tilde{x}_{IQR}$	PAES $\tilde{x}_{IQR}$	OMOPSO $\tilde{x}_{IQR}$	
Schaffer	$7.226_{5.2e-2}$	$10.604_{2.1e-1}$	$1.393_{6.7e-2}$	$0.142_{3.3e-3}$	$4.367_{1.5e-1}$	+
Fonseca	$4.619_{1.7e-2}$	$17.652_{2.4e-1}$	$1.816_{9.9e-2}$	$1.387_{3.2e-2}$	$4.015_{1.1e-1}$	+
Kursawe	$4.753_{4.2e-2}$	$17.213_{2.3e-1}$	$1.730_{7.3e-2}$	$1.127_{4.3e-2}$	$0.572_{1.9e-2}$	+
ZDT1	$5.016_{3.0e-2}$	$17.316_{3.1e-1}$	$2.983_{1.2e-1}$	$1.241_{7.7e-2}$	$11.643_{8.5e-1}$	+
ZDT2	$5.129_{6.9e-2}$	$15.857_{3.6e-1}$	$2.850_{1.9e-1}$	$1.167_{6.5e-2}$	$8.194_{5.6e-1}$	+
ZDT3	$5.036_{2.5e-2}$	$17.161_{3.8e-1}$	$2.699_{9.2e-2}$	$1.139_{7.4e-2}$	$2.379_{2.5e-1}$	+
ZDT4	$5.496_{1.1e-1}$	$12.067_{4.8e-1}$	$1.900_{1.2e-1}$	$0.791_{4.1e-2}$	$1.552_{4.3e-1}$	+
ZDT6	$5.177_{3.1e-2}$	$12.113_{2.8e-1}$	$1.922_{7.6e-2}$	$1.042_{1.2e-1}$	$3.358_{7.6e-1}$	+
ConstrEx	$4.710_{3.0e-2}$	$15.190_{2.4e-1}$	$1.282_{7.1e-2}$	$1.261_{3.6e-1}$	$0.957_{3.4e-2}$	+
Srinivas	$4.246_{1.2e-1}$	$25.617_{9.9e-1}$	$3.081_{1.5e-1}$	$2.128_{2.8e-2}$	$13.203_{1.6e+0}$	+
Osyczka2	$4.973_{3.9e-2}$	$14.887_{5.0e-1}$	$1.798_{1.7e-1}$	$0.818_{1.2e-1}$	$0.369_{7.3e-2}$	+
Tanaka	$4.950_{4.9e-2}$	$12.858_{2.0e-1}$	$0.749_{1.9e-2}$	$0.420_{1.4e-2}$	$0.249_{4.6e-3}$	+

6 Comparative Study of Metaheuristics for Solving MOPs

We designed jMetal to facilitate the implementation of metaheuristic algorithms for solving MOPs and to allow fair performance comparison. In this section, we choose five metaheuristics developed with jMetal and compare them applying the methodology used in the previous section. The algorithms are three evolutionary algorithms (SPEA2, NSGA-II, and PAES), a particle swarm optimization algorithm (OMOPSO), and a scatter search (AbYSS). We briefly describe PAES, OMOPSO, and AbYSS in the following:

- PAES (Pareto Archived Evolution Strategy) is a metaheuristic proposed by Knowles and Corne [7]. The algorithm is based on a simple (1+1) evolution strategy. To find diverse solutions in the Pareto optimal set, PAES uses an external archive of nondominated solutions, which is also used to decide about the new candidate solutions. An adaptive grid is used as density estimator in the archive. We have implemented a real coded version of PAES, applying a polynomial mutation operator.

- OMOPSO is a particle swarm optimization algorithm for solving MOPs [10]. Its main features include the use of the crowding distance of NSGA-II to filter out leader solutions, the use of mutation operators to accelerate the convergence of the swarm, and the concept of ϵ-dominance to limit the number of solutions produced by the algorithm. We consider here the leader population obtained after the algorithm has finished as its result.

- AbYSS (Archive based hYbrid Scatter Search) is an adaptation of the scatter search metaheuristic to the multi-objective domain [9]. It uses an external archive to maintain the diversity of the found solutions; the archive is similar to the one employed by PAES, but using the crowding distance of NSGA-II instead of PAES's adaptive grid. The algorithm incorporates operators of the evolutionary algorithms domain, including polynomial mutation and simulated binary crossover in the improvement and solution combination methods, respectively.

