

Beginner's Guide to the MOEA Framework

David Hadka

Version 2.9

This document is a preview of the Beginner's Guide to the MOEA Framework. Some pages have been removed. Use the link below to purchase the complete guide.

[Online Store](#)

Copyright 2011-2015 David Hadka. All Rights Reserved.

Thank you for purchasing this book! All revenue received from book sales helps fund the continued development and maintenance of the MOEA Framework. We sincerely appreciate your support and hope you find this software useful in your endeavors.

Contents

1	Background	1
1.1	Multiobjective Problem	3
1.2	Pareto Optimality	4
1.3	Multiobjective Evolutionary Algorithms	5
1.4	Measuring Quality	8
1.5	The MOEA Framework	14
1.6	Getting Help	15
2	Setup and First Example	17
2.1	Installing Java	17
2.2	Installing Eclipse	17
2.3	Setting up the MOEA Framework	17
2.4	Your First Example	20
2.5	Running from Command Line	26
2.6	Plotting Results	26
3	Constrained Optimization	29
3.1	Constrained Optimization Example	30
3.2	The Knapsack Problem	33
3.3	Feasibility	39
4	Choice of Optimization Algorithm	41
4.1	Running Different Algorithms	41
4.2	Parameterization	43
4.3	Comparing Algorithms	47
4.4	Runtime Dynamics	52
5	Customizing Algorithms	57
5.1	Manually Running Algorithms	57
5.2	Custom Initialization	59
5.3	Custom Algorithms	62
5.4	Creating Service Providers	64
5.5	Hyperheuristics	67

5.6	Custom Types and Operators	70
5.7	Learning the API	76
6	The Diagnostic Tool	77
6.1	Using the Diagnostic Tool	77
6.2	Adding Custom Algorithms	82
6.3	Adding Custom Problems	86
7	Subsets, Permutations, and Programs	89
7.1	Subsets	89
7.2	Permutations	92
7.3	Programs	95
7.3.1	Type-based Rule System	95
7.3.2	Defining the Problem	98
7.3.3	Custom Operators	103
7.3.4	Ant Problem	104
8	Integers and Mixed Integer Programming	107
8.1	Two Representations	107
8.2	Mixed Integer Programming	111
9	I/O Basics	115
9.1	Printing Solutions	115
9.2	Files	117
9.3	Checkpoints	118
9.4	Creating Reference Sets	119
10	Performance Enhancements	123
10.1	Multithreading	123
10.2	Termination Conditions	128
10.3	Native Compilation	128
10.4	Standard I/O	131
10.5	A Note on Concurrency	134
A	List of Algorithms	135
B	List of Variation Operators	143
B.1	Real-Valued Operators	144
B.2	Binary / Bit String Operators	148
B.3	Permutations	149
B.4	Subsets	149
B.5	Grammars	150
B.6	Program Tree	150
B.7	Generic Operators	150

C List of Problems	153
Bibliography	167

Chapter 1

Background

Optimization is the process of identifying the best solution among a set of alternatives (Miettinen, 1999). Whereas single objective optimization employs a single criterion for identifying the best solution among a set of alternatives, multiobjective optimization employs two or more criteria. The criteria used to compare solutions are known as *objectives*. As multiple objectives can conflict with one another — i.e., improving one objective leads to the deterioration of another — there is, generally speaking, no single optimal solution to multiobjective problems.

As an example, Figure 1.1 shows the tradeoff between two objectives: (1) cost and (2) error. The shaded region depicts the set of candidate solutions to this hypothetical problem. The top-left region contains low cost, high error candidate solutions. The bottom-right region contains high cost, low error candidate solutions. Between these two extremes lie the various degrees of tradeoff between the two objectives, where increases in cost lead to reduced error.

Figure 1.1 demonstrates a fundamental issue in multiobjective optimization. Given that there is no single optimal solution, rather a multitude of potential solutions with varying degrees of tradeoff between the objectives, decision-makers are subsequently responsible for exploring this set of potential solutions and identifying the solution(s) to be implemented. While ultimately the selection of the final solution is the responsibility of the decision-maker, optimization tools should assist this decision process to the best of their ability. For instance, it may prove useful to identify points of diminishing returns. For example, Figure 1.2 identifies the region where a large increase in cost is necessary to impart a marginal decrease in error. To perform this type of analysis, it is necessary to provide the decision-maker with an enumeration or approximation of these tradeoffs. This strategy of enumerating or approximating the tradeoffs is known as *a posteriori* optimization (Coello Coello et al., 2007), and is the focus of this book.

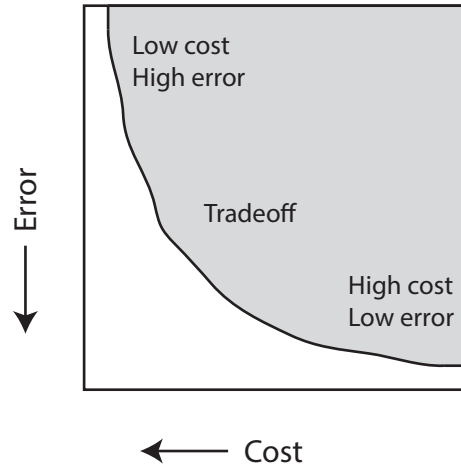


Figure 1.1: Example of the tradeoff between two objectives: (1) cost and (2) error. A tradeoff is formed between these two conflicting objectives where increases in cost lead to reduced error. All figures in this dissertation showing objectives include arrows pointing towards the ideal optimum.

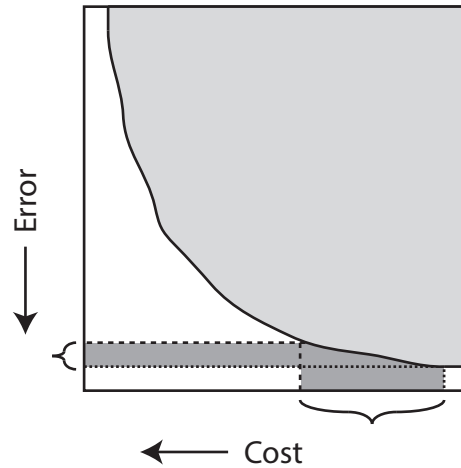


Figure 1.2: Example showing the effect of diminishing returns, where a large increase in cost is necessary to impart a marginal reduction in error.

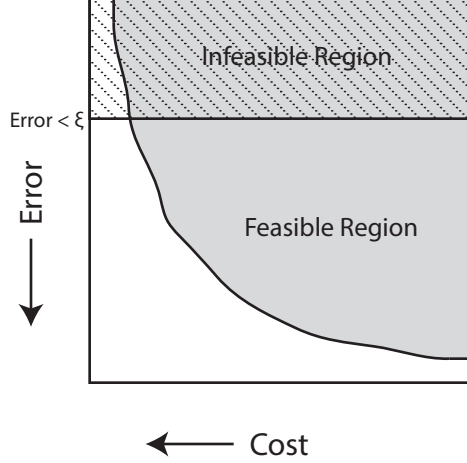


Figure 1.3: Example showing how constraints define an infeasible region (the hashed region). Valid solutions to the optimization problem are only found in the feasible region.

1.1 Multiobjective Problem

We can express the idea of a multiobjective problem (MOP) with M objectives formally as:

$$\begin{aligned} & \underset{\mathbf{x} \in \Omega}{\text{minimize}} && F(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_M(\mathbf{x})) \\ & \text{subject to} && c_i(\mathbf{x}) = 0, \forall i \in \mathcal{E}, \\ & && c_j(\mathbf{x}) \leq 0, \forall j \in \mathcal{I}. \end{aligned} \tag{1.1}$$

We call \mathbf{x} the *decision variables*, which is the vector of variables that are manipulated during the optimization process:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_L \end{bmatrix} \tag{1.2}$$

Decision variables can be defined in a variety of ways, but it is common to see the following types (Bäck et al., 1997):

- **Real-Valued:** 2.71
- **Binary:** 001100010010100001011110101101110011
- **Permutation:** 4, 2, 0, 1, 3

In some applications, it is possible for the number of decision variables, L , to not be a fixed value. In this book, however, we assume that L is constant for a given problem.

The decision space, Ω , is the set of all decision variables. The MOP may impose restrictions on the decision space, called *constraints*. As an example, in Figure 1.3, a hypothetical

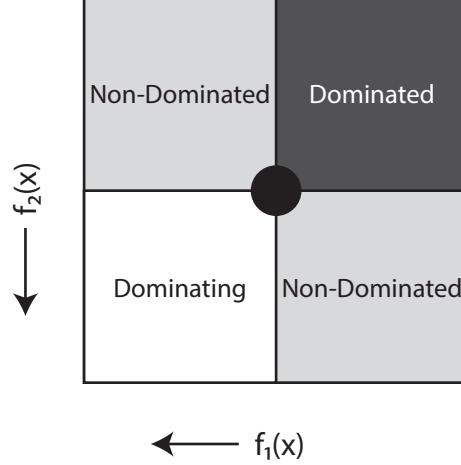


Figure 1.4: Depiction of the various Pareto dominance regions. These regions are relative to each solution, which is centered in the figure. The *dominated* region is inferior in all objectives, the *dominating* region is superior in all objectives and the *non-dominated* region is superior in one objective but inferior in the other.

constraint would prevent any solutions from exceeding an error threshold. In this manner, constraints inform the optimization process as to which solutions are infeasible or impractical. Equation (1.1) shows that zero or more constraints $c_i(\mathbf{x})$ can be defined to express both equality and inequality constraints. The sets \mathcal{E} and \mathcal{I} define whether the constraint is an equality or inequality constraint. The set of all decision variables in Ω which are feasible (i.e., satisfy all constraints) define the *feasible region*, Λ .

1.2 Pareto Optimality

The notion of optimality used today is adopted from the work of Francis Ysidro Edgeworth and Vilfredo Pareto (Coello Coello et al., 2007), and is commonly referred to as *Pareto optimality*. Pareto optimality considers solutions to be superior or inferior to another solution only when it is superior in all objectives or inferior in all objectives, respectively. The tradeoffs in an MOP are captured by solutions which are superior in some objectives but inferior in others. Such pairs of solutions which are both superior and inferior with respect to certain objectives are called *non-dominated*, as shown in Figure 1.4. More formally, the notion of Pareto optimality is defined by the Pareto dominance relation:

Definition 1 A vector $\mathbf{u} = (u_1, u_2, \dots, u_M)$ **Pareto dominates** another vector $\mathbf{v} = (v_1, v_2, \dots, v_M)$ if and only if $\forall i \in \{1, 2, \dots, M\}, u_i \leq v_i$ and $\exists j \in \{1, 2, \dots, M\}, u_j < v_j$. This is denoted by $\mathbf{u} \prec \mathbf{v}$.

Two solutions are non-dominated if neither Pareto dominates the other (i.e., $\mathbf{u} \not\prec \mathbf{v}$ and $\mathbf{v} \not\prec \mathbf{u}$). The set of all non-dominated solutions is captured by the Pareto optimal set and

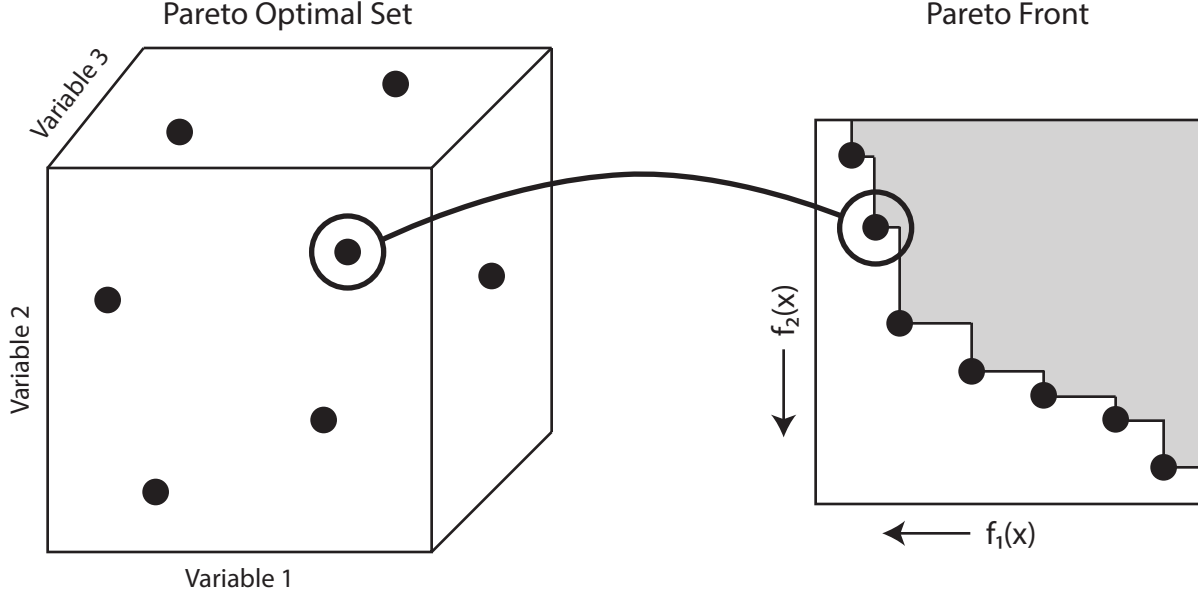


Figure 1.5: Shows a hypothetical mapping between a 3-dimensional Pareto optimal set and its associated 2-dimensional Pareto front. The shaded region in the Pareto front shows the space dominated by the Pareto front.

the Pareto front. The former contains the decision variables while the latter contains the objectives.

Definition 2 For a given multiobjective problem, the **Pareto optimal set** is defined by

$$\mathcal{P}^* = \{\mathbf{x} \in \Omega \mid \neg \exists \mathbf{x}' \in \Lambda, F(\mathbf{x}') \prec F(\mathbf{x})\}$$

Definition 3 For a given multiobjective problem with Pareto optimal set \mathcal{P}^* , the **Pareto front** is defined by

$$\mathcal{PF}^* = \{F(\mathbf{x}) \mid \mathbf{x} \in \mathcal{P}^*\}$$

In this dissertation, the Pareto dominance relation is applied to the objectives. For convenience, we use $\mathbf{x} \prec \mathbf{y}$ interchangeably with $F(\mathbf{x}) \prec F(\mathbf{y})$.

Figure 1.5 shows an example Pareto optimal set and Pareto front, and the resulting mapping between the two. The Pareto optimal set defines the decision variables, whereas the Pareto front captures the objectives and their tradeoffs via Pareto optimality.

1.3 Multiobjective Evolutionary Algorithms

Evolutionary algorithms (EAs) are a class of search and optimization algorithms inspired by processes of natural evolution (Holland, 1975). A broad overview of the design and development of EAs is provided in Bäck et al. (1997). The outline of a simple EA is shown

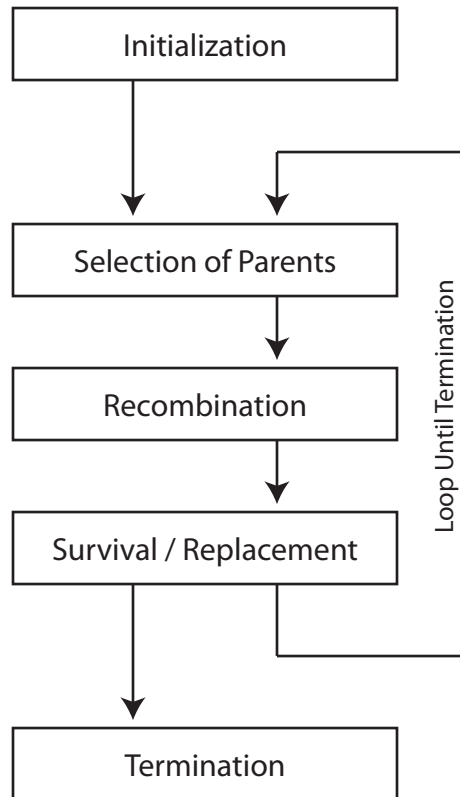


Figure 1.6: The outline of a simple EA. EAs begin with an initialization process, where the initial search population is generated. They next enter a loop of selecting parent individuals from the search population, applying a recombination operator (such as crossover and mutation in genetic algorithms) to generate offspring, and finally updating the search population with these offspring using a replacement strategy. This loop is repeated until some termination condition is met, usually after a fixed number of objective function evaluations (NFE). Upon termination, the EA reports the set of optimal solutions discovered during search.

in Figure 1.6. EAs begin with an initialization process, where the initial search population is generated. They next enter a loop of selecting parent individuals from the search population, applying a recombination operator to generate offspring, and finally updating the search population with these offspring using a replacement strategy. This loop is repeated until some termination condition is met, usually after a fixed number of objective function evaluations (NFE). Upon termination, the EA reports the set of optimal solutions discovered during search.

The behavior of the selection, recombination and survival/replacement processes typically depend on the “class” of EA. For instance, genetic algorithms (GAs) apply crossover and mutation operators that mimic genetic reproduction via DNA (Holland, 1975). Particle swarm optimization (PSO) algorithms simulate flocking behavior, where the direction of travel of each individual is steered towards the direction of travel of surrounding individuals (Kennedy and Eberhart, 1995). While the behavior of each class may be vastly different, they all share a common attribute of utilizing a search population.

Their ability to maintain a population of diverse solutions makes EAs a natural choice for solving MOPs. Early attempts at solving MOPs involved using aggregation-based approaches (Bäck et al., 1997). In aggregation-based approaches, the decision-maker defines an aggregate fitness function that transforms the MOP into a single objective problem, which can subsequently be solved with an EA. Two commonly-used aggregate fitness functions are linear weighting:

$$F_L(\mathbf{x}) = \sum_{i=1}^M \lambda_i f_i(\mathbf{x}), \quad (1.3)$$

and the weighted Chebyshev method:

$$F_T(\mathbf{x}) = \max_{i=1,2,\dots,M} (\lambda_i |z_i^* - f_i(\mathbf{x})|), \quad (1.4)$$

where $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_M)$ are the weights and $\mathbf{z}^* = (z_1^*, z_2^*, \dots, z_M^*)$ is a reference point identifying the decision-maker’s goal (note: this reference point need not be a feasible solution).

Coello Coello et al. (2007) discusses the advantages and disadvantages of aggregate fitness approaches. The primary advantage is the simplicity of the approach and the ability to exploit existing EAs to solve MOPs. In addition, appropriately defined aggregate fitness functions can provide very good approximations of the Pareto front. However, poorly-weighted aggregate fitness functions may be unable to find non-convex regions of the Pareto front. This is problematic since selecting appropriate weights is non-trivial, especially if the relative worth of each objective is unknown or difficult to quantify. Lastly, in order to generate multiple Pareto optimal solutions, aggregate fitness approaches need to be run with differing weights to generate solutions across the entire Pareto front.

These limitations lead to the development of multiobjective evolutionary algorithms (MOEAs) that search for multiple Pareto optimal solutions in a single run. The first MOEA to search for multiple Pareto optimal solutions, the Vector Evaluated Genetic Algorithm (VEGA), was introduced by Schaffer (1984). VEGA was found to have problems similar

to aggregation-based approaches, such as an inability to generate concave regions of the Pareto front. Goldberg (1989) was first to suggest the use of Pareto-based selection, but this concept was not applied until 1993 in the Multiobjective Genetic Algorithm (MOGA) (Fonseca and Fleming, 1993). Between 1993 and 2003, several *first-generation* MOEAs were introduced demonstrating important design concepts such as elitism, diversity maintenance and external archiving. Notable first-generation algorithms include the Niche-Pareto Genetic Algorithm (NPGA) (Horn and Nafpliotis, 1993), the Non-dominated Sorting Genetic Algorithm (NSGA) (Srinivas and Deb, 1994), the Strength Pareto Evolutionary Algorithm (SPEA) (Zitzler and Thiele, 1999), the Pareto-Envelope based Selection Algorithm (PESA) (Corne and Knowles, 2000) and the Pareto Archived Evolution Strategy (PAES) (Knowles and Corne, 1999). Many of these MOEAs have since been revised to incorporate more efficient algorithms and improved design concepts. To date, Pareto-based approaches outnumber aggregate fitness approaches (Coello Coello et al., 2007). For a more comprehensive overview of the historical development of MOEAs, please refer to the text by Coello Coello et al. (2007).

1.4 Measuring Quality

When running MOEAs on a MOP, the MOEA outputs an approximation of the Pareto optimal set and Pareto front. The approximation of the Pareto front, called the *approximation set*, can be used to measure the quality of an MOEA on a particular problem. In some situations, such as with contrived test problems, a *reference set* of the globally optimal solutions may be known. If known, the reference set can be used to measure the absolute performance of an MOEA. If not known, the approximation sets from multiple MOEAs can be compared to determine their relative quality.

There is no consensus in the literature of the appropriate procedure with which to compare approximation sets. These procedures, called *performance metrics*, come in two forms: (1) unary and (2) binary performance metrics (Zitzler et al., 2002c). Unary performance metrics produce a single numeric value with which to compare approximation sets. Unary performance metrics have the advantage of permitting the comparison of approximation sets without requiring the actual approximation set, as one need only compare the numeric values. Binary performance metrics, on the other hand, compare pairs of approximation sets, identifying which of the two approximation sets is superior. In order to allow comparisons across studies, this book uses only unary performance metrics.

Zitzler et al. (2002b) contend that the number of unary performance metrics required to determine if one approximation set is preferred over another must be at least the number of objectives in the problem. Because different MOEAs tend to perform better in different metrics (Bosman and Thierens, 2003), Deb and Jain (2002) suggest only using metrics for the two main functional objectives of MOEAs: proximity and diversity. The following outlines several of the commonly-used unary performance metrics. For details of these performance metrics see Coello Coello et al. (2007).

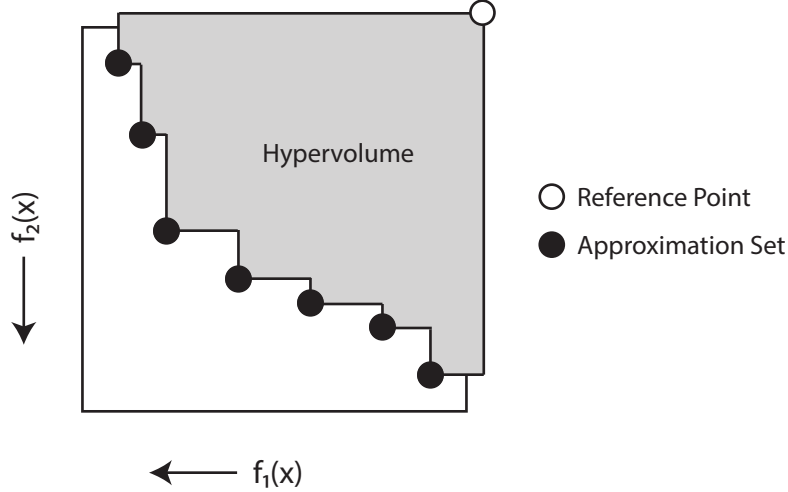


Figure 1.7: Hypervolume measures the volume of the space dominated by the approximation set, bounded by a reference point. This reference point is typically the nadir point (i.e., the worst-case value for each objective) of the reference set plus some fixed delta. This delta ensures extremal points contribute non-zero hypervolume.

Hypervolume As shown in Figure 1.7, the hypervolume metric computes the volume of the space dominated by the approximation set. This volume is bounded by a reference point, which is usually set by finding the nadir point (i.e., the worst-case objective value for each objective) of the reference set plus some fixed increment. This fixed increment is necessary to allow the extremal points in the approximation set to contribute to the hypervolume. Knowles and Corne (2002) suggest the hypervolume metric because it is compatible with the outperformance relations, scale independent, intuitive, and can reflect the degree of outperformance between two approximation sets.

The major disadvantage of the hypervolume metric is its runtime complexity of $O(n^{M-1})$, where n is the size of the non-dominated set. However, Beume and Rudolph (2006) provide an implementation with runtime $O(n \log n + n^{M/2})$ based on the Klee’s measure algorithm by Overmars and Yap. This implementation permits computing the hypervolume metric on moderately sized non-dominated sets up to $M = 8$ objectives in a reasonable amount of time. Further improvements by While et al. (2012) improve the expected runtime further, allowing the efficient calculation of hypervolume with ten or more objectives.

Generational Distance Generational distance (GD) is the average distance from every solution in the approximation set to the nearest solution in the reference set, as shown in Figure 1.8. As such, it measures proximity to the reference set. GD by itself can be misleading, as an approximation set containing a single solution in close proximity to the reference set produces low GD measurements, and is often combined with diversity measures in practice (Hadka and Reed, 2012).

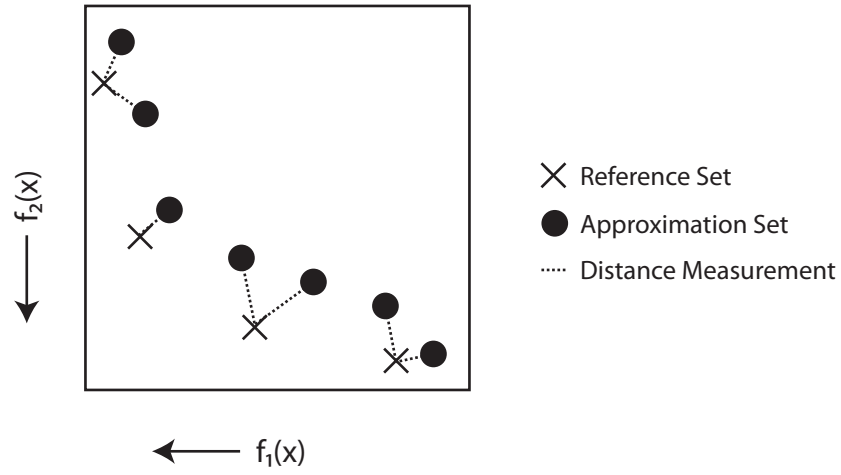


Figure 1.8: Generational distance is the average distance from every solution in the approximation set to the nearest solution in the reference set.

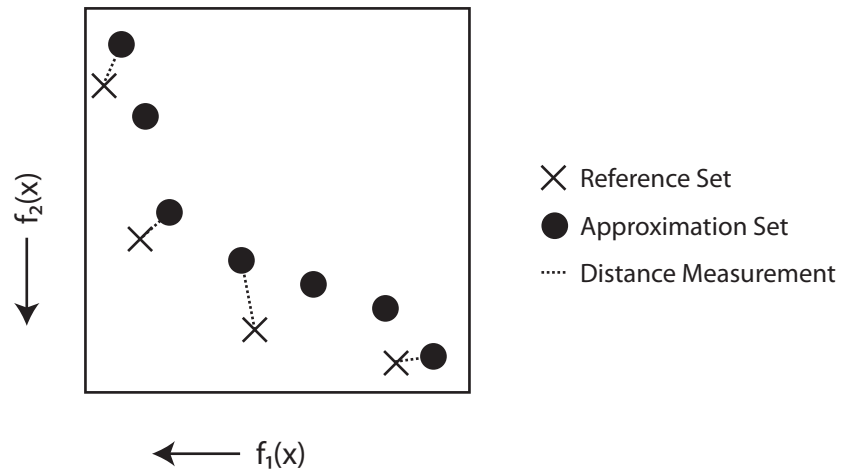


Figure 1.9: Inverted generational distance is the average distance from every solution in the reference set to the nearest solution in the approximation set.

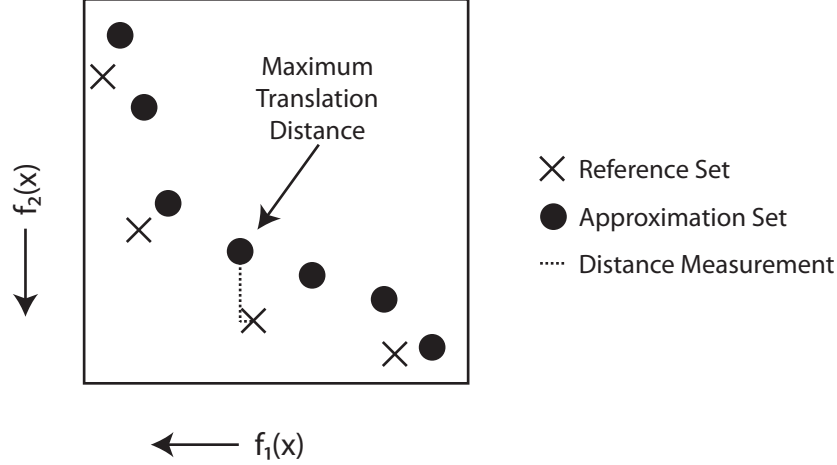


Figure 1.10: ϵ_+ -indicator (also known as the additive ϵ -indicator) is the smallest distance ϵ that the approximation set must be translated by in order to completely dominate the reference set (Coello Coello et al., 2007).

Inverted Generational Distance As its name indicates, the inverted generational distance (IGD) is the inverse of GD — it is the average distance from every solution in the reference set to the nearest solution in the approximation set. IGD measures diversity, as shown in Figure 1.9, since an approximation set is required to have solutions near each reference set point in order to achieve low IGD measurements (Coello Coello et al., 2007).

ϵ_+ -Indicator The additive ϵ -indicator (ϵ_+ -indicator) measures the smallest distance ϵ that the approximation set must be translated by in order to completely dominate the reference set, as shown in Figure 1.10. One observes that good proximity and good diversity both result in low ϵ values, as the distance that the approximation needs to be translated is reduced. However, if there is a region of the reference set that is poorly approximated by the solutions in the approximation set, a large ϵ is required. Therefore, we claim the ϵ_+ -indicator measures the *consistency* of an approximation set (Hadka and Reed, 2013). An approximation set must be free from large gaps or regions of poor approximation in order to be consistent.

Spacing Spacing, shown in Figure 1.11, measures the uniformity of the spacing between solutions in an approximation set (Coello Coello et al., 2007). An approximation set that is well-spaced will not contain dense clusters of solutions separated by large empty expanses. Note that, since spacing does not involve a reference set in its calculation, an approximation can register good spacing while having poor proximity to the reference set. It is therefore recommended to use spacing in conjunction with a performance metric for proximity.