In the algorithms where they are applicable, we use the following parameter settings. The crossover probability is 0.9, the mutation probability is 1/L (being L the number of decision variables), the distribution indexes for crossover and mutation are 20, and archive length is 100.

In Table 3 we include the execution times obtained by the five metaheuristics when solving the benchmark problems. The programs have been executed in a PC equipped with an Intel Pentium 4, 3 GHz processor and 1 GB of main memory, running Open Suse Linux 10.1 (kernel 2.6.16.13-4 smp); we have used J2SE, build 1.5.0_06-b05. The times reported allow us to determine that PAES is the fastest algorithm: it obtains the best values in eighth out of the twelve experiments. The simplicity of this metaheuristic permits to solve all the problems in a range between 0.142 and 2.128 secs. Although OMOPSO achieves the best times in four problems, it is the more irregular algorithm, requiring very long times in problems ZDT1, ZDT2, and Srinivas. The second fastest algorithm is AbYSS, which requires 3.81 secs in the worst case. The slowest metaheuristics with a great difference is SPEA2, whose computing times are between 10.604 and 25.617 secs.

We study now the values obtained after applying the generational distance metric, which are included in Table 4. OMOPSO provides the best values in five out of the twelve problems; however, this algorithm fails in solving the problem ZDT4 (see the high value of the metric in this problem). PAES also finds difficulties to solve ZDT4. The other algorithms are more robust, and none of them is clearly better than the rest.

These conclusions are also applicable to the results of the Hypervolume metric (see Table 5), where OMOPSO yields the best values in seven out of the twelve considered problems. AbYSS is the second best algorithm, obtaining the best indicators in three problems and the second best values in seven out of the twelve MOPS.

Table 4: Comparison among the five metaheuristics. Generational distance metric.

Problem	NSGA-II $\bar{x}_{IQR}$	SPEA2 $\bar{x}_{IQR}$	AbYSS $\bar{x}_{IQR}$	PAES $\bar{x}_{IQR}$	OMOPSO $\bar{x}_{IQR}$	
Schaffer	2.393e-4 $_{1.8e-5}$	2.374e-4 $_{1.8e-5}$	2.320e-4 $_{1.8e-5}$	2.395e-4 $_{1.9e-5}$	2.330e-4 $_{2.0e-5}$	+
Fonseca	3.481e-4 $_{4.7e-5}$	2.181e-4 $_{2.9e-5}$	2.344e-4 $_{3.1e-5}$	4.704e-4 $_{9.6e-5}$	1.210e-4 $_{9.1e-6}$	+
Kursawe	2.126e-4 $_{3.4e-5}$	1.586e-4 $_{2.0e-5}$	1.541e-4 $_{1.9e-5}$	3.926e-4 $_{1.3e-4}$	4.665e-4 $_{8.1e-5}$	+
ZDT1	2.174e-4 $_{4.9e-5}$	2.211e-4 $_{2.8e-5}$	1.873e-4 $_{3.5e-5}$	2.338e-4 $_{1.4e-4}$	1.335e-4 $_{4.2e-5}$	+
ZDT2	1.666e-4 $_{4.3e-5}$	1.770e-4 $_{4.8e-5}$	1.077e-4 $_{5.3e-5}$	1.950e-4 $_{1.6e-4}$	6.352e-5 $_{1.5e-5}$	+
ZDT3	2.107e-4 $_{2.0e-5}$	2.320e-4 $_{2.0e-5}$	1.948e-4 $_{2.3e-5}$	1.948e-4 $_{1.0e-4}$	2.030e-4 $_{2.7e-5}$	+
ZDT4	5.159e-4 $_{3.5e-4}$	5.753e-4 $_{4.4e-4}$	5.683e-4 $_{5.0e-4}$	4.891e-2 $_{1.4e-1}$	7.692e-1 $_{4.6e-1}$	+
ZDT6	1.034e-3 $_{1.4e-4}$	1.750e-3 $_{2.9e-4}$	5.503e-4 $_{2.1e-5}$	6.865e-4 $_{3.8e-3}$	1.026e-3 $_{3.0e-2}$	+
ConstrEx	2.951e-4 $_{4.6e-5}$	2.027e-4 $_{2.2e-5}$	2.255e-4 $_{3.7e-5}$	3.446e-4 $_{1.2e-4}$	1.516e-4 $_{2.0e-5}$	+
Srinivas	1.919e-4 $_{5.5e-5}$	1.115e-4 $_{3.6e-5}$	5.943e-5 $_{2.4e-5}$	2.847e-4 $_{1.1e-4}$	5.167e-5 $_{3.2e-5}$	+
Osyczka2	1.044e-3 $_{9.2e-5}$	1.472e-3 $_{2.2e-4}$	1.147e-3 $_{1.6e-2}$	7.468e-3 $_{2.0e-2}$	1.788e-3 $_{7.1e-4}$	+
Tanaka	7.637e-4 $_{1.2e-4}$	6.417e-4 $_{1.2e-4}$	7.786e-4 $_{9.2e-5}$	7.626e-4 $_{2.9e-4}$	7.583e-4 $_{9.7e-5}$	+