In academic works, it is common to see results published using GD, hypervolume and ϵ_+ -indicator. These three metrics record proximity, diversity and consistency, respectively,

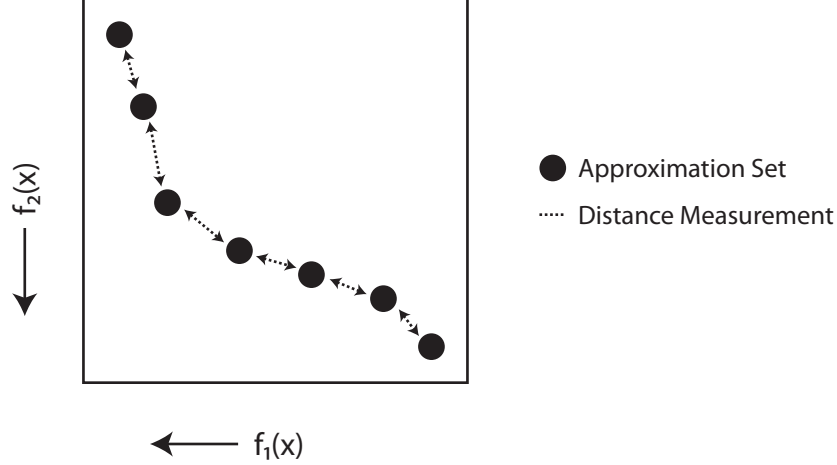


Figure 1.11: Spacing measures the uniformity of the spacing between solutions in an approximation set.

which we claim are the three main functional objectives of MOEAs (Fonseca and Fleming, 1996). Figure 1.12 provides a graphical representation of the importance of the ϵ_+ -indicator and consistency. MOEAs are expected to produce high-quality solutions covering the entire extent of the tradeoff surface, with few gaps or regions of poor approximation.

In order to report these performance metrics consistently, all performance metrics are normalized. This normalization converts all performance metrics to reside in the range $[0, 1]$, with 1 representing the optimal value. First, the reference set is normalized by its minimum and maximum bounds so that all points in the reference set lie in $[0, 1]^N$, the N -dimensional unit hypercube. Second, each approximation set is normalized using the same bounds. Third, the performance metrics are calculated using these normalized sets. Finally, the performance metrics are transformed by the following equations to ensure a value of 1 represents the optimal value achievable by the metric. Hypervolume is transformed with:

$$\mathcal{M}(A_p^s) = \widehat{\mathcal{M}}(A_p^s) / \mathcal{M}^*, \quad (1.5)$$

where $\widehat{\mathcal{M}}$ represents the raw metric value. GD and the ϵ_+ -indicator are transformed with:

$$\mathcal{M}(A_p^s) = \max(1 - \widehat{\mathcal{M}}(A_p^s), 0). \quad (1.6)$$

When solving test problems, the reference set is known analytically. For most real-world problems, however, the reference set is not available. In these situations, it is often necessary to construct a reference set from the union of all approximation sets generated during experimentation. Then, performance metrics can be evaluated relative to this combined reference set.

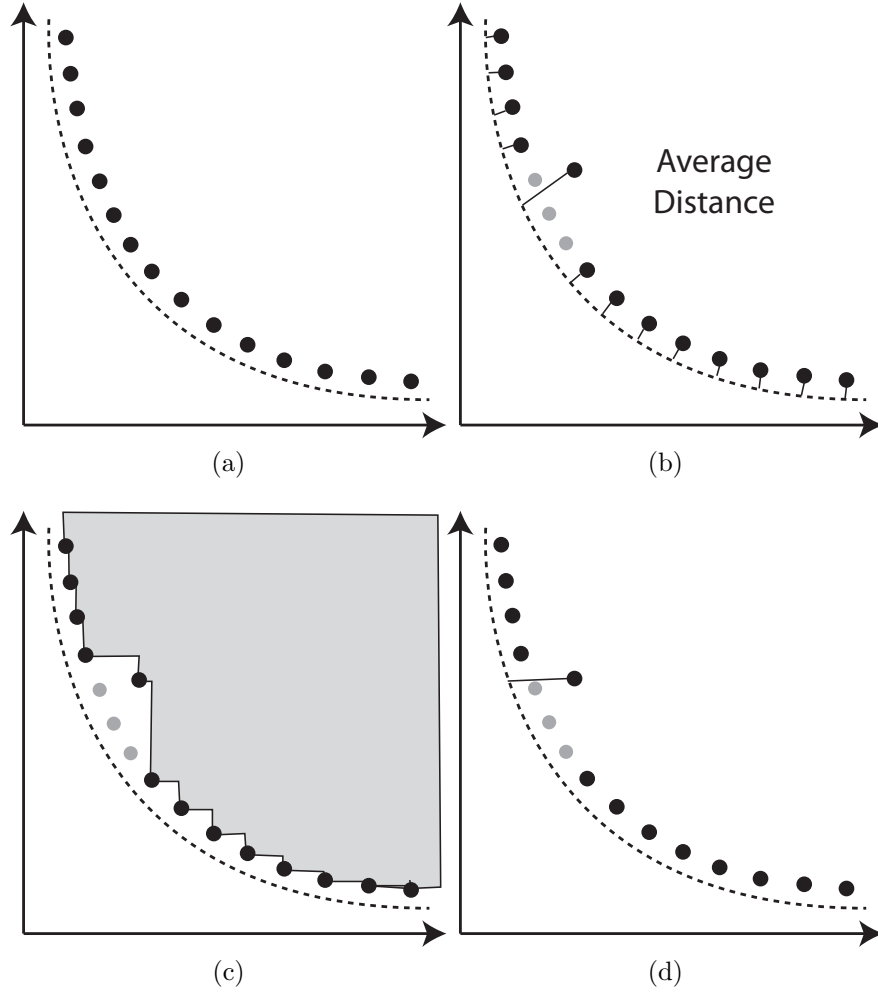


Figure 1.12: Demonstrates the importance of ϵ -indicator as a measure of consistency. (a) A good approximation set to the reference set, indicated by the dashed line. (b) Generational distance averages the distance between the approximation set and reference set, reducing the impact of large gaps. The missing points are shaded light gray. (c) The change in hypervolume due to a gap is small relative to the entire hypervolume. (d) ϵ -Indicator easily identifies the gap, reporting a metric 2-3 times worse in this example.

1.5 The MOEA Framework

The MOEA Framework is a free and open source Java library for developing and experimenting with multiobjective evolutionary algorithms (MOEAs) and other general-purpose optimization algorithms. We will be using the MOEA Framework throughout this book to explore multiobjective optimization. Its key features includes:

Fast, reliable implementations of many state-of-the-art multiobjective evolutionary algorithms. The MOEA Framework contains internally NSGA-II, NSGA-III, ϵ -MOEA, ϵ -NSGA-II, PAES, PESA2, SPEA2, IBEA, SMS-EMOA, GDE3, SMPSO, OMOPSO, CMA-ES, and MOEA/D. These algorithms are optimized for performance, making them readily available for high performance applications. By also supporting the JMetal and PISA libraries, the MOEA Framework provides access to 30 multiobjective optimization algorithms.

Extensible with custom algorithms, problems and operators. The MOEA Framework provides a base set of algorithms, test problems and search operators, but can also be easily extended to include additional components. Using a Service Provider Interface (SPI), new algorithms and problems are seamlessly integrated within the MOEA Framework.

Modular design for constructing new optimization algorithms from existing components. The well-structured, object-oriented design of the MOEA Framework library allows combining existing components to construct new optimization algorithms. And if needed functionality is not available in the MOEA Framework, you can always extend an existing class or add new classes to support any desired feature.

Permissive open source license. The MOEA Framework is licensed under the free and open GNU Lesser General Public License, version 3 or (at your option) any later version. This allows end users to study, modify, and distribute the MOEA Framework freely.

Fully documented source code. The source code is fully documented and is frequently updated to remain consistent with any changes. Furthermore, an extensive user manual is provided detailing the use of the MOEA Framework in detail.

Extensive support available online. As an actively maintained project, bug fixes and new features are constantly added. We are constantly striving to improve this product. To aid this process, our website provides the tools to report bugs, request new features, or get answers to your questions.

Over 1200 test cases to ensure validity. Every release of the MOEA Framework undergoes extensive testing and quality control checks. And, if any bugs are discovered that survive this testing, we will promptly fix the issues and release patches.

1.6 Getting Help

This beginner's guide is the most comprehensive resource for learning about the MOEA Framework. Additional resources are available on our website at <http://www.moeaframework.org>. This website also has links to file bugs or request new features. If you still can not find an answer to your question, feel free to contact us at support@moeaframework.org.

This document is a preview of the Beginner's Guide to the MOEA Framework. Some pages have been removed. Use the link below to purchase the complete guide.

[Online Store](#)

Chapter 2


Setup and First Example

In this chapter, we will setup the MOEA Framework on your computer and demonstrate a simple example. These instructions are tailored for Windows users, but similar steps can be taken to install the MOEA Framework on Linux or Mac OS.

2.1 Installing Java

The MOEA Framework runs on Java version 6 or any later version. Since we will need to compile examples, you will need to install the Java Development Kit (JDK) for version 6 or later. For Windows users, we recommend using Oracle's JDK available at <http://www.oracle.com/technetwork/java/javase/>. Make sure you download the JDK and not the JRE (Java Runtime Environment).

2.2 Installing Eclipse

If this is your first time using the MOEA Framework, we suggest using Eclipse to build projects. Eclipse is a free development environment for writing, debugging, testing, and running Java programs. First, download Eclipse from <http://www.eclipse.org/>. To install Eclipse, simply extract the ZIP archive to a location on your computer (e.g., your desktop). Within the extracted folder, run  `eclipse.exe`. First-time users of Eclipse may be prompted to select a workspace location. The default location is typically fine. Click the checkbox to no longer show this dialog and click Ok.

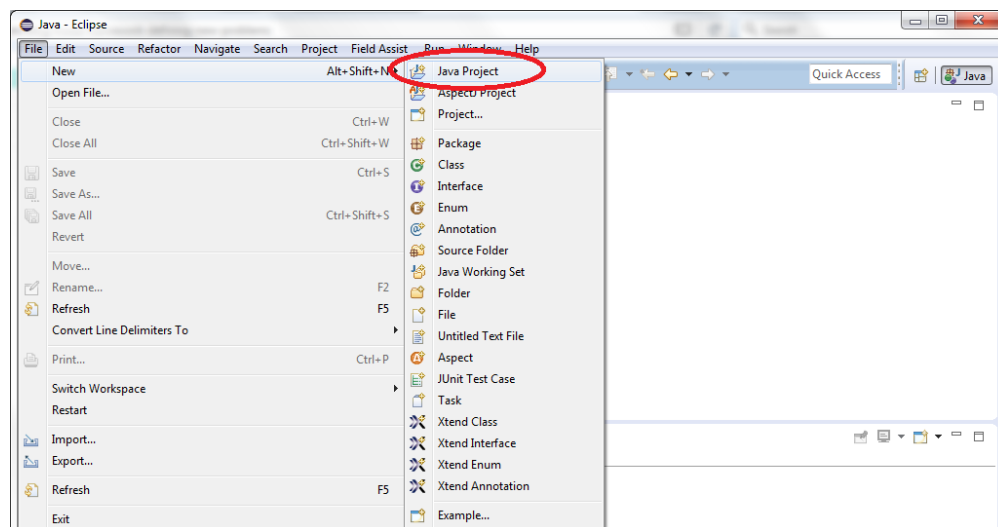
2.3 Setting up the MOEA Framework

We recommend starting with this book's supplemental materials, which includes a full installation of the MOEA Framework and all of the code samples found in this book. The supplemental materials can be downloaded by following this link: [Preview - Link Removed](#).

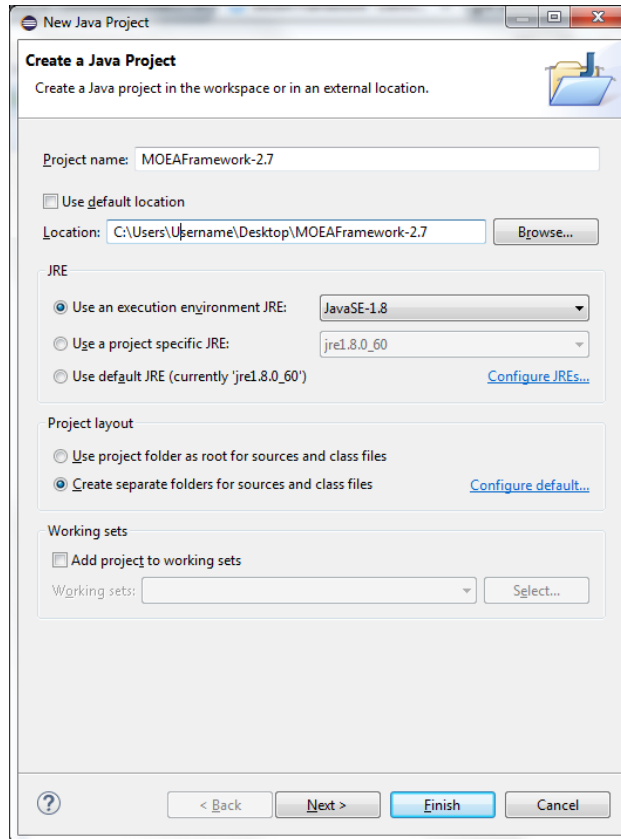
Note: that ends with the letter O and not the number 0. As you read this book, find the appropriate Java file within the 📁book folder to follow along.

Alternatively, you can download the MOEA Framework's compiled binaries from <http://www.moeaframework.org/> and manually type in the examples. The compiled binaries are distributed as a compressed TAR file (.tar.gz) and need to be extracted. We recommend using 7-Zip, a free and open source program, which can be downloaded from <http://www.7-zip.org/>. Extract to your Desktop or any other convenient location.

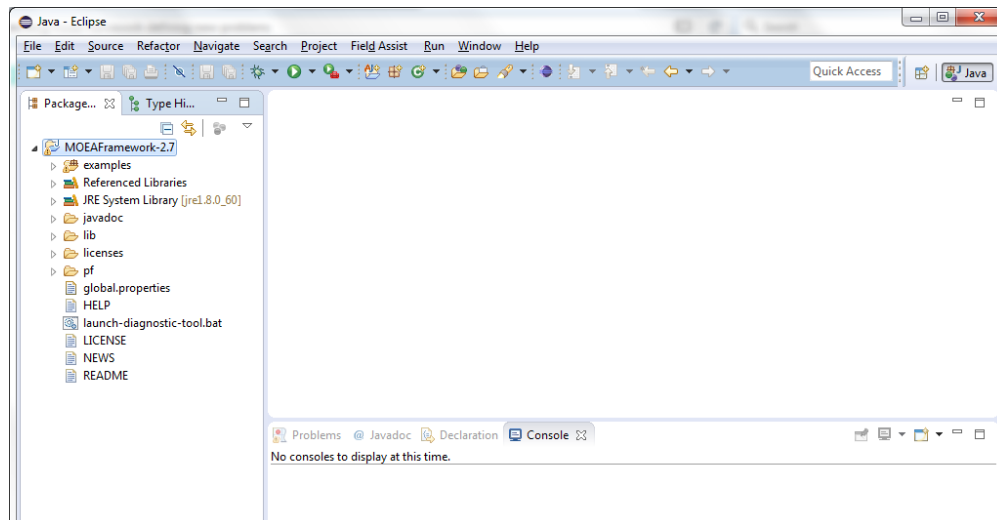
Next, we need to create a project within Eclipse. Select File → New Java Project from the menu.



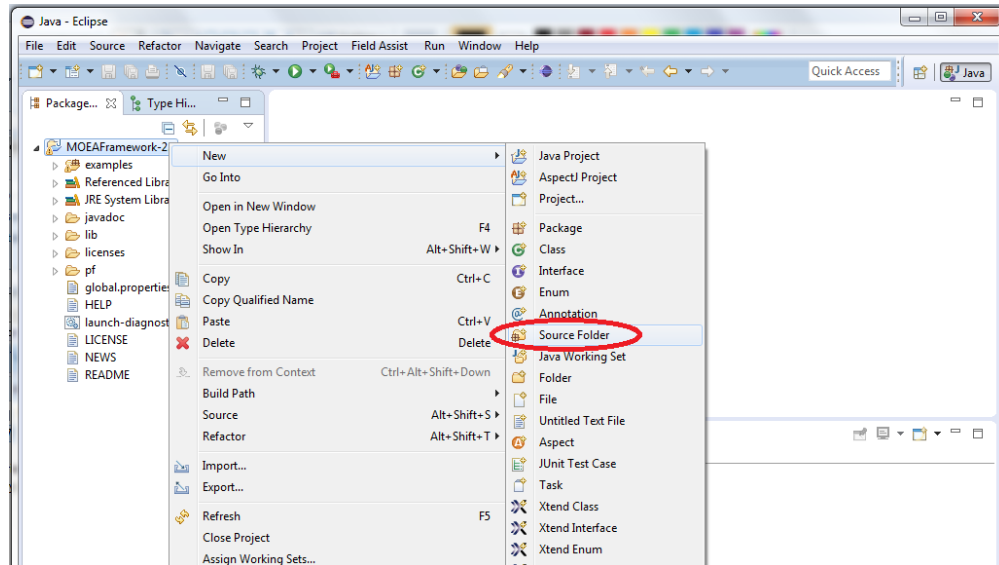
In the window that appears, uncheck "Use default location". Click the "Browse..." button and select the extracted MOEA Framework folder. Click Finish.



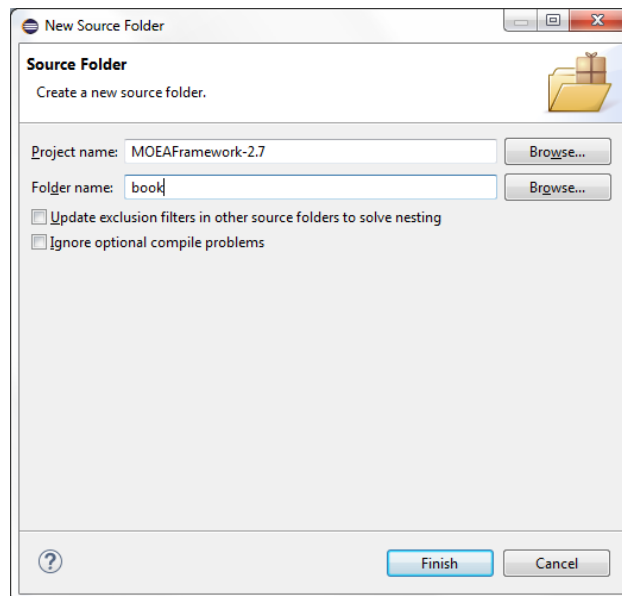
The MOEA Framework project will now appear within the "Package Explorer" in Eclipse, as shown below.



If you are using the supplemental materials, you can skip down to the next section titled "Your First Example." Otherwise, we will now create a source folder where our examples will reside. Right-click on the project and select New → Source Folder.

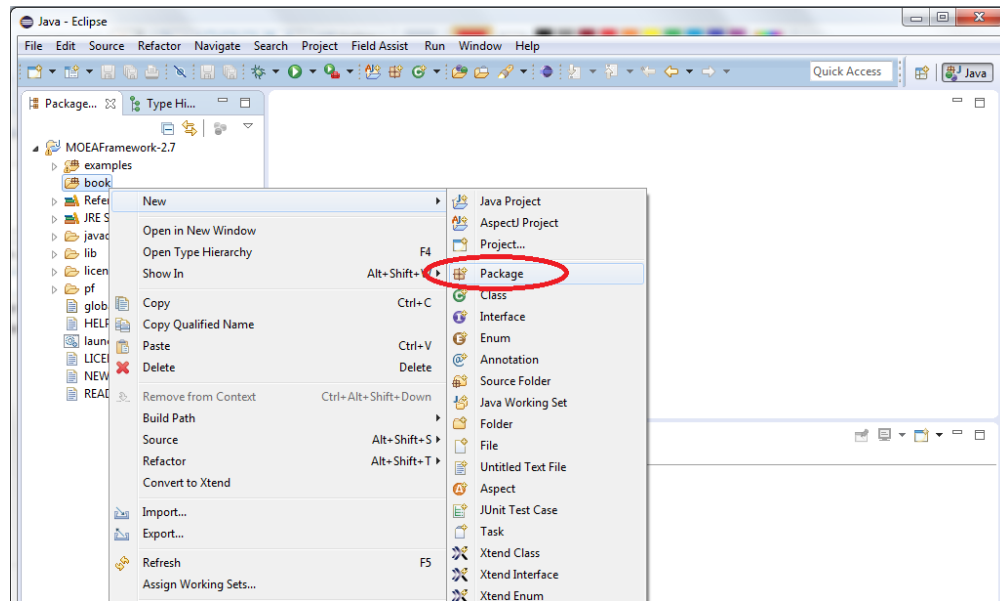


Give the new source folder the name "book" and click Finish.

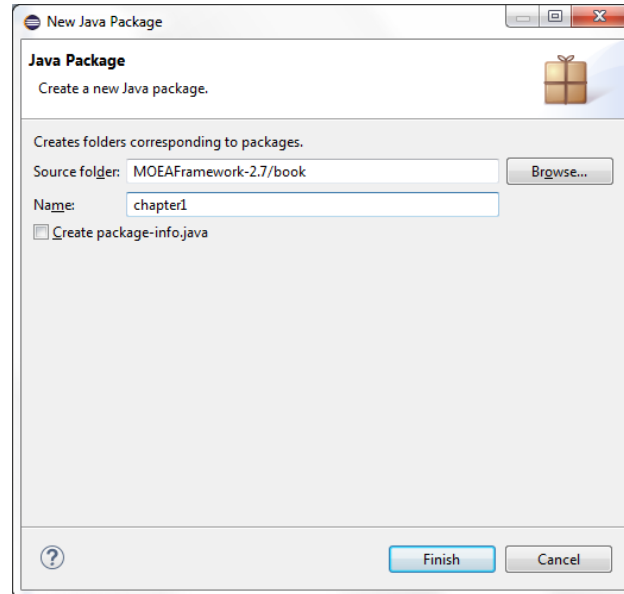


2.4 Your First Example

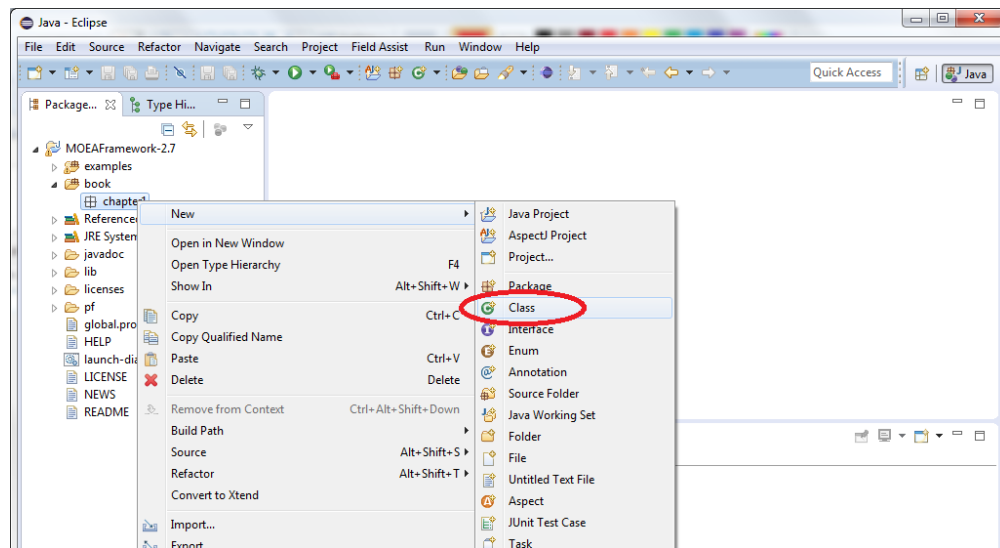
In Java, packages are used to organize source code into a hierarchical structure. We will organize the examples from each chapter into its own package. Thus, for the first chapter, we will create a package named `chapter2`. Right-click on the source folder we just created and select `New` → `Package`.



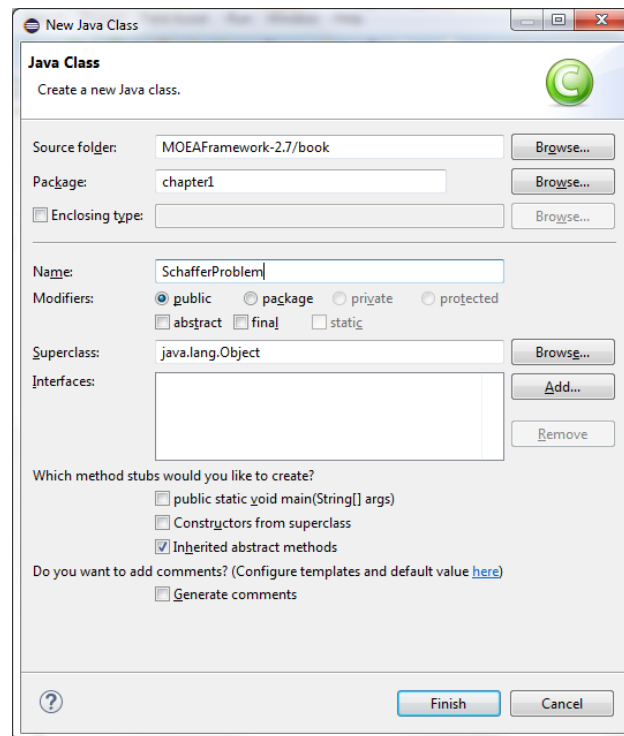
Enter the name `chapter2` and click Finish.



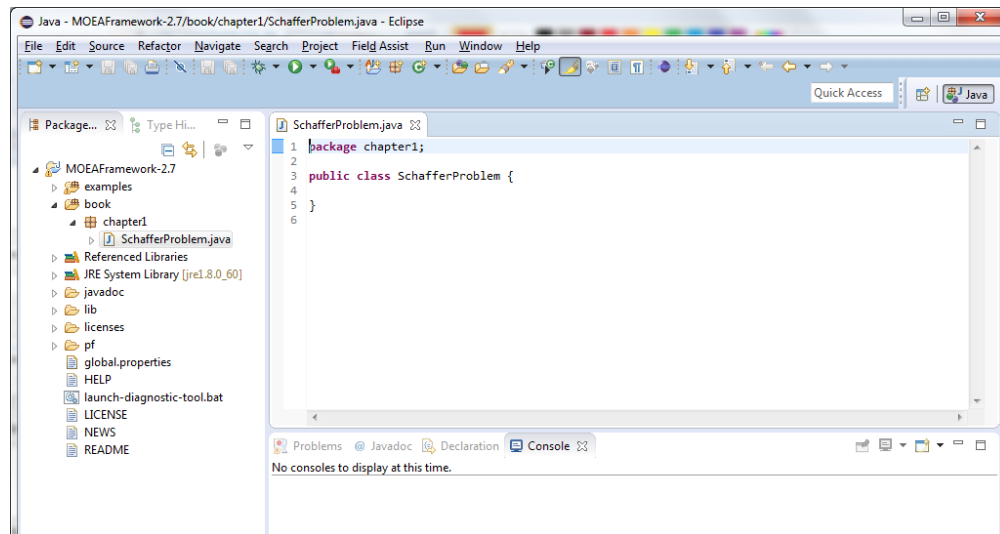
Finally, we create the Java file for the actual code. In Java, these are called classes. Right-click the `chapter2` folder and select `New` → `Class`.



Type the name SchafferProblem and click Finish.



At this point, your Eclipse workspace will contain one Java file named SchafferProblem.java and that file will be opened in the text editor within Eclipse, as shown below.



Now we can begin defining the problem.

```
1 package chapter2;
2
3 import org.moeaframework.core.Solution;
4 import org.moeaframework.core.variable.EncodingUtils;
5 import org.moeaframework.problem.AbstractProblem;
6
7 public class SchafferProblem extends AbstractProblem {
8
9     public SchafferProblem() {
10         super(1, 2);
11     }
12
13     @Override
14     public void evaluate(Solution solution) {
15         double x = EncodingUtils.getReal(solution.getVariable(0));
16
17         solution.setObjective(0, Math.pow(x, 2.0));
18         solution.setObjective(1, Math.pow(x - 2.0, 2.0));
19     }
20
21     @Override
22     public Solution newSolution() {
23         Solution solution = new Solution(1, 2);
24         solution.setVariable(0, EncodingUtils.newReal(-10.0, 10.0));
25         return solution;
26     }
27
28 }
```

MOEAFramework/book/chapter2/SchafferProblem.java

The anatomy of a problem is as follows. First, it must implement the Problem interface. Rather than implement the Problem interface directly, it is often more convenient to extend the AbstractProblem class, as seen on Line 7. Three methods are required when extending AbstractProblem: the constructor, newSolution, and evaluate. The constructor, shown on lines 9-11, is responsible for initializing the problem. For this problem, we call **super** (1, 2) to indicate this problem will consist of one decision variable and two objectives. The newSolution method, shown on lines 14-18, generates a prototype solution for the problem. A prototype solution describes each decision variable, and where applicable, any bounds on the variables. For this problem, we create a single real-valued decision variable bounded by $[-10, 10]$. Thus, on line 15 we create the prototype solution with one variable and two objectives (e.g., **new** Solution(1, 2)), assign the variable on line 16 (e.g., solution.setVariable(0, EncodingUtils.newReal(-10.0, 10.0))); and return the solution on line 17. Finally, we define the evaluate method on lines 21-26, which is responsible for computing the objectives for a given candidate solution. On line 22, we read the value of the decision variable (e.g., EncodingUtils.getReal(solution.getVariable(0))), and on lines 24 and 25 evaluate the two objectives. For the Schaffer problem, the two objectives are $f_1(x) = x^2$ and $f_2(x) = (x - 2)^2$. Type this code into the `SchafferProblem.java` file and save.