Table 5: Comparison among the five metaheuristics: Hypervolume metric

Problem	NSGA-II $\bar{x}_{IQR}$	SPEA2 $\bar{x}_{IQR}$	AbYSS $\bar{x}_{IQR}$	PAES $\bar{x}_{IQR}$	OMOPSO $\bar{x}_{IQR}$	
Schaffer	8.294e-1 $_{2.0e-4}$	8.294e-1 $_{1.7e-4}$	8.299e-1 $_{4.8e-5}$	8.280e-1 $_{4.3e-4}$	8.299e-1 $_{4.3e-5}$	+
Fonseca	3.081e-1 $_{5.1e-4}$	3.107e-1 $_{3.1e-4}$	3.104e-1 $_{3.1e-4}$	3.053e-1 $_{1.1e-3}$	3.125e-1 $_{2.7e-5}$	+
Kursawe	3.996e-1 $_{2.8e-4}$	4.009e-1 $_{2.1e-4}$	4.011e-1 $_{2.5e-4}$	3.942e-1 $_{1.9e-3}$	3.978e-1 $_{5.3e-4}$	+
ZDT1	6.594e-1 $_{3.5e-4}$	6.600e-1 $_{3.5e-4}$	6.614e-1 $_{2.8e-4}$	6.573e-1 $_{2.2e-3}$	6.614e-1 $_{3.2e-4}$	+
ZDT2	3.261e-1 $_{4.9e-4}$	3.263e-1 $_{7.4e-4}$	3.282e-1 $_{3.8e-4}$	3.241e-1 $_{1.4e-3}$	3.283e-1 $_{2.9e-4}$	+
ZDT3	5.148e-1 $_{2.1e-4}$	5.141e-1 $_{3.4e-4}$	5.158e-1 $_{3.4e-3}$	5.057e-1 $_{4.9e-2}$	5.148e-1 $_{8.1e-4}$	+
ZDT4	6.542e-1 $_{4.6e-3}$	6.518e-1 $_{1.0e-2}$	6.545e-1 $_{7.1e-3}$	6.403e-1 $_{2.0e-2}$	0.000e+0 $_{0.0e+0}$	+
ZDT6	3.883e-1 $_{2.4e-3}$	3.785e-1 $_{4.3e-3}$	4.004e-1 $_{1.7e-4}$	3.968e-1 $_{2.5e-3}$	4.013e-1 $_{7.0e-5}$	+
ConstrEx	7.743e-1 $_{4.3e-4}$	7.757e-1 $_{2.9e-4}$	7.761e-1 $_{2.6e-4}$	7.718e-1 $_{4.5e-3}$	7.765e-1 $_{2.5e-4}$	+
Srinivas	5.380e-1 $_{5.0e-4}$	5.400e-1 $_{2.0e-4}$	5.407e-1 $_{1.2e-4}$	5.357e-1 $_{1.2e-3}$	5.407e-1 $_{8.8e-5}$	+
Osyczka2	7.451e-1 $_{7.8e-3}$	7.308e-1 $_{3.4e-2}$	7.443e-1 $_{3.5e-1}$	3.771e-1 $_{1.9e-1}$	7.070e-1 $_{1.4e-2}$	+
Tanaka	3.076e-1 $_{4.2e-4}$	3.085e-1 $_{7.5e-4}$	3.076e-1 $_{4.6e-4}$	3.019e-1 $_{2.7e-3}$	3.063e-1 $_{6.5e-4}$	+

Our purpose in this section has been to show how jMetal can be used to make a comparative study among metaheuristics for solving MOPs. Although this study is a preliminary one, it allows us to deduce that reference metaheuristics such as NSGA-II and SPEA2 can be improved by new algorithms. For this reason, there is room for designing new techniques, which have to be deeply analyzed to assess their performance. In this context, jMetal is a valuable software that can be used with advantages against other systems to carry out this kind of studies.

7 Conclusions and Future Work

We have presented jMetal, a Java-based framework for developing metaheuristics for solving multi-objective optimization problems. The main goal driving this work is to provide a tool specifically designed to implement this kind of algorithms, which can also be used to carry out fair comparative studies among different techniques. We have described the architecture of jMetal and the implementation of NSGA-II has been used as a case study.

We have compared the original implementations of two state-of-the-art metaheuristics for solving MOPs, NSGA-II and SPEA2, against their versions developed under jMetal with the idea in mind of testing the accuracy of our software. The results of a set of experiments using twelve bi-objective MOPs reveal that the jMetal versions of these algorithms are competitive, and in many cases they provide very similar values to the ones obtained by the original algorithms. As an example of the use of jMetal to analyze different metaheuristics, we have included a comparative study of three evolutionary algorithms, a scatter search, and a particle swarm optimization algorithm.

As a line of future work, we intend to use jMetal to carry out a deeper comparison among metaheuristics for solving MOPs, analyzing more algorithms and a wider set of problems, and applying additional performance metrics.

References

[1] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA - A Platform and Programming Language Independent Interface for Search Algorithms. In *Conference on Evolutionary Multi-Criterion Optimization (EMO 2003)*, pages 494–508, 2003.

[2] Christian Cagné and Marc Parizeau. Genericity in evolutionary computation software tool: Principles and case-study. *International Journal on Artificial Intelligence Tools*, 15(2):173–194, 2006.

[3] Sebastien Cahon, El-Ghazali Talbi, and Nordine Melab. Paradiseo: A framework for parallel and distributed metaheuristics. In *nternational Parallel and Distributed Processing Symposium (IPDPS'03)*, page 144a, 2003.

[4] C.A. Coello, D.A. Van Veldhuizen, and G.B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Genetic Algorithms and Evolutionary Computation. Kluwer Academic Publishers, 2002.

[5] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

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

[7] J. Knowles and D. Corne. The pareto archived evolution strategy: A new baseline algorithm for multiobjective optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation*, pages 9–105, Piscataway, NJ, 1999. IEEE Press.

[8] Antonio J. Nebro, Francisco Luna, and Enrique Alba. New ideas in applying scatter search to multiobjective optimization. In C.A. Coello, A. Hernández, and E. Zitler, editors, *Third International Conference on Evolutionary MultiCriterion Optimization, EMO 2005*, volume 3410 of *Lecture Notes in Computer Science*, pages 443–458. Springer, 2005.

[9] Antonio J. Nebro, Francisco Luna, Enrique Alba, Andreas Beham, and Bernabé Dorronsoro. AbYSS: Adapting Scatter Search for Multiobjective Optimization. Technical Report ITI-2006-2, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos, 2006.

[10] Margarita Reyes and Carlos A. Coello Coello. Improving pso-based multi-objective optimization using crowding, mutation and ϵ-dominance. In C.A. Coello, A. Hernández, and E. Zitler, editors, *Third International Conference on Evolutionary MultiCriterion Optimization, EMO 2005*, volume 3410 of *LNCS*, pages 509–519. Springer, 2005.

[11] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional, second edition edition, 2004.

[12] Felix Streichert and Holger Ulmer. Javaeva: A java based framework for evolutionary algorithms. Technical Report WSI-2005-06, Centre for Bioinformatics Tübingen (ZBIT) of the Eberhard-Karls-University, Tübingen, 2005.

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

[14] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 2001.

[15] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.