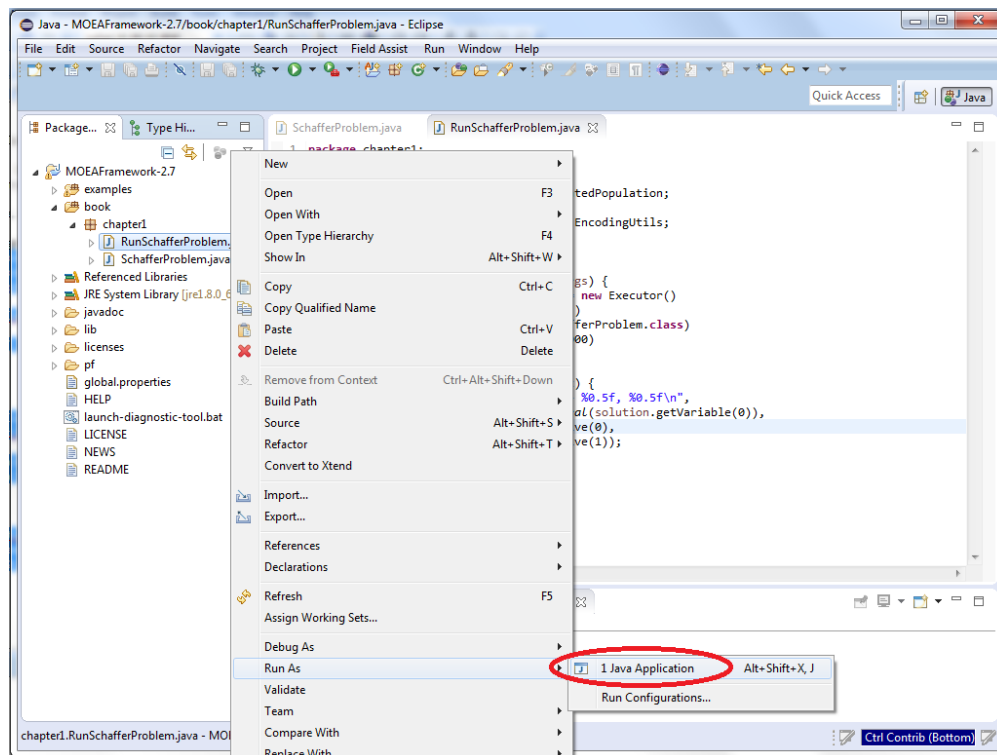
At this point, the problem is defined, but we also need to create the code to solve the problem. To begin, create a new class called RunSchafferProblem with the following code:

```

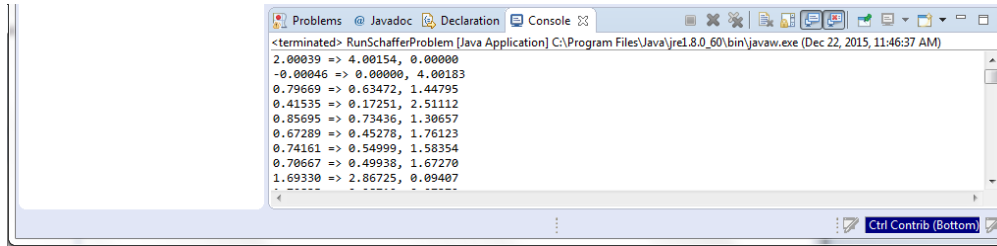
1 package chapter2;
2
3 import org.moeaframework.Executor;
4 import org.moeaframework.core.NondominatedPopulation;
5 import org.moeaframework.core.Solution;
6 import org.moeaframework.core.variable.EncodingUtils;
7
8 public class RunSchafferProblem {
9
10     public static void main(String[] args) {
11         NondominatedPopulation result = new Executor()
12             .withAlgorithm("NSGAII")
13             .withProblemClass(SchafferProblem.class)
14             .withMaxEvaluations(10000)
15             .run();
16
17         for (Solution solution : result) {
18             System.out.printf("%.5f => %.5f, %.5f\n",
19                 EncodingUtils.getReal(solution.getVariable(0)),
20                 solution.getObjective(0),
21                 solution.getObjective(1));
22         }
23     }
24
25 }
```


On line 10, we create a the main method. In Java, main methods are the starting points for applications. This is the first method that is invoked when we run the application. To solve our problem, we will use the `Executor` class. The executor is responsible for creating instances of optimization algorithms and using them to solve problems. It is a sophisticated class, but at the bare minimum it requires three pieces of information: 1) the name of the optimization algorithm, 2) the problem, and 3) the maximum number of function evaluations (NFE) permitted to solve the problem. We set the values on lines 12-14 and call `run()` on line 15 to solve the problem. The result, a Pareto approximation set, is saved on line 10 to a `NondominatedPopulation`. Lines 17-22 format and print the Pareto approximate set to the screen. Each line on the output is a Pareto approximate solution where the left-most value is the decision variable and the two values to the right of `=>` are the objective values.

Run this application by right-clicking the file `RunSchafferProblem.java` and selecting `Run` → `Java Application`.



The output will appear in the Console within Eclipse and should appear similar to below.



Congratulations, you have just successfully optimized a problem using the MOEA Framework!

2.5 Running from Command Line

If you are not using Eclipse to run these examples, they can also be run manually from the command line. On Windows, open a new Command Prompt window and change the directory to the MOEA Framework folder. Then type the following commands:

```
javac -cp "lib/*;book" book/chapter2/SchafferProblem.java
javac -cp "lib/*;book" book/chapter2/RunSchafferProblem.java
java -cp "lib/*;book" chapter2.RunSchafferProblem
```

The first two lines compile the two class files we created. Note the use of the `-cp "lib/*;book"` argument that specifies the Java classpath. This tells Java where to locate any referenced files. We will be referencing files in the `lib` folder, which contains all of the MOEA Framework libraries, and the `book` folder, which contains the files we are compiling. The last line runs the example. We again must specify the classpath, but note that we are running the class `chapter2.RunSchafferProblem`. This is the full class path for our problem. It consists of the package (e.g., `chapter2`), followed by a period, followed by the class name (e.g., `RunSchafferProblem`).

2.6 Plotting Results

In the previous example, we output the two objectives to the console. Outputting the raw data like this is useful as you can save the data to a text file, load the data into Excel, etc. For problems like this with only two objectives, we can plot the solutions as points in a scatter plot, as shown below:

```
1 package chapter2;
2
3 import org.moeaframework.Executor;
4 import org.moeaframework.analysis.plot.Plot;
5 import org.moeaframework.core.NondominatedPopulation;
6
```

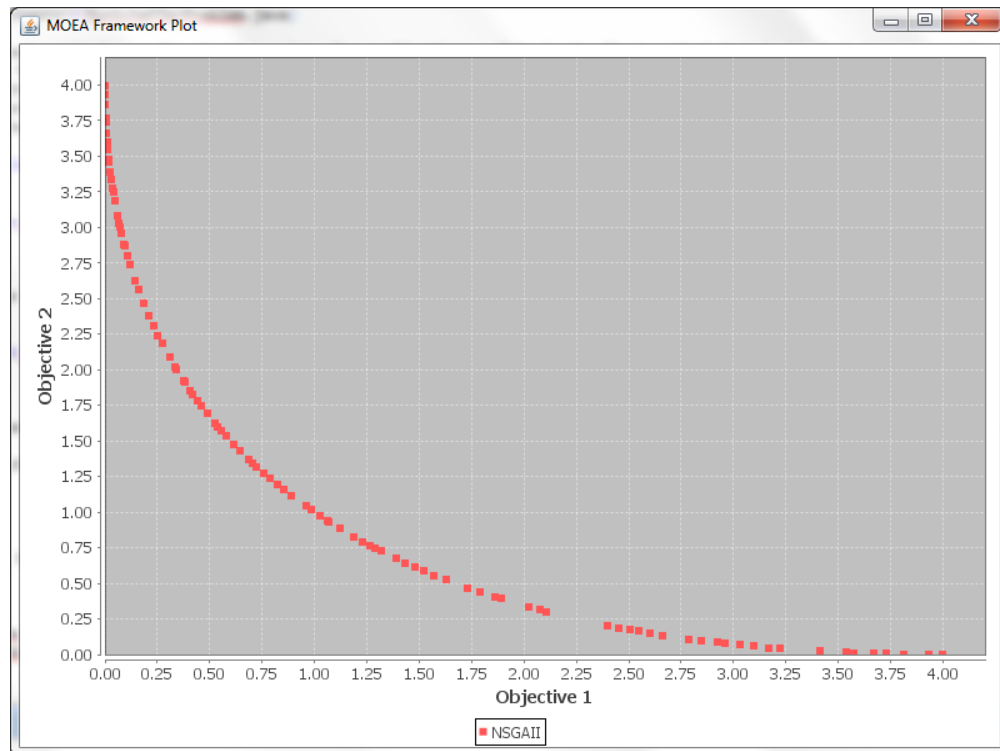
```

7 public class PlotSchafferProblem {
8
9     public static void main(String[] args) {
10         NondominatedPopulation result = new Executor()
11             .withAlgorithm("NSGAII")
12             .withProblemClass(SchafferProblem.class)
13             .withMaxEvaluations(10000)
14             .run();
15
16         new Plot()
17             .add("NSGAII", result)
18             .show();
19     }
20
21 }

```

MOEAFramework/book/chapter2/PlotSchafferProblem.java

Running this code produces the following plot:



Now we can see the shape of the Pareto approximation set produced by the optimization algorithm.

This document is a preview of the Beginner's Guide to the MOEA Framework. Some pages have been removed. Use the link below to purchase the complete guide.

[Online Store](#)

Chapter 3

Constrained Optimization

In the previous chapter, we solved the two objective Schaffer problem. This was an unconstrained problem, meaning that any solution we generate is a feasible design. Many real-world problems have constraints, either caused by physical limitations (e.g., maximum operating temperature), monetary (e.g., available capital), are risk-based (e.g., maximum failure rate for a product), etc. In general, there are two types of constraints: equality and inequality. Equality constraints are of the form $g(x) = c$ for some constant c . Inequality constraints are of the form $h(x) \geq d$ for some constant d . For example, takes the Srinivas problem as defined below.

$$F = (f_1(x, y), f_2(x, y)), \text{ where} \quad \left| \begin{array}{l} -20 \leq x, y \leq 20, \\ 0 \geq x^2 + y^2 - 225, \\ 0 \geq x - 3y + 10 \end{array} \right.$$

On the left, we see the two objective functions that we are minimizing. On the right, we have the constraints. Note they are all inequality constraints.

The MOEA Framework represents constraints a bit differently. Instead of needing to know the constraint formula, the MOEA Framework simply says “set the constraint value to 0 if the solution is feasible, set it to any non-zero value if the constraint is violated.” For example, take the constraint $0 \geq x - 3y + 10$. You would typically express this constraint within the MOEA Framework using the ternary if-else expression $x - 3y + 10 \leq 0 ? 0 : x - 3y + 10$. This expression begins with the comparator $x - 3y + 10 \leq 0$ that tests if the constraint is satisfied. If satisfied, the resulting is 0. Otherwise, the result is $x - 3y + 10$. Why do we set the value to $x - 3y + 10$ when the constraint is violated? It is useful in optimization to know how far a solution is from the feasibility boundary. By setting the constraint value to smaller values the closer a solution likes in proximity to the feasibility boundary, the optimization algorithm can guide search towards the feasible region. For the Srinivas problem, we would evaluate the problem as follows:

```
1 public void evaluate(Solution solution) {  
2     double x = EncodingUtils.getReal(solution.getVariable(0));
```

```

3      double y = EncodingUtils.getReal(solution.getVariable(1));
4      double f1 = Math.pow(x - 2.0, 2.0) + Math.pow(y - 1.0, 2.0) + 2.0;
5      double f2 = 9.0*x - Math.pow(y - 1.0, 2.0);
6      double c1 = Math.pow(x, 2.0) + Math.pow(y, 2.0) - 225.0;
7      double c2 = x - 3.0*y + 10.0;
8
9      solution.setObjective(0, f1);
10     solution.setObjective(1, f2);
11     solution.setConstraint(0, c1 <= 0.0 ? 0.0 : c1);
12     solution.setConstraint(1, c2 <= 0.0 ? 0.0 : c2);
13 }

```

Lets try another example. Suppose we have the constraint $x^2 + y \leq 10$. The trick here is to remember that we want to assign non-zero values when the constraint is violated. It is useful to convert the constraint into the form $h(x) \leq 0$, or $x^2 + y - 10 \leq 0$. The resulting constraint calculation would be:

```

1      double c = Math.pow(x, 2.0) + y - 10;
2      solution.setConstraint(0, c <= 0.0 ? 0.0 : c);

```

Great! You'll probably notice that there is an additional constraint in our Srinivas problem: $-20 \leq x, y \leq 20$. This is different from equality and inequality constraints because these are constraints placed on the decision variables. In the MOEA Framework, we do not need to explicitly specify this as a constraint. As shown below, we set these bounds when defining the decision variables.

```

1      public Solution newSolution() {
2          Solution solution = new Solution(2, 2, 2);
3
4          solution.setVariable(0, EncodingUtils.newReal(-20.0, 20.0));
5          solution.setVariable(1, EncodingUtils.newReal(-20.0, 20.0));
6
7          return solution;
8      }

```

3.1 Constrained Optimization Example

Ok, now we can fully define the constrained Srinivas problem. From the problem statement, we see that the Srinivas problem has two decision variables, two objectives, and two constraints. The two decision variables are real-valued and bounded by $[-20, 20]$. The equations for the objectives and constraints are given.

Within Eclipse, create a new package named `chapter3` and create the class `SrinivasProblem`. Enter the following code:

```
1 package chapter3;
2
3 import org.moeaframework.core.Solution;
4 import org.moeaframework.core.variable.EncodingUtils;
5 import org.moeaframework.problem.AbstractProblem;
6
7 public class SrinivasProblem extends AbstractProblem {
8
9     public SrinivasProblem() {
10         super(2, 2, 2);
11     }
12
13     @Override
14     public void evaluate(Solution solution) {
15         double x = EncodingUtils.getReal(solution.getVariable(0));
16         double y = EncodingUtils.getReal(solution.getVariable(1));
17         double f1 = Math.pow(x - 2.0, 2.0) + Math.pow(y - 1.0, 2.0) + 2.0;
18         double f2 = 9.0*x - Math.pow(y - 1.0, 2.0);
19         double c1 = Math.pow(x, 2.0) + Math.pow(y, 2.0) - 225.0;
20         double c2 = x - 3.0*y + 10.0;
21
22         solution.setObjective(0, f1);
23         solution.setObjective(1, f2);
24         solution.setConstraint(0, c1 <= 0.0 ? 0.0 : c1);
25         solution.setConstraint(1, c2 <= 0.0 ? 0.0 : c2);
26     }
27
28     @Override
29     public Solution newSolution() {
30         Solution solution = new Solution(2, 2, 2);
31
32         solution.setVariable(0, EncodingUtils.newReal(-20.0, 20.0));
33         solution.setVariable(1, EncodingUtils.newReal(-20.0, 20.0));
34
35         return solution;
36     }
37
38 }
```

MOEAFramework/book/chapter3/SrinivasProblem.java

We already explained the components of this code. The primary difference is the addition of the constraints. First, on lines 10 and 28, we pass in three arguments instead of two. The third argument indicates the number of constraints (e.g., **super**(2, 2, 2) and **new** Solution(2, 2, 2)). Secondly, on lines 23-24 we set the constraints. Again, the constraint is 0 when a solution is feasible.

As before, we also need a class to run this example. Create the class

RunSrinivasProblem with the code below:

```
1 package chapter3;
2
3 import org.moeaframework.Executor;
4 import org.moeaframework.core.NondominatedPopulation;
5 import org.moeaframework.core.Solution;
6
7 public class RunSrinivasProblem {
8
9     public static void main(String[] args) {
10         NondominatedPopulation result = new Executor()
11             .withAlgorithm("NSGAII")
12             .withProblemClass(SrinivasProblem.class)
13             .withMaxEvaluations(10000)
14             .run();
15
16         for (Solution solution : result) {
17             if (!solution.violatesConstraints()) {
18                 System.out.format("%10.3f      %10.3f%n",
19                     solution.getObjective(0),
20                     solution.getObjective(1));
21             }
22         }
23     }
24 }
25 }
```

MOEAFramework/book/chapter3/RunSrinivasProblem.java

There are a few changes to this code from the previous chapter. First, on line 12 we use the new `SrinivasProblem` class. Second, on line 17, we check if the solutions are feasible prior to printing the output. Most algorithms will only store feasible solutions, but it's always good practice to check if a solution violates any constraints. With this code input, you can then run the `RunSrinivasProblem` class and view the output. We could alternatively plot the results for easier viewing:

```
1 package chapter3;
2
3 import org.moeaframework.Executor;
4 import org.moeaframework.analysis.plot.Plot;
5 import org.moeaframework.core.NondominatedPopulation;
6
7 public class PlotSrinivasProblem {
8
9     public static void main(String[] args) {
10         NondominatedPopulation result = new Executor()
11             .withAlgorithm("NSGAII")
12             .withProblemClass(SrinivasProblem.class)
13             .withMaxEvaluations(10000)
```



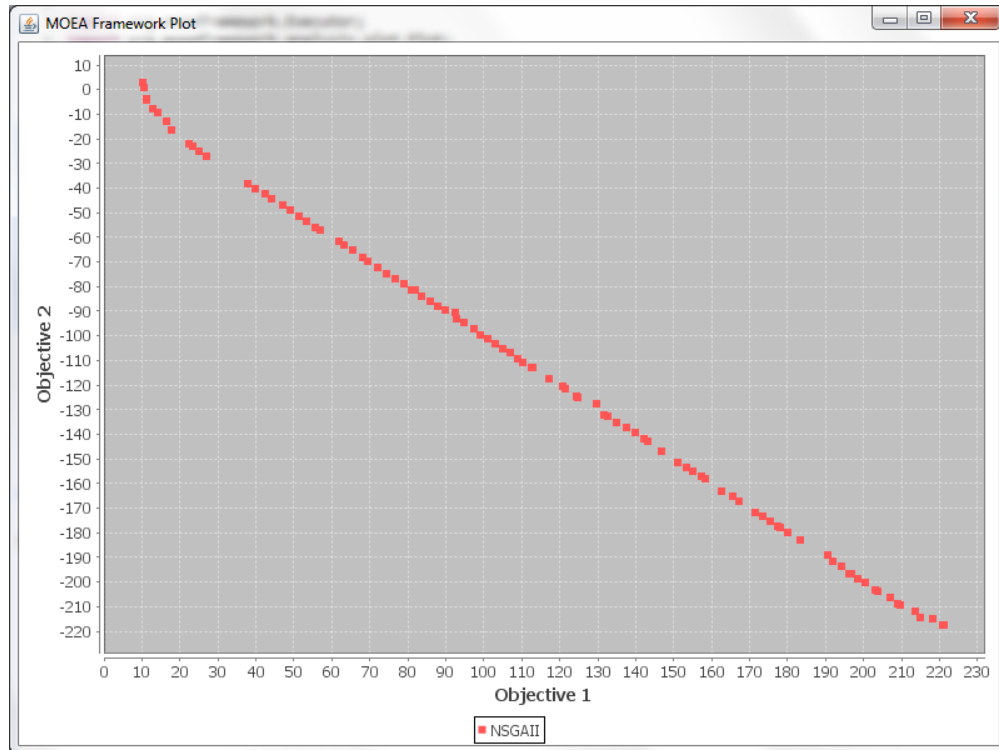
```

14     .run();
15
16     new Plot()
17         .add("NSGAI", result)
18         .show();
19 }
20
21 }

```

MOEAFramework/book/chapter3/PlotSrinivasProblem.java

which produces the following plot:



3.2 The Knapsack Problem

Ok, now for a more complex example. Have you ever heard of the famous Knapsack problem? If not, check out the Wikipedia page at http://en.wikipedia.org/wiki/Knapsack_problem for more details. This is a famous combinatorial problem that involves choosing which items to place in a knapsack to maximize the value of the items carried without exceeding the weight capacity of the knapsack. More formally, we are given N items. Each item has a profit, $P(i)$, and weight, $W(i)$, for $i = 1, 2, \dots, N$. Let $d(i)$ represent our decision to place the i -th item in the knapsack, where $d(i) = 1$ if the item is put into the knapsack and $d(i) = 0$ otherwise. If the knapsack has a weight capacity of C , then the knapsack problem is defined as:

$$\text{Maximize } \sum_{i=1}^N d(i) * P(i) \text{ such that } \sum_{i=1}^N d(i) * W(i) \leq C$$

The summation on the left (which we are maximizing) calculates the total profit we gain from the items placed in the knapsack. The summation on the right side is a constraint that ensures the items placed in the knapsack do not exceed the weight capacity of the knapsack.

Lets make it a little more interesting, after all this is a library for multiobjective optimization. Instead of having one knapsack, lets have two (in fact, this can be generalized to any number of knapsacks). Additionally, the profit and weights vary depending on which knapsack is holding each item. For example, an item will have a profit of \$25 and a weight of 5 pounds in the first knapsack, but will have a profit of \$15 and a weight of 8 pounds in the second knapsack. (It may seem unusual that the weight changes, but that is how the problem is defined in the literature.) Thus, profit is now defined by $P(i, j)$ and weight by $W(i, j)$, where the $j = 1, 2$ term is the knapsack index. Lastly, each knapsack defines its own capacity, C_1 and C_2 . Combining all of this, the multiobjective knapsack problem is formally defined as:

$$\begin{aligned} &\text{Maximize } \sum_{i=1}^N d(i) * P(i, 1) \text{ such that } \sum_{i=1}^N d(i) * W(i, 1) \leq C_1 \text{ and} \\ &\text{Maximize } \sum_{i=1}^N d(i) * P(i, 2) \text{ such that } \sum_{i=1}^N d(i) * W(i, 2) \leq C_2 \end{aligned}$$

This problem is a bit different from the others we have seen thus far. For the knapsack problem, we are picking items to fit into the knapsack. The bit string representation works well for situation where we are making many yes/no decisions (yes if it is included in the knapsack). For example, if we have 5 items, we can represent the decision to include each item using a bit string with 5 bits. Each bit in the string corresponds to an item, and is set to 1 if the item is included and 0 if the item is excluded. For instance, the bit string 00110 would place items 3 and 4 inside the knapsacks, excluding the rest. Our `newSolution` method is defined as follows:

```

1  public Solution newSolution() {
2      Solution solution = new Solution(1, nsacks, nsacks);
3      solution.setVariable(0, EncodingUtils.newBinary(nitems));
4      return solution;
5  }
```

Observe on line 3 how we create the bit string representation (also called a binary string) with `nitems` bits. Also note that the 5 bits are contained within a single decision variable, so we define this problem with only a single decision variable.

Now for the `evaluate` method. Summing up the profits is straightforward. But there is also a constraint we must deal with. We must ensure the weight of the items does not exceed the capacity of the knapsack. Thus, we need to sum up the weights of the selected items and compare to the capacity. The resulting method is shown below:

```

1 public void evaluate(Solution solution) {
2     boolean[] d = EncodingUtils.getBinary(solution.getVariable(0));
3     double[] f = new double[nsacks];
4     double[] g = new double[nsacks];
5
6     // calculate the profits and weights for the knapsacks
7     for (int i = 0; i < nitems; i++) {
8         if (d[i]) {
9             for (int j = 0; j < nsacks; j++) {
10                 f[j] += profit[j][i];
11                 g[j] += weight[j][i];
12             }
13         }
14     }
15
16     // check if any weights exceed the capacities
17     for (int j = 0; j < nsacks; j++) {
18         if (g[j] <= capacity[j]) {
19             g[j] = 0.0;
20         } else {
21             g[j] = g[j] - capacity[j];
22         }
23     }
24
25     // negate the objectives since Knapsack is maximization
26     solution.setObjectives(Vector.negate(f));
27     solution.setConstraints(g);
28 }

```

Note the comment on line 25. We are negating the objective values since our objectives are being maximized. The MOEA Framework is designed to minimize objectives. To handle maximized objectives, simply negate the value. Minimizing the negated value is equivalent to maximizing the original value. Take caution, however, as the output from the MOEA Framework will include the negative values. You should always negate the outputs to restore them to their correct sign.

Below is the full implementation of the Knapsack problem. We have configured this instance for a simple problem with 2 knapsacks and 5 items. Copy this code into the KnapsackProblem class.

```

1 package chapter3;
2
3 import org.moeaframework.core.Solution;
4 import org.moeaframework.core.variable.EncodingUtils;
5 import org.moeaframework.problem.AbstractProblem;
6 import org.moeaframework.util.Vector;
7
8 public class KnapsackProblem extends AbstractProblem {

```

```

9
10 /**
11  * The number of sacks.
12  */
13 public static int nsacks = 2;
14
15 /**
16  * The number of items.
17  */
18 public static int nitems = 5;
19
20 /**
21  * Entry {@code profit[i][j]} is the profit from including item {@code j}
22  * in sack {@code i}.
23  */
24 public static int[][] profit = {
25     {2, 5},
26     {1, 4},
27     {6, 2},
28     {5, 1},
29     {3, 3}
30 };
31
32 /**
33  * Entry {@code weight[i][j]} is the weight incurred from including item
34  * {@code j} in sack {@code i}.
35  */
36 public static int[][] weight = {
37     {3, 3},
38     {4, 2},
39     {1, 5},
40     {5, 3},
41     {5, 2}
42 };
43
44 /**
45  * Entry {@code capacity[i]} is the weight capacity of sack {@code i}.
46  */
47 public static int[] capacity = { 10, 8 };
48
49 public KnapsackProblem() {
50     super(1, nsacks, nsacks);
51 }
52
53 public void evaluate(Solution solution) {
54     boolean[] d = EncodingUtils.getBinary(solution.getVariable(0));
55     double[] f = new double[nsacks];
56     double[] g = new double[nsacks];
57
58     // calculate the profits and weights for the knapsacks
59     for (int i = 0; i < nitems; i++) {

```

```

60     if (d[i]) {
61         for (int j = 0; j < nsacks; j++) {
62             f[j] += profit[j][i];
63             g[j] += weight[j][i];
64         }
65     }
66 }
67
68 // check if any weights exceed the capacities
69 for (int j = 0; j < nsacks; j++) {
70     if (g[j] <= capacity[j]) {
71         g[j] = 0.0;
72     } else {
73         g[j] = g[j] - capacity[j];
74     }
75 }
76
77 // negate the objectives since Knapsack is maximization
78 solution.setObjectives(Vector.negate(f));
79 solution.setConstraints(g);
80 }
81
82 public Solution newSolution() {
83     Solution solution = new Solution(1, nsacks, nsacks);
84     solution.setVariable(0, EncodingUtils.newBinary(nitems));
85     return solution;
86 }
87
88 }

```

MOEAFramework/book/chapter3/KnapsackProblem.java

Next, copy the following code into the RunKnapsackProblem class.

```

1 package chapter3;
2
3 import java.io.IOException;
4
5 import org.moeaframework.Executor;
6 import org.moeaframework.core.NondominatedPopulation;
7 import org.moeaframework.core.Solution;
8 import org.moeaframework.examples.ga.knapsack.Knapsack;
9 import org.moeaframework.util.Vector;
10
11 public class RunKnapsackProblem {
12
13     public static void main(String[] args) throws IOException {
14         NondominatedPopulation result = new Executor()
15             .withProblemClass(Knapsack.class)
16             .withAlgorithm("NSGAII")
17             .withMaxEvaluations(10000)
18             .distributeOnAllCores()

```

```

19         .run();
20
21     for (int i = 0; i < result.size(); i++) {
22         Solution solution = result.get(i);
23         double[] objectives = solution.getObjectives();
24
25         // negate objectives to return them to their maximized form
26         objectives = Vector.negate(objectives);
27
28         System.out.println("Solution " + (i+1) + ":");
29         System.out.println("    Sack 1 Profit: " + objectives[0]);
30         System.out.println("    Sack 2 Profit: " + objectives[1]);
31         System.out.println("    Binary String: " + solution.getVariable(0));
32     }
33 }
34
35 }

```

MOEAFramework/book/chapter3/RunKnapsackProblem.java

As before, we specify the problem class when creating the `Executor`. Observe that we do not need to tell the algorithm about any new features of our problem. It automatically detects that the problem uses a bit string representation and constructs the algorithm with the appropriate crossover and mutation operators. Also note on line 18 that we call `distributeOnAllCores()`, which enables multithreading. For problems with time-consuming evaluations, you can gain substantial performance improvements by spreading the work across multiple processors on your computer. The MOEA Framework automatically handles this for you.

Running this example will produce output similar to the following. Since this is a multiobjective problem, there is typically no single optimal solution. Instead, there are several options, all equally “good”. Solution 1 has the maximum profit of 9.0 for Sack 1, but the worst profit of 5.0 for Sack 2.

```

Solution 1:
    Sack 1 Profit: 9.0
    Sack 2 Profit: 5.0
    Binary String: 01010
Solution 2:
    Sack 1 Profit: 5.0
    Sack 2 Profit: 8.0
    Binary String: 00110
Solution 3:
    Sack 1 Profit: 7.0
    Sack 2 Profit: 6.0
    Binary String: 00011
Solution 4:
    Sack 1 Profit: 6.0
    Sack 2 Profit: 7.0
    Binary String: 01100

```

In this Knapsack problem, we hard-coded the profits, weights, and capacities. A more generalized implementation is available with the MOEA Framework in the `examples` folder.

3.3 Feasibility

You may encounter problems that are severely constrained where the majority of the search space is infeasible. Under these circumstances, it becomes very challenging for an optimization algorithm to find feasible solutions. When this happens, the results from the MOEA Framework will contain only infeasible solutions. It is important to check if solutions are infeasible with the `solution.violatesConstraints()` method. This can be a challenging problem to address. In some cases, the problem can be reformulated or represented in a different way to relax the constraints. This is outside the scope of this beginner's guide, but more details can be found in academic literature.

This document is a preview of the Beginner's Guide to the MOEA Framework. Some pages have been removed. Use the link below to purchase the complete guide.

[Online Store](#)

Appendix A

List of Algorithms

This appendix lists the available algorithms, a short description of the distinct features of the algorithm, and a list of the parameters and default values. Most of these algorithms support a variety of crossover and/or mutation operators. In those cases, refer to Appendix B for a list of the operators and their parameters.

CMA-ES

CMA-ES is a sophisticated covariance matrix adaptation evolution strategy algorithm for real-valued global optimization (Hansen and Kern, 2004; Igel et al., 2007). CMA-ES produces offspring by sampling a distribution formed by a covariance matrix, hence the name, and updating the covariance matrix based on the surviving offspring. Single and multi-objective variants exist in the literature and both are supported by the MOEA Framework.

Parameters	Description	Default Value
lambda	The offspring population size	100
cc	The cumulation parameter	1
cs	The step size of the cumulation parameter	1
damps	The damping factor for the step size	1
ccov	The learning rate	1
ccovsep	The learning rate when in diagonal-only mode	1
sigma	The initial standard deviation	0.5
diagonalIterations	The number of iterations in which only the covariance diagonal is used	0
indicator	Either " hypervolume " or " epsilon " to specify the use of the hypervolume or additive-epsilon indicator. If unset, crowding distance is used	Unset
initialSearchPoint	Initial guess at the starting location (comma-separated values). If unset, a random initial guess is used	Unset

¹Parameter value is derived from other settings. See Igel et al. (2007) for details.

ϵ -MOEA

ϵ -MOEA is a steady-state MOEA that uses ϵ -dominance archiving to record a diverse set of Pareto optimal solutions Deb et al. (2003). The term steady-state means that the algorithm evolves one solution at a time. This is in contrast to generational algorithms, which evolve the entire population every iteration. ϵ -dominance archives are useful since they ensure convergence and diversity throughout search Laumanns et al. (2002). However, the algorithm requires an additional ϵ parameter which is problem dependent. The ϵ parameter controls the granularity or resolution of the solutions in objective space. Smaller values produce larger, more dense sets while larger values produce smaller sets. In general, the ϵ values should be chosen to yield a moderately-sized Pareto approximate set.

Parameter	Description	Default Value
populationSize	The size of the population	100
epsilon	The ϵ values used by the ϵ -dominance archive, which can either be a single value or a comma-separated array	Problem dependent

ϵ -NSGA-II

ϵ -NSGA-II combines the generational search of NSGA-II with the guaranteed convergence provided by an ϵ -dominance archive Kollat and Reed (2006). It also features randomized restarts to enhance search and find a diverse set of Pareto optimal solutions. During a random restart, the algorithm empties the current population and fills it with new, randomly-generated solutions.

Parameter	Description	Default Values
populationSize	The size of the population	100
epsilon	The ϵ values used by the ϵ -dominance archive, which can either be a single value or a comma-separated array	Problem dependent
injectionRate	Controls the percentage of the population after a restart this is “injected”, or copied, from the ϵ -dominance archive	0.25
windowSize	Frequency of checking if a randomized restart should be triggered (number of iterations)	100
maxWindowSize	The maximum number of iterations between successive randomized restarts	100
minimumPopulationSize	The smallest possible population size when injecting new solutions after a randomized restart	100
maximumPopulationSize	The largest possible population size when injecting new solutions after a randomized restart	10000

GDE3

GDE3 is the third version of the generalized differential evolution algorithm Kukkonen and Lampinen (2005). The name differential evolution comes from how the algorithm evolves offspring. It randomly selects three parents. Next, it computes the difference (the differential) between two of the parents. Finally, it offsets the remaining parent by this differential.

Parameter	Description	Default Values
populationSize	The size of the population	100
de.crossoverRate	The crossover rate for differential evolution	0.1
de.stepSize	Control the size of each step taken by differential evolution	0.5

IBEA

IBEA is a indicator-based MOEA that uses the hypervolume performance indicator as a means to rank solutions Zitzler and Künzli (2004). Indicator-based algorithms are based on the idea that a performance indicator, such as hypervolume or additive ϵ -indicator, highlight solutions with desirable qualities. The primary disadvantage of indicator-based methods is that the calculation of the performance indicator can become computationally expensive, particularly as the number of objectives increases.

Parameter	Description	Default Value
populationSize	The size of the population	100
indicator	The indicator function (e.g., "hypervolume", "epsilon")	"hypervolume"

MOEA/D

MOEA/D is a relatively new optimization algorithm based on the concept of decomposing the problem into many single-objective formulations . Several version of MOEA/D exist in the literature. The most common variant seen in the literature, MOEA/D-DE (Li and Zhang, 2009), is the default implementation in the MOEA Framework.

An extension to MOEA/D-DE variant called MOEA/D-DRA introduced a utility function that aimed to reduce the amount of “wasted” effort by the algorithm (Zhang et al., 2009). This variant is enabled by setting the `updateUtility` parameter to a non-zero value.

Parameter	Description	Default Value
populationSize	The size of the population	100
de.crossoverRate	The crossover rate for differential evolution	0.1
de.stepSize	Control the size of each step taken by differential evolution	0.5
pm.rate	The mutation rate for polynomial mutation	1/ N
pm.distributionIndex	The distribution index for polynomial mutation	20.0
neighborhoodSize	The size of the neighborhood used for mating, given as a percentage of the population size	0.1
delta	The probability of mating with an individual from the neighborhood versus the entire population	0.9
eta	The maximum number of spots in the population that an offspring can replace, given as a percentage of the population size	0.01
updateUtility	The frequency, in generations, at which utility values are updated. If set, this uses the MOEA/D-DRA variant; if unset, then the MOEA/D-DE variant is used	Unset

NSGA-II

NSGA-II is one of the first and most widely used MOEAs (Deb et al., 2000). It enhanced its predecessor, NSGA, by introducing fast non-dominated sorting and using the more computationally efficient crowding distance metric during survival selection.

Parameter	Description	Default Value
populationSize	The size of the population	100

NSGA-III

NSGA-III is the many-objective successor to NSGA-II, using reference points to direct solutions towards a diverse set (Deb and Jain, 2014). The number of reference points is controlled by the number of objectives and the `divisions` parameter. For an M -objective problem, the number of reference points is:

$$H = \binom{M + \text{divisions} - 1}{\text{divisions}} \quad (\text{A.1})$$

The authors also propose a two-layer approach for divisions for many-objective problems where an outer and inner division number is specified. To use the two-layer approach, replace the `divisions` parameter with `divisionsOuter` and `divisionsInner`.

Parameter	Description	Default Value
populationSize	The size of the population. If unset, the population size is equal to the number of reference points	Unset
divisions	The number of divisions	Problem dependent

OMOPSO

OMOPSO is a multiobjective particle swarm optimization algorithm that includes an ϵ -dominance archive to discover a diverse set of Pareto optimal solutions (Sierra and Coello Coello, 2005). This implementation of OMOPSO differs slightly from the original author's implementation in JMetal due to a discrepancy between the author's code and the paper. The paper returns the ϵ -dominance archive while the code returns the leaders. This discrepancy causes a small difference in performance.

Parameter	Description	Default Value
populationSize	The size of the population	100
archiveSize	The size of the archive	100
maxEvaluations	The maximum number of evaluations for adapting non-uniform mutation	25000
mutationProbability	The mutation probability for uniform and non-uniform mutation	$1/N$
perturbationIndex	Controls the shape of the distribution for uniform and non-uniform mutation	0.5
epsilon	The ϵ values used by the ϵ -dominance archive	Problem dependent

PAES

PAES is a multiobjective version of evolution strategy (Knowles and Corne, 1999). PAES tends to underperform when compared to other MOEAs, but it is often used as a baseline algorithm for comparisons. Like PESA-II, PAES uses the adaptive grid archive to maintain a fixed-size archive of solutions.

Parameter	Description	Default Value
archiveSize	The size of the archive	100
bisections	The number of bisections in the adaptive grid archive	8
pm.rate	The mutation rate for polynomial mutation	$1/N$
pm.distributionIndex	The distribution index for polynomial mutation	20.0

PESA-II

PESA-II is another multiobjective evolutionary algorithm that tends to underperform other MOEAs but is often used as a baseline algorithm in comparative studies (Corne et al., 2001). It is the successor to PESA (Corne and Knowles, 2000). Like PAES, PESA-II uses the adaptive grid archive to maintain a fixed-size archive of solutions.

Parameter	Description	Default Value
populationSize	The size of the population	10
archiveSize	The size of the archive	100
bisections	The number of bisections in the adaptive grid archive	8

Random

The random search algorithm simply randomly generates new solutions uniformly throughout the search space. It is not intended as an “optimization algorithm” *per se*, but as a way to compare the performance of other MOEAs against random search. If an optimization algorithm can not beat random search, then continued use of that optimization algorithm should be questioned.

Parameter	Description	Default Value
populationSize	This parameter only has a use when parallelizing evaluations; it controls the number of solutions that are generated and evaluated in parallel	100
epsilon	The ϵ values used by the ϵ -dominance archive, which can either be a single value or a comma-separated array (this parameter is optional)	Unset

SMPSO

SMPSO is a multiobjective particle swarm optimization algorithm (Nebro et al., 2009).

Parameter	Description	Default Value
populationSize	The size of the population	100
archiveSize	The size of the archive	100
pm.rate	The mutation rate for polynomial mutation	$1/N$
pm.distributionIndex	The distribution index for polynomial mutation	20.0

SMS-EMOA

SMS-EMOA is an indicator-based MOEA that uses the volume of the dominated hypervolume to rank individuals (Beume et al., 2007).

Parameter	Description	Default Value
populationSize	The size of the population	100
offset	The reference point offset for computing hypervolume	100

SPEA2

SPEA2 is an older but popular benchmark MOEA that uses the so-called “strength-based” method for ranking solutions (Zitzler et al., 2002a). The general idea is that the strength or quality of a solution is related to the strength of solutions it dominates.

Parameter	Description	Default Value
populationSize	The size of the population	100
offspringSize	The number of offspring generated every iteration	100
k	Crowding is based on the distance to the k -th nearest neighbor	1

VEGA

VEGA is considered the earliest documented MOEA. While we provide support for VEGA, other MOEAs should be preferred as they exhibit better performance. VEGA is provided for its historical significance (Schaffer, 1985).

Parameter	Description	Default Value
populationSize	The size of the population	100

Appendix B

List of Variation Operators

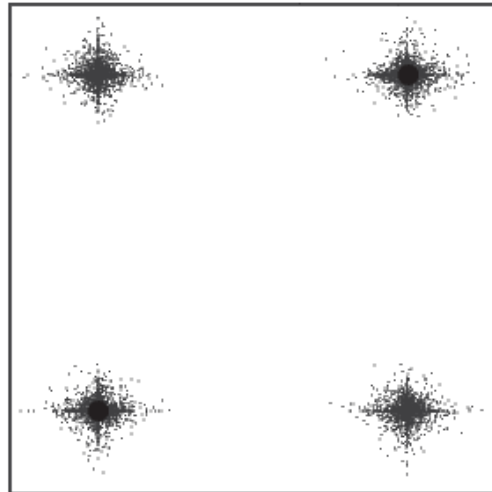
The following crossover and mutation operators are supported by the MOEA Framework.

Representation	Type	Abbr.
Real / Integer	Simulated Binary Crossover	sbx
Real / Integer	Polynomial Mutation	pm
Real / Integer	Differential Evolution	de
Real / Integer	Parent-Centric Crossover	pcx
Real / Integer	Simplex Crossover	spx
Real / Integer	Unimodal Normal Distribution Crossover	undx
Real / Integer	Uniform Mutation	um
Real / Integer	Adaptive Metropolis	am
Binary	Half-Uniform Crossover	hux
Binary	Bit Flip Mutation	bf
Permutation	Partially-Mapped Crossover	pmx
Permutation	Element Insertion	insertion
Permutation	Element Swap	swap
Subset	Subset Crossover	ssx
Subset	Subset Replacement	replace
Grammar	Single-Point Crossover for Grammars	gx
Grammar	Uniform Mutation for Grammars	gm
Program	Branch (Subtree) Crossover	bx
Program	Point Mutation	ptm
Any	Single-Point Crossover	1x
Any	Two-Point Crossover	2x
Any	Uniform Crossover	ux

B.1 Real-Valued Operators

Simulated Binary Crossover (SBX)

SBX attempts to simulate the offspring distribution of binary-encoded single-point crossover on real-valued decision variables (Deb and Agrawal, 1994). It accepts two parents and produces two offspring. An example of this distribution, which favors offspring nearer to the two parents, is shown below.



The distribution index controls the shape of the offspring distribution. Larger values for the distribution index generates offspring closer to the parents.

Parameters	Description	Default Value
sbx.rate	The probability that the SBX operator is applied to a decision variable	1.0
sbx.distributionIndex	The shape of the offspring distribution	15.0

Polynomial Mutation (PM)

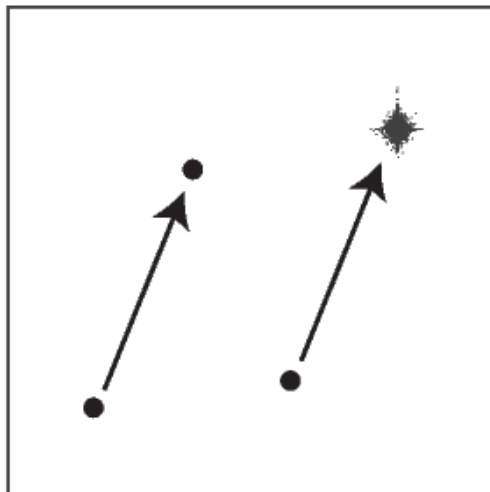
PM attempts to simulate the offspring distribution of binary-encoded bit-flip mutation on real-valued decision variables (Deb and Goyal, 1996). Similar to SBX, PM favors offspring nearer to the parent. It is recommended each decision variable is mutated with a probability of $1/N$, where N is the number of decision variables. This results in one mutation per offspring on average.

The distribution index controls the shape of the offspring distribution. Larger values for the distribution index generates offspring closer to the parents.

Parameters	Description	Default Value
pm.rate	The probability that the PM operator is applied to a decision variable	$1/N$
pm.distributionIndex	The shape of the offspring distribution (larger values produce offspring closer to the parent)	20.0

Differential Evolution (DE)

Differential evolution works by randomly selecting three distinct individuals from a population. A difference vector is calculated between the first two individuals (shown as the left-most arrow in the figure below), which is subsequently applied to the third individual (shown as the right-most arrow in the figure below).

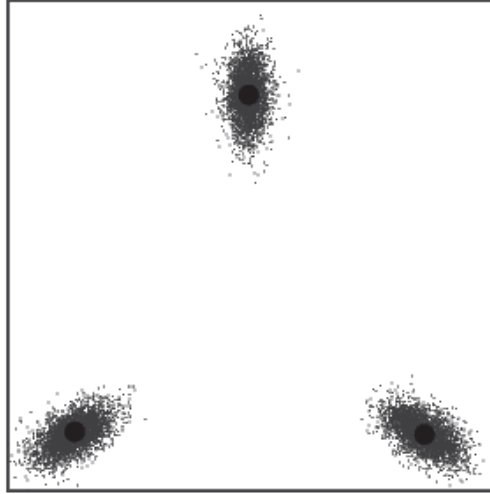


The scaling factor parameter adjusts the magnitude of the difference vector, allowing the user to decrease or increase the magnitude in relation to the actual difference between the individuals (Storn and Price, 1997). The crossover rate parameter controls the fraction of decision variables which are modified by the DE operator.

Parameters	Description	Default Value
de.crossoverRate	The fraction of decision variables modified by the DE operator	0.1
de.stepSize	The scaling factor or step size used to adjust the length of each step taken by the DE operator	0.5

Parent Centric Crossover (PCX)

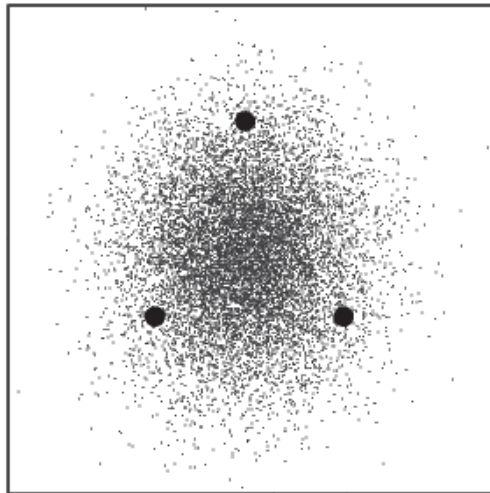
PCX is a multiparent operator, allowing a user-defined number of parents and offspring (Deb et al., 2002). Offspring are clustered around the parents, as depicted in the figure below.



Parameters	Description	Default Value
pcx.parents	The number of parents	10
pcx.offspring	The number of offspring generated by PCX	2
pcx.eta	The standard deviation of the normal distribution controlling the spread of solutions in the direction of the selected parent	0.1
pcx.zeta	The standard deviation of the normal distribution controlling the spread of solutions in the directions defined by the remaining parents	0.1

Unimodal Distribution Crossover (UNDX)

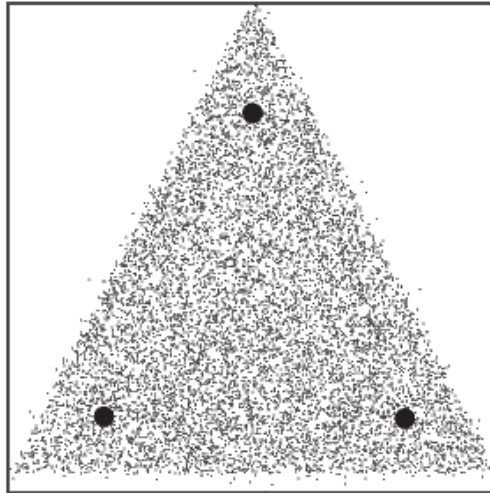
UNDX is a multiparent operator, allowing a user-defined number of parents and offspring (Kita et al., 1999; Deb et al., 2002). Offspring are centered around the centroid, forming a normal distribution whose shape is controlled by the positions of the parents, as depicted in the figure below.



Parameters	Description	Default Value
undx.parents	The number of parents	10
undx.offspring	The number of offspring generated by UNDX	2
undx.zeta	The standard deviation of the normal distribution controlling the spread of solutions in the orthogonal directions defined by the parents	0.5
undx.eta	The standard deviation of the normal distribution controlling the spread of solutions in the remaining orthogonal directions not defined by the parents. This value is divided by \sqrt{N} prior to use, where N is the number of decision variables.	0.35

Simplex Crossover (SPX)

SPX is a multiparent operator, allowing a user-defined number of parents and offspring (Tsutsui et al., 1999; Higuchi et al., 2000). The parents form a convex hull, called a simplex. Offspring are generated uniformly at random from within the simplex. The expansion rate parameter can be used to expand the size of the simplex beyond the bounds of the parents. For example, the figure below shows three parent points and the offspring distribution, clearly filling an expanded triangular simplex.



Parameters	Description	Default Value
spx.parents	The number of parents	10
spx.offspring	The number of offspring generated by UNDX	2
spx.epsilon	The expansion rate	3

Uniform Mutation (UM)

Each decision variable is mutated by selecting a new value within its bounds uniformly at random. The figure below depicts the offspring distribution. It is recommended each decision

variable is mutated with a probability of $1/N$, where N is the number of decision variables. This results in one mutation per offspring on average.

Parameters	Description	Default Value
um.rate	The probability that the UM operator is applied to a decision variable	$1/N$

Adaptive Metropolis (AM)

AM is a multiparent operator, allowing a user-defined number of parents and offspring (Vrugt and Robinson, 2007; Vrugt et al., 2009). AM produces normally-distributed clusters around each parent, where the shape of the distribution is controlled by the covariance of the parents.

Internally, the Cholesky decomposition is used to update the resulting offspring distribution. Cholesky decomposition requires that its input be positive definite. In order to guarantee this condition is satisfied, all parents must be unique. In the event that the positive definite condition is not satisfied, no offspring are produced and an empty array is returned by

Parameters	Description	Default Value
am.parents	The number of parents	10
am.offspring	The number of offspring generated by AM	2
am.coefficient	The jump rate coefficient, controlling the standard deviation of the covariance matrix. The actual jump rate is calculated as $(am.coefficient/\sqrt{n})^2$	2.4

B.2 Binary / Bit String Operators

Half Uniform Crossover (HUX)

Half-uniform crossover (HUX) operator. Half of the non-matching bits are swapped between the two parents.

Parameters	Description	Default Value
hux.rate	The probability that the UM operator is applied to a binary decision variable	1.0

Bit Flip Mutation (BF)

Each bit is flipped (switched from a 0 to a 1, or vice versa) using the specified probability.

Parameters	Description	Default Value
bf.rate	The probability that a bit is flipped	0.01

B.3 Permutations

Partially Mapped Crossover (PMX)

PMX is similar to two-point crossover, but includes a repair operator to ensure the offspring are valid permutations (Goldberg and Jr., 1985).

Parameters	Description	Default Value
pmx.rate	The probability that the PMX operator is applied to a permutation decision variable	1.0

Insertion Mutation

Randomly selects an entry in the permutation and inserts it at some other position in the permutation.

Parameters	Description	Default Value
insertion.rate	The probability that the insertion operator is applied to a permutation decision variable	0.3

Swap Mutation

Randomly selects two entries in the permutation and swaps their position.

Parameters	Description	Default Value
swap.rate	The probability that the swap operator is applied to a permutation decision variable	0.3

B.4 Subsets

Subset Crossover (SSX)

SSX is similar to HUX crossover for binary strings, where half of the non-matching members are swapped between the two subsets.

Parameters	Description	Default Value
ssx.rate	The probability that the SSX operator is applied to a subset decision variable	0.9

Replace Mutation

Randomly replaces one of the members in the subset with a non-member.

Parameters	Description	Default Value
replace.rate	The probability that the replace operator is applied to a subset decision variable	0.3

B.5 Grammars

Grammar Crossover (GX)

Single-point crossover for grammars. A crossover point is selected in both parents with the tail portions swapped.

Parameters	Description	Default Value
gx.rate	The probability that the GX operator is applied to a grammar decision variable	1.0

Grammar Mutation (GM)

Uniform mutation for grammars. Each integer codon in the grammar representation is uniformly mutated with a specified probability.

Parameters	Description	Default Value
gm.rate	The probability that the GM operator is applied to a grammar decision variable	1.0

B.6 Program Tree

Subtree Crossover (BX)

Exchanges a randomly-selected subtree from one program with a compatible, randomly-selected subtree from another program.

Parameters	Description	Default Value
gm.rate	The probability that the BX operator is applied to a program tree decision variable	1.0

Point Mutation (PTM)

Mutates a program by randomly selecting nodes in the expression tree and replacing the node with a new, compatible, randomly-selected node.

Parameters	Description	Default Value
gm.rate	The probability that the PTM operator is applied to a program tree decision variable	1.0

B.7 Generic Operators

Generic operators can be applied to any type. They work by simply swapping the value of the decision variable between the parents.

One-Point Crossover (1X)

A crossover point is selected and all decision variables to the left/right are swapped between the two parents. The two children resulting from this swapping are returned.

Parameters	Description	Default Value
1x.rate	The probability that one-point crossover is applied to produce offspring	1.0

Two-Point Crossover (2X)

Two crossover points are selected and all decision variables between the two points are swapped between the two parents. The two children resulting from this swapping are returned.

Parameters	Description	Default Value
2x.rate	The probability that two-point crossover is applied to produce offspring	1.0

Uniform Crossover (UX)

Crossover operator where each index is swapped with a specified probability.

Parameters	Description	Default Value
ux.rate	The probability that uniform crossover is applied to produce offspring	1.0

Preview - Pages Removed

Bibliography

- Bäck, T., Fogel, D. B., and Michalewicz, Z. (1997). *Handbook of Evolutionary Computation*. Taylor & Francis, New York, NY.
- Beume, N., Naujoks, B., and Emmerich, M. (2007). Sms-emoa: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669.
- Beume, N. and Rudolph, G. (2006). Faster S-metric calculation by considering dominated hypervolume as Klee’s measure problem. In *Second International Association of Science and Technology for Development (IASTED) Conference on Computational Intelligence*, pages 231–236, San Francisco, CA.
- Bosman, P. A. and Thierens, D. (2003). The balance between proximity and diversity in multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 7(2):174–188.
- Coello Coello, C. A., Lamont, G. B., and Van Veldhuizen, D. A. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer Science+Business Media LLC, New York, NY.
- Corne, D. W., Jerram, N. R., Knowles, J. D., and Oates, M. J. (2001). PESA-II: Region-based selection in evolutionary multiobjective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 283–290, San Francisco, CA.
- Corne, D. W. and Knowles, J. D. (2000). The Pareto envelope-based selection algorithm for multiobjective optimization. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature (PPSN VI)*, pages 839–848, Paris, France.
- Deb, K. and Agrawal, R. B. (1994). Simulated binary crossover for continuous search space. Technical Report No. IITK/ME/SMD-94027, Indian Institute of Technology, Kanpur, India.
- Deb, K., Anand, A., and Joshi, D. (2002). A computationally efficient evolutionary algorithm for real-parameter optimization. *Evolutionary Computation*, 10:371–395.

- Deb, K. and Goyal, M. (1996). A combined genetic adaptive search (geneas) for engineering design. *Computer Science and Informatics*, 26(4):30–45.
- Deb, K. and Jain, H. (2014). 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*, 18(4):577–601.
- Deb, K. and Jain, S. (2002). Running performance metrics for evolutionary multi-objective optimization. KanGAL Report No. 2002004, Kanpur Genetic Algorithms Laboratory (KanGAL), Indian Institute of Technology, Kanpur, India.
- Deb, K., Mohan, M., and Mishra, S. (2003). A fast multi-objective evolutionary algorithm for finding well-spread Pareto-optimal solutions. KanGAL Report No. 2003002, Kanpur Genetic Algorithms Laboratory (KanGAL), Indian Institute of Technology, Kanpur, India.
- Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2000). A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.
- Fonseca, C. M. and Fleming, P. J. (1993). Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the Fifth International Conference Genetic Algorithms (ICGA 1993)*, pages 416–423, Urbana-Champaign, IL.
- Fonseca, C. M. and Fleming, P. J. (1996). On the performance assessment and comparison of stochastic multiobjective optimizers. In *Parallel Problem Solving from Nature (PPSN IV)*, pages 584–593, Berlin, Germany.
- Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Reading, MA.
- Goldberg, D. E. and Jr., R. L. (1985). Alleles, loci, and the traveling salesman problem. In *1st International Conference on Genetic Algorithms and Their Applications*.
- Hadka, D. and Reed, P. (2012). Diagnostic assessment of search controls and failure modes in many-objective evolutionary optimization. *Evolutionary Computation*, 20(3):423–452.
- Hadka, D. and Reed, P. (2013). Borg: An auto-adaptive many-objective evolutionary computing framework. *Evolutionary Computation*, 21(2):231–259.
- Hansen and Kern (2004). Evaluating the cma evolution strategy on multimodal test functions. In *Eighth International Conference on Parallel Problem Solving from Nature PPSN VIII*, pages 282–291.
- Hansen, M. P., Hansen, M. P., Jaszkievicz, A., and Jaszkievicz, A. (1998). Evaluating the quality of approximations to the non-dominated set. Technical Report IMM-REP-1998-7, Technical University of Denmark.

- Higuchi, T., Tsutsui, S., and Yamamura, M. (2000). Theoretical analysis of simplex crossover for real-coded genetic algorithms. In *Parallel Problem Solving from Nature (PPSN VI)*, pages 365–374.
- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.
- Horn, J. and Nafpliotis, N. (1993). Multiobjective optimization using the Niche Pareto Genetic Algorithm. IlliGAL Report No. 93005, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois, Urbana-Champaign, IL.
- Igel, C., Hansen, N., and Roth, S. (2007). Covariance matrix adaptation for multi-objective optimization. *Evolutionary Computation*, 15:1–28.
- Kennedy, J. and Eberhart, R. (1995). Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, Perth, Australia.
- Kita, H., Ono, I., and Kobayashi, S. (1999). Multi-parental extension of the unimodal normal distribution crossover for real-coded genetic algorithms. In *Proceedings of the 1999 Congress on Evolutionary Computation (CEC 1999)*, pages 1581–1588, Washington, DC.
- Knowles, J. and Corne, D. (2002). On metrics for comparing non-dominated sets. In *Congress on Evolutionary Computation (CEC 2002)*, pages 711–716, Honolulu, HI.
- Knowles, J. D. and Corne, D. W. (1999). Approximating the nondominated front using the Pareto Archived Evolution Strategy. *Evolutionary Computation*, 8:149–172.
- Kollat, J. B. and Reed, P. M. (2006). Comparison of multi-objective evolutionary algorithms for long-term monitoring design. *Advances in Water Resources*, 29(6):792–807.
- Kukkonen, S. and Lampinen, J. (2005). GDE3: The third evolution step of generalized differential evolution. In *The 2005 IEEE Congress on Evolutionary Computation (CEC 2005)*, pages 443–450, Guanajuato, Mexico.
- Laumanns, M., Thiele, L., Deb, K., and Zitzler, E. (2002). Combining convergence and diversity in evolutionary multi-objective optimization. *Evolutionary Computation*, 10(3):263–282.
- Li, H. and Zhang, Q. (2009). Multiobjective optimization problems with complicated Pareto sets, MOEA/D and NSGA-II. *IEEE Transactions on Evolutionary Computation*, 13(2):284–302.
- Miettinen, K. M. (1999). *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, Norwell, MA.

- Nebro, A. J., Durillo, J. J., García-Nieto, J., Coello Coello, C. A., Luna, F., and Alba, E. (2009). SMPSO: A new PSO-based metaheuristic for multi-objective optimization. In *IEEE Symposium on Computational Intelligence in Multicriteria Decision-Making (MCDM 2009)*, pages 66–73, Nashville, TN.
- Schaffer, D. J. (1984). *Multiple Objective Optimization with Vector Evaluated Genetic Algorithms*. PhD thesis, Vanderbilt University, Nashville, TN.
- Schaffer, D. J. (1985). Multiple objective optimization with vector evaluated genetic algorithms. In *1st International Conference on Genetic Algorithms*, pages 93–100.
- Sierra, M. R. and Coello Coello, C. A. (2005). Improving PSO-based multi-objective optimization using crowding, mutation and ϵ -dominance. In *Evolutionary Multi-Criterion Optimization (EMO 2005)*, pages 505–519, Guanajuato, Mexico.
- Srinivas, N. and Deb, K. (1994). Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248.
- Storn, R. and Price, K. (1997). Differential evolution — a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359.
- Tsutsui, S., Yamamura, M., and Higuchi, T. (1999). Multi-parent recombination with simplex crossover in real coded genetic algorithms. In *Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 657–664, Orlando, FL.
- Vrugt, J. A. and Robinson, B. A. (2007). Improved evolutionary optimization from genetically adaptive multimethod search. *Proceedings of the National Academy of Sciences*, 104(3):708–711.
- Vrugt, J. A., Robinson, B. A., and Hyman, J. M. (2009). Self-adaptive multimethod search for global optimization in real-parameter spaces. *IEEE Transactions on Evolutionary Computation*, 13(2):243–259.
- While, L., Bradstreet, L., and Barone, L. (2012). A fast way of calculating exact hypervolumes. *IEEE Transactions on Evolutionary Computation*, 16(1):86–95.
- Zhang, Q., Liu, W., and Li, H. (2009). The performance of a new version of MOEA/D on CEC09 unconstrained MOP test instances. In *Congress on Evolutionary Computation (CEC 2009)*, pages 203–208, Trondheim, Norway.
- Zitzler, E. and Künzli, S. (2004). Indicator-based selection in multiobjective search. In *Parallel Problem Solving from Nature (PPSN VIII)*, pages 832–842, Birmingham, UK.
- Zitzler, E., Laumanns, M., and Thiele, L. (2002a). *SPEA2: Improving the Strength Pareto Evolutionary Algorithm For Multiobjective Optimization*. International Center for Numerical Methods in Engineering (CIMNE), Barcelona, Spain.

- Zitzler, E., Laumanns, M., Thiele, L., Fonseca, C. M., and da Fonseca, V. G. (2002b). Why quality assessment of multiobjective optimizers is difficult. In *Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 666–674, New York, NY.
- Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271.
- Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2002c). Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.